From Information System Requirements to Designs: A Mapping Framework

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

November 1989

FORTH/CSI/TR/1989/020

Technical Report Series
Computer Science Institute, FORTH†
P.O.Box 1385, Heraklio, Crete, 711-10 Greece
Tel.: +30.81.229302, Fax: +30.81.229342, Telex: 262389 CCI GR
E-mail – UUCP: mcvaex/ariadneyannis

This work was partially supported by the ESPRIT project DAIDA (No 892).
For two of the authors (Lawrence Chang and John Mylopoulos) the Institution is: Department of Computer Science,
University of Toronto, Canada
† "Foundation of Research and Technology – Hellas," formerly "Research Center of Crete" (RCC).
From Information System Requirements to Designs: A Mapping Framework

Lawrence Chung², Panagiotis Katalagaris³, Manolis Marekakis³, Michalis Mertikas⁴, John Mylopoulos², Yannis Vassiliou³

²Department of Computer Science, University of Toronto, 10 King’s College Road, Toronto, Ontario, Canada M5S 1A4
³Institute of Computer Science, Foundation for Research and Technology – Hellas, P. O. Box 1585, GR – 711 10, Heraklion, Crete, Greece.

Abstract

Information system development should address adequate representation of requirements specifications and their use in generating designs. In our framework, requirements specifications are assumed to include a functional description of what the information system is intended to do, how it will interact with its environment, what information it will manage and how that information relates to the system’s environment.

The generation of a design is achieved by mapping elements of the requirements model into one or more corresponding design objects. This mapping process is guided by two considerations. Locally, the process is directed by dependency types among requirements and design objects which determine allowable mappings for a particular requirements object. Globally, the process is guided by non-functional requirements, such as accuracy and security requirements on the intended system, which are represented as goals describing desirable properties of the intended system. Satisficing methods for these goals are used to guide local mapping decisions.

The paper includes the description of a prototype implementation — called IRIS — of aspects of the proposed mapping framework and illustrates its features through a sample session. The implementation was carried out within the DAIDA project at the Institute of Computer Science of the Foundation for Research and Technology, Crete.

¹This paper is based on results from the DAIDA project, funded in part by the European Commission through the ESPRIT programme under contract no. 892; financial support for this research was also received from the Institute of Computer Science of the Foundation for Research and Technology – Hellas (FORTH), the Natural Sciences and Engineering Research Council of Canada and the University of Toronto.
Contents

1 Introduction .......................................................... 2
   1.1 Context .......................................................... 2
   1.2 The Problem ...................................................... 4
   1.3 The Solution ...................................................... 5
   1.4 Related Work ..................................................... 5
   1.5 Paper Structure .................................................. 7

2 Requirements and Design Languages ............................. 7
   2.1 Requirements Modelling Languages: Telos .................. 7
   2.2 Conceptual Design Languages: TDL .......................... 11

3 A Dependency-Based Mapping Framework ....................... 13
   3.1 Representation of Mapping Dependencies and Types ...... 15
   3.2 Representation of Design Objects ......................... 17
   3.3 The Mapping Process and System Architecture of IRIS .... 20
   3.4 Mapping Refinements ......................................... 20
   3.5 A Sample Session with IRIS and Implementation Status ... 22

4 Dealing with Non-Functional Requirements .................. 28
   4.1 A Model of Accuracy .......................................... 28
   4.2 Goal-Directed Mapping ....................................... 33

5 Conclusions ......................................................... 36

Bibliography .......................................................... 38
List of Figures

1.1 The DAIDA Architecture ................................................. 3
3.1 Basic Telos Representation of the Mapping Task .................. 14
3.2 Two-layer representation of the design model in Telos .......... 18
3.3 General Architecture of the Mapping Assistant .................. 20
3.4 Telos Object Selection ................................................. 24
3.5 Attribute Display ...................................................... 25
3.6 Dependency Graph from the Mapping of &GenExpenseReport ..... 27
4.1 An Example of information flow ...................................... 29
4.2 The goal tree generated for the example ......................... 35
1 Introduction

1.1 Context

**Theme** Large information systems, such as those for hospital admission, inventory control, airline reservation, and public administration, constitute a steadily growing and strategic component of the software market. Such systems are traditionally built using a three-stage process consisting of requirements modelling and analysis, conceptual design, and implementation. These three stages have appeared in various forms both in the data engineering literature (e.g., [New Orleans78]) and in the software engineering literature (e.g., [Royce70], [Sommerville89]).

This paper focuses on the mapping of information system requirements specification into a design. Its goal is to present a framework for interactively generating a design from a requirements specification, with the human designer making decisions within a formal, computer-assisted framework. To establish this goal, the paper first refines the notions of requirements modelling and design for information systems and describes appropriate languages for each. A formal framework is then presented where the generation of a design is achieved by mapping elements of the requirements model into one or more corresponding design objects. This mapping process is guided by two considerations. Locally, the process is directed by dependency types among requirements and design objects which determine allowable mappings for a particular requirements object. Globally, the process is guided by non-functional requirements, such as accuracy and security requirements on the intended system, which are represented as goals describing desirable properties of the intended system. Satisficing methods for these goals are used to guide local mapping decisions.

The paper includes the description of a prototype implementation — called IRIS\(^2\) — of aspects of the proposed mapping framework and illustrates its features through a sample session. The implementation was carried out within the DAIDA project at the Institute of Computer Science of the Foundation for Research and Technology, Crete.

The stated aim of the DAIDA project has been the development of a software engineering environment for building and maintaining information systems. The structure of the environment was based on the following premises:

1. Suitable languages need to be adopted, rather than developed from scratch, for requirements modelling, design, and implementation.

2. The environment needs to include a design record management system which maintains a complete design record for an information system, including its requirements and design specifications, its implementation as well as the interdependencies among them.

3. The environment needs to offer tools and corresponding methodologies for mapping requirements specifications onto designs and designs onto implementations. A proof of concept is also required, in the form of a prototype implementation.

The overall architecture of the DAIDA environment is shown on Figure 1.1. Telos, TDL, and DBPL\(^3\) 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, in the mapping of that model to a design, and later on an implementation. The GKBMS also maintains a

---

\(^2\)According to Greek mythology, Iris was a goddess, serving as personal assistant to Zeus.

\(^3\)More on the former languages later in the paper. DBPL is a database programming language developed at the University of Frankfurt, which offers a Modula-2-like programming framework extended with sophisticated relational database management facilities [Schmidt88].
complete 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 described in [Borgida89], while the GKBMS component is presented in [Jarke88] [Jarke89]. A companion paper to this one, [Borgida90], describes the mapping framework from designs to implementations proposed by DAIDA. Finally, details about IRIS, the prototype mapping assistant developed at FORTH, can be found in [Mertikas89] and [Katalagarianos90].

Requirements Specifications and System Designs A requirements specification, according to the DAIDA framework, includes both functional and non-functional requirements. Functional requirements for information systems are assumed to include a description of the intended functionality of the system under development (hereafter the system model), but also a description of the environment within which the system will eventually function (called in the sequel the environment model or world model) as well as 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 adopt languages which offer facilities for "world modelling", such as knowledge representations or semantic data models.

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 ([Roman85]).

To illustrate the distinction between functional and non-functional requirements, consider a hypothetical expense report system for research projects: functional requirements for the environment

---

4. This premise is well supported in the literature. See, for example, [Zave81], [Jackson81], and [Greenspan84], among others.
may state that members of the projects should participate in various meetings held in various countries and submit their expense summaries together with the actual expense vouchers, while functional requirements for the embedded system may specify the generation of monthly expense reports for each member and for each project. Functional requirements for the interaction may require expense summaries in the environment to be submitted via electronic mail by each member. Non-functional requirements, on the other hand, 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.

1.2 The Problem

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 these refinements, the generation of a system design is an extremely difficult and error-prone 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 chosen for requirements modelling and system design hopefully share representation features, such as generalization hierarchies. Methods need to be developed for the exploitation of such commonalities in order to simplify the mapping task.

4. Justification of designs: Each design should be, as much as possible, consistent to the requirements specification from which it was generated. However, no formal framework is available to date for justifying design decisions on the basis of functional and non-functional requirements.

It should be noted that not all the parts of a requirements specification are mapped down to a corresponding design. The mapping applies to components of only the systems model. The rest of the requirements specification prescribes the nature of interactions between the information system and its environment and the meaning of the information maintained by the system.

5“Prescribes” in the sense that the environment is expected to behave consistently with the requirements specification. For instance, a requirement specification may state that a student registration system is updated weekly. When the environment doesn't behave according the requirements specification, problems arise which may render the system useless (because, for instance, its databases are out of date) or even inoperational (for example, because of missing data).
1.3 The Solution

This paper outlines a *dependency-based, goal-oriented* framework to the mapping problem. The framework is *dependency-based* in the sense that the mapping of parts of the requirements specification into a design is guided by predefined allowable dependencies. Data entities in the design, for example, may only be derived from, and therefore dependent upon, entities in the functional requirements model, while activities may only be mapped onto transactions or scripts.

At the same time, the framework 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 framework 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 framework (e.g., accuracy, security, operational, performance). The feasibility of this framework is illustrated in the paper for accuracy requirements only. However, the proposed treatment applies to other types of non-functional requirements.6

It is important to note that the proposed framework can be used as the basis for CASE tools that aid the information system designer by confirming the consistency of her decisions with standard mapping knowledge and by maintaining information on interdependencies between decisions and elements of requirements and design specifications. These interdependencies can serve as basis for the justification of designs and for the rational evolution of the system under development.

1.4 Related Work

In the field of software development, much attention has been given to the generation of efficient implementations from formal specifications, either manually or automatically. For example, [Spivey87] and [Worlsworth87] describe a software development methodology based on a set-theoretic formalism named Z, while [Abrial88] describes a tool (named the B tool) that can serve as proof assistant in establishing that an implementation generated through a sequence of transformations is indeed consistent with a given Z specification.7 Another project in the same league is the Cornell Program Synthesizer [Teitelman81] which inspired the Nuprl proof development system [Constable86]. Generally, such work focuses on programming-in-the-small applications and is most relevant to the mapping problem from designs to implementations. A fine survey of AI-related work on this problem, which has traditionally been called "automatic programming" can be found in [Barstow87], while [Hayes87] gives an excellent comparative account of several formal specification methodologies.

As one of the foremost advocates for the need of world modelling in software development, Jackson [Jackson83] proposes a methodology called JSD (Jackson's System Design) which is, in fact, in commercial use. JSD starts with a world model and produces a functional system design. However, JSD, as other diagrammatic approaches, adopts an ad hoc set of modelling features and is often quite informal in the specifications it starts with and produces. The work reported here may be seen as an attempt to go beyond the JSD methodology using concepts from knowledge representation and artificial intelligence.

Although the modelling concepts adopted are not as powerful as in DAIDA, the work by Bubenko [Bubenko81] is close to DAIDA in spirit, using layers of formal software specifications as a basis for the definition of a software development process. The first layer of Bubenko's proposal comprises an abstract world model, called "understanding level", and the second layer comprises a conceptual system design. In order to define model behaviour in non-procedural terms, time is considered as the most essential concept and the world model is described in an extended time perspective rather than

---

6Indeed, we already have results on its application for the representation and use of security requirements.
7Both Z and the B tool play an important role for the design-to-implementation mapping assistant of DAIDA [Borgida89].
in points of procedure invocation. Unlike JSD and the framework reported here, however, Bubenko offers no methodology for mapping the first layer to the second.

The Gandalf project at CMU [Haberman86] includes a component called SMILE which consists of a multi-user software engineering environment. SMILE is intended to offer intelligent assistance to the software developer. However, unlike DAIDA whose user support is based on a knowledge base (managed by the GKBMS), the knowledge used by SMILE is hardcoded and cannot be changed or extended easily. [Kaiser87] describes a more recent effort to generalize SMILE by including an objectbase and a model of the software development activity which represent explicitly some of the hardcoded knowledge of SMILE. This effort is similar in spirit to that of DAIDA. The two projects diverge in that DAIDA focuses on information systems and takes full advantage of its narrow focus in selecting languages, in developing tools, and in adopting methodologies.

The CHI project [Smith85] [Green85], a descendant of PSI project carried out earlier at Stanford, is led by Green at Kestrel Institute [Green76], and is based on a wide-spectrum language, called V, that is executable. The starting point here for the software development process is a formal specification consisting of logical assertions which is mapped, via several transformations, to an efficient LISP implementation [Westfold84]. However, the logical assertions that constitute the initial formal specification tend to focus on the behaviour of the software-to-be-built rather than its intended environment and information content and CHI relies on computational notions such as sequencing, compute-versus-store, etc., to make them executable. Thus, CHI, like other projects focusing on programming-in-the-small applications, bypasses completely the requirements modelling and analysis phase and only addresses the problem of going from a less deterministic description of the software-to-be-built to a completely deterministic (and efficient) one. Also, as pointed out earlier, the CHI project does not offer much guidance on how conceptual system designs are produced. On the other hand, the project does have to its credit a commercial product called REFINE.

The TI (Transformational Implementation) project [Balzer81] [Fickas85] led by Balzer at USC ISI is one of the foremost automatic programming projects in the world, attempting to develop software through the interaction of system builders and automated assistants. A main theme of the project is the efficient implementation of conceptual system designs constructed in terms of a wide-spectrum executable language, called Gist. Gist is based on state-transitions coupled with the entity-relationship model. Specification refinements are achieved by constraining the allowable state transitions. While treating each design component as a goal to fulfill, the project has generated a set of design alternatives (or methods) and a set of criteria for design decisions that form the backbone for a software development methodology.

KATE is one of several recent projects which address programming-in-the-large. It is led by Fickas at the University of Oregon [Fickas87] [Fickas88] and its scope extends that of the TI project by considering requirements specifications. A fuller specification gradually evolves from an incomplete and inconsistent one, and that is later transformed into a specification that can serve as starting point for the TI framework. A major difference between DAIDA on one hand and TI and KATE on the other is that Gist is a wide-spectrum executable language while Telos, used in DAIDA to represent knowledge about an information system design record, is a declarative and definitely non-executable knowledge representation language. Moreover, although Fickas acknowledges the need to consider non-functional requirements in generating conceptual system designs, no concrete framework has been offered as yet for this task.

In conclusion, there are three distinguishing features of the research reported here compared to other research reported in the literature. Firstly, the focus of the work is information systems rather than software development in general. Secondly, the problem being addressed is the generation of designs from requirements specifications rather than the generation of implementations from designs.

---

Footnote 8: A language is wide-spectrum if it offers constructs for specifications ranging from highly abstract and non-procedural to ones that are implementation-oriented [Bauer76].
Finally, this work adopts a state-of-the-art knowledge representation language (Telos) to represent the knowledge required to assist the mapping process.

1.5 Paper Structure

Section 2 briefly introduces Telos and TDL and outlines the nature of requirements and conceptual designs for information systems. Section 3 describes the dependency-based, goal-oriented mapping framework adopted and presents through sample sessions the prototype implementation that has been completed. Section 4 addresses the problem of representation and use of non-functional requirements, focusing on accuracy, while section 5 summarizes the results of this work and outlines directions for further research.

2 Requirements and Design Languages

In order to illustrate our approach to system development, we sketch the main features of the requirements and design languages adopted. It is important to stress at this point that the mapping framework presented depends primarily on the ontological and structural features which determine the source and target levels of the mapping, rather than the particular languages chosen.

2.1 Requirements Modelling Languages: Telos

As indicated in the introduction, requirements specifications prescribe not only the behaviour of the system under development, but also the environment within which it will function. They also include a model of the world about which information is maintained in the system. The language to be adopted, then, must have facilities for “world modelling”, including the notions of entities, relationships, time, activities, and the like. In addition, it was felt that the language must offer associated inference mechanisms so that questions can be answered and properties proven about a given requirements specification within a formal setting. Lastly, it was decided that the language must support structuring mechanisms such as aggregation, generalization and classification to facilitate the task of building (usually large) requirements specifications. The particular language adopted for the DAIDA project and this paper is Telos [Mylopoulos80]\(^4\). In addition to the features mentioned above, Telos includes some novel features, such as its treatment of attributes, which are promoted to first class status, and the provision of special representational and inferential facilities for temporal knowledge.

A Telos knowledge base consists of propositions, which are atomic units that can be used to represent an entity or an elementary binary relationship in the application domain. A Telos proposition is a 4-tuple \(<source, label, target, time>\), representing an atomic fact. For example, the fact “John's age was 17 during 1990” might be represented by the proposition <John, age, 17, 1990>. Individuals too, like “John” and “17” are represented by propositions. Propositions can be combined to form complex descriptions, consisting of categorized (multi-valued) attributes, constraints, and deductive rules. These descriptions are in turn organized through classification and generalization. Moreover, descriptions in the 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 the instantiation hierarchy. Starting with the bottom layer of tokens, the hierarchy is extended with the layers of simple classes — having tokens as

\(^4\)Telos evolved from RML, a requirements modelling language based on knowledge representation ideas [Greenspan84].
instances, $M_1$, classes — having simple classes as instances. $M_2$, classes, .... The top layer of the hierarchy includes $Omega$, classes which have instances from several layers below.

Telos allows the designer to view a requirements specification as an historical account of events and activities in the application. This historical account is facilitated by Telos since every proposition includes a time interval representing the lifetime of the denoted entity or relationship. Time intervals can be related to each other in a knowledge base through temporal relations such as $before$, $during$, and $overlaps$ [Allen83]. The inference mechanisms built into Telos support temporal reasoning with respect to such relations, such as "If A is before B and B before C then A is before C".

As an example of Telos use, consider the following description of a (research) project, treated here as an activity:

```
IndividualClass Project in ActivityClass with
  input, single
    budg: Budget
  output, single
    finalReport: ResearchReport
  control, single
    nm: Name
    beginTime, endTime: Date
  control
    participant: Organization
    workers: Employee
  part
    meet: HaveMeeting;
    genExpenseReport: GenExpenseReport
  precondition
    : ProposalAccepted(this) {"this" refers to an instance of Project *}
  activationCondition
    : $now during startTime$
  postCondition
    : $budg.amountLeft = 0 after endTime$
  terminationCondition
    : $now after endTime$
  necessary
    : $meet during this$
    : {"expense reports are produced every six months *}
    : $(\forall x/GenExpenseReport)
      (x meets this.beginTime=6mo) or
      (Exists y/GenExpenseReport) (x meets y + 6mo)$

end Project
```

Project is declared here as an instance of IndividualClass and ActivityClass, both predefined metaclasses, hence its instances are individuals and represent on-going activities, within some organization. The generic attributes (attribute classes) associated with Project are listed within the declaration using a <label>: <type> syntax. For instance, budg: Budget allows instances of Project to have budg attributes with values which are instances of the class Budget. Generic attributes are grouped into attribute categories which impose constraints on their instances. In the example, input attributes must exist at the time a project begins and are "consumed" by the project\textsuperscript{10}. Attributes are in general multi-valued, but can be constrained to accept a single value.

According to its definition, Project has two sub-activities: HaveMeeting refers to those meeting activities attended by a project's members; while GenExpenseReport refers to those activities

\textsuperscript{10}This feature is borrowed from SADT [Ross77]. SADT is a trademark of Softech, Inc.
intended to generate expense reports. In addition, Project has a number of constraints that are classified into attribute categories such as preCondition, activationCondition, and necessary. As suggested by their name, preconditions must be true at the time of the creation of another Project instance. Activation conditions, on the other hand, trigger the creation of Project instances when they become true, while necessary conditions are constraints to be enforced at all times for Project instances.

In our example, the precondition ProposalAccepted(thid) simply checks that the predicate ProposalAccepted is true for an entity about to become an instance of Project. The activation condition, on the other hand, is true when the (atomic) interval referred to by the special variable now falls completely within the time interval specified as start time for the project. Moving on to the necessary constraints of Project, all meet activities for a project must occur within the lifetime of the project, and a GenExpenseReport activity occurs every six months after the initiation of the project.

Note that the attribute categories of output, input, single, preCondition, etc., are not built into the language. Rather, they can be defined as attribute metaclasses, thanks to the special treatment of attributes within the Telos representational framework. For example, single can be defined as an attribute metaclass with a constraint that specifies something like:

"Let Attr be an instance of single (here, Attr is a single-valued attribute):
If attr1, attr2 are instances of Attr, then from(attr1) ≠ from(attr2)"

where from(attr) refers to the source of the attribute attr. For purposes of presentation, it is assumed that a number of attribute categories used in the sequel have already been defined.

The class Project above is part of our sample requirements specification, called research expense management system, used throughout the paper. Note that the class Project is part of a functional requirements specification. More specifically, the class is part of the environment model of the functional requirements specification in the sense that it describes an activity which takes place in the environment of the intended system and which affects the operation of the system.

Within a research expense management system, among other things, members of projects participate in meetings (held in different countries) and submit their expense summaries together with actual expense vouchers. ExpenseSummary below is another entity class of the environment model representing expense summaries. The attributes of expense summaries specify the participant involved, the meeting, the amount, and the date or period when the expense was incurred:

IndividualClass ExpenseSummary in S_class, EntityClass with
necessary, single
  amount: Money
  participant: Member
  meet: HaveMeeting
  expDate: Date
end ExpenseSummary

Next we present fragments of a system model description. It is assumed that monthly expense reports must be generated by the system for each member, for each meeting, and for each project. Throughout the discussion, entity and activity classes that are part of the system model are declared to be instances of SystemClass. Note that within the system, &Project is chosen not to be an activity but rather an entity class. Allowing for multiple representations (views) in requirements is considered good modelling practice. 

11As a notational convenience, classes which are part of the system model within a requirements specification have identifiers prefixed by "&".
IndividualClass &MonMbrExpRpt
  in S_Class, ActivityClass, SystemClass
  isA &GenExpenseReport with

  input
    exp: &Expense
    m: &Month
  control
  pers: &Employee
  output
    exrep: &MonMbrExpRpt
  activationCondition
    : (Exists t/Date)(now during t and
      LastDayOfMonth(t, m))

end &MonMbrExpRpt

This is a system activity that generates monthly reports. Its inputs are expense reports, its output the monthly report and it is activated every last day of the month.

IndividualClass &Project in S_Class, EntityClass, SystemClass with
  rep [with imports  participant, workers, nm, beginTime, endTime, budg, mt]
    : Project
  necessary, single
  nn: &Name
  beginTime, endTime: &Date
  budg: &Number
  budgetLeft: &Number
  necessary
    participant: &Organization
    workers: &Employee
  attribute  {* meetings held for this project *}
    mt: &Meeting
end &Project

IndividualClass &Expense in S_Class, EntityClass, SystemClass with
  rep [with imports  amount, participant, expDate, meet]
    : ExpenseSummary
  necessary, single
    amount: &Money;
    participant: &Employee;
    expDate: &Date;
    meet: &Meeting
  ...
end &Expense

&Project is a system class which will maintain information within the system under development about on-going projects. Note that, rep is an attribute metaclass that postulates a one-to-one correspondence between instances of &Project within the system model, and those of Project within the environment model.

Lastly we describe the interaction between the environment and the system through the interaction model of the functional specification. Expense summaries in the environment will be submitted via electronic mail by each member in the environment to an interface activity in the embedded system.

IndividualClass &ReceiveExpense
2.2 Conceptual Design Languages: TDL

Design specifications, according to the DAIDA world view, offer a conceptual design for the information system by structuring the data and transactions which constitute the system according to their intended meaning rather than their implementation. The Taxis Design Language (TDL in the sequel) supports the development of such specifications by offering a uniform semantic data model for describing data, transactions, and long-term processes. As with many other semantic data models, TDL model offers the notions of entities and relationships along with structuring mechanisms like aggregation, generalization, and classification. A major advantage of semantic data models over traditional record-based approaches is the direct and natural correspondence between the structure of a given design and (aspects of) the application domain, which facilitates the design, access, and interpretation of the information handled by the system [Borgida85].

TDL offers a variety of data classes for modeling the entities that are relevant to the application domain, including conventional data types (Integer, String, enumerated, and subrange types), but also labelled Cartesian products (or aggregate classes), whose instances have equality determined structurally, and entity classes which have their extensions (collections of instances) externally updated. Thus, calendar months can be represented as an enumerated class, addresses as Cartesian products of Number, Street, City, and Country, and employees as an entity class. As with other semantic data models, entities have an intrinsic, unchanging identity. Each type of data class has associated attribute categories which indicate the kinds of attributes that are applicable to their instances. For example, UNCHANGING attributes cannot be modified, once assigned a value; CHANGING attributes, in contrast, may get different values at states change; UNIQUE attributes are required to have different values for each instance of the class. Subclass hierarchies are supported and their use is encouraged throughout the design.

To effect state changes, functions and two types of procedures are also offered: transactions, intended to define atomic database operations, and scripts, intended to model long-term processes. TDL attempts to maintain uniformity by casting transactions in the same mold as entities: A transaction is viewed as an entity, with its parameters, conditions, and actions defined by its attributes. As in Simula and its derivatives, procedure definitions become class descriptions; hence, procedure invocations are represented with instances of these classes. Most innovatively, transactions are also organized into generalization hierarchies; for example, organizing a research project meeting is a specialization of organizing scientific meetings in general. Transaction parameters appear as attributes with the attribute categories IN and OUT indicating whether the parameter is used to communicate information into or out of the transaction. Each transaction specifies a set of allowable state transitions through pre/postcondition constraints on the values of the state variables (attribute functions, parameters, and class extensions). In TDL such constraints are expressed through the attribute categories GIVEN (conditions required to hold in the initial state when the transaction is invoked) and GOALS (conditions required to hold in the final state). Both conditions are expressed as logical assertions, where goals relate values in both the initial and final state. The insertion and removal of objects from the extents of classes is expressed through two attribute categories, PRODUCES and CONSUMES. In order to detect integrity constraint violations, or to avoid roll-backs, transactions
have INITIAL and FINAL constraints which are checked dynamically before and after each transaction execution. Failure by one of these constraints leads to the invocation of exception handlers.

TDL assumes frame axioms for its transactions [Nilsson74]. This means that the semantics of transactions have a built-in assumption which states that system data that are not affected by a transaction are assumed to remain unchanged. Frame axioms are not built into Teles. This constitutes perhaps the most fundamental difference the two languages. Transactions also constitute a point of difference between TDL and Taxis [Mylopoulos80], which was the design language originally adopted for DAIDA. Taxis offers a procedural sublanguage for the definition of transactions. This, it was felt, preempts implementation alternatives and makes the inclusion of both design and implementation levels within the DAIDA framework partly redundant. Accordingly, the procedural sublanguage of Taxis was discarded to be replaced by pre/postcondition assertions. In addition, unlike Taxis, TDL offers set-theoretic expressions which makes its expression sublanguage more like that of Teles.

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.

Scripts are integrated completely into the TDL framework: script classes are organized into a subclass hierarchy according to their generality/specificity, have states and transitions defined in terms of attributes, and have instances that can be accessed through the same facilities used to access instances of entity classes. Among other things, this allows queries concerning the currently executing set of scripts. Through scripts, the designer has access to a triggering mechanism which allows the expression of statements such as:

"IF there is a message with data about a new expense THEN create a new instance of class Expense; deal with possible errors as follows ...;"

"IF 6 months have passed THEN invoke the appropriate GenerateProjectSummary procedures; format and print the results in a report ...;"

A simple example of a TDL specification follows and is intended to illustrate several features of the language. The example consists of three entity class definitions (Employee, Expense, Project) and one transaction definition (GenMonExpRep).

ENTITY CLASS Employee ISA Person WITH
UNCHANGING
    eName : String;
CHANGING
    worksOn : SETOF Project;
END

In the above definition, Employee is a subclass of Person with two attributes of eName and worksOn.

ENTITY CLASS Expense WITH
UNCHANGING
    mt: Meeting;
    ...;
CHANGING
    transactionTime, validityTime: TimeInterval;
INITIAL
    validSubmission? : True /* 'THIS' is a variable ranging over Expense */
    BEFORE (THIS.transactionTime.from, mt.endTime Plus 1mo);
INVARIANT

12
validExp? : True
  (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);
END

The invariant validExp? in the Expense definition enforces the projects on which an employee works to be a subset of the projects his company is engaged in. The constraint that each instance of a class needs to have a unique attribute value is illustrated by the UNIQUE attribute category and id attribute below:

ENTITY CLASS Project WITH
  UNCHANGING
  name : String;
  ...
  UNIQUE
  id: (name)
END

Based on a set of expenses (SETOF Expense), the following GenMonExpRep transaction produces an expense report (expRep), and deletes the expenses from the Expense class (by CONSUMES).

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

A detailed description of TDL can be found in [Borgida00].

3 A Dependency-Based Mapping Framework

In its most elementary form, the mapping process from requirements specifications to system designs involves a one-to-one mapping from entity and activity classes in the system model into corresponding classes of a design specification. Thus each design class depends on a corresponding requirements class which constrains its attributes, instances, and generalizations as well as specializations. An obvious starting point for the formalization of the mapping process is to represent the types of dependencies that may exist within the system, the kinds of objects they can inter-relate, and the constraints on these objects imposed by different dependency types.

Once such dependency types have been defined, they guide the mapping process by circumscribing the set of allowable design objects that can be associated through a dependency with a given requirements object. In other words, once the designer has selected an object for mapping/remapping, the mapping assistant can offer guidance by highlighting the applicable mapping dependency types. The designer can either select one of the dependency types offered or create a new one. If one of the
dependency types is selected, the system sets up a dependency which is an appropriate instance of the selected dependency type and guides the designer further with respect to the requirements object attributes, its classification hierarchy ancestors and descendants, or both. In addition, the assistant ensures the correctness of the designer’s decisions by checking for possible mistakes.

This approach to the mapping problem relies heavily on the features of Telos, not only for describing the system requirements, but also for providing the uniform representational formalism required to support the mapping process. Thus, Telos is used to represent all artifacts playing a role in the mapping process, including requirements objects, design objects (called thereafter, abstract TDL objects), as well as the dependency objects (both generic/types and specific/instances) that represent design decisions.

Figure 3.1 illustrates the basic framework of the mapping process, represented in terms of Telos classes. Dependencies are modelled as Telos entity classes and metaclasses. For instance, NonMbrExpRptToTDLGenMonExpRep.Dep models the mapping of the Telos activity class &NonMbrExpRpt to the TDL data class TDLGenMonExpRep. Dependency types are modelled by metaclasses which impose constraints on their instances. For example, the metaclass Activity.Transaction.Dep constrains the mapping of (requirements) activity classes to (design) transaction classes.

In the sequel, we fully describe the mapping assistant, called IRIS in its prototype implementation [Katalagarianos90] [Vassiliou90], by presenting in turn the representation of mapping dependencies and dependency types, the representation of design objects, the mapping process as well as the system architecture adopted for IRIS. In addition, some mapping refinements are demonstrated through a sample session with IRIS.
3.1 Representation of Mapping Dependencies and Types

Dependency types are predefined in the mapping tool, while dependencies are application specific instantiations. For presentation purposes, dependencies are grouped into a number of categories:

Classification dependencies: These are represented as metaclasses which impose constraints on the mapping of categories of requirements objects. A simple example of such a classification dependency limits the mapping of activities to transaction or script design objects. For mnemonic reasons, a dependency name is comprised of the types of the source requirements object and the target TDL object (separated by "_"). Each dependency metaclass has at least two necessary attributes, named respectively `telosObject` and `abstractTdlObject`, which indicate the inter-dependent requirements and design objects.

The most general dependency metaclass is `Telos_TDL_Dep`, shown below. The attribute `isaDep` models the mapping of `isA` relationships.

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

The dependency metaclass `Entity_EntityClass_Dep` models the mapping of a requirements entity class `EntityClass` (via attribute `telosObject`), into a design entity class `TDL_EntityClass` (via attribute `abstractTdlObject`):

```plaintext
IndividualClass Entity_EntityClass_Dep
    in M1_Class, Omega_Dep
    isa Entity.DATAClass_Dep with
        necessary
            telosObject: EntityClass;
            abstractTdlObject: TDL_EntityClass;
            attributeDep: DataAttribute_Dep;
        attribute
            isaDep: Entity_EntityISA_Dep
end Entity_EntityClass_Dep
```

The above dependency is an instance of `M1.Class` and `Omega.Dep`, a specialization of `Entity.DATAClass_Dep`. Its attributes maintain information on the requirements and design objects at hand, but also the dependencies among the attributes of these objects (through `attributeDep`) as well as their `isA` relationships (through `isaDep`). The dependency class `ExpenseToTDLExpense_Dep` is an example of a classification dependency which is established from the mapping of the requirements entity class `Expense`:

```plaintext
IndividualClass ExpenseToTDLExpense_Dep
    in S_Class, Entity_EntityClass_Dep
    isa Dep_S_Class with
        telosObject
            : Expense
        abstractTdlObject
            : TDLExpense
        attributeDep
            : ExpenseAmount_Dep;
```

15
Attribute dependencies. These prescribe the allowable mappings of attributes. For instance, the dependency NecessarySingle_Changing_Dep models the mapping of Telos necessary and single attribute to a TDL changing attribute.

IndividualClass NecessarySingle_Changing_Dep
in Mi_Class, Omega_Dep
isa EntitySingle_Dep with

attribute
  TelosObject: Necessary;
abstractTdlObject: Changing
integrityConstraint
  : $ (Forall x/NecessarySingle_Changing_Dep)
  (exists existingDep/Entity_EntityClass_Dep)
  (existingDep.telosObject=from(x.telosObject) and
  (existingDep.attributeDep = x)) $  
end NecessarySingle_Changing_Dep

Similarly, 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.

IndividualClass OutputSingle_Produces_Dep
in Mi_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

IsA dependencies and Instantiation dependencies. In addition to the above types of dependencies, two other types of dependencies were deemed useful. IsA dependencies constraint the mapping of isA hierarchies while Instantiation dependencies deal with the mapping of instantiation hierarchies.

Returning to our example, suppose that the activity class MonMbrExpRpt has been selected for mapping. The dependency types in the system suggest three possibilities: map into a script, a transaction, or a function. This is achieved in the current implementation by highlighting the nodes Activity_Script_Dep, Activity_Transaction_Dep, and Activity_Function_Dep in the dependency hierarchy. Suppose that the designer decides to map MonMbrExpRpt into a TDL transaction class. An instance of the dependency metaclass Activity_Transaction_Dep is then created. The description of this dependency directs further mapping actions. The necessary attribute attributeDep requires the mapping of the attribute(s) of the activity class; Otherwise, an inconsistency will be detected in the dependency knowledge base. The attribute isaDep will model the
mappings of isa relations of &MonMbrExpRpt if it has any. The Activity.Transaction_Dep is defined as a specialization of Activity.Procedure_Dep from which it inherits a number of attributes.

IndividualClass Activity_Procedure_Dep
    in Mi_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 Mi_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

Then, the instance MonMbrExpRptToTDLGenMonExpRep_Dep will be created in the mapping of &MonMbrExpRpt.

IndividualClass MonMbrExpRptToTDLGenMonExpRep_Dep
    in S_Class, Activity_Transaction_Dep
    isa Dep_S_Class with
        telosObject
            : &MonMbrExpRpt
        abstractTdLObject
            : TDLGenMonExpRep
        attributeDep
            : MonMbrExpRpt_exp_Dep;
            : MonMbrExpRpt_m_Dep;
            : MonMbrExpRpt_pers_Dep;
            : MonMbrExpRpt_exrep_Dep
    end MonMbrExpRptToTDLGenMonExpRep_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.

3.2 Representation of Design Objects

The features, syntax, and aspects of the semantics of TDL have been modelled 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 of TDL_Object. Among TDL metaclasses, the most general is TDL_MetaClass with specializations TDL_DataClass, TDL_Procedure, and TDL_Script. The Telos representation of transaction classes is illustrated below in terms of a series of instantiations of the metaclass TDL_TransactionClass:
Figure 3.2 Two-layer representation of the design model in Teles.

Thus, 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 particular application. The organization of abstract TDL objects is depicted in Figure 3.2, where the Telos representation of transaction classes is illustrated as instantiations of the metaclass TDL.TransactionClass.

Attribute categories of the design language have been defined as Telos attribute metaclasses organized along isa hierarchies. TDL.AttributeClass is 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 Mi_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\[12\]. The Telos attribute class Produces shown below is the attribute meta-class modelling the TDL attribute category Produces. The from component of each attribute class specifies the valid objects where that attribute class can 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_DataClass;
    end Produces

Note that multi-valued attributes are represented in TDL in terms of the *setOf* feature, while in Telos attributes are assumed to be multi-valued, but can be constrained to be single-valued by classifying them under *single*.

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

AttributeClass Justification in M1_Class, TDL_Object with
    components
        from: TDL_MetaClass;
        label: justification;
        to: Telos_TDL_Dep
    end Justification

Then, the TDL transaction GenMonExpRep defined in Section 2, is represented in Telos by the class TDLGenExpRep:

IndividualClass TDLGenMonExpRep
    in S_Class, TDL_TransactionClass, TDL_Object with
    tdl_in
        exp: TDLExpenses
        locs, single
        pers: TDL_String;
        m: TDL_Mon
        produces, single
        expr: TDLMonEmplExpRep
    end TDLGenMonExpRep

---

\[12\] Since the *when* component of all definitions below is *AllTime*, it will be omitted from definitions that follow.
3.3 The Mapping Process and System Architecture of IRIS

The mapping process sketched below can be characterized as computer-assisted design. Given a complete definition of a requirements specification, including a system model, the designer generates a design with the help of the mapping assistant. Generation of a design requires the following mapping steps:

1. IRIS displays those components of the system model that have not yet been mapped;
2. The designer selects a system model component to map;
3. IRIS displays candidate dependency types;
4. The designer selects a dependency type;
5. IRIS displays a template of a design entity for the dependency type selected;
6. The designer completes the definition of the design entity.

Architecturally, IRIS is depicted in Figure 3.3. The knowledge relevant to the mapping task is shown on the right of the Figure, while the user interacts with IRIS in an abstract data type fashion. A graphical representation of objects is employed at the user interface.

3.4 Mapping Refinements

Software development rarely proceeds in a strictly top-down fashion. Instead, requirements are elaborated while the mapping process is in progress. These elaborations lead to new mapping tasks.
for newly created or even re-defined requirements objects. An enhanced process that allows the interleaving of elaborations and mappings would proceed until the designer is satisfied:

1. The designer chooses one of the three options — creating (a new), refining (an existing), and mapping a system model object;

2. If mapping is chosen, IRIS and the designer repetitively go through the mapping steps described above until either all the descriptions in a system model are mapped into a design or the designer determines mapping is temporarily complete.

The above description presents a more realistic mapping process. However, this process is still unsatisfactory in that it doesn’t offer any methodological guidance for the generation of a requirements specification or its mapping into a design.

To introduce such methodological guidance, we adopt the framework proposed in [Goldman82]. Assuming that specifications describe allowable states and state transitions, Goldman identifies three basic types of refinements:

- **Structural refinements** that change the information structure associated with a state description,

- **Temporal refinements** that change the number of states by decomposing a state transition into sub-transitions, thus introducing intermediate states, and

- **Coverage refinements** that either expand or restrict the set of allowable states.

These refinement types have been further revised to take full advantage of the rich representational features offered by Teles and TDL:

**Structural refinements** include three types of refinement:

- **isa refinements** change the lattice structure of isa hierarchies by inserting a new class as a leaf node of the isa hierarchy or as a generalization of a collection of existing classes.

- **attribute refinements** introduce changes to attribute classes, i.e., values of attribute definitions, by generalizing or specializing the type of a given attribute, or by inserting or removing attributes associated with a given class.

- **instanceOf refinements** effect changes along the instanceOf dimension by introducing new class definitions at the metaclass level or higher. Note that TDL does not support metaclasses and that any metaclasses used in the system model will have to be “compiled” into structural and assertional information associated with its instances before a corresponding design is generated [Mertikas89].

**Temporal refinements** change the number of states by decomposing a state transition, activity in the case of our requirements specifications, into sub-transitions. For instance, the activity, &ReceiveExpense, may be split into four sub-activities: Receive that accepts attribute values for expense summaries, ErrChk that checks the validity of each of the accepted attribute values, Compare that compares two expense summaries, one based on ExpViaEmail and the other based on ExpInCmo, and Transfer that transfers ExpenseViaEmail to Expense. Note that if the designer sees no need for further temporal refinement for Receive, ErrChk, Compare and Transfer, dependency-based mapping will be called for and &ReceiveExpense may be mapped down into a script that coordinates

---

13While [Goldman82] uses the term refinement for adding the amount of detail and abstraction or contraction for reducing the amount of detail, we use the term for adding or reducing the amount of detail as well as for changing the detail.
the four refined activities.

Coverage refinements either expand or restrict the set of possible states and state transitions. For instance, consideration of failure modes for a certain activity may lead to the introduction of other activities:

Members submit their expense summaries:

(i) Members submit consistent summaries or
(ii) Members submit inconsistent summaries or
(iii) Members do not submit summaries.

The detection of and recovery from violations of the above activity may result in the creation of a new class InvalidExpense and an activity, InformViolation, that informs appropriate persons in case of a violation.

As might be expected, one type of refinement usually triggers other refinements of the same or other types. A complete implementation of the mapping assistant proposed in this paper will include definitions of refinement types which aid the generation of a requirements specification. These refinement types can be seen as complementary to the dependency types discussed earlier.

Many mapping issues require hard decisions by designers. While our approach offers certain guidance for resolving the issues, due to space limitations, we do not describe in this paper the nature of the issues and mechanisms needed to resolve them (See [Chung89], [Katalagarianos80], [Vassiliou90] for detail).

3.5 A Sample Session with IRIS and Implementation Status

Early on in the implementation of IRIS, and consistent with other components of the DAIDA project, it was decided to use graphical representations to give the designer 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 into a design. As demonstrated in Figure 3.4, the basic IRIS interface consists of a Telos area, a TDL area, and a dependency types area.

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:

```plaintext
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;
...
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. The first involves adopting the isA hierarchy provided by TDL. In this case we simply map each Telos Activity to a TDL Transaction as follows:
An alternative way is to eliminate this hierarchy from the design specification and to create instead the following transaction:

```
TRANSACTION GenMonExpRep WITH
IN
  pr: Proj;
  exp: SETUP Expense;
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 [Katalagari and others, 2000]. For the example, `&MonExpRpt` 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 object 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 one of Simple classes (`S.classes`), M1.classes, or Meta_Meta.Classes which include M2.classes up to Omega.classes.

   Next, a specific object is selected for mapping. To complete the mapping, the designer needs to first review its complete description. The result of selecting a particular class is shown in Figure 3.4. As `&MonExpRpt` 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.

   **Action**: Select mapping with inheritance

   **Effect**: Attribute displayer comes up (Figure 3.5). 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 true, IRIS 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 already mapped.

2. **Start the mapping process for the object**

   The next step is to decide how that object is going to be mapped. As shown in Figure 3.5, by selecting the class node `&MonExpRpt`, the designer is offered a menu item of map.
Figure 3.4 Telos Object Selection.
Figure 3.5 Attribute Display.
Action: Select map

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

3. Selecting mapping rules, creating TDL object, and mapping attributes

Proceeding in a similar fashion, the designer arrives at the situation shown in Figure 3.6. Next, the designer needs to decide which rule to choose among the highlighted nodes. Then, the system highlights the applicable ones for this mapping task. When an appropriate node is selected, the system suggests that the Telos MonNbrExpRpt can be mapped to a TDL transaction. At this point, the designer can either overwrite the suggestion by providing her own design or go ahead with the suggestion. Assuming that the designer agrees, a TDL editor comes up with the frame of the TDL object to be created. After the designer fills up the name slot with, say MonNbrExpRpt, the dependency corresponding to the mapping performed appears on the attribute displayer.

The next step is to proceed with the mapping of the attributes including the inherited ones. For each attribute link selected by the designer, the system suggests a set of mapping alternatives in the mapping area. When the designer selects one of the mapping alternatives, the mapping assistant provides the TDL attribute type on the TDL editor. After the designer fills up the slot for the TDL attribute name, the mapping assistant shows the dependency corresponding to the mapping of attribute in the attribute displayer.

4. Marking the end of a particular mapping

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

Action: Select the button TELOS Tell within the TDL editor.

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

Action: Select Syntax Check and then Consistency within the TDL editor.

Effect: TDL object is Told to the TDL KB. Class MonNbrExpRpt is marked as mapped on the attribute displayer. Figure 3.6 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 other requirements 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, the system recommends 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.

The implemented prototype of IRIS runs on SUN workstations and is implemented on top of BIM-Prolog, while the windows and graphics interface have been implemented in SunView and Pixrect. The graphical, window-based interfaces used by this mapping assistant were developed by GFI.

At the time this paper was written, the implementation included approximately 30 dependency types. It is estimated that a reasonably complete implementation will require 50 – 100 types, and comparable numbers of decomposition and refinement methods.

---

14 A longer example of an IRIS session can be found in [Vassiliou90].
Figure 3.6 Dependency Graph from the Mapping of &CanExpense

Transaction MonAmExpRpt with
In
exp: SETOF EXPENSE;
pr: PROJ;
Locals
per: STRING;
m: MOD;
Produces
expRep: DntExpRpt;
END

INPUT file read..............

TRYING to parse input file ...
INPUT file parsed successfully.

TRYING to create the internal representation,
Working with definition number 1.
4 Dealing with Non-Functional Requirements

The complexity of an information system is determined partly by its functionality and partly by global requirements on its cost, performance, reliability, maintainability, portability, and the like. These non-functional requirements are invariably stated informally during requirements analysis, are often contradictory\textsuperscript{15}, difficult to enforce during software development and impossible to validate for the user once the final system has been built. This section proposes a framework for the representation of non-functional requirements as well as their use to rationalize the software development process.

In essence, non-functional requirements are treated as (possibly contradictory) goals that guide the refinements in the mapping process and justify particular design decisions. To link design decisions to global goals — say, system security — we need methods for goal decomposition and others for “achieving” goals through particular design decisions. Actually, a particular design decision rarely “achieves” a non-functional requirement. More often than not, design decisions contribute positively or negatively towards a particular non-functional requirement and for the rest of the discussion we will speak of goal satisfying \textsuperscript{[Simon31]}\textsuperscript{16} to suggest that generated software is expected to fulfill within acceptable limits, rather than absolutely, non-functional requirements. The job of the mapping assistant is then (i) to record inter-dependencies, positive or negative, between goals and design decisions, (ii) to store and make available to the designer decomposition methods for different types of goals, (iii) to maintain the goal tree which has the initial non-functional requirements as root node and records the decomposition of each goal into subgoals, and (iv) to maintain a record of how goals/subgoals are affected by individual design decisions.

The framework described in this section assumes that the methods for goal decomposition and satisfying are specific to the type of non-functional requirement they are intended to treat. To demonstrate the nature and the power of the framework, accuracy requirements are treated in detail throughout. An analogous treatment has been done for security requirements.

4.1 A Model of Accuracy

Accuracy as a requirement on an information system reflects the faithfulness of the information maintained by the system with respect to the application domain. A fundamental premise in constructing a model for this type of requirement is that accuracy of a piece of information totally depends on the way that information is manipulated within the system as well as its environment.

Consider, once again, the project expense management example. Figure 4.1 shows a plausible information flow network that models the manner in which agents in the application domain pass around information items. After issuing a ticket, an airline company sends the ticket to a project member who gives a copy of the ticket, after using it, to the secretary. The secretary issues a reimbursement request form with the ticket stub as a supporting document to the central management office by mail and at the same time inserts the expense summary into the information system via electronic mail. In turn, the central management office enters the reimbursement request form into the information system and a reimbursement is issued after a suitably irritating delay. At the end of each month, the information system produces a monthly expense report and sends it to the project leader.

This simple, but common, scenario includes a number of agents (persons or systems) who participate by maintaining, transmitting, and manipulating information. The information may involve

\textsuperscript{15}The rule of thumb seems to be that initially the user expects the intended software system to do everything perfectly and cost nothing; it is the job of the systems analyst to not only formulate realistic functional and non-functional requirements but also educate the user.

\textsuperscript{16}Simon introduced the term to refer to decision methods that look for good or satisfactory solutions instead of optimal ones. The term is used here in a broader sense since in the context of non-functional requirements, it is unclear what “solutions” mean or under what conditions solutions are optimal.
Figure 4.1 An example of information flow.
an entity’s properties ("Brian is a research scientist", "Matthias is an engineer") or attributes ("Bill is John’s manager", "Maria earns $50 000"). In addition to agents, the scenario includes channels used to transmit information. Individual links in the information flow network shown in Figure 4.1 represent unit flows.

An information flow network may involve sequential or parallel paths. Intuitively, we expect that accuracy is adversely affected by long sequential paths, as intermediaries may introduce inaccuracy, while parallel paths enhance accuracy since the same message is sent to its destination through more than one route. One of the aims in developing a model for accuracy was to formally account for these intuitions.

What does it mean for information to be accurate? Firstly, we assume that accuracy is a fundamental (semantic) attribute of information chunks (hereafter, information items), in the same sense that weight, density or volume are fundamental (physical) attributes of material chunks. If \( x \) is an information item, \( A[x] \) specifies the degree of confidence in \( x \), i.e., confidence in the proposition that \( x \) faithfully describes the application domain. Even though this might be possible for narrow domains, \( A \) will not be treated here as a probabilistic, fuzzy, or even quantitative measure. Instead, we attempt to develop a qualitative model which leads to both decomposition rules for accuracy goals and design methods that enhance accuracy goals.

In the context of DAIDA, information items maintained by the system to be built have a definite structure which can be used to clarify the nature of the accuracy measure:

- \( A[\text{InstanceOf}(e, c, t)] \), measures the confidence that the (application domain) entity represented by token \( e \) has the property represented by class \( c \) during time interval (represented by) \( t \);
- \( A[\text{attr}(e, v, t)] \) measures the confidence that the entity represented by \( e \) has attribute \( \text{attr} \) with value (the entity represented by) \( v \) during time interval \( t \);
- \( A[e] \) measures the confidence in the assertion that there exists one and only one entity represented by \( e \) and there is no other \( e’ \) which represents the same entity.

These definitions allow the formalization of accuracy for particular individual entities, their attributes and properties.

A then is a function,

\[
A : I \rightarrow D, \quad \text{where}
\]

\[
I = E \cup A \cup In,
\]

\( E \) is the set of token entities represented in the information system,

\( A \) is the set of token attributes,

\( In \) is the set of instance relationships between tokens and data classes,

and, finally,

\[
D \quad \text{is an partially ordered set with minimum element } \iota \text{ and maximum element } \alpha, \text{ representing respectively total lack of and complete confidence in the accuracy of an information item.}
\]

The symbol " \( > \) " represents a partial order relation among the elements.

A single predicate is next introduced to simplify the representation of accuracy requirements. Let \( X \subseteq I \):

\[
\text{Accurate}_{d}[X] \equiv \bigwedge_{x \in X} A[x] \geq d
\]
for some threshold value $d$, which will generally be ignored in the rest of the discussion. Accurate is the basic predicate used to express accuracy requirements, such as:

\[
\text{Accurate}\{x, \text{attr}(x, v, t), \text{instanceOf}(x, C, t') \mid x \text{ such that instanceOf}(x, \text{Accounts}, q), \text{for some } q, \text{ and balance}(x, y, q') \text{ and } y > \$1\,000\,000}\}
\]

Note that here we assume that "client accounts being accurate" refers to all attributes and properties of those accounts.

To simplify the notation, we introduce set-theoretic functions, extension, attributes, and properties:

\[
\text{extension}(C, t) = \{x \mid \text{instanceOf}(x, C, t') \text{ and } t' \text{ during } t\}
\]

\[
\text{attributes}(e, t) = \{\text{attr}(e, v, t') \mid t' \text{ during } t\}
\]

\[
\text{properties}(e, t) = \{\text{instanceOf}(e, C, t') \mid t' \text{ during } t\}
\]

Another function can be obtained by using a set valued function, $\wedge$. For instance, $C^\wedge \text{attr}$ yields the class which is the value of the definition for the attribute attr of the class C. Then

\[
\text{extension}(C^\wedge \text{attr}, t) = \{\text{attr}(e, v, t') \mid \text{instanceOf}(e, C, t') \text{ and } t' \text{ during } t\}
\]

which enables us to introduce other useful notations such as:

\[
\text{extension}(C^\wedge \text{attributes}, t) = \{\text{attributes}(e, t) \mid \text{instanceOf}(e, C, t') \text{ and } t' \text{ during } t\}
\]

With these functions, the client accounts example can be expressed as

\[
\text{Accurate}\{x, \text{properties}(e, t), \text{attributes}(e, t) \mid \begin{align*}
& x \in \text{extension}(\text{Accounts}, \text{AllTime}) \text{ and } \\
& \text{balance}(x, y, t') \text{ for some } t' \text{ and } y > \$1\,000\,000 \text{ and } \\
& t' \text{ during } t\}
\end{align*}
\]

We can apply the Accurate predicate to more complex concepts, as will be seen later, such as

\[
\text{Accurate}[C] \equiv \text{Accurate}[\text{extension}(C, \text{AllTime})]
\]

\[
\text{Accurate}[C^\wedge \text{attr}] \equiv \text{Accurate}[\text{extension}(C^\wedge \text{attr}, \text{AllTime})]
\]

\[
\text{Accurate}[C^\wedge \text{attributes}] \equiv \text{Accurate}[\text{extension}(C^\wedge \text{attributes}, \text{AllTime})]
\]

More sophisticated notions of accuracy can be introduced by making additional assumptions about the accuracy measure, e.g., that $A$ is a fuzzy function, or by introducing several accuracy predicates, such as AbsolutelyAccurate, HighlyAccurate, MarginallyAccurate, etc., which use different threshold intervals within $D$.

Note that accuracy is treated extensionally rather than intensionally here. In other words, an accuracy requirement for, say, most-highly-rated client accounts is interpreted as a requirement on the particular accounts maintained by the information system, not the description of the concept of mostHighlyRatedClientAccounts included in the system specification\footnote{An intensional treatment of accuracy would attempt to measure the accuracy of information items such as generic descriptions; the description of the HaveMeeting activity, for example, may be accurate or inaccurate depending on its declared (generic) sub-activities, constraints, parameters, etc.. This notion of accuracy seems appropriate when one attempts to measure the faithfulness of the world model — that is part of a functional requirement — to the application domain.}.
Treating an accuracy requirement as a goal to be satisfied leads to another class of systematic refinements whose structure forms a *goal tree*. Each node in the tree represents a goal, with the root node representing the initial accuracy goal. Moreover, each node/goal has an associated set of successor nodes/subgoals which either conjunctively or disjunctively satisfy the goal associated with the parent node. Thus goal trees are *AND (OR)* trees [Nilsson71] with goal satisfaction replaced by the more flexible notion of goal satisficing. Within such a framework, *goal refinement* corresponds to the conjunctive or disjunctive decomposition of a goal into a set of successor goals. Goal refinements may involve the instantiation of a goal decomposition method, leading to a conjunctive decomposition. Alternatively, several competing methods may be applicable for a goal, leading to its decomposition or satisficing. Such situations lead to disjunctive goals. Since the selection of a refinement may involve decision-making on the part of the designer, each method may be associated with a set of *selection criteria* that partly formalize and facilitate the selection process.

Below we list some obvious methods for goal decomposition:

- **subclass method**: In order to prove \( \text{Accurate}[C \wedge \text{attributes}] \) (\( \text{Accurate}[C \wedge \text{attr}z] \)), where \( C \) is a class with specializations \( C_1, C_2, ..., C_n \) such that \( \text{extension}[C, t] = \bigcup_{i=1}^{n} \text{extension}[C_i, t] \), prove \( \bigwedge_{i=1}^{n} \text{Accurate}[C_i \wedge \text{attributes}] \) (\( \text{Accurate}[C \wedge \text{attr}z] \)).

- **derivedInfo method**: In order to prove \( \text{Accurate}[x] \), where \( x \) is derived from \( y_1, y_2, ..., y_n \) through a function, \( \text{derivationFunction} \), prove \( \bigwedge_{i=1}^{n} \text{Accurate}[y_i] \).

- **subset method**: In order to prove \( \text{Accurate}[S] \), where \( S \) is a set with subsets \( S_1, S_2, ..., S_n \) such that \( S = \bigcup_{i=1}^{n} S_i \), prove \( \bigwedge_{i=1}^{n} \text{Accurate}[S_i] \).

- **superset method**: In order to prove \( \text{Accurate}[S'] \), where \( S' \subset S \), prove a stronger proposition, \( \text{Accurate}[S] \).

Some methods for goal satisficing include:

- **consistency checking**: the system enforces an appropriate set of integrity constraints; these may include measures against frequently-occurring errors and may be added to the design even if they are not in the requirements specification;

- **confirmation**: sender confirms information he/she sent, thereby neutralizing leaky (or unreliable) channels;

- **validation**: double-checking, by someone other than sender — redundancy is the key here;

- **certification**: comprehensive validation to meet predetermined standards;

- **verification**: checking with application domain to confirm that information item is accurate — making \( A[x] \) have as value \( c \) or \( \ell \);

- **auditing**: an accuracy auditor goes through suspicious information items;

- **better routing**: establishing channels that are less leaky.

In general, an information system development task will have a number of associated non-functional requirements of various types, not just accuracy ones. Some of the goal decomposition or satisficing methods for these other requirements may adversely affect accuracy goals. Here are some examples of methods with such adverse effects:

- **frequent modification**: frequent updates, insertions and deletions;

- **uncontrolled modification**: direct modification by casual users;
• some security goals:
  
  – *sole authorizer*: access to be approved by a single, unique authorizer (the unavailability
    of the authorizer leads to update failure);
  
  – *sole accessor*: access by a single, unique agent (failure to perform a needed operation
    induces inaccuracy);

• some security satisfying methods: for instance, a *lengthy identification* method requiring a
  lengthy procedure leads to temporal delays in a needed update.

4.2 Goal-Directed Mapping

As suggested in Section 2, a functional requirements specification treats the information system under
development as one of the agents present in the intended system environment. One of the functions
of the environment model within the functional requirements specification is the description of the
structure and the behavior of information flow in the application domain.

Consider again the report generation example of Section 2 and assume that
*Accurate[&MonExpRpt^attributes]* is the root node of the goal tree representing non-functional re-
quirements (in what follows, *attributes* is abbreviated by *att*). Further assume that *&GenMonRpt*,
declared in Section 2, has been specialized by *&GenMonProjRpt, &GenMonMtgRpt* and *&GenMonMbrRpt,*
*&GenMonMbrRpt*). The root goal can then be refined using the *subclass* method mentioned earlier:

\[
\text{Accurate}[\&\text{MonExpRpt}^\text{att}] \rightarrow \text{Accurate}[\&\text{MonProjExpRpt}^\text{att}] \land \\
\text{Accurate}[\&\text{MonMtgExpRpt}^\text{att}] \land \\
\text{Accurate}[\&\text{MonMbrExpRpt}^\text{att}]
\]

The result of this decomposition is that *Accurate[&MonExpRpt^att]* has been decomposed into
three subgoals of *Accurate[&MonProjExpRpt^att]*, *Accurate[&MonMtgExpRpt^att]* and
*Accurate[&MonMbrExpRpt^att]*. Now each of these subgoals needs to be satisfied. For instance, satis-
fying the subgoal of *Accurate[&MonProjExpRpt^att]* requires that all attributes of *MonProjExpRpt* be
accurate. At this point, the designer indicates that each *MonProjExpRpt* in *MonProjExpRpt* (from
here on, assume that the argument *Accurate* is universally quantified) has five attributes: month
(*mon*), project (*proj*), total monthly expense (*exp*), project budget (*budg*) and remaining project
budget (*budgetLeft*). Then, the goal of *Accurate[&MonProjExpRpt^att]* is decomposed into:

\[
\text{Accurate}[\&\text{MonProjExpRpt}^\text{att}] \rightarrow \text{Accurate}[\&\text{MonProjExpRpt}.\text{mon}] \land \\
\text{Accurate}[\&\text{MonProjExpRpt}.\text{proj}] \land \\
\text{Accurate}[\&\text{MonProjExpRpt}.\text{exp}] \land \\
\text{Accurate}[\&\text{MonProjExpRpt}.\text{budg}] \land \\
\text{Accurate}[\&\text{MonProjExpRpt}.\text{budgetLeft}]
\]

Note that the designer, at this point, may choose either to map *MonProjExpRpt* into TDL classes
through a dependency-based mapping or to continue to make further refinements in order to satisfice
the initially posted goal.

While trying to satisfy the subgoal of *Accurate[&MonProjExpRpt^att]*, the designer may choose
to indicate that *MonProjExpRpt^exp* is *derived* information, where the formula for the derivation
function is

\[
\text{monProjExpRpt.exp} \leftarrow \text{SUM}(\{\text{ex in Expense} \mid \\
\text{ex.mtg.when = monProjExpRpt.mon and} \\
\text{ex.proj = monProjExpRpt.proj}\}).\text{ECUamount})
\]
It is then possible to instantiate the \textit{derivedInfo} decomposition method:

\[
\text{Accurate[monProjExpRpt.exp]} \quad \text{Accurate[exp.mtg]} \land \\
\quad \text{Accurate[mtg.when]} \land \\
\quad \vdots \\
\quad \text{Accurate[Expense]} \land \\
\quad \text{Accurate[derivationFunction]}
\]

The designer may now choose either to map \textit{derivationFunction} into a function, transaction or script, according to the dependency types available in the system, or to continue to make further goal refinements dealing with activation conditions, termination conditions, etc. Figure 4.2 shows the results of a dependency-based mapping for \textit{derivationFunction}. Firstly, \textit{derivationFunction} is mapped into a TDL function which is called by the newly introduced transaction \texttt{addMonProjExp} that computes \texttt{monProjExpRpt.exp} (as well as the \texttt{nonProjExpRpt.budgetLeft}) for a \texttt{monProjExpRpt} and inserts it (with appropriate \texttt{mcn}, \texttt{proj}, and \texttt{budg} values) into \texttt{nonProjExpRpt}. Note that, although not shown here, \texttt{addMonProjExp} may have pre- and/or postconditions thereby improving the accuracy of the derived information. Secondly, since monthly project expense summaries need to be produced at the end of each month, a script needs to be introduced which includes a single transition. The transition is activated at the end of each month.

Every information item in the information system is either received from an external agent or derived from what already resides within the system\textsuperscript{18}. Let’s continue considering the subgoals of \texttt{Accurate[monProjExpRpt.exp]}. Our mapping assistant and the designer need to go through a refinement process for \texttt{exp.ECUamount} similar to that of \texttt{nonProjExpRpt.exp}: the designer provides a derivation function with all the sources, the mapping assistant decomposes the goal of \texttt{Accurate[exp.ECUamount]} into subgoals, etc. The designer indicates that \texttt{exp.ECUamount} is derived, while other attribute values, including \texttt{exp.mtg}, \texttt{exp.when}, and each instance of \texttt{Expense}, are received from members (or their secretaries).

Now focus on the refinement of \texttt{Accurate[extension(Expense)]}. When asked to identify all transmission routes for expenses, the designer indicates that expenses can be based either on summaries received via electronic mail (\texttt{expViaEmail}, directly sent by a secretary) or on data entered at the central management office (\texttt{expInCmo}, based upon the reimbursement request form sent by a secretary). Then an expense will be accurate if either \texttt{expViaEmail} or \texttt{expInCmo} is accurate. We see here two competing methods. Using as selection criterion the length of the flow path we have,

\[
\text{Accurate[expense]} \quad \leftarrow \quad \text{Accurate[expViaEmail]}
\]

As will be explained later in this section, we note that if a \textit{verification} method (comparing certain information received from more than one agent) is needed, both \texttt{expViaEmail} and \texttt{expInCmo} must be accurate.

A received information item is assumed to be accurate if its creation and transmission are accurate. Unfortunately, ensuring that the creation and transmission of an information item is accurate is in many cases costly and impractical. Accordingly, the designer may choose to resign himself to using several satisficing methods for \texttt{Accurate[expViaEmail]}. Clearly, using more methods leads to increased confidence that the information maintained by the system is indeed accurate.

\[
\text{Accurate[expViaEmail]} \quad \leftarrow \quad \text{validation(expViaEmail)} \lor \\
\quad \text{verification(expViaEmail)} \lor \\
\quad \text{certification(expViaEmail)}
\]

\textsuperscript{18}The \texttt{system clock} may be viewed as an external agent.
Figure 4.2 The goal tree generated for the example.
Here, we need a **coverage refinement** for expenses, i.e., a refinement which considers the case when something goes wrong with expenses. Inaccuracy of received information about expenses will be detected by **ErrChk, Compare**, or by any other goal satisfying method. As discussed in Section 3, a coverage refinement usually results in structural refinements. In order to allow temporary storage of inaccurate expenses, we need to create a class for instances of either expViaEmail or InvalidExpViaEmail (entity classification) and treat it as a specialization of Expense class (leaf specialization). Here, we see a simple example of filling in details about requirements violations as well as recovery from these violations.

Figure 4.2 also illustrates how a potential **temporal refinement** might be achieved for an activity. If **consistency checking** method and **verification** method are selected, received expViaEmail information (by Receive activity) goes through error checking (ErrChk) against various constraints (utilizing the dependency-based mapping, the designer may choose to map the constraints into preconditions of ErrChk, if ErrChk is mapped into a transaction), compared with expInCmc (Compare), and then transferred to Expense class (Transfer). Thus, &ReceiveExpense is decomposed into Receive, ErrChk, Compare, and Transfer. As with &GenNonProjRpt, this temporal refinement shows an example of filling in details about cause-effect chains.

The success (or lack thereof) of goal satisfying methods relies on the cooperation between the system and agents in the environment. Overall responsibility for accuracy is vested in the **accuracy auditor**, who contributes to the **procedure manual**, which is initially drafted during the design process. The manual indicates policies that the agents in the environment should obey in order to satisfy the subgoals selected. It affects the organizational structure and the interaction between the system and agents during usage of the system. For instance, if a **verification** method is selected, the procedure manual indicates that a member must transfer his expense information to the information system (expViaEmail) and to the central management office which will enter the same information into the information system (expInCmc). The effect of this procedure is to open communication channels between the member, the system, and the office. Especially when few methods are selected, the manual indicates the duties of agents in satisfying the non-selected ones.

## 5 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 mapping framework encompasses a number of novel features. Non-functional requirements are represented as goals in terms of which design decisions are justified. A number of different classes of knowledge are identified, including:

- **dependency types**, which constrain the mapping of requirements objects into design ones,
- **decomposition methods**, which provide rules for decomposing goals representing non-functional requirements into simpler goals,
- **satisficing methods**, which relate, positively or negatively, design decisions with goals of different types,

- **refinement methods**, which help the developer of a requirements specification elaborate by adding detail.

In addition, a control structure is suggested which deploys these knowledge sources to guide and rationalize the software development process while leaving overall control to the human software developer. Lastly a prototype implementation of the mapping assistant, IRIS, is described, giving a "feel" for the type of services offered.

The framework described in this paper requires further refinement in several directions. Firstly, the model of accuracy used to represent and reason about accuracy requirements needs further elaboration and exploration to discover its limitations. Analogous research is required for other classes of non-functional requirements. For each such class, decomposition and satisficing methods need to be discovered and added to the system's knowledge base.

A second major area of future work involves the refinement of the control structure built into the mapping assistant. This control structure can be basically software development process-oriented, where the system model is mapped down to a system design in a (hopefully) disciplined fashion and different decisions are associated positively or negatively with goals they affect. Alternatively, it can be goal-driven, where the generation of the goal tree guides the generation of the design. Other control regimes between those extremes are also possible. We propose to resolve this issue empirically by experimenting with different alternatives once a more complete implementation is available.

**Acknowledgements**

Many thanks are due to several persons at the University of Crete, where much of the research reported here was carried out, at the University of Toronto, where research on the design of information systems has been going on for almost fifteen years, also colleagues participating in the DAIDA project and representing BIM (Belgium), GFI (France), SCS (Germany), the University of Frankfurt (Germany), and the University of Passau (Germany). They contributed helpful advice, insightful comments, encouragement and plain good humor throughout the project.

Special thanks are due to Matthias Jarke (University of Passau) not only for his suggestions concerning this work, but also for his inspired technical leadership of DAIDA. We are equally grateful to Alex Borgida (Rutgers University) for encouragement and advice throughout. In addition, we acknowledge the fine implementation work by several graduate and undergraduate students at the University of Crete including Thodoros Topaloglou and Manolis Koubarakis (Telos), Lydia Kavraki and Maria Mamalaki (TDL), and Kostas Haritonidis (Graphical User Interface of IRIS). Sol Greenspan maintained an ongoing interest and provided continuous moral support and insights. Our gratitude is also due to Brian Nixon, Eric Yu, and members of the Taxis group at the University of Toronto for their input and interest in this work.

Finally, we wish to thank the anonymous referees for suffering through an earlier longer version of the paper and providing us with many helpful comments.
Bibliography


