University of Alberta

Library Release Form

Name of Author: Iqbal A. Goralwalla

Title of Thesis: Temporality in Object Database Management Systems

Degree: Doctor of Philosophy

Year this Degree Granted: 1998

. . . .

Iqbal A. Goralwalla
Department of Computing Science
University of Alberta
Edmonton, Alberta T6G 2H1
Canada

Date: . . . .

University of Alberta

Temporality in Object Database Management Systems

by

Iqbal A. Goralwalla

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Spring 1998

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Temporality in Object Database Management Systems** submitted by Iqbal A. Goralwalla in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

. . . .

Dr. M. Tamer Özsu

. . . .

Dr. Duane Szafron

. . . .

Dr. Ramez Elmasri

. . . .

Dr. David Wishart

. . . .

Dr. Peter van Beek

. . . .

Dr. Li-Yan Yuan

Date: . . . .

Abstract

Conventional databases represent the state of an enterprise at one particular point in time. That is, they contain only current data. As a database changes, out-of-date information, representing past states of the enterprise, is discarded. However, temporal support is a requirement posed by many database applications, such as office information systems, engineering databases, and multimedia systems. Most of the research on modeling time has concentrated on the definition of a particular temporal model and its incorporation into a (relational or object-oriented) database management system. This research has led to the definition and design of a multitude of temporal models. Many of these assume a specific set of temporal features, and therefore do not incorporate sufficient functionality or extensibility to meet the varying temporal requirements of today's applications. Instead, similar functionality is re-engineered every time a temporal model is created for a new application. This suggests a need for combining the diverse features of time under a single infrastructure that allows design reuse. In this thesis, an object-oriented framework that provides such a unified infrastructure is presented. An object-oriented approach allows one to capture the complex semantics of time by representing it as a basic entity. Furthermore, the typing and inheritance mechanisms of object-oriented systems directly enable the various notions of time to be reflected in a single framework. The framework can then be tailored to accommodate the temporal needs of different applications, and existing temporal models can be derived by making a series of design decisions through subclass specialization. The framework can also be used to derive a series of more general temporal models that meet the needs of a growing number of emerging applications. Furthermore, the framework can be used to compare and analyze different temporal object models with respect to the design dimensions. This helps identify the strengths and weaknesses of the different models according to the temporal features they support.

The framework is then used to instantiate a single temporal object model which has multiple facets of time. There have been many temporal object model proposals (for example, [RS91, SC91, WD92, KS92, CITB92, BFG97]). These models differ in the functionality that they offer, however as in relational systems, they assume a set of fixed notions of time. The temporal object model developed in this thesis consists of an extensible set of primitive time types with a rich set of behaviors to consistently and uniformly model the diverse features of time. The model is one possible implemention of the temporal framework and provides concrete and consistent semantics for the different temporal features which is necessary for their coexistence.

The establishment of a temporal object model provides a foundation from which investigations are carried out on using temporality to model other database features such as schema evolution. The issues of schema evolution and temporal object models have traditionally been considered to be orthogonal and handled independently. This is unrealistic because to properly model applications that need incremental design and experimentation (such as CAD, software design process), the evolutionary histories of the schema objects should be traceable. In this thesis, a method for managing schema changes by exploiting the functionality of the temporal object model is proposed. The result is a uniform treatment of schema evolution and temporal support for many object database management systems applications that require both.

Acknowledgements

All praises are due to God who guides and protects me. Indeed, His Bounties on me cannot be enumerated. This thesis is for His sake.

I would like to sincerely thank my supervisor, Dr. M. Tamer Özsu, for his advise, guidance, and support during my research. His patient encouragement and enthusiasm in publishing the results of my research is very much appreciated.

Many thanks go to Dr. Duane Szafron, my co-supervisor, for all his help and support during my research. His detailed comments and advise on every aspect of my research helped me to think rigorously.

I would also like to thank the members of my committee, Dr. Peter van Beek, Dr. Li-Yan Yuan, Dr. Ramez Elmasri, and Dr. David Wishart for their willingness to be on my thesis committee and for their invaluable comments and suggestions.

It was a rewarding experience working within the database research group at the University of Alberta. I am especially grateful to Yuri Leontiev for all those hours we spent discussing various aspects of the temporal model, Paul Iglinski for listening and helping me with ObjectStore and C++ related questions, Kaladhar Voruganti and Vincent Oria for all the interesting discussions we had, Wade Holst for his help in developing the temporal framework toolkit, and last but not least Anne Nield for always saying “yes” to my never ending requests.

Finally, I express my sincere gratitude to my wife and parents for all their love, support and encouragement. Their patience while I completed my education helped carry this dream to reality.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

A database contains data pertaining to an organization and its activities. It forms a data repository from which information is extracted for various purposes. Databases in general carry the most recent data. As changes occur, out-of-date data, representing past states of the enterprise, are overwritten and are no longer available. Conventional databases can be viewed as *snapshot* databases in that they represent the state of an enterprise at one particular time. However, time is an attribute of most real-world phenomena. Events occur at specific points in time; objects and the relationships among objects exist and change over time. The ability to model the temporal dimension of the real world is essential for many applications such as econometrics, banking, inventory control, medical records, real-time systems, multimedia, airline reservations, versions in CAD/CAM applications, statistical and scientific data, etc. To support the temporal information needs of these applications, the database should possess a temporal dimension to store and manipulate time varying data.

In the last decade there has been extensive research activity on temporal databases. An initial summary of research projects can be found in [Sno86] which also includes a bibliography on temporal databases. Additional bibliographies on temporal databases are given in [SS88, Soo91]. These bibliographies were updated by Kline [Kli93]. A further update to Kline's bibliography appeared in [TK96]. Certain research directions in temporal databases are highlighted in [Sno90]. Further areas of temporal database research are detailed in [TCG+93] which gives comprehensive treatment of the state-of-the-art in temporal databases as of 1993. Reports on two international workshops on temporal databases can be found in [Pea94] and [SJS95]. Research on temporal object-oriented databases is surveyed and critically compared in [Sno95a]. A further survey on temporal and real-time databases is given in [ÖS95]. The state of art of temporal database system implementations is summarized in [Böh95].

The early research on temporal databases concentrated mainly on extending the relational model [Cod70] to handle time in an appropriate manner. These extensions can be grouped into two main categories. The first approach uses First Normal Form (1NF) relations in which special time attributes are added to a relation and the history of each attribute is modeled by several 1NF tuples [Ari86, LJ88, NA89, Sar90, Sno87]. This approach is known as *tuple timestamping*. The second approach uses Non-First Normal Form (N1NF) relations in which time is attached to attribute values of a relation and the history of an attribute is modeled by a single N1NF tuple [CC87, Gad88, Tan86]. This approach is known as *attribute timestamping*. TQUEL [Sno87] is a prototype implementation on the INGRES database management system (DBMS) which demonstrates the tuple timestamping approach. A memory resident prototype implementation using attribute timestamping (TDBMS) is reported in [Gor92]. TDBMS has been used to study the performance of the attribute and tuple timestamping approaches [GTÖ95]. The results reported in this study provide useful insight into the design of temporal databases.

The notion of time, with its multiple facets, is difficult (if not impossible) to represent in the relational model since it does not adequately capture data or application semantics. This is substantiated by most of the relational temporal models that only support a discrete and linear model of time. The general limitation of the relational model in supporting complex applications has led to research into next-generation data models, specifically object data models. The research on temporal models has generally followed this trend (for example, [RS91, SC91, WD92, KS92, PM92, CITB92, BFG97]). Temporal object models can more accurately capture the semantics of complex objects and treat time as a basic component.

1.2 Scope and Contributions

Most of the temporal database research has concentrated on the definition of a particular temporal model and its incorporation into a relational DBMS (RDBMS) or object-oriented DBMS (ODBMS). Many of these do not incorporate sufficient functionality to meet the varying requirements that many applications have for temporal support. This thesis describes the development of a temporal framework that exploits object-oriented features to model the diverse aspects of time. The framework can then be tailored to reflect the temporal needs of a given class of applications. The framework is also used to instantiate a particular temporal object model which has multiple facets of time. The model is subsequently used to manage advanced DBMS functionality, such as schema evolution. This section provides an overview of the contributions in each of these areas.

1.2.1 Temporal Framework Issues

The relational and object-oriented approaches to model temporal information have led to the definition and design of a multitude of temporal models. Many of these assume a set of fixed notions about time, and therefore do not incorporate sufficient functionality or extensibility to meet the varying temporal requirements of today's applications. Instead, similar functionality is re-engineered every time a temporal model is created for a new application. Wuu & Dayal [WD92] provide an abstract *time* type to model the most general semantics of time which can then be subtyped (by the user or database designer) to model the various notions of time required by specific applications. However, this requires significant support from the user, including specification of the temporal schema.

Although most temporal models were designed to support the temporal needs of a particular application, or group of similar applications, if we look at the functionality offered by the temporal models at an abstract level, there are notable similarities in their temporal features:

- Each temporal model has one or more temporal primitives, namely, *time instant*, *time interval*, *time span*, etc. The *discrete* or the *continuous* domain is used by each temporal model as a temporal domain over the primitives.

- Some temporal models require their temporal primitives to have the same underlying *granularity*, while others support multiple granularities and allow temporal primitives to be specified in different granularities.

- Most temporal models support a *linear* model of time, while a few support a *branching* model. In the former, temporal primitives are totally ordered, while in the latter they have a partial order defined on them.

- All temporal models provide some means of modeling historical information about real-world entities and/or histories of entities in the database. Two of the most popular types of histories that have been employed are *valid* and *transaction* time histories [Sno87].

These commonalities suggest a need for combining the diverse features of time under a single infrastructure that is extensible and allows design reuse. In this thesis, an object-oriented framework [JF88] is presented that provides such a unified infrastructure. An object-oriented approach allows us to capture the complex semantics of time by representing it as a basic entity. In addition, the typing and inheritance mechanisms of object-oriented systems directly enable the various notions of time to be reflected in a single framework. Furthermore, temporal models and applications requiring temporal support can be built using the pre-existing components of the framework. This allows reuse of a small number of types, thereby minimizing the code needed to develop new models. The temporal framework has been implemented in C++ on Sun Solaris. Additionally, a graphical user interface (GUI) has been implemented in perl/Tk as a front-end to the temporal framework. The GUI allows a user to tailor the framework by selecting the desired temporal features. The fundamental contributions of the temporal framework are as follows:

1. A design space for temporal models. This involves identifying the design dimensions and their temporal features, and exploring the dependencies within and among the design dimensions to structure the design space. The design space is then represented by exploiting object-oriented features to model the different aspects of time (Section 2.3).

2. The temporal framework can be tailored to accommodate real-world applications that have different temporal needs (Section 2.4).

3. The various existing temporal object models can be represented within the framework (Section 2.4).

4. The framework can be used to analyze and compare the different temporal object models based on the design dimensions. This gives an indication of how temporal object models range in their provision of different temporal features of a design dimension (Section 2.5).

1.2.2 Temporal Object Model Issues

The design and development of the temporal object model, as an example instantiation of the temporal framework, is conducted within the context of the TIGUKAT[1] system [ÖPS+95]. The TIGUKAT temporal object model provides concrete and consistent semantics for the different temporal features of the framework which is necessary for their coexistence. The behavioral and uniform features of the TIGUKAT object model are exploited in order to incorporate time uniformly. The philosophy behind adding temporality to the TIGUKAT object model is to accommodate multiple applications which have different type semantics requiring various notions of time. Consequently, the TIGUKAT temporal object model consists of an extensible set of primitive time types with a rich set of behaviors to consistently and uniformly model the diverse features of time. Chapter 3 provides more details on the TIGUKAT temporal object model. The fundamental contributions of the TIGUKAT temporal object model are the following:

1. The model manages both anchored (time instant, time interval) and unanchored (time span) temporal data of multiple granularities. In supporting temporal data that is specified in different granularities, numerous approaches have been proposed to deal with the issues of converting temporal data from one granularity to another [CC87, WJL91, WJS93, WBBJ97, BP85, MPB92, MMCR92, Sno95b]. The emphasis, however, has only been on granularity conversions with respect to anchored temporal data. This is because a granularity in these approaches is modeled as an *anchored* partitioning of the time axis, thereby making it difficult to deal with granularity conversions in unanchored temporal data. The TIGUKAT temporal object model

[1] TIGUKAT (tee-goo-kat) is a term in the language of Canadian Inuit people meaning "objects." The Canadian Inuit, commonly known as Eskimos, are native to Canada with an ancestry originating in the Arctic regions of the country.

provides a novel approach to the treatment of granularity in temporal data. A granularity is modeled as a special kind of unanchored temporal primitive that can be used as a unit of time. That is, a granularity is modeled as a *unit unanchored temporal primitive*. Granularities are accommodated within the context of *calendars* and granularity conversions are presented and discussed in terms of unanchored durations of time. This allows consistent modeling and operation on unanchored temporal data comprised of different and mixed granularities. Specifically, the TIGUKAT temporal object model provides a means to represent unanchored temporal data, procedures to convert the temporal data to a given granularity, canonical forms for the data, and operations between the data. The model also provides a means to represent anchored temporal data at different granularities and gives the semantics of operations on anchored temporal data.

2. Both discrete and continuous domains of time are supported. This is in contrast to previous work which deals with only a single domain of time which is usually discrete [Sno92]. The need to support different time domains in a general object model is the emerging consensus in the temporal database research community [DSS94]. In the TIGUKAT temporal object model, discrete temporal primitives with different granularities and continuous temporal primitives are consistently represented.

3. The notion of a *timeline* to represent an axis over which time can be perceived is supported. Different types of orderings which give timelines their structural characteristics are identified. As a result, both linear and branching orders are supported. Most temporal object models support only linear time. Furthermore, timelines can be comprised of time intervals or time instants or both. This facilitates the definition of both homogeneous and heterogeneous timelines, thereby allowing events that take place at particular moments of time and those that take place within a duration of time to be modeled on the same timeline. This feature of heterogeneous timelines has not been addressed before in the temporal database research community.

4. Since the TIGUKAT model is behavioral, different dimensions of time (e.g., valid and transaction time dimensions) are represented using separate behaviors in contrast to structurally combining them in a single behavior. Thus, the approach of modeling different dimensions of time in TIGUKAT is purely behavioral and encapsulates the structure within behaviors.

1.2.3 Schema Evolution Issues

The provision of time in the TIGUKAT object model establishes a platform from which temporality can be used to investigate advanced database features such as schema evolution. The issues of schema evolution and temporal object models are generally considered to be orthogonal and are handled independently. This is unrealistic because to properly model applications that need incremental design and experimentation (such as CAD, software design process), the evolutionary histories of the schema objects should be traceable. In this thesis, a method for managing schema evolution by exploiting the functionality of the TIGUKAT temporal object model is presented. Given that the applications supported by ODBMSs need support for incremental development and experimentation with changing and evolving schema, a temporal domain is a natural means for managing changes in schema and ensuring consistency of the system. The result is a uniform treatment of schema evolution and temporal support for many ODBMS applications that require both.

Schema evolution is the process of allowing changes to schema without loss of information. Typical schema changes include adding and dropping behaviors (properties) defined on a type, and adding and dropping subtype relationships between types, to name a few. The meta-model of TIGUKAT is uniformly represented within the object model itself, providing reflective capabilities [PÖ93]. One result of this uniform approach is that schema objects (e.g., types) are objects with well-defined behaviors. The approach of keeping track of the changes to a type is the same as that for keeping track of the changes to objects. By defining appropriate behaviors on the meta-architecture, the evolution of schema is supported. Any changes in schema object definitions involve changing the history of certain behaviors to reflect the changes. For example, adding a new behavior to a type changes the history of the type's interface to include the new behavior. The old interface of the type is still accessible at a time before the change was made.

Using time to maintain and manage schema changes gives substantial flexibility in the software design process. It enables the designers to retrieve the interface of a type that existed at any time in the design phase, reconstruct the super(sub)-lattice of a type as it was at a certain time (and subsequently the type lattice of the object database at that time), and trace the implementations of a certain behavior in a particular type over time.

A change to the schema of an object database system necessitates corresponding changes to the underlying object instances in order to ensure the overall consistency of the system. Change propagation deals with reflecting changes to the individual objects by *coercing* them to coincide with the new schema definition. Two main approaches have been proposed to deal with coercing object instances to reflect the changed schema: immediate and deferred object coercions. Immediate object coercion results in suspension of all running programs until all objects have been coerced, while deferred object coercion leads to delays each time an object is accessed.

In this thesis, the strategy for change propagation supports both deferred object update semantics and immediate object update semantics. The granularity of object coercion is based on individual behaviors. That is, individual behaviors defined on the type of an object can be coerced to a new definition for that object when the object is accessed, leaving the other behaviors to retain their old definitions. This is in contrast to other models where an object is converted in its entirety to a changed type. The approach taken in this thesis has two distinct advantages depending on whether deferred or immediate update semantics are used. If deferred update semantics are used, the "behavior-at-a-time" coercion results in an even "lazier" update semantics – a behavior application to an object results in the update of only part of the object's structure. Updates due to other behavior changes are delayed until they are needed by other behavior applications. If immediate update semantics are used, then the update can be done more quickly since the system knows that changes to the affected type are localized to the single behavior that was just changed. This is important because the major drawback of immediate update semantics is the speed of update. Another identifying characteristic of the propagation model is that a historical record of the coerced behaviors is maintained for each object so that even if behaviors are coerced to reflect an update to an object, older definitions of the behaviors can still be accessed for each object.

1.3 Organization

The remainder of this thesis is organized into four chapters defining the temporal framework, the temporal object model, schema evolution management, and a summary chapter which contains concluding remarks and future research directions.

Chapter 2 presents the temporal framework by identifying the design space for temporal models. The design space is classified across four design dimensions (key abstractions). Interactions within and between the dimensions are identified in order to structure the design space. The design space is then represented by exploiting object-oriented techniques. Object-oriented types are used to model the design dimensions and their temporal features, while object-oriented properties (abstractions of methods and attributes in traditional object-oriented terminology) are used to model the relationships between the design dimensions and the operations on their temporal features. The chapter also describes the functionality of the temporal toolkit and illustrates how the toolkit can be used to tailor the temporal framework in order to accommodate the temporal needs of different applications, and temporal models. Object-oriented techniques are then used to compare and analyze different temporal object models with respect to the design dimensions of the framework. Finally, the implementation details of the framework are outlined.

Chapter 3 presents an example instantiation of the temporal framework within the context of the TIGUKAT temporal object model, demonstrating a single temporal model with multiple facets of time. The model consists of an extensible set of primitive time types with a rich set of behaviors to consistently and uniformly model the design space for temporal object models.

Chapter 4 examines the issue of managing schema evolution using a temporal object model. A method for managing schema changes and propagating the changes to underlying instances by exploiting the functionality of the TIGUKAT temporal object model is presented.

Chapter 5 presents conclusions and contributions of this thesis. The results are summarized and avenues for future research are outlined.

Chapter 2

An Object-Oriented Framework for Temporal Data Models

2.1 Overview

In this chapter[1], an object-oriented framework is presented that provides a unified infrastructure for the diverse notions of time. The temporal framework consists of abstract object-oriented types and properties (abstractions of methods and attributes in traditional object-oriented terminology). The types are used to model the different temporal features in each of the design dimensions, while the properties are used to model the different operations on each temporal feature and to represent the dependencies between the design dimensions. These types and properties could then be used by any temporal model to define the semantics of their specific notion of time. The framework can be considered as an extension to the work of Wuu & Dayal [WD92] in that it provides the user or database designer with explicit types and properties to model the diverse features of time. The specification of a schema for modeling time is complex, and certainly not trivial. It is therefore imperative for temporal object models to have a temporal infrastructure from which they can choose the temporal features they need.

A parallel can be drawn between the temporal framework presented in this thesis and similar (albeit on a much larger scale) approaches used in *Choices* [CJR87] and *cmcc* [ATGL96]. *Choices* is a framework for operating system construction which was designed to provide a family of operating systems that could be reconfigured to meet diverse requirements posed by an application or a user. *cmcc* is an optimizing compiler that makes use of frameworks to facilitate code reuse for different modules of a compiler. Similar to *Choices* and *cmcc*, the temporal framework can be regarded as an attempt to construct a family of temporal models. The framework can then be tailored to reflect a temporal model or application which need certain notions of time. A particular temporal model or application

[1]This chapter has appeared as [GÖS98] and portions of an earlier version have appeared as [GÖS97b].

would be one of the many "instances" of the framework.

A similar objective to the temporal framework is pursued by Wuu & Dayal [WD92] who provide an abstract *time* type to model the most general semantics of time which can then be subtyped (by the user or database designer) to model the various notions of time required by specific applications. The temporal framework presented in this thesis subsumes the work of Wuu & Dayal in that it provides the user or database designer with explicit types and properties to model the diverse features of time. The approach of Wuu & Dayal requires significant support from the user, including specification of the temporal schema, which is a complex, and non-trivial task. It is therefore imperative for temporal object models to have a temporal infrastructure from which users can choose the temporal features they need.

The diverse features of time are also identified in [Sno95a]. The focus however, is on comparing various temporal object models and query languages based on their ability to support valid and transaction time histories. In this thesis, the generic aspects of temporal models can be captured and described using a single framework. In [PLL96] a temporal reference framework for multimedia synchronization is proposed and used to compare existing temporal specification schemes and their relationships to multimedia synchronization. The focus however, is on different forms of temporal specification, and not on different notions of time. The model of time used concentrates only on temporal primitives and their representation schemes.

The rest of this chapter is organized as follows. Section 2.2 gives an overview of object-oriented frameworks. The architecture of the temporal framework is presented in Section 2.3. Section 2.4 shows how the temporal framework can be tailored to accommodate different applications and temporal models. Finally, Section 2.5 describes the comparison of different temporal object models using the temporal framework.

2.2 Object-Oriented Frameworks

A framework characterizes the reusable architectural design of a system; the kinds of objects in the system and how they interact. Frameworks consist of collections of abstract classes that capture the common aspects of applications in a certain problem domain, and the way in which the instances of these classes collaborate. More specifically, a framework is a set of classes that embodies an abstract design for solutions to a family of related problems [JF88]. The framework can then be specialized and/or instantiated to implement a particular problem. With frameworks, software developers don't have to start from scratch each time they write an application. Frameworks are built from a collection of objects, so both the design and code of a framework can be reused [Tal94]. This is useful in software development since it saves time and effort. The Model-View-Controller of Smalltalk-80 [KP88] and the Unidraw graphical editor [VL90] are two examples of frameworks that have been developed for graphical user interfaces. Frameworks have also been used in other domains such as operating systems [CJR87], network protocols [HJE95], and compilers [ATGL96], to name a few.

2.3 The Architecture of the Temporal Framework

In order to accommodate the varying requirements that many applications have for temporal support, the design dimensions that span the design space for temporal models are first identified. Next, the components or features of each design dimension are identified. Finally, the interactions between the design dimensions are explored in order to structure the design space. These steps produce a framework which consists of abstract and concrete object types, and properties. The types are used to model the different design dimensions and their corresponding components. The properties are used to model the different operations on each component, and to represent the relationships (constraints) between the design dimensions. The framework classifies design alternatives for temporal models by providing types and properties that can be used to define the semantics of many different specific notions of time.

2.3.1 Design Dimensions

The design alternatives for temporal models can be classified along four design dimensions:

1. *Temporal Structure* – provides the underlying ontology and domains for time.
2. *Temporal Representation* – provides a means to represent time so that it is human readable.
3. *Temporal Order* – gives an ordering to time.
4. *Temporal History* – allows events and activities to be associated with time.

There are two parts to the description of a design dimension. First, a set of temporal features that the design dimension encompasses is defined. Second, the relationships between the temporal features is explored and the resulting design space for the design dimension is described. The design space consists of an architectural overview of abstract and concrete types corresponding to the temporal features, and a design overview which describes some of the key properties (operations) defined in the interface of the types. The properties are not described in detail since many of these are traditional temporal operations that have already appeared in the literature on temporal databases.

The availability of commonly used object-oriented features is assumed – *atomic entities* (reals, integers, strings, etc.); *types* for defining common features of objects; *properties* (which represent *methods* and *instance variables*) for specifying the semantics of operations that may be performed on objects; *classes* which represent the extents of types; and *collections* for supporting general heterogeneous groupings of objects. In this chapter, a reference prefixed by "T_" refers to a type, and "*P_*" to a property. A type is represented by a rounded box. An abstract type is shaded with a black triangle in its upper left corner, while a concrete type is unshaded. In Figures 2.5, 2.8, 2.9, and 2.18 the rectangular boxes are objects. Objects have an outgoing edge for each property applicable to the object which is labeled with the name of the property and which leads to an object resulting from the application of the property to the given object. A circle labeled with the symbols { } represents a container object and has outgoing edges labeled with "∈" to each member object.

2.3.1.1 Temporal Structure

The first question about a temporal model is "what is its underlying temporal structure?" More specifically, what are the temporal primitives supported in the model, what temporal domains are available over these primitives, and what is the temporal determinacy of the primitives? Indeed, the temporal structure dimension with its various constituents forms the basic building block of the design space of any temporal model since it is comprised of the basic temporal features that underlie the model. An overview of the features of a temporal structure is given and the relationships that exist between them are then identified.

Components

1. **Temporal Primitives**

   Temporal primitives can either be *anchored* (*absolute*) or *unanchored* (*relative*) [Sno92]. For example, 31 *July* 1995 is an anchored temporal primitive since its exact location on the time axis is known, whereas 31 *days* is an unanchored temporal primitive since it can stand for any block of 31 consecutive days on the time axis.

   There is only one unanchored primitive, called the *span*. A span is a duration of time with a known length, but no specific starting and ending anchor points. There are two anchored primitives, the *instant* (*moment*, *chronon*) and the *interval*. An instant is a specific anchored moment in time, e.g., 31 *July* 1995. An interval is a duration of time between two specific anchor points (instants) which are the lower and upper bounds of the interval, e.g., [15 *June* 1995, 31 *July* 1995].

2. **Temporal Domain**

   The temporal domain of a temporal structure defines a scale for the temporal primitives. A temporal domain can be *continuous* or *discrete*. Discrete domains map temporal primitives to the set of integers. That is, for any temporal primitive in a discrete time domain, there is a unique successor and predecessor. Continuous domains map temporal primitives to the set of real numbers. Between any two temporal primitives of a continuous time domain, another temporal primitive exists. Most of the research in the context of temporal databases has assumed that the temporal domain is discrete. Several arguments in favor of using a discrete temporal domain are made by Snodgrass [Sno92] including the imprecision of clocking instruments, compatibility with natural language references, possibility of modeling events which have duration, and practicality of implementing a continuous temporal data model. However, Chomicki [Cho94] argues that the continuous (dense) temporal domain is very useful in mathematics and physics. Furthermore, continuous time provides a useful abstraction if time is thought of as discrete but with instants that are very close. In this case, the set of time instants may be very large which in turn may be difficult to implement efficiently. Chomicki further argues that query evaluation in the context of constraint databases [KKR90, Rev90] has been shown to be easier in continuous domains than in discrete domains. Continuous temporal domains have also been used to facilitate full abstract semantics in reasoning about concurrent programs [BKP86]. A general framework for supporting temporal primitives (instants, intervals, sets of intervals) that allows seamless integration of dense and discrete temporal domains of time over a linearly ordered, unbounded point structure is given in [GLÖS97].

3. **Temporal Determinacy**

   There are many real world cases which have complete knowledge of the time or the duration of a particular activity. For example, the time interval allowed for students to complete their *Introduction to Database Management Systems* examination is known for certain. This is an example of a determinate temporal primitive. However, there are cases when the knowledge of the time or the duration of a particular activity is known only to a certain extent. For example, the exact time instant when the Earth was formed is unknown, though one may speculate on an approximate time for this event. In this case, the temporal primitive is indeterminate. Indeterminate temporal information is also prevalent in various sources such as granularity, dating techniques, future planning, and unknown or imprecise event times [DS93]. Since the ultimate purpose of a temporal model is to represent real temporal information, it is desirable for such a model to be able to capture both determinate and indeterminate temporal primitives.

Design Space

Figure 2.1 shows the building block hierarchy of a temporal structure. The basic building block consists of anchored and unanchored temporal primitives. The next building block provides a domain for the primitives that consists of discrete or continuous temporal primitives. Finally, the last building block of Figure 2.1 adds determinacy. Thus, a temporal structure can be defined by a series of progressively enhanced temporal primitives.

Figure 2.2 gives a detailed hierarchy of the different types of temporal primitives that exist in each of the building blocks of Figure 2.1. Based on the features of a tem-

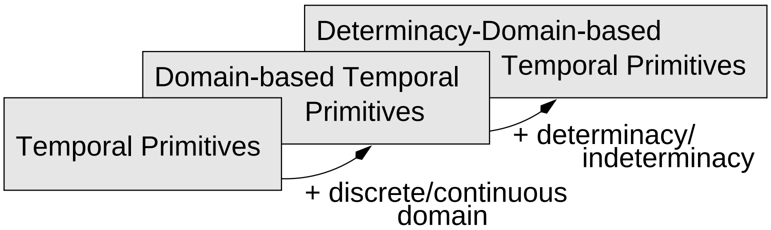


Figure 2.1: Building a Temporal Structure

poral structure, its design space consists of 11 different kinds of temporal primitives. These are the determinacy-domain-based temporal primitives shown in Figure 2.2 and described below.

Continuous time instants and intervals. Continuous instants are just points on the (continuous) line of all anchored time specifications. They are totally ordered by the relation "later than." Since, in theory, continuous instants have infinite precision, they cannot have a period of indeterminacy. Therefore, continuous indeterminate time instants do not exist in Figure 2.2. However, continuous intervals can be determinate or indeterminate. The difference between them is that a continuous determinate interval denotes that the activity associated with it occurs during the *whole* interval, while a continuous indeterminate interval denotes that the activity associated with it occurs *sometime* during the interval. Continuous intervals have lower and upper bounds which are continuous instants.

Discrete time instants and intervals. Assume that somebody has been on a train the whole day of 5 January 1987. This fact can be expressed using a determinate time instant $5\ January\ 1987_{det}$ (which means *the whole day of*). However, the fact that somebody is leaving for Paris on 5 January 1987 can be represented as an indeterminate time instant $5\ January\ 1987_{indet}$ (which means *some time on that day*). Hence, each discrete time instant is either *determinate* or *indeterminate*, corresponding to the two different interpretations. Determinate and indeterminate discrete time instants can subsequently be used to form *discrete time intervals*. Determinate (indeterminate) time instants can be used as boundaries of determinate (indeterminate) time intervals.

Time spans. Discrete and continuous determinate spans represent complete information about a duration of time. A discrete determinate span is a summation of distinct granularities with integer coefficients e.g., $5\ days$ or $2\ months\ +\ 5\ days$. Similarly, a continuous determinate span is a summation of distinct granularities with real coefficients e.g., $0.31\ hours$ or $5.2\ minutes\ +\ 0.15\ seconds$.

Discrete and continuous indeterminate spans represent incomplete information about a duration of time. They have lower and upper bounds that are determinate spans. For example, $1\ day\ \sim\ 2\ days$ is a discrete indeterminate span that

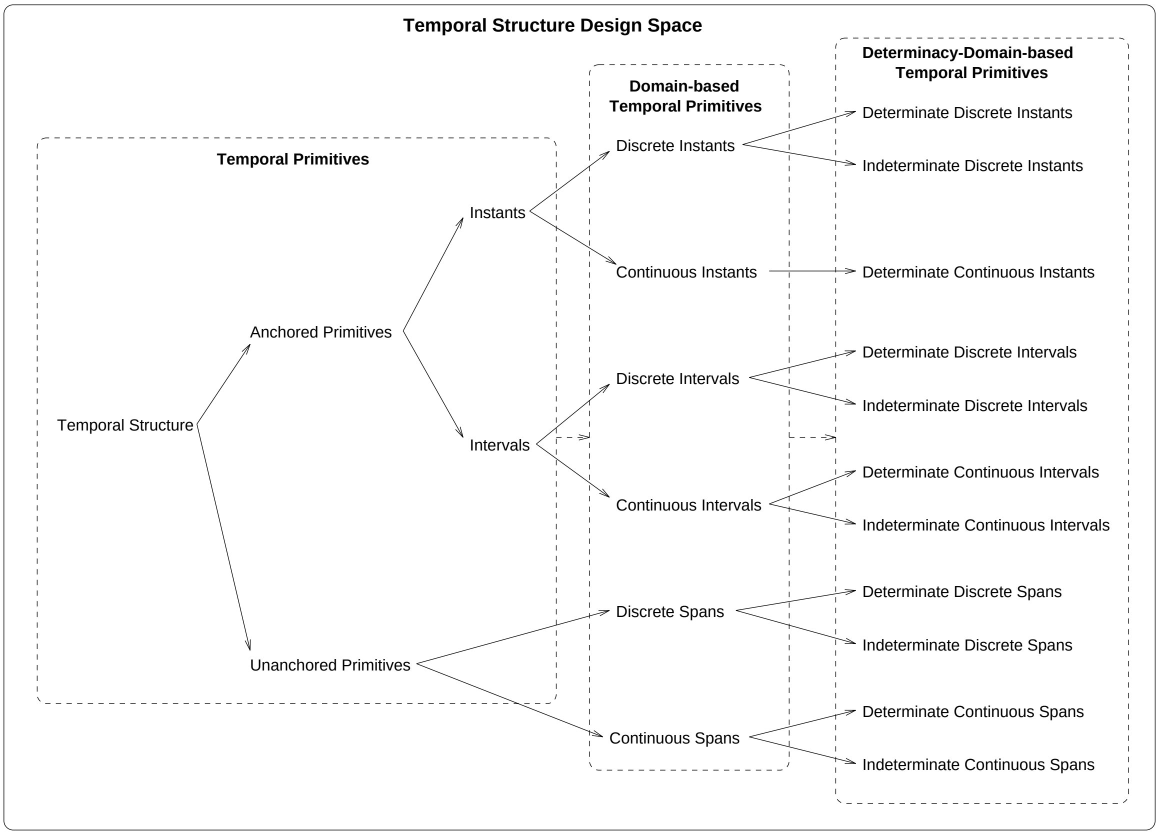


Figure 2.2: Design Space of a Temporal Structure

can be interpreted as "a time period between one and two days."

The mapping of the temporal structure to an object type hierarchy is given in Figure 2.3 which shows the types and generic properties that are used to model various kinds of determinacy-domain-based temporal primitives.

Properties defined on time instants allow an instant to be compared with another instant; an instant to be subtracted from another instant to find the time duration between the two; and a time span to be added to or subtracted from an instant to return another instant. Furthermore, properties *P_calendar* and *P_calElements* are used to link time instants to calendars which serve as a representational scheme for temporal primitives (see Section 2.3.1.2). *P_calendar* returns the calendar which the instant belongs to and *P_calElements* returns a list of the calendric elements in a time instant. For example *P_calendar* applied to the time instant 15 June 1995 would return *Gregorian*, while the application of *P_calElements* to the same time instant would return $(1995, June, 15)$.

Similarly, properties defined on time intervals include unary operations which return the lower bound, upper bound and length of the interval; ordering operations which define Allen's interval algebra [All84]; and set-theoretic operations.

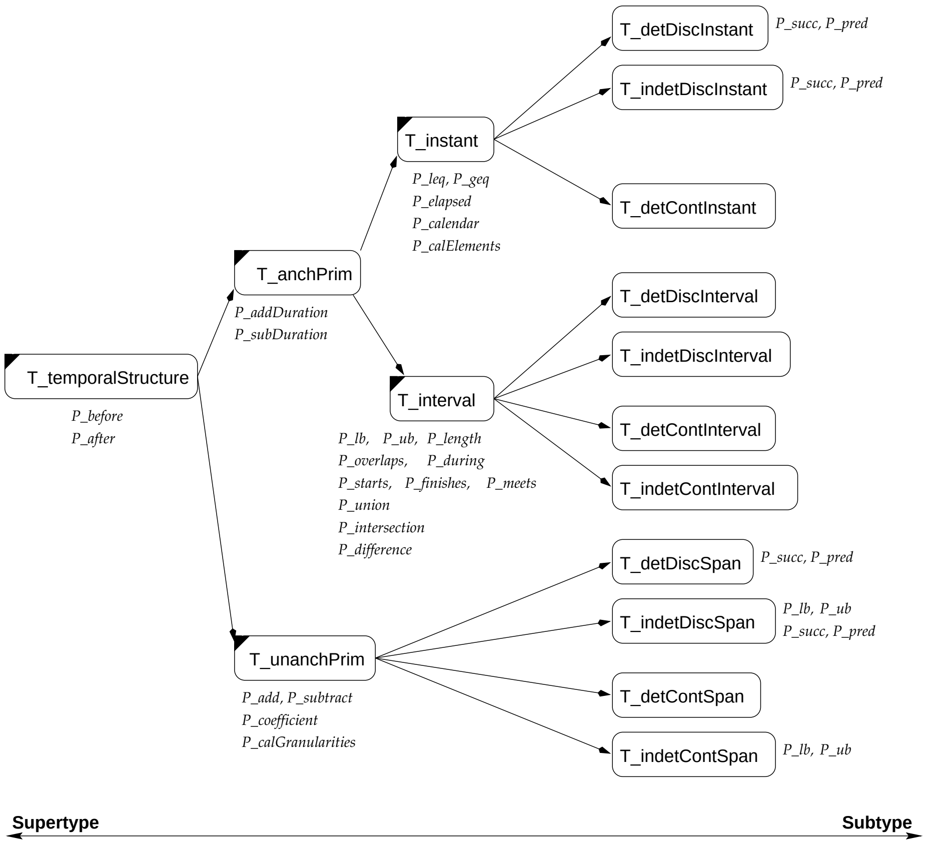


Figure 2.3: The Inheritance Hierarchy of a Temporal Structure

Properties defined on time spans enable comparison and arithmetic operations between spans. The *P_before* and *P_after* properties are refined for time spans to model the semantics of $<$ and $>$, respectively. Additionally, properties *P_coefficient* and *P_calGranularities* are used as representational properties and provide a link between time spans and calendars (see Section 2.3.1.2). *P_coefficient* returns the (real) coefficient of a time span given a specific calendric granularity. For example, $(5\ days)\cdot$ *P_coefficient*(day) returns 5.0. *P_calGranularities* returns a collection of calendric granularities in a time span. For example, the property application $(1\ month{+}5\ days)\cdot$ *P_calGranularities* returns $\{day, month\}$.

It can be noted that the properties *P_succ* and *P_pred* are defined in all the types involving discrete instant and span primitives (see Figure 2.3). This redundancy can be eliminated by refactoring the concerned types and using multiple inheritance. More specifically, an abstract type called `T_discrete` can be introduced, and the properties *P_succ* and and *P_pred* defined on it. All the types involving discrete primitives can then be made subtypes of `T_discrete`. A similar approach can be used to factor the types that define properties *P_lb* and *P_ub*. An abstract type called `T_bounds` can be introduced with the properties *P_lb* and *P_ub* defined on it. The `T_interval` type and the types involving indeterminate spans can then be made subtypes of `T_bounds`. The concept of multiple subtyping hierarchies to collect semantically related types together and avoid the duplication of properties has been reported in [HKOS96]. For example, the unanchored primitives hierarchy can be re-structured as shown in Figure 2.4.

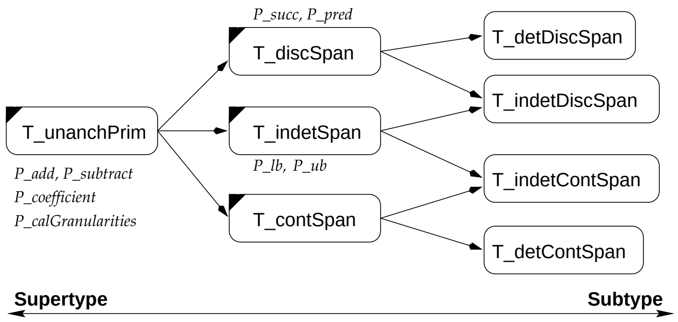


Figure 2.4: Multiple Subtyping Hierarchy for Unanchored Temporal Primitives

2.3.1.2 Temporal Representation

Components. For human readability, it is important to have a representational scheme in which the temporal primitives can be made human readable and usable. This is achieved by means of calendars. Common calendars include the *Gregorian* and *Lunar* calendars. Educational institutions also use *Academic* calendars.

Calendars are comprised of different time units of varying granularities that enable the representation of different temporal primitives. In many applications, it is desirable to have multiple calendars that have different calendric granularities. For example, in financial trading, multiple calendars with different time units and operations need to be available to capture the semantics of financial data [CS93, CSS94]. Extensive calendar support is also required in time series management [DDS94, LEW96].

Design Space. A calendar is composed of an origin, a set of calendric granularities, and a set of conversion functions. The origin marks the start of a calendar[2]. Calendric granularities define the reasonable time units (e.g., *minute*, *day*, *month*) that can be used in conjunction with this calendar to represent temporal primitives. A calendric granularity also has a list of *calendric elements*. For example in the Gregorian calendar, the calendric granularity *day* has the calendric elements *Sunday*, *Monday*, ..., *Saturday*. Similarly in the Academic calendar, the calendric granularity *semester* has the calendric elements *Fall*, *Winter*, *Spring*, and *Summer*. The conversion functions establish the conversion rules between calendric granularities of a calendar.

[2]The definition of a calendar in this thesis is different from that defined in [CS93, CSS94, LEW96] where structured collections of time intervals are termed as "calendars." The definition in this thesis adheres closely to the human understanding of a calendar. However, the extensibility feature of the framework allows other notions of calendars to be incorporated easily under the temporal representation design dimension.

Since all calendars have the same structure, a single type, called `T_calendar` can be used to model different calendars, where instances represent different calendars. The basic properties of a calendar are, *P_origin*, *P_calGranularities*, and *P_functions*. These allow each calendar to define its origin, calendric granularities, and the conversion functions between different calendric granularities. Calendric granularities and their conversion functions are treated in detail within the context of the TIGUKAT temporal model (see Section 3.2).

Example 2.1 Figure 2.5 shows four instances of `T_calendar` – the **Gregorian**, **Lunar**, **Academic**, and **Fiscal** calendars. The origin of the Gregorian calendar is given as the span 1582 *years* from the start of time since it was proclaimed in 1582 by Pope Gregory XIII as a reform of the Julian calendar. The calendric granularities in the Gregorian calendar are the standard ones, **year**, **month**, **day**, etc. The origin of the Academic calendar shown in Figure 2.5 is assumed to be the span 1908 *academicYears* having started in the year 1908, which is the establishment date of the University of Alberta. The Academic calendar has similar calendric granularities as the Gregorian calendar and defines a new calendric granularity of **semester**. The semantics of the Lunar and Fiscal calendars could similarly be defined. □

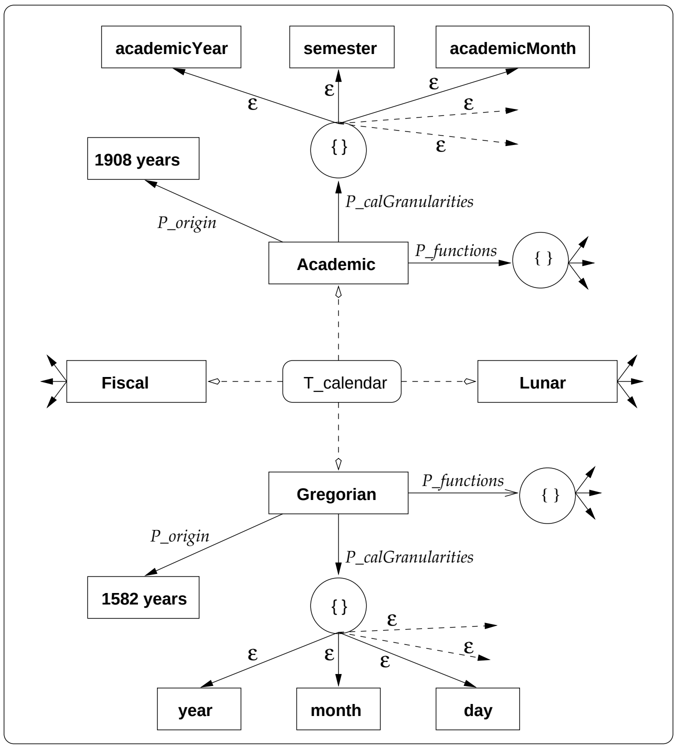


Figure 2.5: Temporal Representational Examples

2.3.1.3 Temporal Order

The means of designing the temporal structure and the temporal representation of a temporal model leads to the next step of providing an ordering scheme for the temporal primitives. This constitutes the third building block of the design space.

Components. A temporal order can be classified as being *linear* or *branching*. In a linear order, time flows from past to future in an ordered manner. In a branching order, time is linear in the past up to a certain point, when it branches out into alternate futures. The structure of a branching order can be thought of as a tree defining a partial order of times. The trunk (stem) of the tree is a linear order and each of its branches is a branching order. The linear model is used in applications such as office information systems. The branching order is useful in applications such as computer aided design and planning or version control which allow objects to evolve over a non-linear (branching) time dimension (e.g., multiple futures, or partially ordered design alternatives).

Design Space. The different types of temporal orders are dependent on each other. A *sub-linear* order is one in which the temporal primitives (time intervals) are allowed to overlap, while a *linear* order is one in which the temporal primitives (time intervals) are not allowed to overlap. Every linear order is also a sub-linear order. A branching order is essentially made up of sub-linear orders. The relationship between temporal orders is shown in Figure 2.6.

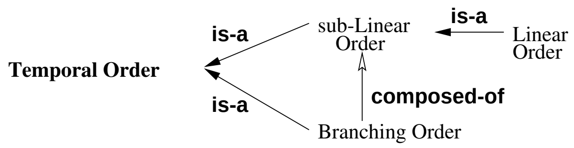


Figure 2.6: Temporal Order Relationships

The hierarchy in Figure 2.7 gives the various types and properties which model different temporal orders[3].

Example 2.2 Consider the operations that take place in a hospital on any particular day. It is usually the case that at any given time multiple operations are taking place. Assume that an eye cataract surgery took place between 8am and 10am, a brain tumor surgery took place between 9am and 12pm, and an open heart surgery took place between 7am and 2pm on a certain day. Figure 2.8 shows a pictorial representation of operationsOrder, which is an object of type `T_subLinearOrder`. operationsOrder consists of the time intervals [08:00,10:00], [09:00,12:00], [07:00,14:00], and does not belong to any branching timeline. As seen in the figure, operationsOrder consists of intervals (representing the time periods during

[3] *Periodic* temporal orders are not considered in this thesis. However, these can easily be incorporated as a subtype of `T_temporalOrder`.

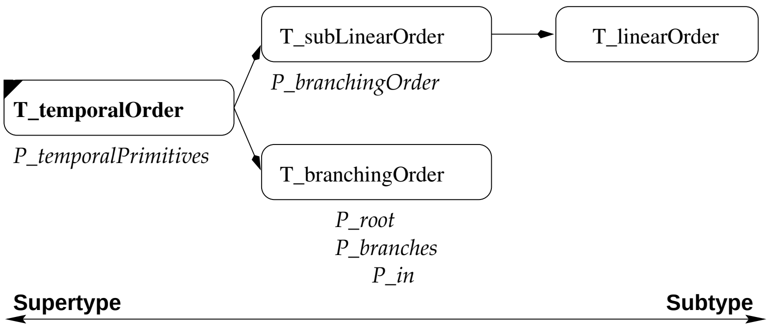


Figure 2.7: The Hierarchy of Temporal Orders

which the different surgeries took place) that overlap each other. Hence, operationsOrder is an example of a sub-linear order. □

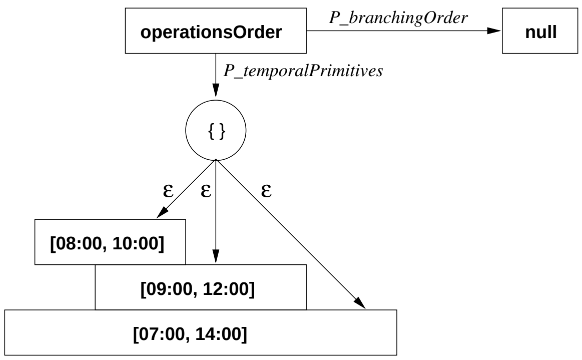


Figure 2.8: An Example of a Sub-Linear Order.

Example 2.3 To illustrate the use of objects of type `T_linearOrder` which are total linear temporal orders, consider a patient with multiple pathologies, for example as a result of diabetes. The patient has to attend several special clinics, each on a different day. Hence, it follows that since the patient cannot attend more than one special clinic on any day, the temporal order of the patient's special clinics visit history is linear and totally ordered. Suppose the patient visited the opthalmology clinic on 10 January 1995, the cardiology clinic on 12 January 1995, and the neurology clinic on 3 February 1995. Figure 2.9 shows a pictorial representation of specialClinicOrder, which is an object of type `T_linearOrder`. As seen in the figure, specialClinicOrder is totally ordered as its time intervals do not overlap. □

Example 2.4 Consider an observational pharmacoeconomic analysis of the changing trends, over a period of time, in the treatment of a chronic illness such as asthma [GÖS97a]. The analysis would be performed using information gathered over a time period. At a fixed point during this period new guidelines for the treatment of asthma were released. At that

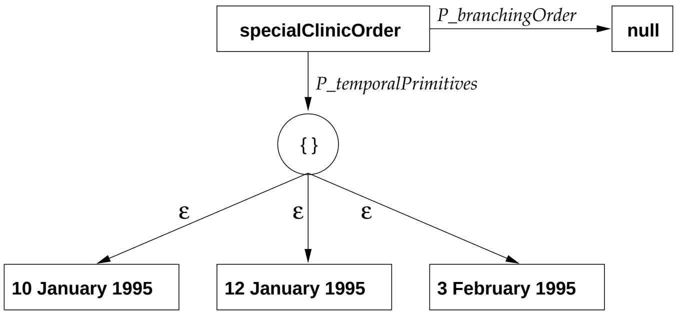


Figure 2.9: An Example of a Linear Order.

point the population of patients known to have asthma are divided into those whose doctors continue the old established treatment, and those whose doctors, in accordance with new recommendations, change their treatment. Thus, the patients are divided into two groups, each group undergoing a different treatment for the same illness. The costs and benefits accrued over the trial period for each treatment are calculated. Since such a study consists of several alternative treatments to an illness, a branching timeline is the natural choice for modeling the timeline of the study. The point of branching is the time when the new guidelines for the treatment of the illness are implemented. Figure 2.10 shows the branching timeline for such a medical trial history.

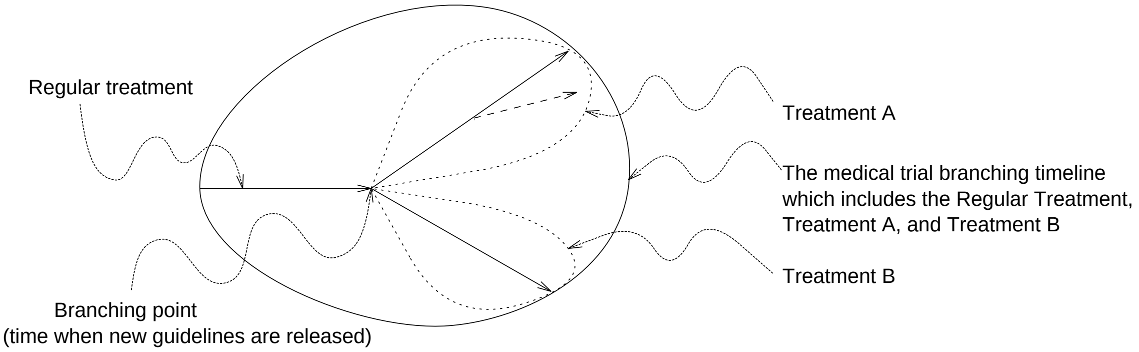


Figure 2.10: An Example of a Branching Order.

The same branching timeline could as easily handle the situation where different versions of a particular treatment, say Treatment A, are implemented based on certain parameters. In this case, the "Treatment A" branch would in turn branch at a certain point into different Treatment A versions. This situation is also depicted in Figure 2.10. □

2.3.1.4 Temporal History

So far various features of time have been considered; its structure, the way it is represented, and how it is ordered. The final building block of the design space of temporal models makes it possible to associate time with entities to model different temporal histories.

Components. One requirement of a temporal model is an ability to represent and manage real-world entities as they evolve over time and assume different states (values). The set of these values forms the *temporal history* of the entity.

Two basic types of temporal histories are considered in databases which incorporate time. These are *valid* and *transaction* time histories [SA85]. Valid time denotes the time when an entity is effective (models reality), while transaction time represents the time when a transaction is posted to the database. Usually valid and transaction times are the same. Other temporal histories include *event* time [RS91, CK94] and *decision* time [EGS93] histories. Event (decision) time denotes the time the event occurred in the real-world. Valid, transaction, and event times have been shown to be adequate in modeling temporal histories [CK94].

Design Space. Since valid, transaction, and event time histories have different semantics, they are orthogonal. Figure 2.11 shows the various types that could be used to model these different histories. A temporal history consists of objects and their associated timestamps.

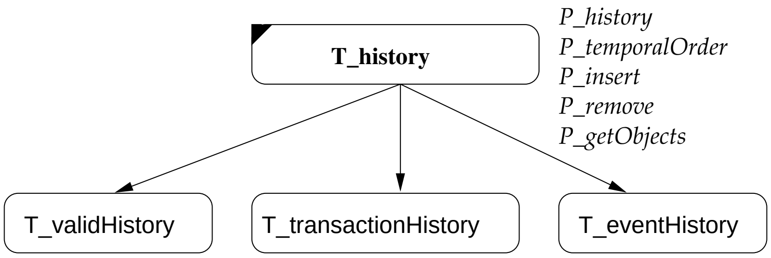


Figure 2.11: The Types and Properties for Temporal Histories

Property *P_history* defined on `T_history` returns a collection of all *timestamped* objects that comprise the history. A history object also knows the temporal order of its temporal primitives. The property *P_temporalOrder* returns the temporal order (which is an object of type `T_temporalOrder`) associated with a history object. The temporal order basically orders the time intervals (or time instants) in the history. Another property defined on history objects, *P_insert*, timestamps and inserts an object in the history. Property *P_remove* drops a given object from the history at a specified temporal primitive. The *P_getObjects* property allows the user to get the objects in the history at (during) a given temporal primitive. The properties defined on `T_history` are refined in `T_validHistory`, `T_transactionHistory`, and `T_eventHistory` types to model the semantics of the different kinds of histories. Moreover, each history type can define additional properties, if necessary, to model its particular semantics. The clinical example described in Section 2.4.2 illustrates the use of the properties defined on `T_history`.

2.3.2 Relationships between Design Dimensions

In the previous section the building blocks (design dimensions) for temporal models were described and the design space of each dimension was identified. In this section, the interactions between the design dimensions are explored. This will enable the building blocks to be put together and will give structure to the design space for temporal models.

A temporal history is composed of entities which are ordered in time. This temporal ordering is over a collection of temporal primitives in the history, which in turn are represented in a certain manner. Hence, the four dimensions can be linked via the "has-a" relationship shown in Figure 2.12.

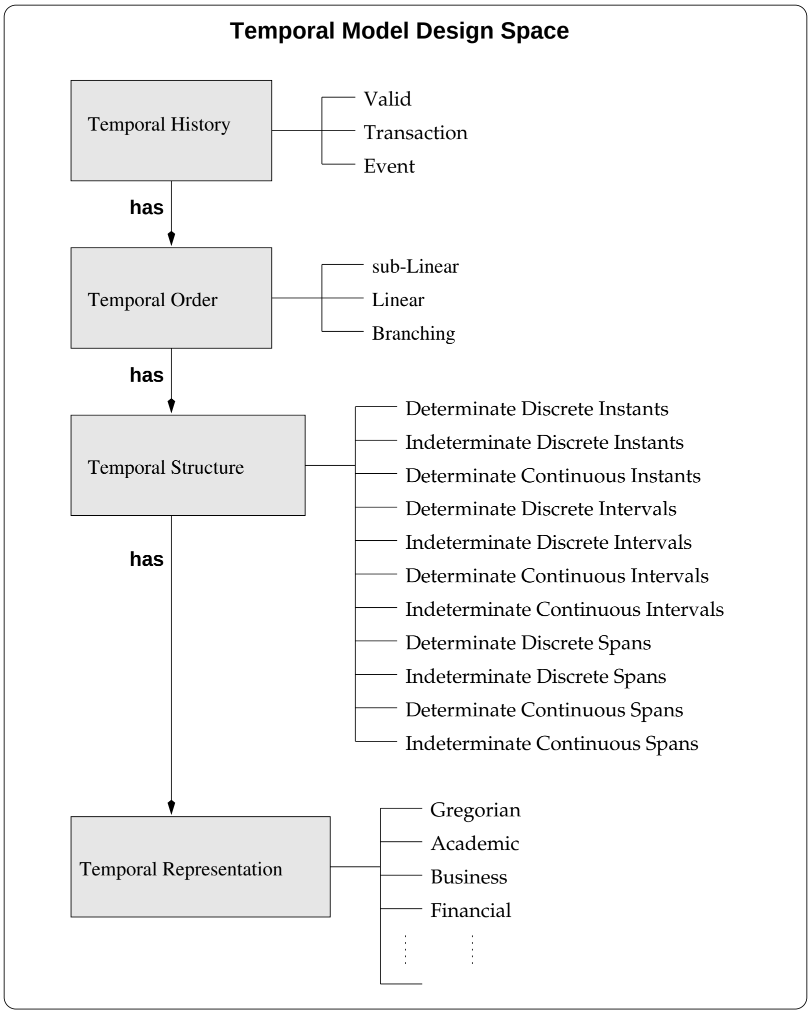


Figure 2.12: Design Space for Temporal Models

Basically, a temporal model can be envisioned as having a notion of time, which has an underlying temporal structure, a means to represent the temporal structure, and different temporal orders to order the temporal primitives within a temporal structure. This notion of time, when combined with application objects, can be used to represent various temporal histories of the objects in the temporal model.

Figure 2.12 gives the design space for temporal models. A temporal model can support one or more of valid, transaction, event, and user-defined histories. Each history in turn has a certain temporal order. This temporal order has properties which are defined by the type of temporal history (linear or branching). A linear history may or may not allow overlapping of anchored temporal primitives that belong to it. If it does not allow overlapping, then such a history defines a total order on the anchored temporal primitives that belong to it. Otherwise, it defines a partial order on its anchored temporal primitives. Each order can then have a temporal structure which is comprised of all or a subset of the 11 different temporal primitives that are shown in Figure 2.2. Finally, different calendars can be defined as a means to represent the temporal primitives.

The four dimensions are modeled in an object system by the respective types shown in Figure 2.13. The "has a" relationship between the dimensions is modeled using the properties shown in the figure. An object of `T_temporalHistory` represents a temporal history. Its temporal order is obtained using the *P_temporalOrder* property. A temporal order is an object of type `T_temporalOrder` and has a certain temporal structure which is obtained using the *P_temporalPrimitives* property. The temporal structure is an object of type `T_temporalStructure`. The property *P_calendar* gives the instance of `T_calendar` which is used to represent the temporal structure.

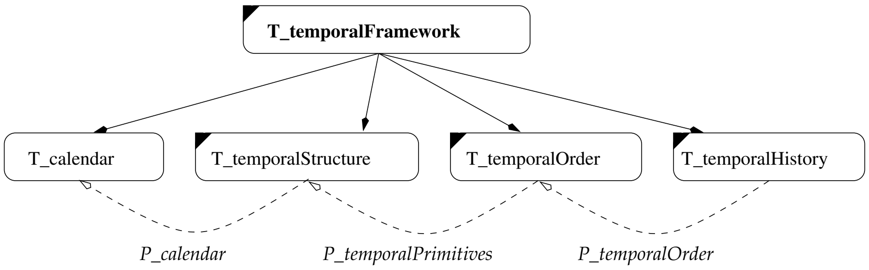


Figure 2.13: Relationships between Design Dimensions Types

The relationships shown in Figure 2.13 provide a temporal framework which encompasses the design space for temporal models. The detailed type system, shown in Figure 2.14, is based on the design dimensions identified in Section 2.3 and their various features which are given in Figures 2.3, 2.7, and 2.11. As described in Section 2.3.1.1, refactoring of types and multiple inheritance can be used to handle identical properties that are defined over different types in the inheritance hierarchy shown in Figure 2.14. The framework can now be tailored for the temporal needs of different applications and temporal models. This is illustrated in Section 2.4.

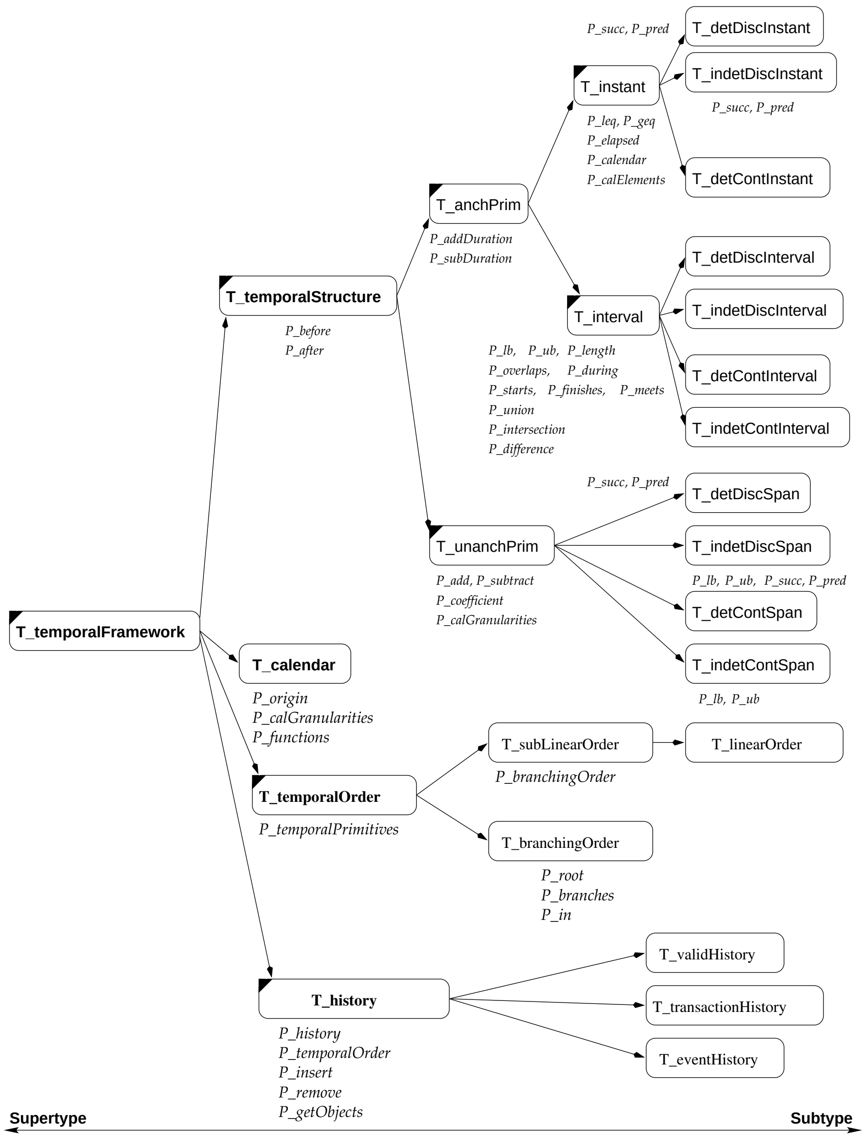


Figure 2.14: The Inheritance Hierarchy for the Temporal Framework

2.3.3 Temporal semantics in relational vs object-oriented databases

In temporal relational databases the semantics of the different notions of time usually appear in the query model. For example, the temporal relations in the database can either be point-based or interval-based and the underlying query language would have to be designed to capture point-based or interval-based semantics.

In temporal object-oriented databases much of the semantics of the different notions of time is captured by the properties (behaviors) defined on the various types that model temporal features, as shown in Figure 2.14. For example, to retrieve the length of an interval, one would just apply the *P_length* property on the interval. No explicit construct would be needed in the underlying query language. In temporal relational databases however, an explicit construct, or operation would have be defined in the query language to return the length of an interval. The semantics of the properties in the temporal framework are well defined in the literature. For example, the semantics of the ordering properties (for example, *P_overlaps*, *P_meets*) on intervals are well defined by Allen in [All84].

2.4 Tailoring the Temporal Framework

This section illustrates how the temporal framework that is defined in Section 2.3 can be tailored to accommodate applications and temporal models which have different temporal requirements. The first sub-section describes a toolkit that aids users in tailoring the temporal framework. The use of the toolkit is demonstrated in subsequent sub-sections. In the next two sub-sections, examples of two real-world applications that have different temporal needs are given. The last sub-section gives an example of a temporal object model and shows how the model can be derived from the temporal framework.

2.4.1 A Toolkit for the Temporal Framework

In this section a toolkit for using the temporal framework is described. The toolkit is comprised of a graphical user interface which provides a user with the ability to choose different temporal features that pertain to a particular application or group of applications. The toolkit has been implemented in perl/Tk and its temporal type hierarchy is shown in Figure 2.15. The type hierarchy is a little different from the type hierarchy of the temporal framework given in Figure 2.14. These differences are due to the implementation environment and are discussed in Section 2.6.

In order to select the desired temporal features, a user simply clicks on the types associated with the features. For each type selected, the toolkit highlights all other types that are related with the selected type. The related types are of two kinds: the first kind are those related to the selected type by virtue of inheritance (these are basically all the supertypes of the selected type); the second kind are those related to the selected type by virtue of the design dimension relationships shown in Figure 2.13. The toolkit provides a facility

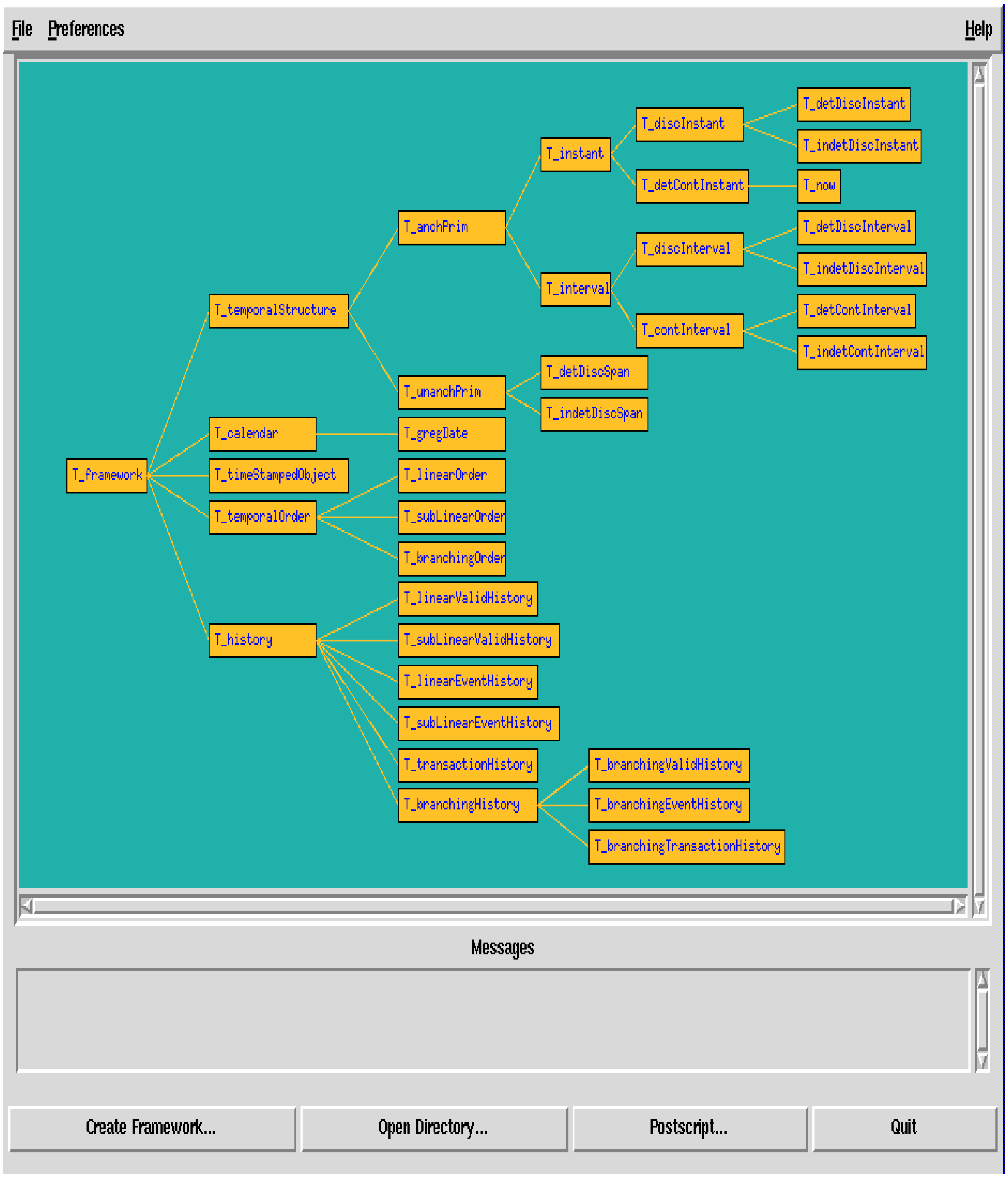


Figure 2.15: The Temporal Framework Toolkit

(the `Create Framework` button) whereby all the header and source C++ files associated with the highlighted types can be stored in a specified directory. Two additional files are also copied to the directory. The first file is a standard `Makefile` which allows the user to compile all the source files. The second file is an exemplary `main.C` file which illustrates how objects of the various temporal framework types can be created and manipulated.

Figure 2.16 shows the types highlighted when `T_branchingTransactionHistory` is selected. The types `T_branchingHistory`, `T_history`, and `T_framework` are supertypes of `T_branchingTransactionHistory`. The type `T_transactionHistory` is highlighted because the root of a branching transaction history is an object of type `T_transactionHistory`. Every branching transaction history has a branching order which is why `T_branchingOrder`, and subsequently its supertype `T_temporalOrder` are highlighted. The type `T_linearOrder` is highlighted because the root of a branching order is an object of type `T_linearOrder`. The type `T_now` and all its supertypes are highlighted because the timestamps in a transaction history are restricted to the current date. Since the type `T_detContInstant` defines an operation that returns the elapsed time (which is a time span) between two instants, the type `T_detDiscSpan` (and its supertype) is also highlighted. Finally, the types `T_gregDate` and `T_calendar` are highlighted since they represent human-readable timestamps in the branching history.

The tailoring process described above is recursive in that the toolkit allows a user to specify a directory from which an already tailored type hierarchy can be read (using the `Open Directory` button). For example, the type hierarchy shown in Figure 2.17 consists of the highlighted types of Figure 2.16. The user can now select the desired types from the tailored type hierarchy. The types highlighted in red in Figure 2.17 are as a result of selecting the `T_transactionHistory` type.

2.4.2 Clinical Data Management

This section describes a real-world example from clinical data management that illustrates the four design dimensions and the relationships between them which were discussed in Section 2.3.

During the course of a patient's illness, different blood tests are administered. It is usually the case that multiple blood tests of the patient are carried out on the same day. Suppose the patient was suspected of having an infection of the blood, and therefore had two different blood tests on 15 January 1995. These were the diagnostic hematology and microbiology blood tests. As a result of a very raised white cell count the patient was given a course of antibiotics while the results of the tests were awaited. A repeat hematology test was ordered on 20 February 1995. Suppose each blood test is represented by an object of the type `T_bloodTest`. The valid history of the patient's blood tests can then be represented in the object database as an object of type `T_validHistory`. This object is called bloodTestHistory. To record the hematology and microbiology blood tests, the objects mi-

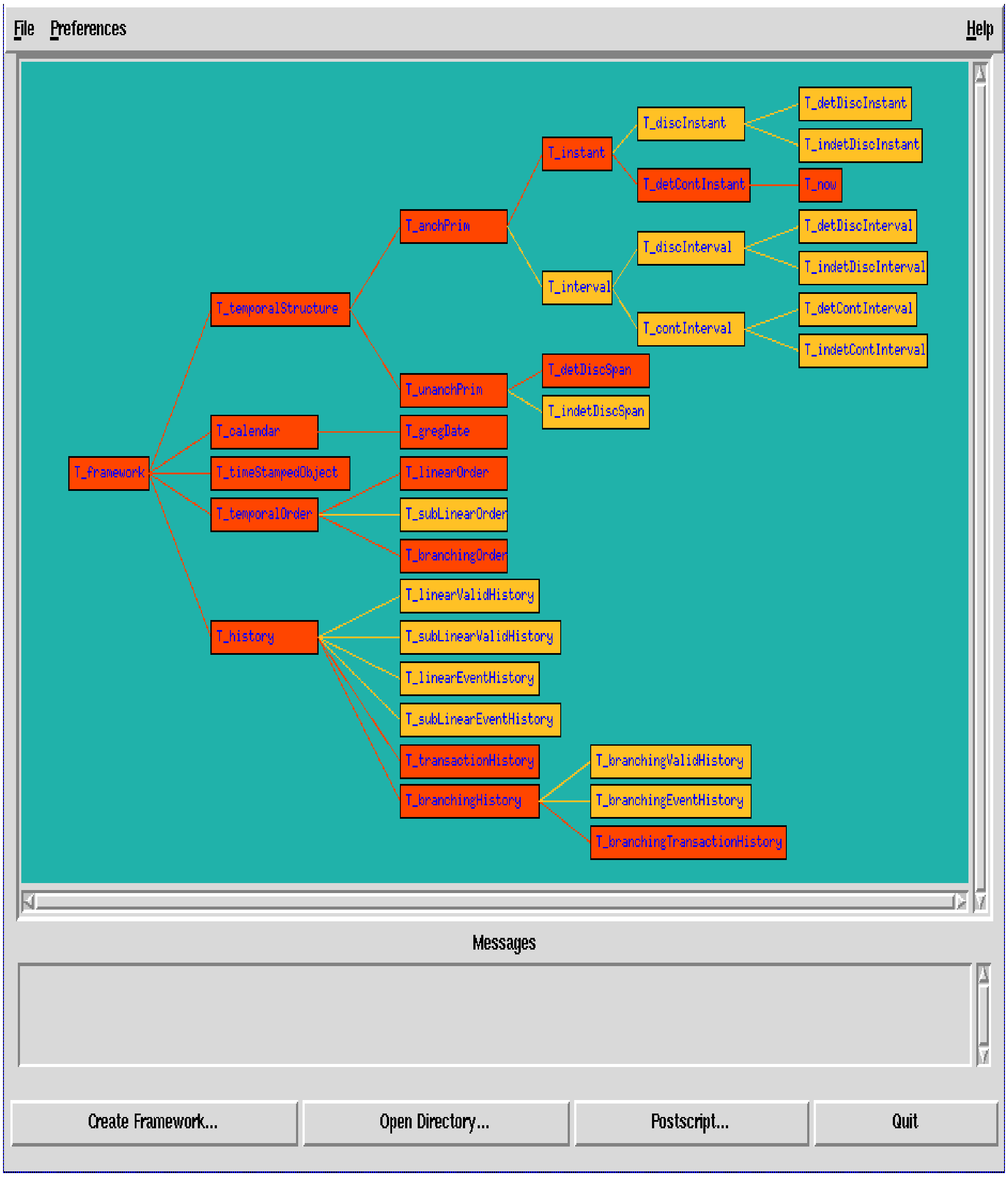


Figure 2.16: Tailoring the Temporal Framework

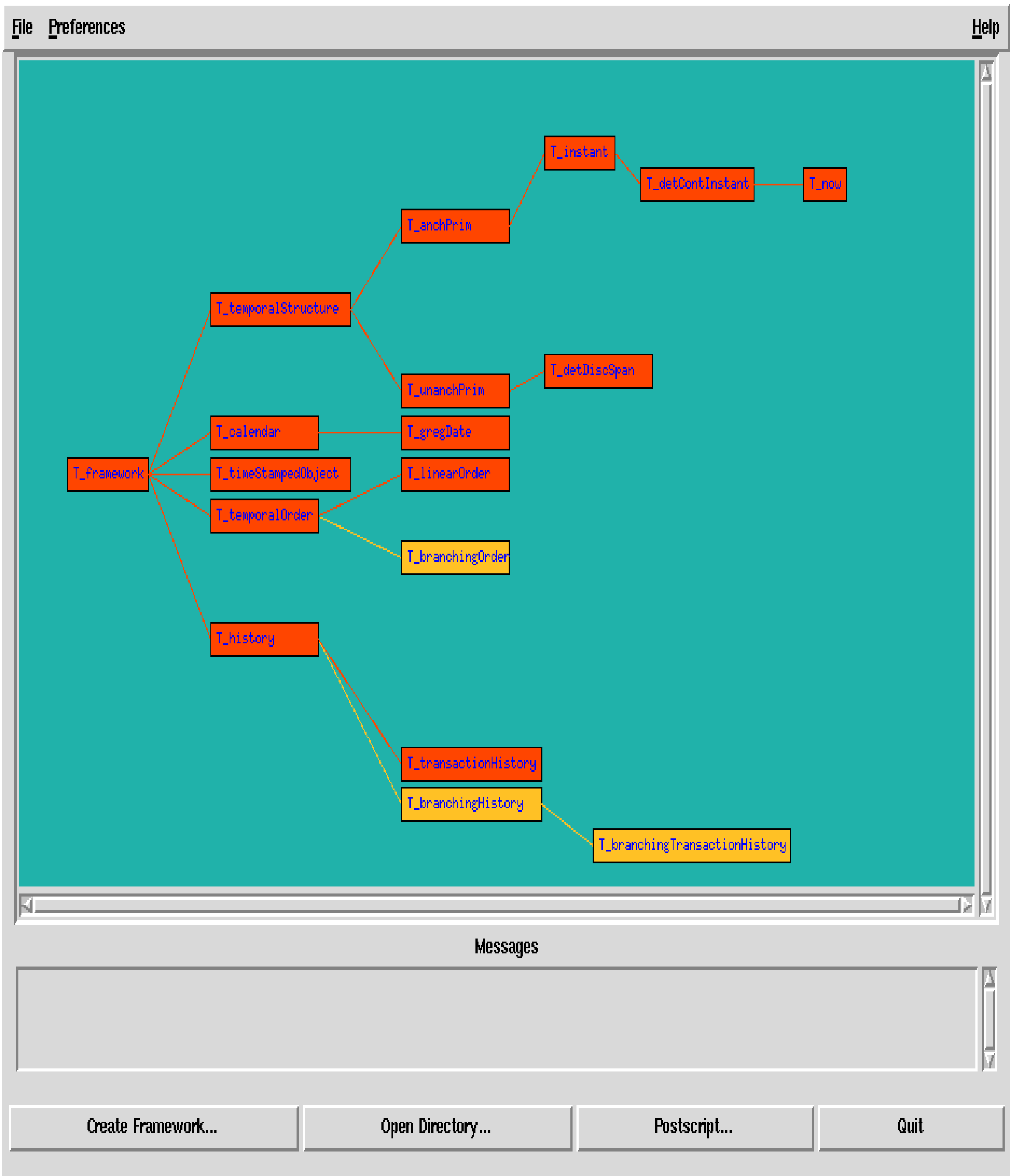


Figure 2.17: Recursively Tailoring the Temporal Framework

crobiology, hematology1, and hematology2 with type `T_bloodTest` are first created and then entered into the object database using the following property applications:

$$\begin{aligned}
\textsf{bloodTestHistory}.P_insert(\textsf{microbiology}, 15\ January\ 1995)\\
\textsf{bloodTestHistory}.P_insert(\textsf{hematology1}, 15\ January\ 1995)\\
\textsf{bloodTestHistory}.P_insert(\textsf{hematology2}, 20\ February\ 1995)
\end{aligned}$$

If subsequently there is a need to determine which blood tests the patient took in January 1995, this would be accomplished by the following property application:

$$\textsf{bloodTestHistory}.P\_getObjects([1\ January\ 1995,\ 31\ January\ 1995])$$

This would return a collection of timestamped objects of `T_bloodTest` representing all the blood tests the patient took in January 1995. These objects would be the (timestamped) hematology1 and the (timestamped) microbiology.

Figure 2.18 shows the different temporal features that are needed to keep track of a patient's blood tests over the course of a particular illness. The figure also illustrates the relationships between the different design dimensions of the temporal framework.

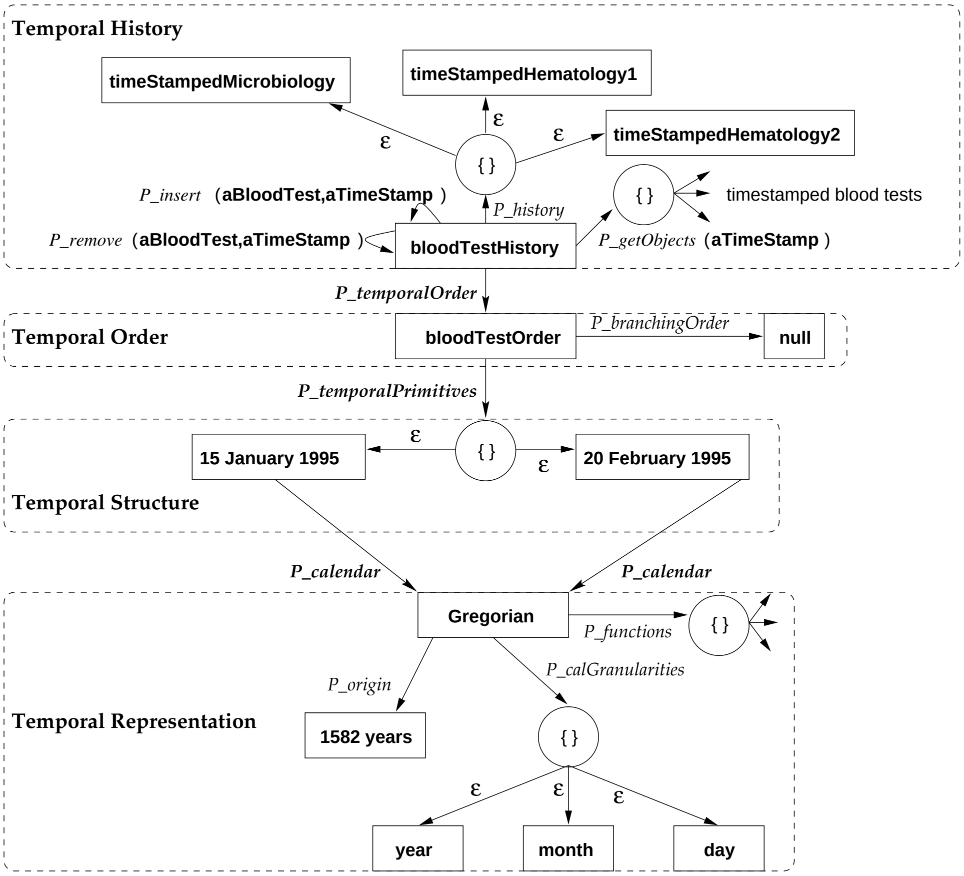


Figure 2.18: A Patient's Blood Test History

The patient has a blood test history represented by the object bloodTestHistory. The *P_history* property when applied to bloodTestHistory results in a collection object whose members are the timestamped objects timeStampedMicrobiology, timeStampedHematology1, and timeStampedHematology2. The *P_insert* property updates the blood test history (bloodTestHistory) by inserting an object of type `T_bloodTest` at a given anchored temporal primitive. Similarly, the property *P_remove* updates the bloodTestHistory by removing an object of type `T_bloodTest` at a given anchored temporal primitive. The *P_getObjects* property returns a collection of timestamped blood test objects when given an anchored temporal primitive.

Applying the property *P_temporalOrder* to bloodTestHistory results in the object bloodTestOrder which represents the temporal order on different blood tests in bloodTestHistory. bloodTestOrder has a certain temporal structure which is obtained by applying the *P_temporalPrimitives* property. Finally, the primitives in the temporal structure are represented using the Gregorian calendar, Gregorian and the calendric granularities year, month, and day.

Consider now the various temporal features required to represent the different blood tests taken by a patient. Anchored, discrete, and determinate temporal primitives are required to model the dates on which the patient takes different blood tests. These dates are represented using the Gregorian calendar. Since the blood tests take place on specific days, the temporal primitives during which the patient took blood tests form a total order. Lastly, a valid time history is used to keep track of the different times the blood tests were carried out. To support these temporal features, the toolkit described in Section 2.4.1 can be used to reconfigure the temporal framework with the appropriate types and properties. This is shown in Figure 2.19.

2.4.3 Time Series Management

The management of time series is important in many application areas such as finance, banking, and economic research. One of the main features of time series management is extensive calendar support [DDS94, LEW96]. Calendars map time points to their corresponding data and provide a platform for granularity conversions and temporal queries. Therefore, the temporal requirements of a time series management system include elaborate calendric functionality (which allows the definition of multiple calendars and granularities) and variable temporal structure (which supports both anchored and unanchored temporal primitives, and the different operations on them).

Figure 2.20 shows how the temporal requirements of a time series management system can be modeled using the types and properties of the temporal framework. It can be noted from the figure that only the temporal structure and temporal representation design dimensions are used to represent the temporal needs of a time series. This demonstrates that it is not necessary for an application requiring temporal features to have all four

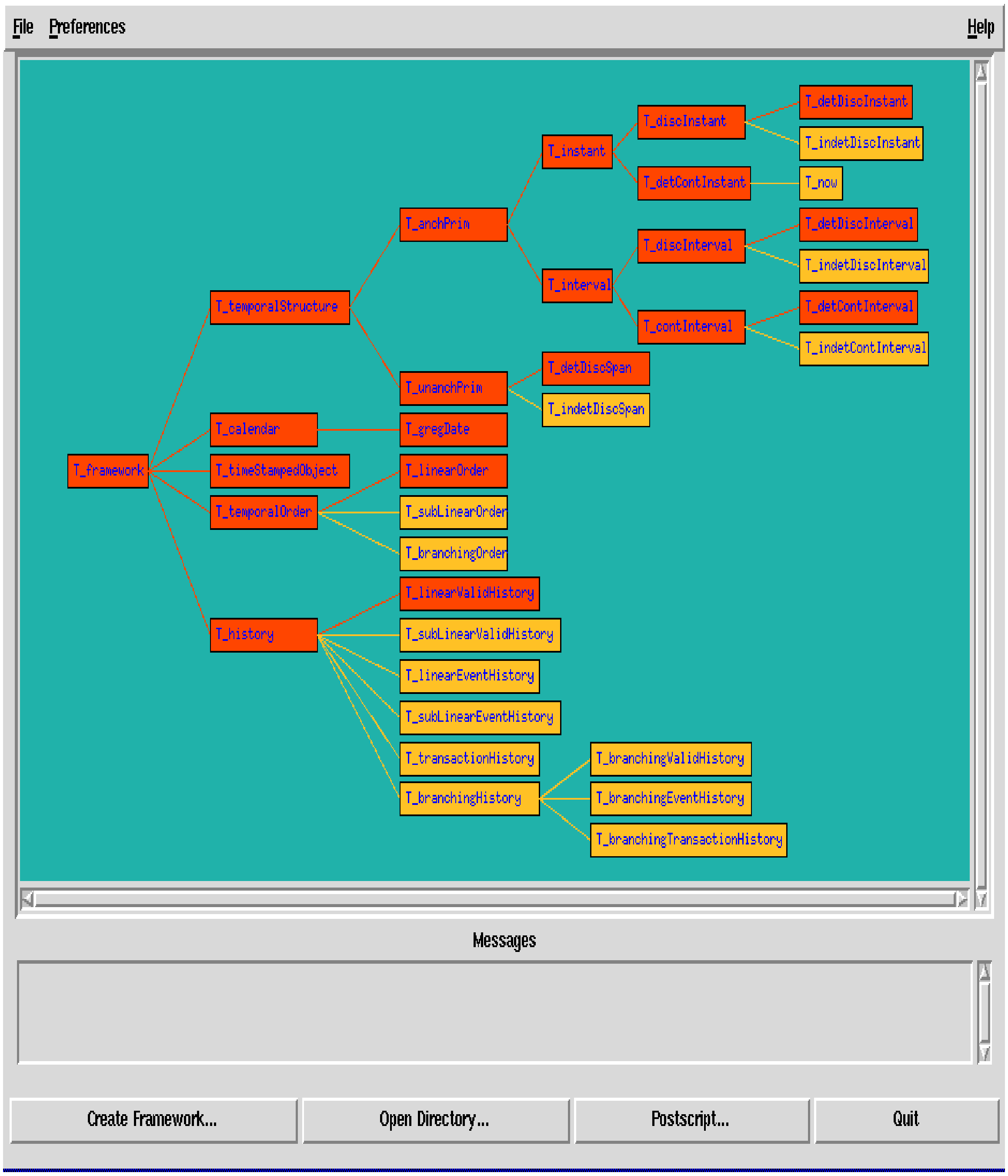


Figure 2.19: The Temporal Framework Inheritance Hierarchy for the Clinical Application

design dimensions in order to be accommodated in the framework. One or more of the design dimensions specified in Section 2.3.1 can be used as long as the design criteria shown in Figure 2.12 hold.

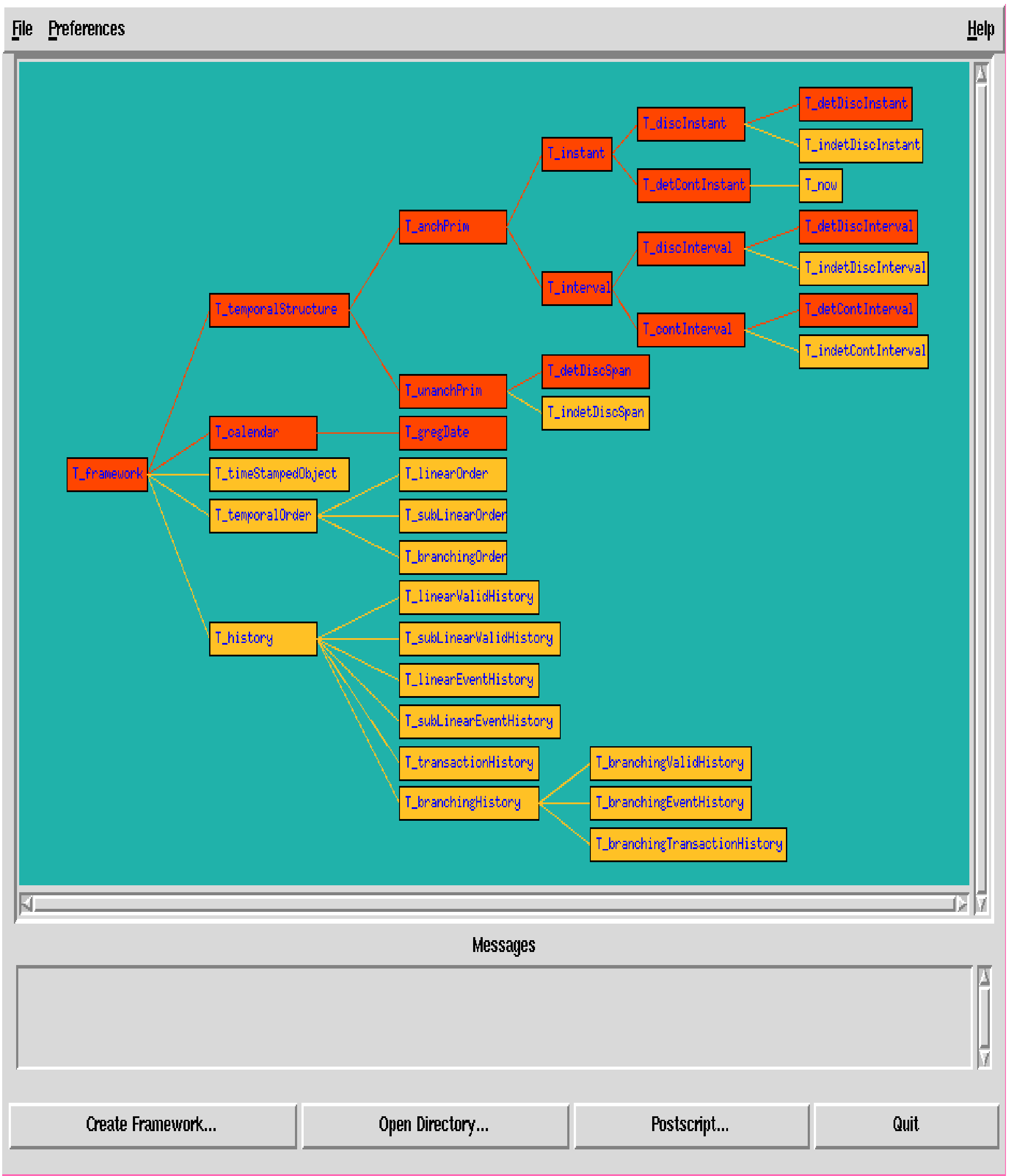


Figure 2.20: The Temporal Framework Inheritance Hierarchy for Time Series Management

2.4.4 TOODM - A Temporal Object-Oriented Data Model

This section illustrates how the temporal framework can accommodate the temporal features of different temporal object models. Rose & Segev's temporal object-oriented data model (TOODM) [RS91] is chosen as a representative model since it uses object types and inheritance to model temporality. An overview of the temporal features of TOODM is first given followed by a description of how these features can be derived using the types and properties of the temporal framework. There is no doubt that TOODM has more functionality to offer in addition to temporality, but capturing the additional non-temporal functions is beyond the scope of this framework and this thesis.

2.4.4.1 Overview of Temporal Features

TOODM was designed by extending an object-oriented entity-relationship data model to incorporate temporal structures and constraints. The functionality of TOODM includes: specification and enforcement of temporal constraints; support for past, present, and future time; support for different type and instance histories; and allowance for retro/proactive updates. The type hierarchy of the TOODM system used to model temporality is given in Figure 2.21. The boxes with a dashed border represent types that have been introduced to model time, while the rest of the boxes represent basic types.

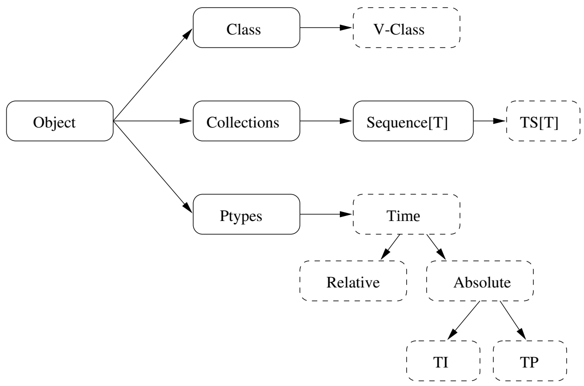


Figure 2.21: System Defined Temporal Types in TOODM

The `Object` type is the root of the type tree. The type `V-Class` is used to represent user-defined versionable classes. More specifically, if the instance variables, messages/methods, or constraints of a type are allowed to change (maintain histories), the type must be defined as a subtype of `V-Class`.

The `Ptypes` type models primitive types and is used to represent objects which do not have any instance variables. `Ptypes` usually serve as domains for the instance variables of other objects. The `Time` primitive type is used to represent temporal primitives. The `TP` type represents time points, while the `TI` type represents time intervals. Time points can have different calendar granularities, namely *Year*, *Month*, *Day*, *Week*, *Hour*, *Minute*, and

Structure			Representation	Order	History
Primitives	Domain	Determinacy			
Anchored	Continuous	Determinate	Gregorian Calendar	Total Linear	Valid
Unanchored					Transaction
					Event

Table 2.1: Temporal Design Dimension Features of TOODM

Second.

The `TS[T]` type represents a time sequence which is a collection of objects ordered on time. `TS[T]` is a parametric type with the type `T` representing a user or system defined type upon which a time sequence is being defined. For every time-varying attribute in a (versionable) class, a corresponding subclass (of `TS[T]`) is defined to represent the time sequence (history) of that attribute. For example, if the salary history of an employee is to be maintained, a subclass (e.g., `TS[Salary]`) of `TS[T]` has to be defined so that the salary instance variable in the employee class (which is defined as a subclass of `V-Class`) can refer to it to obtain the salary history of a particular employee.

2.4.4.2 Representing the Temporal Features of TOODM in the Temporal Framework

TOODM supports both anchored and unanchored primitives. These are modeled by the `Absolute` and `Relative` types shown in Figure 2.21. The anchored temporal primitives supported are time instants and time intervals. A continuous time domain is used to perceive the temporal primitives. Finally, the temporal primitives are determinate.

Time points and time intervals are represented by using the Gregorian calendar with granularities *Year*, *Month*, *Day*, *Week*, *Hour*, *Minute*, and *Second*. Translations between granularities in operations are provided, with the default being to convert to the coarser granularity. A (presumably total) linear order of time is used to order the primitives in a temporal sequence. TOODM combines time with facts to model different temporal histories, namely, valid, transaction, and event time histories. Table 2.1 summarizes the temporal features (design space) of TOODM according to the design dimensions for temporal models that were described in Section 2.3.1.

Figure 2.22 shows the type system instance of our temporal framework that corresponds to the TOODM time types shown in Figure 2.21 and described in Table 2.1.

The `Time` primitive type is represented using the `T_temporalStructure` type. The `TP` and `TI` types are represented using the `T_instant` and `T_interval` types, respectively. Similarly, the `Relative` type is represented using the `T_unanchPrim` type. Since TOODM supports continuous and determinate temporal primitives, the (concrete) types `T_detContInstant`, `T_detContInterval`, and `T_detContSpan` are used to model continuous and determinate instants, intervals, and spans, respectively.

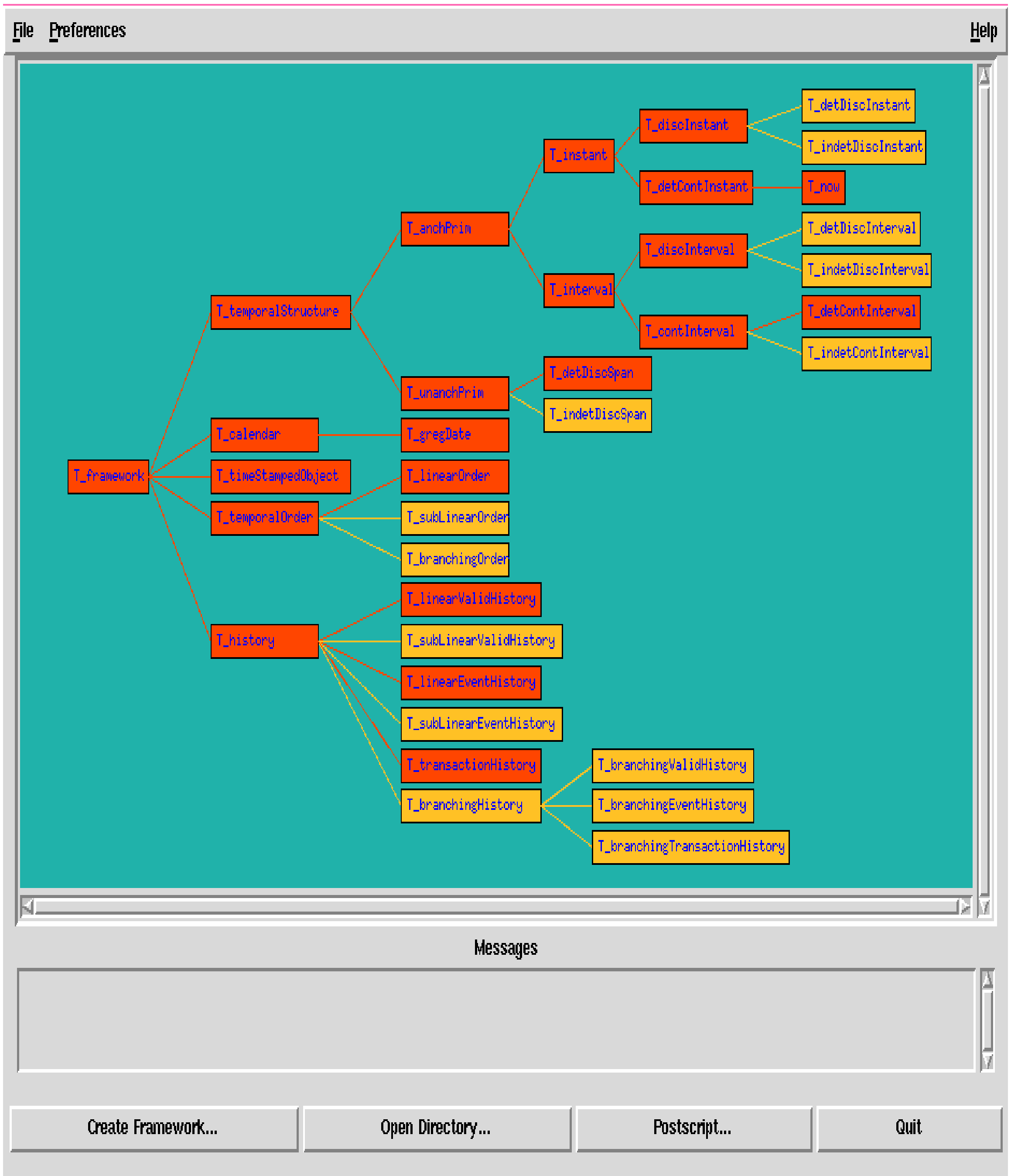


Figure 2.22: The Temporal Framework Inheritance Hierarchy for TOODM

The Gregorian calendar and its different calendric granularities are modeled using the `T_calendar` type. Time points and time intervals are ordered using the `T_linearOrder` type. Time sequences represented by the `TS[T]` type are modeled by the history types in the temporal framework. More specifically, valid time (vt), record time (rt), and event time (et) are modeled using the `T_validHistory`, `T_transactionHistory`, and `T_eventHistory` types.

TOODM models valid, transaction and event histories all together in one structure as shown by the `TS[Salary]` type in the previous section. The temporal framework, however, provides different types to model valid, transaction, and event histories to allow their respective semantics to be modeled. Moreover, it uses properties to access the various components of histories. For example, to represent the valid history of an employee's salary an object of type `T_validHistory` is first created. The *P_insert* property then inserts objects of type `T_integer` (representing salary values) and objects of type `T_interval` (representing time intervals) into the salary valid history object. The transaction and event time histories of the salary are similarly represented, except in these histories the *P_insert* property inserts timestamps which are time instants (i.e., objects of type `T_instant`).

2.5 Comparison of Temporal Object Models

In this section the temporal framework is used to compare and analyze the temporal object models that have appeared in recent literature. In particular, the [RS91, SC91, KS92, PM92, CITB92, BFG97] temporal object models are considered. The work of Wuu & Dayal [WD92] and Cheng & Gadia [CG93] (which follows a similar methodology as [WD92]) are not considered since they do not provide concrete notions of time in their models. In [WD92, DW92], variables and quantifiers are used to range over time. Abstract notions of time and abstract time types are used to facilitate the modeling of various notions of time. However, it is not clear how these abstract types fit in the primitive type lattice. Neither are any behaviors defined on these abstract types. Essentially, the user (or database designer) is burdened with the responsibility of defining most of the temporal model which includes specifying the temporal schema and specifying the queries.

Object models which support versioning using time [KGBW90, WLH90, SRH90, Sci94] usually follow a structural embedding of temporality within type definitions. Thus, the notion of temporal objects is lost since the model knows nothing about temporality. Moreover, most temporal version models use the `Date` function call which is provided by the system. For example, though the EXTRA-V version model [Sci94] supports "valid" and "transaction" time, it does so by timestamping attributes using system provided dates. This is limited in scope as no semantics of the various notions of time are provided. Since these models are not "temporal object models" in the strict sense of the term, they are not included in this study.

2.5.1 Overview of Temporal Object Models

Rose and Segev [RS91, RS93a, RS93b] extend an object-oriented based entity-relationship data model to incorporate time. Their work is described in Section 2.4.4.

OSAM*/T [SC91] is a temporal object-oriented knowledge model which provides a conceptual basis for representing behavioral and structural properties of objects. Behavioral properties are represented in terms of methods and knowledge rules, while structural properties are represented in terms of structured associations with other objects. The emphasis of the model is on temporal knowledge rules which are represented as conceptual objects that can evolve (through updates) over time. Histories of temporal knowledge rules and other object instances are captured by object instance time-stamping. OSAM*/T uses the discrete time domain to perceive time, in which time is isomorphic to natural numbers and is represented as a time sequence. Valid time is used to record the changes of object instances as they evolve in time. Object histories are recorded by time stamping object instances using time intervals.

Käfer and Schöning propose a temporal object-oriented data model called TMAD [KS92], to deal with history management of complex objects. They extend a general object-oriented data model (MAD), which allows for overlapping complex objects, by a temporal dimension. Objects of TMAD are called *molecule histories*, and a temporal query returns a set of molecule histories. A molecule history describes the evolution of an object in time. The lifespan of this evolution is called the *validity interval.* A molecule history consists of all the states the molecule has had in this interval. Each state is called a *time slice*. A temporal complex object has a history which is comprised of time slices. To retrieve and manipulate temporal complex-objects TMAD defines a query language with specific temporal constructs. In order to preserve the previous values of data, TMAD defines a new update operation which inserts copies of the latest state of all modified objects into the database without deletion of the old states. Each state is marked with a validity interval which has to be specified by the user. However, the transaction time is automatically recorded by the system.

Chu et. al., propose a temporal evolutionary object-oriented data model (TEDM) for medical image data in [CITB92]. Images such as X-rays, CT scans, etc. and the image features related to a patient's body structure are represented as objects in the data model. TEDM enhances the traditional object-oriented constructs with a new set of constructs to describe the evolutionary behavior of objects. Evolutionary object constructs consist of *Evolution* which models the characteristics of an object as it evolves over time; *Fusion* which models the cases when an object fuses with different objects to form a new object; and *Fission* which takes care of situations in which an object splits into two or more independent objects. Temporal relation object constructs include constructs that represent the temporal relationships between an object and its supertype, and object constructs that show the temporal relationships between the life spans of peer objects at the same level in a type hierarchy. Objects in TEDM can be either versionable or non-versionable. Versionable objects are simply collections of non-versionable objects associated with linearly ordered time intervals. TEDM adopts different time dimensions to time stamp objects. Valid time is represented using a time interval and denotes the duration in which the object was valid, record time is when the object is recorded in the database, and event time is when the event about an object actually occurs.

Pissinou and Makki [PM92] extend an object-oriented data model, called the 3 Dimensional Information Space (3DIS), to incorporate the semantics of time and thereby support temporal data and the temporal evolution of data. All entities in the Temporal 3DIS (T-3DIS) databases are treated as objects at various discrete time intervals. The T-3DIS model extends the 3DIS model to include temporal information on objects, mappings, and types. Each object is associated with a *valid time temporal version* $\{[t_a, t_b], \{versionnumbers\}\}$ which gives the *lifespan* of an object for a given version or a sequence of versions. The interval $[t_a, t_b]$ denotes the valid time of the temporal version. The version number(s) together with the time interval model the evolution of an object. Temporal relationships between objects are modeled by (domain-object($[t_i, t_j], v$), (mapping-object($[t_k, t_l], v$), (domain-object($[t_m, t_n], v$)), where v indicates a version number. This helps in determining the lifespan of a mapping (interval $[t_k, t_l]$), which then helps in determining temporal inter-object relationships. A set of temporal operations is defined on objects in T-3DIS that allows viewing, insertion, deletion, and modification of temporal objects.

Bertino et. al., present T_Chimera, a temporal extension to the Chimera object-oriented data model [BFG97]. The main objectives of T_Chimera are to represent on a formal basis several issues that arise from the introduction of time in an object-oriented database. These include both temporal and object-oriented issues. T_Chimera extends the set of Chimera types with the notion of a *temporal type*. Temporal types uniformly represent variables for which the history of changes over time is recorded and variables for which only the current value is kept. For each Chimera type T, a corresponding temporal type (denoted as $temporal(T)$) is defined. The domain of time is assumed to be isomorphic to natural numbers and is modeled using the `time` type. Instances of the $temporal(T)$ type are partial functions from instances of the `time` type to instances of the type T. An interval denoted as $[t_1, t_2]$ includes all time instants between t_1 and t_2, including t_1 and t_2. Union, intersection, and inclusion operations are defined on time intervals and have the usual semantics of set operations. T_Chimera also provides notions of consistency between an object with respect to its class taking into account that both the object state and the classes the object belongs to (objects can migrate during their lifespan from one class to another) vary over time. Formal definitions of object equality, referential integrity, and inheritance in the context of both temporal and non-temporal objects are also given.

2.5.2 Classification of Temporal Object Models

The temporal features of the different temporal object models discussed in Section 2.5.1 are summarized in Tables 2.1 and 2.2. The criteria in comparing different temporal object models is based on the design dimensions identified in Section 2.3.1. It is true that the models may have other (salient) temporal differences, but the concern in this work is comparing their temporal features in terms of the framework defined in Section 2.3.

Similar to the methodology used in Section 2.3, object-oriented techniques are used this time to classify temporal object models according to each design dimension. This will give an indication of how temporal object models range in their provision of different temporal features of a design dimension – from the most powerful model (i.e., the one having the most number of temporal features) to the least powerful model (i.e., the one having the least number of temporal features).

Model	Structure			Representation	Order	History
	Primitives	Domain	Determinacy			
OSAM*/T	Anchored	Discrete	Determinate	N/A	Linear	Valid
TMAD	Anchored	Discrete	Determinate	Gregorian Calendar	Linear	Valid
						Transaction
TEDM	Anchored	Discrete	Determinate	N/A	Linear	Valid
						Transaction
						Event
T-3DIS	Anchored	Discrete	Determinate	Gregorian Calendar	Partial	Valid
T-Chimera	Anchored	Discrete	Determinate	N/A	Linear	Valid

Table 2.2: Design Dimension Features of different Temporal Object Models

Temporal Structure. It can be noticed from Tables 2.1 and 2.2 that most of the models support a very simple temporal structure, consisting of anchored primitives which are discrete and determinate. In fact, all models in Table 2.2 support the *same* temporal structure, which consists of discrete and determinate anchored temporal primitives. These primitives can be accommodated in the temporal framework by the `T_anchPrim`, `T_instant`, `T_detDiscinstant`, `T_interval`, and `T_detDiscInterval` types, and their respective properties. The temporal structure of TOODM is slightly enhanced with the presence of unanchored primitives. TOODM is also the only model that supports the continuous temporal domain.

Figure 2.23 shows how the type inheritance hierarchy is used to classify temporal object models according to their temporal structures. The temporal structures of OSAM*/T, TMAD, TEDM, T-3DIS, and T-Chimera can be modeled by a single type – that representing temporal primitives that are anchored, discrete, and determinate. This means that any of these models can be used to provide temporal support for applications that need a temporal structure comprised of anchored temporal primitives which are discrete and determinate. Similarly, the temporal structure of TOODM can be modeled by a type which represents anchored and unanchored temporal primitives that are continuous and determinate. This implies that TOODM is the only model that can support applications requiring a continuous time domain, and unanchored temporal primitives.

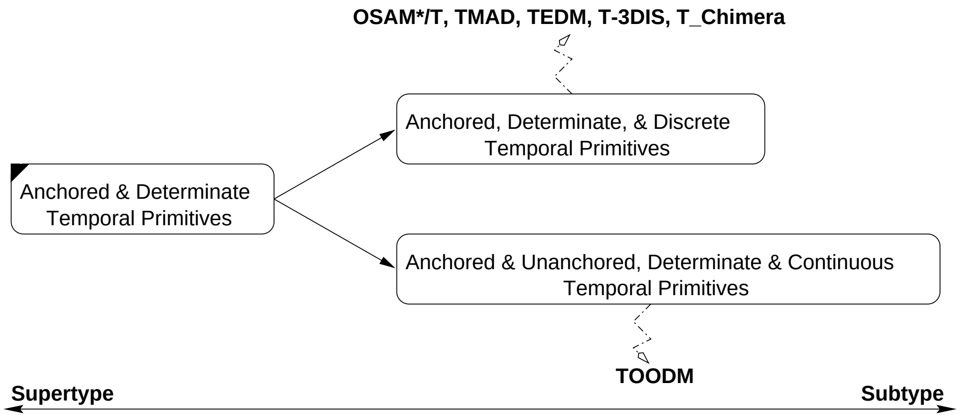


Figure 2.23: Classification of Temporal Object Models according to their Temporal Structures

Temporal Representation. Temporal primitives in the OSAM*/T [SC91], TEDM [CITB92], and T-Chimera [BFG97] models are simply represented using natural numbers. The models do not provide any additional representational scheme which supports calendars and different granularities. The granularity of the temporal primitives is dependent on the application using the model. When a calendric representational scheme is provided for the temporal primitives, it is comprised of a single underlying calendar, which is usually Gregorian. This is the case in the TOODM [RS91], TMAD[KS92], and T-3DIS [PM92] models.

Temporal Order. All models shown in Tables 2.1 and 2.2, except T-3DIS, support a linear temporal order. The T-3DIS model supports a sub-linear temporal order. These temporal orders are accommodated in the temporal framework using the `T_subLinearOrder` and `T_linearOrder` types. Figure 2.24 shows how the models can be classified in an inheritance type hierarchy according to their temporal orders. The type that models a partial linear order of time sits at the root of the hierarchy and represents the T-3DIS model. Since a total linear order is also a partial order, the models supporting total linear orders can be represented by a direct subtype of the root type.

Temporal History. Tables 2.1 and 2.2 show how the temporal object models range in their support for the different types of temporal histories. Figure 2.25 shows how the models can be classified according to the temporal histories they support using a type inheritance hierarchy. The root type in Figure 2.25 represents the models which only support valid time histories. These are the OSAM*/T, T-3DIS, and T-

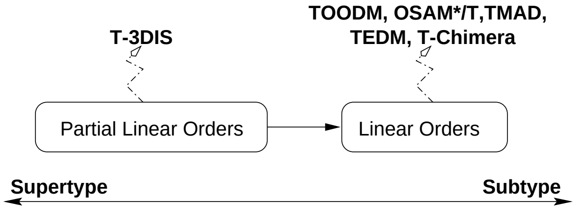


Figure 2.24: Classification of Temporal Object Models according to their Temporal Orders

Chimera models. A direct subtype of the root type inherits the valid time history and provides transaction time history as well. This type represents the TMAD model. Similarly, the rest of the subtypes inherit different histories from their supertypes and add new histories to their type as shown in Figure 2.25. From Figure 2.25, it can be seen that applications requiring only valid time histories can be supported by all models; applications requiring valid and transaction time can be supported by the TMAD, TEDM, and TOODM models; and applications requiring valid, transaction, and event time can be supported by the TEDM and TOODM models.

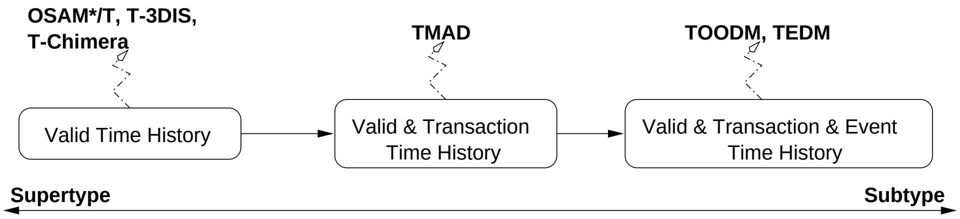


Figure 2.25: Classification of Temporal Object Models according to their Temporal Histories

Overall Classification. Having classified the temporal object models according to the individual design dimensions, the models can now be treated as points in the design space and the object-oriented inheritance hierarchy can be used to compare the models on all the temporal features of the design dimensions that they support. Figure 2.26 gives an inheritance hierarchy in which types are used to represent the different models, and the temporal features supported by the models are used as the criterion for inheritance.

The abstract type at the root of the hierarchy represents the least powerful temporal object model which supports a temporal structure comprised of anchored primitives which are discrete and determinate, no temporal representational scheme, a partial linear order, and a valid time history. This type has two immediate subtypes. The first subtype represents the OSAM*/T and the T-Chimera models. It inherits all the features of the root type and refines its partial linear order to a total linear order. Similarly, the second subtype represents the T-3DIS model, inherits all the features of the root type, and adds a representational scheme which supports the Gregorian calendar. The type representing OSAM*/T and T-Chimera also has two subtypes. The first subtype represents the TEDM model and has all the features of its supertype with the additional features of transaction and event time histories. The second subtype (which is also a subtype of the type representing T-3DIS from which it inherits the representational scheme) represents the TMAD model. This type has the additional feature of the transaction time history. A direct subtype of the types representing TEDM and TMAD represents the TOODM model. The type representing TOODM inherits the representational scheme from the type representing TMAD and the event time history from the type representing TEDM. It also adds unanchored primitives and the continuous time domain to its temporal structure. From Figure 2.26 it can reasonably be concluded that OSAM*/T and T-Chimera are the two least powerful temporal object models since they provide the least number of temporal features. The TOODM model is the most powerful since it provides the most number of temporal features.

The comparison of different temporal object models made in this section shows that there is significant similarity in the temporal features supported by the models. In fact, the temporal features supported by OSAM*/T and T-Chimera are identical. The temporal features of TEDM are identical to those of OSAM*/T and T-Chimera in the temporal structure, temporal representation, and temporal order design dimensions. These commonalities substantiate the need for a temporal framework which combines the diverse features of time under a single infrastructure that allows design reuse.

It can also be noted that temporal object models have not really taken advantage of the richness of their underlying object model in supporting alternate features of a design dimension. They have assumed a set of fixed notions of time. From a range of different temporal features, a single temporal feature is supported in most of the design dimensions. As such, not much advantage has been gained over the temporal relational models in supporting applications that have different temporal needs. For example, engineering applications like CAD would benefit from a branching time model, while time series and financial applications require multiple calendars and granularities. In Chapter 3, a temporal object model is presented that aims to exploit object-oriented technology in supporting a wide range of applications with diverse temporal needs.

2.6 Implementation of the Temporal Framework

The temporal framework architecture described in Section 2.3 is general enough to be implemented on most object-oriented systems. In this section, a prototype implementation of the temporal framework in C++ on Sun Solaris is described. The implementation is constrained by the functionality of C++, and as such the implemented type hierarchy of the temporal framework is a little different from the hierarchy shown in Figure 2.14. In the following sub-sections, these differences will be highlighted for each design dimension. In

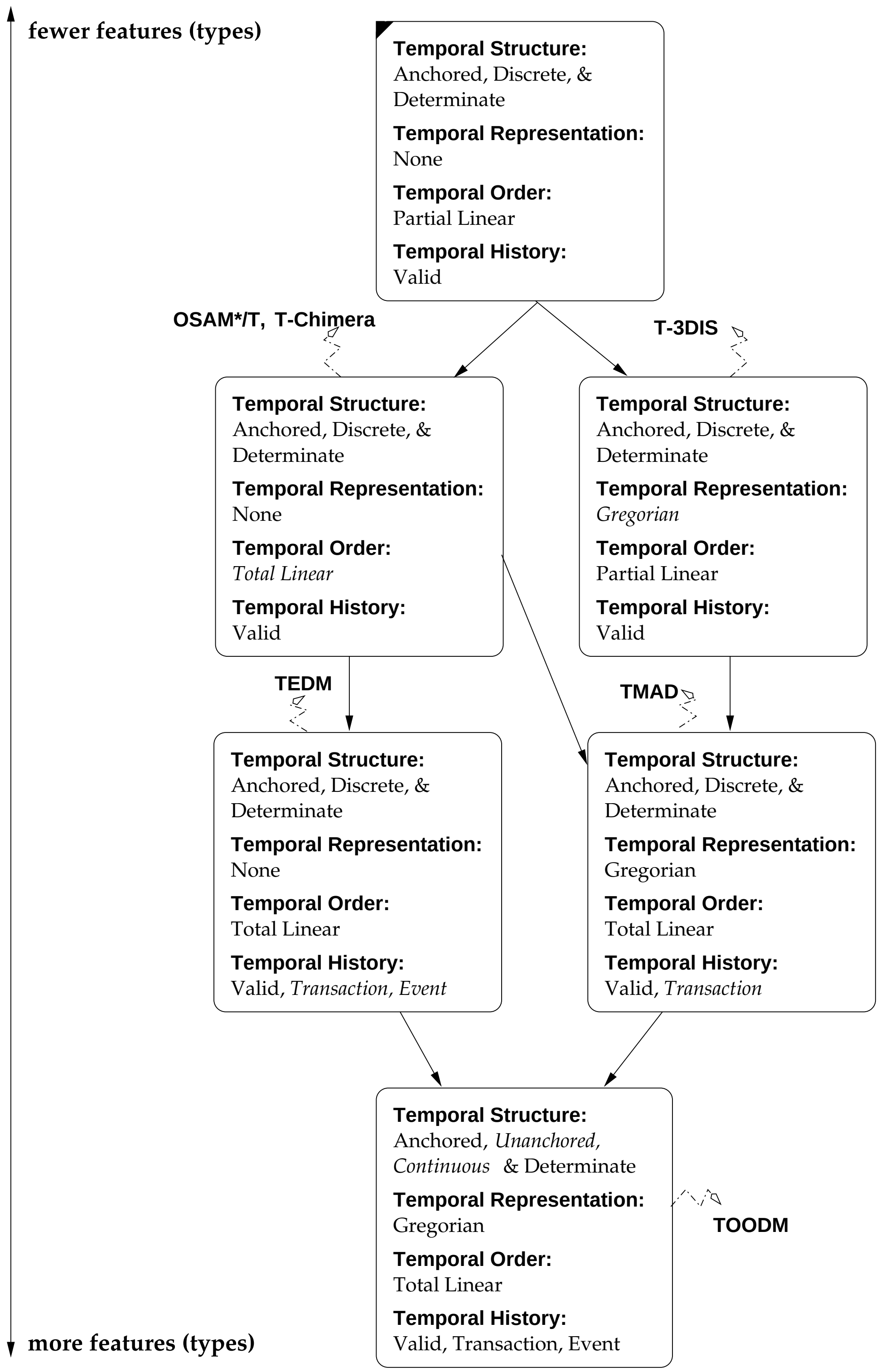


Figure 2.26: Overall Classification of Temporal Object Models

this section, the term "type" is used to refer to the term "class" in C++.

2.6.1 Implementation of Temporal Structure

The implemented type hierarchy for the temporal structure is given in Figure 2.27. This hierarchy is similar to the one shown in Figure 2.3. The additional types `T_discInstant`, `T_discInterval` and `T_contInterval` have been implemented to abstract out properties common to their respective subtypes. The concrete type `T_now` has been defined so that its instant represents the current time. Type `T_now` is used specifically in transaction time histories to timestamp objects with the current time.

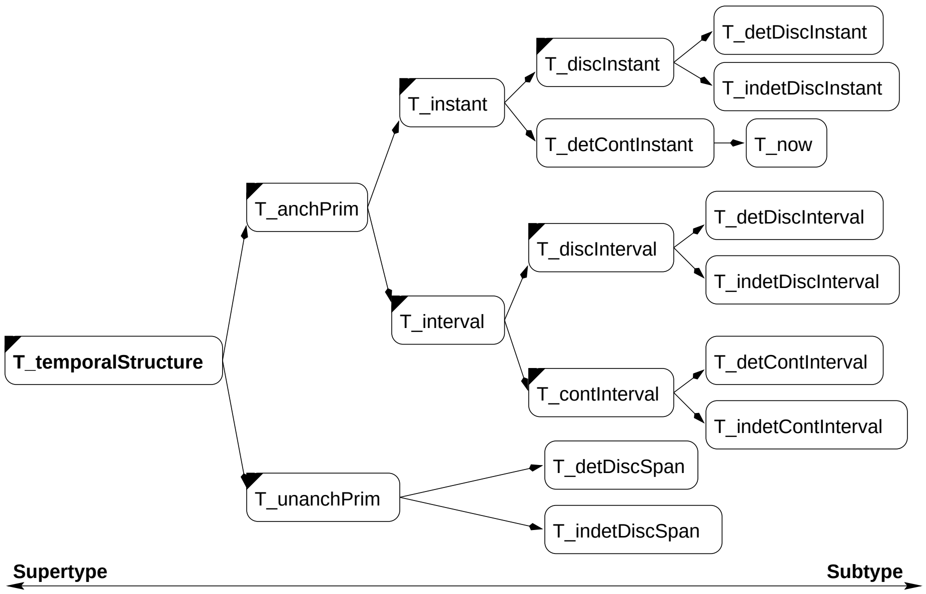


Figure 2.27: The Implementation Inheritance Hierarchy of a Temporal Structure

The type `T_detContInstant` is the most basic concrete type for anchored primitives. It represents a determinate continuous instant which is a point on the global axis of time. All operations pertaining to time instants are defined as behaviors in type `T_detContInstant`. The type also defines constructors that allow a determinate continuous instant to be represented as a gregorian date. Objects of type `T_detContInstant` are also used to construct determinate and indeterminate continuous intervals. These are objects of types `T_detContInterval` and `T_indetContInterval`, respectively. The lower and upper bounds of determinate and indeterminate continuous intervals are determinate continuous instants. The abstract type for continuous intervals, `T_contInterval` defines the different ordering operations on intervals. Two different semantics are associated with the ordering operations; "for sure" and "maybe." These have been implemented to better capture the semantics of the ordering operations when indeterminate continuous intervals are involved. For example, the "maybe" *before* operation when the argument and/or receiver is indeterminate returns `TRUE` if the intervals *overlap* or *meet*. In contrast, the "for sure" *before* operation when the argument and/or receiver is indeterminate returns `TRUE` only if the upper bound of the receiver interval is less than or equal to the lower bound of the argument interval. The set-theoretic operations have been implemented to restrict the argument and receiver intervals to be of the same type. As such, these operations have been implemented in the `T_detContInterval` and `T_indetContInterval` types.

2.6.2 Implementation of Temporal Representation

The implemented type hierarchy for the temporal representation is given in Figure 2.28. For illustration purposes, a simplified version of the Gregorian calendar having the calendric granularities *year*, *month*, and *day* has been implemented as a means of representing temporal primitives. The algorithms used in the implementation of the Gregorian calendar have been proposed by Dershowitz and Reingold [DR90]. Gregorian dates are implemented as *absolute* dates. An absolute date refers to the number of days elapsed since the Gregorian date Sunday, December 31, 1 BC. Thus the Gregorian date January 1, 1 AD is absolute date number 1. An operator, `int()`, has been implemented in `T_gregDate` to cast a calendric date to an integer (absolute date). This has made the implementation of ordering and comparison operations on temporal primitives much easier. Temporal primitives represented as Gregorian dates are simply cast to integers, and the ordering or comparison operations are then carried out on integers. Another operation implemented in `T_gregDate` computes the calendric date from the absolute date.

The semantics of other calendars can be implemented similarly as subtypes of `T_calendar`. The work of Dershowitz and Reingold [DR90] also has algorithms for other calendars, for example, Julian, Islamic, and Hebrew dates.

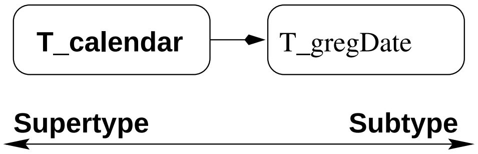


Figure 2.28: The Implementation Inheritance Hierarchy of a Temporal Representation

2.6.3 Implementation of Temporal Order

The implemented type hierarchy for temporal orders is given in Figure 2.29. The temporal order types are implemented as templated types where the type `T_Y` is one of the concrete anchored temporal primitive types. To maintain a sorted order of temporal primitives, the temporal order types make use of sorted templated collections in the Tools.h++ class library that comes with the Sun OS compiler. In particular, the pointer-based collection, RWPtrSortedVector⟨`T_T`⟩ is used. The items in the collection have an ordered relationship with each other and can be accessed by an index number. An insertion sort is used to maintain a sorted order in the collection, and duplicates are allowed. The operators ==

and < must be defined on type `T` in order to ensure the type has well-defined equality and less-than semantics, respectively. The == and < operators are used by operations in RWPtrSortedVector⟨`T_T`⟩ to insert and find objects in the collection.

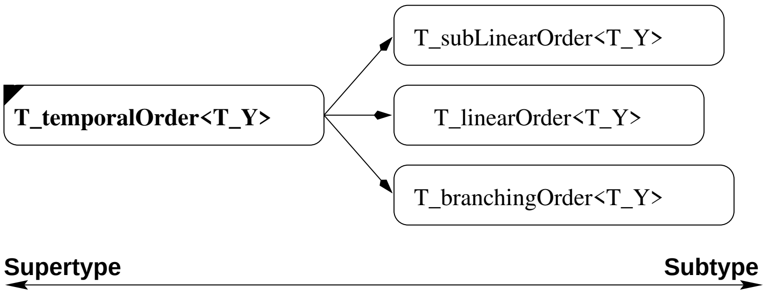


Figure 2.29: The Implementation Inheritance Hierarchy of Temporal Orders

The abstract type `T_temporalOrder`⟨`T_Y`⟩ defines virtual *insert* and *remove* operations which simply call their counterpart operations in RWPtrSortedVector⟨`T_T`⟩. The insertion sort that is implemented in RWPtrSortedVector⟨`T_T`⟩ falls short in maintaining a linear order when intervals are involved. This is because the only criteria used in the sort are equality and less-than semantics. If these criteria are not met, the interval is assumed to have greater-than semantics and inserted in the collection at an appropriate location. This is problematic since the inserted interval may overlap with one or more intervals in the collection. To get around this problem, a method has been implemented in `T_linearOrder`⟨`T_Y`⟩ which ensures that the intervals in the linear collection strictly follow or precede each other. The problem of overlapping intervals does not apply for sub-linear orders since objects of type `T_subLinear`⟨`T_Y`⟩ are partially ordered with respect to each other. Thus, `T_subLinear`⟨`T_Y`⟩ simply uses the *insert* and *remove* operations defined in `T_temporalOrder`⟨`T_Y`⟩.

A branching order is implemented as a triplet comprising of a starting point (which is a determinate continuous instant), a root (which is a linear order), and a collection of branches (which is an ordered collection of branching orders). The *insert* and *remove* operations are redefined in type `T_branchingOrder`⟨`T_Y`⟩. The *insert* operation inserts the temporal primitive in the root of the receiver after ensuring that the primitive is not before the starting point and does not overlap the starting points of any of the receiver branches. Another operation defined in `T_branchingOrder`⟨`T_Y`⟩, *insertBranch*, inserts a branch on the receiver at the point given by a determinate continuous instant. The branch is inserted after ensuring that the timestamps on the root of the receiver do not overlap the starting point of the branch to be inserted.

2.6.4 Implementation of Temporal History

The implemented type hierarchy for the temporal histories is given in Figure 2.30. This hierarchy is quite different from the one shown in Figure 2.11. Additional types have been introduced for valid and event time histories in order to capture the different ordering in these histories. For example, the type T_linearValidHistory⟨T_X, T_Y⟩ represents valid histories that are linearly ordered.

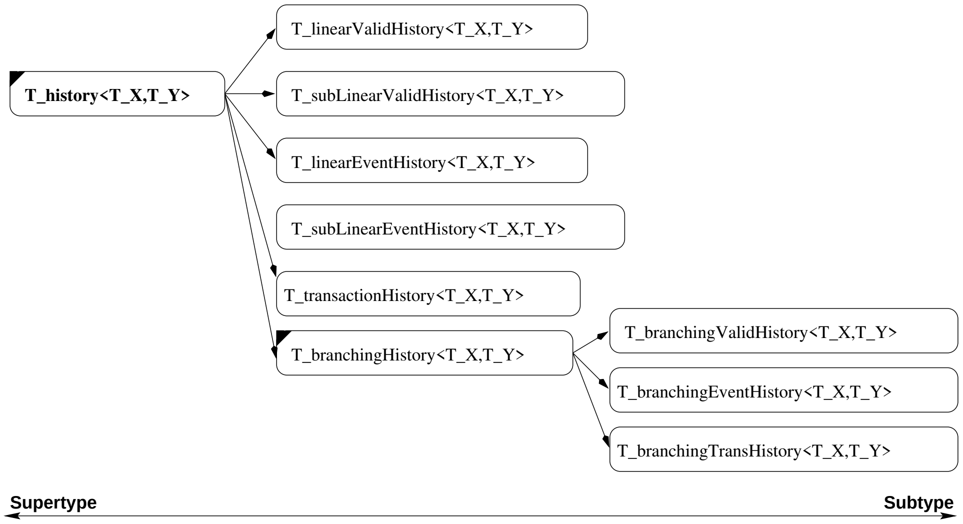


Figure 2.30: The Implementation Inheritance Hierarchy of Temporal Histories

Similar to the temporal order types (see Figure 2.29), the temporal history types are implemented as templated types with two argument types. The type T_X can represent any object, while the type T_Y is one of the concrete anchored temporal primitive types. In type T_transactionHistory, T_Y is restricted to be the type T_now. The description for ensuring linearity outlined in Section 2.6.3 also applies to temporal histories. A temporal history consists of a sorted collection of timestamped objects (of type T_timeStampedObject⟨T_X, T_Y⟩. A timestamped object basically defines operations that return the timestamped object (of type T_X) and the timestamp (of type T_Y). The implementation of the branching histories is similar to that of branching orders described in Section 2.6.3. The type T_branchingHistory⟨T_X, T_Y⟩ has been implemented to abstract out properties common to its subtypes.

Chapter 3

The TIGUKAT Temporal Object Model

This chapter presents the TIGUKAT temporal model[1] as an example instantiation of the temporal framework that incorporates the multiple notions of time given in Figure 2.14. The philosophy behind adding temporality to the TIGUKAT object model [Pet94] is to exploit the richness of the object model in order to accommodate multiple applications which have different type semantics requiring various notions of time [LGÖS97, GÖS97a]. In [LGÖS97], the temporal model is used to manage temporal relationships which is inherent in multimedia data such as video, while in [GÖS97a], the temporal model provides branching temporal histories that are needed to store and retrieve historical information that is commonly found in medical applications such as clinical trials, and in applications such as computer aided design and planning or version control. Consequently, the TIGUKAT temporal object model consists of an extensible set of primitive time types with a rich set of behaviors that implement the temporal framework, providing consistent semantics for the different temporal features which is necessary for their coexistence.

The rest of the chapter is organized as follows. Section 3.1 gives an overview of the TIGUKAT object model. In Sections 3.2-3.5, the temporal features of the TIGUKAT temporal object model are presented as instantiations of the design dimensions of the temporal framework that were identified in Chapter 2.

3.1 Overview of the TIGUKAT Object Model

The TIGUKAT object model [Pet94, ÖPS$^+$95] is purely *behavioral* with a *uniform* object semantics. The model is *behavioral* in the sense that all access and manipulation of objects is based on the application of behaviors to objects. The model is *uniform* in that every component of information, including its semantics, is modeled as a *first-class object* with

[1] Initial work on the model appeared in [GÖ93]. Subsequent enhancements of different aspects of the model presented in this chapter appear in [LGÖS97, GÖS97a].

well-defined behavior. Other typical object modeling features supported by TIGUKAT include strong object identity, abstract types, strong typing, complex objects, full encapsulation, multiple inheritance, and parametric types.

The primitive objects of the model include: *atomic entities* (reals, integers, strings, etc.); *types* for defining common features of objects; *behaviors* for specifying the semantics of operations that may be performed on objects; *functions* for specifying implementations of behaviors over types; *classes* for automatic classification of objects based on type[2]; and *collections* for supporting general heterogeneous groupings of objects. Figure 3.1 shows a simple type lattice that will be used to illustrate the concepts introduced in the rest of the thesis.

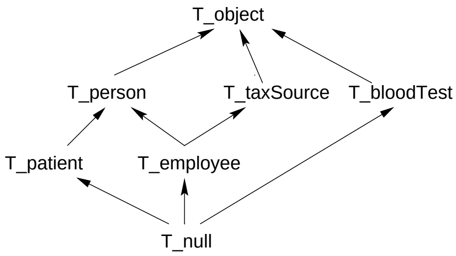


Figure 3.1: Simple type lattice.

In this thesis, a reference prefixed by "T_" refers to a type, "**C_**" to a class, "*B_*" to a behavior, and "T_X< T_Y >" to the type T_X parameterized by the type T_Y. For example, `T_person` refers to a type, **C_person** to its class, *B_age* to one of its behaviors and `T_collection`< `T_person` > to the type of collections of persons. A reference such as joe, without a prefix, denotes some other application specific reference. The type `T_null` in TIGUKAT binds the type lattice from the bottom (i.e., most defined type), while the `T_object` type binds it from the top (i.e., least defined type). `T_null` is introduced to provide, among other things, error handling and null semantics for the model.

The access and manipulation of an object's state occurs exclusively through the application of behaviors. We clearly separate the definition of a behavior from its possible implementations (functions). The benefit of this approach is that common behaviors over different types can have a different implementation in each of the types. This provides direct support for behavior *overloading* and *late binding* of functions (implementations) to behaviors.

The model separates the definition of object characteristics (a *type*) from the mechanism for maintaining instances of a particular type (a *class*). A *type* defines behaviors and encapsulates behavior implementations and state representation for objects created using that type as a template. The behaviors defined by a type describe the *interface* to the objects of that type. A *class* ties together the notions of *type* and *object instances*. Objects of a particular type cannot exist without an associated class and every class is uniquely

[2]Types and their extents are separate constructs in TIGUKAT.

associated with a single type. Object creation occurs only through classes using its associated type as a template for the creation. Thus, a fundamental notion of TIGUKAT is that *objects* imply *classes* which imply *types*.

3.2 Temporal Representation

In this section, the temporal representational scheme of calendars, described in Section 2.3.1.2, is used as a means to make the temporal primitives, described in Section 2.3.1.1, human readable and usable in the TIGUKAT temporal model. A calendar should be able to support multiple granularities since temporal information processed by an ODBMS is usually available in multiple granularities. Such information is prevalent in various sources. For example:

- *clinical data* – Physicians usually specify temporal clinical information for patients with varying granularities [CPP95, CPP96]. For example, "the patient suffered from abdominal pain for 2 hours and 20 minutes on June 15, 1996," "in 1990, the patient took a calcium antagonist for 3 months," "in October 1993, the patient had a second heart seizure."

- *real-time systems* – A process is usually composed of sub-processes that evolve according to times that have different granularities [CMR91]. For example, the temporal evolution of the basin in a hydroelectric plant depends on different sub-processes: the flow of water is measured daily; the opening and closing of radial gates is monitored every minute; and the electronic control has a granularity of microsecond.

- *geographic information systems* – Geographic information is usually specified according to a varying time scale [Flo91]. For example, vegetation fluctuates according to a seasonal cycle, while temperature varies daily.

- *office information systems* – temporal information is available in different time units of the Gregorian calendar [BP85, CR88, MPB92]. For example, employee wages are usually recorded in the time unit of hours while the history of sales are categorized according to months.

Clearly, in many applications, it is desirable to have multiple calendars that have different calendric granularities. As discussed in Section 2.3.1.2, a calendar is comprised of an origin, a set of calendric granularities, and a set of conversion functions. In the following subsections, the concepts of calendric granularity and conversion functions in the TIGUKAT temporal model are defined. The work in this section and Section 3.3 is based on a single calendar. Extensions of the work presented in these sections to incorporate multiple calendar support are given in Appendix A.

3.2.1 Calendric Granularities

In the TIGUKAT temporal model, a novel approach to the treatment of granularity in temporal data is provided. A granularity is modeled as a *unit unanchored temporal primitive* (*unit time span*). More specifically, a granularity is modeled as a special kind of time span that can be used as a unit of time (unanchored durations are discussed in more detail in Section 3.3.2). Granularity conversions are presented and discussed in terms of unanchored durations of time. This feature allows the consistent modeling and operation on unanchored temporal primitives that are comprised of different and mixed granularities. The work in this thesis should be seen as complementing other works on temporal granularity. It fills in the missing piece by allowing unanchored temporal primitives to be specified in different and mixed granularities, and facilitates the conversion of unanchored temporal primitives from one granularity to another.

The inherent problem in adequately supporting unanchored temporal primitives with different granularities in [CC87, WJL91, WJS93, WBBJ97, BP85, MPB92, MMCR92, Sno95b], is that a granularity is treated as an *anchored* partitioning of the time axis. Since unanchored temporal primitives are *independent* of anchored temporal primitives (i.e., their location on the time axis is unknown since they are not anchored at any particular point) problems arise in the conversion of unanchored temporal primitives from one granularity to another when a granularity is modeled as an anchored partitioning of the time axis. In converting (scaling) an unanchored temporal primitive from one granularity to another in TSQL2 [Sno95b], it is noted:

> "... the problem is that a granularity is an anchored partitioning, whereas an interval[3] is unanchored ... the consequence of the unanchored nature of intervals is that whenever an interval is scaled, an indeterminate interval will result, even when an interval is scaled from a finer to a coarser granularity" (page 370 in [Sno95b]).

Since a calendric granularity is a special kind of a time span, it is meaningful to compare two calendric granularities with each other.

Definition 3.1 *Comparison between calendric granularities:* G_A is *coarser* than G_B if $G_A > G_B$ as a time span. Similarly, G_A is *finer* than G_A if $G_A < G_B$ as a time span. ■

Example 3.1 The span of 1 *day* is shorter (<) than the span of 1 *month* and therefore the calendric granularity of days (G_{day}) is finer than the calendric granularity of months (G_{month}) in the Gregorian calendar. Similarly, G_{month} is coarser than G_{day}. □

[3] An *interval* is the basic unanchored temporal primitive in TSQL2. It is similar to a *time span* in this work.

3.2.2 Functions

Associated with a calendar is a list of functions ($\mathcal{F}$) which determine the number of finer calendric elements in coarser calendric elements. For example, assume the presence of a calendar C with the calendric granularities *year*, *month* and *day*. Three functions are defined: The first returns the number of months in a given year; the second returns the number of days in a given month of a given year; and the third maps a given year, month, and day to a real number on a global timeline. Notice that these functions depend on the particular value of a granularity and not just the granularity itself. For example, the number of days in a month depend on the month itself. More generally, let C be a calendar with calendric granularities $G_1, G_2, \ldots, G_n$, where G_1 is the coarsest calendric granularity and G_n is the finest calendric granularity. The following functions are then defined:

Definition 3.2 *Conversion functions:*

$$\begin{aligned}
f_C^1(i_1) &\rightarrow N_{G_2},\ 1 \le i_1 \le p_1 \\
f_C^2(i_1, i_2) &\rightarrow N_{G_3},\ 1 \le i_1 \le p_1, 1 \le i_2 \le p_2 \\
&\vdots \\
f_C^n(i_1, i_2, \ldots, i_n) &\rightarrow R,\ 1 \le i_1 \le p_1, 1 \le i_2 \le p_2, \ldots, 1 \le i_n \le p_n
\end{aligned}$$

where i_j $(1 \le j \le n)$ are natural numbers which correspond to the ordinal number of a calendric element of the j^{th} calendric granularity in calendar C. For example, the ordinal values of the year 1995 and the month September in the Gregorian calendar would be 1995 and 9, respectively. N_{G_x} $(1 \le x \le n)$ is a natural number which stands for the number of G_x's. p_i is the range of calendric elements for each considered granularity. R is a real number[4]. ■

The first function ($f_C^1(i_1)$) gives the number of G_2's in a given calendric element of G_1. The second function ($f_C^2(i_1, i_2)$) gives the number of G_3's defined by a given pair of calendric elements of types G_1 and G_2. The last function ($f_C^n(i_1, i_2, \ldots, i_n)$) maps a calendric element of the finest calendric granularity (G_n) to a real number on an underlying global real timeline, hereafter referred to as G_{tl}. The scale of G_{tl} is dependent on the precision of the respective machine architecture. For simplicity and explanatory purposes in this work, the scale of G_{tl} is assumed to be *seconds*.

Example 3.2 To illustrate the workings of the above functions, suppose one is interested in the number of months in 1995, the number of days in September 1995 and the number of seconds in 12 September 1995 in calendar C. The ordinal values corresponding to the year 1995, the month September, and the day 12, are 1995, 9, and 12, respectively. Then:

$$\begin{aligned}
f_C^1(1995) &\rightarrow 12 \\
f_C^2(1995, 9) &\rightarrow 30 \\
f_C^3(1995, 9, 12) &\rightarrow 86400.0
\end{aligned}$$

[4]We assume the underlying global timeline is real

□

Although the above example is trivial, it illustrates how the conversion functions work. It sets the stage for the more complicated cases that are discussed in the next section.

3.2.3 Conversions between Calendric Granularities

In a temporal model where times with different calendric granularities are supported, one should be able to convert a finer calendric granularity to a coarser calendric granularity, and vice-versa. These conversions are discussed below by first defining two functions. These functions are necessary since the number of units of one granularity contained in a unit of another granularity is not fixed. For example, there are 30 days in September and 31 days in January.

Definition 3.3 *Lower bound factor* $[lbf(G_A, G_B)]$*:* The lower bound factor of G_A and G_B is the minimum number of G_B units that can form 1 G_A unit. ■

Definition 3.4 *Upper bound factor* $[ubf(G_A, G_B)]$*:* The upper bound factor of G_A and G_B is the maximum number of G_B units that can form 1 G_A unit. ■

Example 3.3 $lbf(G_{month}, G_{day}) = 28$ and $ubf(G_{month}, G_{day}) = 31$. Both factors coincide in the case of those granularities that have exact conversions. For instance, $lbf(G_{hour}, G_{minute}) = ubf(G_{hour}, G_{minute}) = 60$. □

The user can define new calendric granularities in terms of existing ones. For example, the new calendric granularity *decade* could be defined in terms of the existing calendric granularity *year* using $lbf(G_{decade}, G_{year}) = ubf(G_{decade}, G_{year}) = 10$.

The derivations of $lbf(G_A, G_B)$ and $ubf(G_A, G_B)$ from the conversion functions defined in Section 3.2.2 when G_A is coarser than G_B, and when G_A is finer than G_B are now given.

Derivation 3.1 *G_A is coarser than G_B:* Let $G_1, \ldots, G_A, \ldots, G_B, \ldots, G_n$ be the totally ordered calendric granularities of calendar C with G_1 being the coarsest calendric granularity and G_n the finest. The following conversion functions are defined in C:

$$\begin{array}{rcl} & \vdots & \\ f_C^A(i_1, \ldots, i_A) & \rightarrow & N_{G_A+1} \\ & \vdots & \\ f_C^B(i_1, \ldots, i_B) & \rightarrow & N_{G_B+1} \\ & \vdots & \end{array}$$

Now, the number of G_B units in any given calendric element i_A is given by the following summation:

$$f_C^{A\rightarrow B}(i_1, \ldots, i_A) = \sum_{j_1=1}^{f_C^A(i_1,\ldots,i_A)} \sum_{j_2=1}^{f_C^{A+1}(i_1,\ldots,i_A,j_1)} \cdots \sum_{j_{B-A-1}=1}^{f_C^{B-2}(i_1,\ldots,i_A,j_1,\ldots,j_{B-A-2})} f_C^{B-1}(i_1, \ldots, i_A, j_1, \ldots, j_{B-A-1})$$

The minimum (maximum) number of G_B units in calendric element i_A is then the minimum (maximum) of the above formula over all $i_1, \ldots, i_A$. More specifically,

$$lbf(G_A, G_B) = \min_{(i_1,\ldots,i_A)\in C} \{f_C^{A\to B}(i_1, \ldots, i_A)\} \quad (3.1)$$

$$ubf(G_A, G_B) = \max_{(i_1,\ldots,i_A)\in C} \{f_C^{A\to B}(i_1, \ldots, i_A)\} \quad (3.2)$$

∎

Example 3.4 Let C be a calendar with the calendric granularities *year*, *month* and *day*. The following functions are defined in C:

$$\begin{aligned} f_C^1(y) &\to N_{months} \\ f_C^2(y, m) &\to N_{days} \\ f_C^3(y, m, d) &\to R \end{aligned}$$

where y, m, and d are ordinal values of calendric elements in the calendric granularities year, month, and day, respectively. Suppose we want to find $lbf(G_{year}, G_{day})$ and $ubf(G_{year}, G_{day})$. The number of days in any year y is given by the summation: $\sum_{m=1}^{f_C^1(y)} f_C^2(y, m)$. The minimum (maximum) number of days in a year is then the minimum (maximum) of this summation over all y. More specifically,

$$lbf(G_{year}, G_{day}) = \min_y\{\sum_{m=1}^{f_C^1(y)} f_C^2(y, m)\} \quad ubf(G_{year}, G_{day}) = \max_y\{\sum_{m=1}^{f_C^1(y)} f_C^2(y, m)\}$$

□

Derivation 3.2 *Minimum and maximum number of G_B in K units of G_A:* Formulas (3.1) and (3.2) calculate the minimum and maximum number of G_B in *one* unit of G_A, respectively. Formulas (3.1) and (3.2) are now generalized to calculate the minimum and maximum number of G_B in K units of G_A, e.g., the minimum and maximum number of days in $2 \cdot G_{month}$ where $K = 2$.

$$lbf(K, G_A, G_B) = \min_{i_1,\ldots,i_A} \{ \sum_{0\leq dist_{k_A}((i_1',\ldots,i_A'),(i_1,\ldots,i_A))\leq K-1} f_C^{A\to B}(i_1', \ldots, i_A')\} \quad (3.3)$$

$$ubf(K, G_A, G_B) = \max_{i_1,\ldots,i_A} \{ \sum_{0\leq dist_{k_A}((i_1',\ldots,i_A'),(i_1,\ldots,i_A))\leq K-1} f_C^{A\to B}(i_1', \ldots, i_A')\} \quad (3.4)$$

The summation in formulas (3.3) and (3.4) is the number of B units in K consecutive A units starting with $(i_1, \ldots, i_A)$. The function $dist_{k_A}((i_1', \ldots, i_A'), (i_1, \ldots, i_A))$ finds the number of k_A units elapsed between $(i_1', \ldots, i_A')$ and $(i_1, \ldots, i_A)$. For example, the number of months elapsed between (1996, February) and (1995, January) is 13. The lower and upper bound factors are then obtained by taking the minimum and maximum of the summation over all $(i_1, \ldots, i_A)$. ∎

Embedding the coefficient K within formulas (3.3) and (3.4) reduces the information lost in the process of calculating the number of G_B units in K units of G_A as compared to first finding the number of G_B units in one unit of G_A and then multiplying it by K to find the number of G_B in K units of G_A. For example, using formulas (3.1) and (3.2) to calculate the minimum and maximum number of days in $2 \cdot G_{month}$ gives 56 and 62, respectively, while formulas (3.3) and (3.4) give 59 and 62, respectively – thereby reducing the information lost by 3 days. Note that for exact conversions, $lbf(K, G_A, G_B) = ubf(K, G_A, G_B) = K \cdot lbf(G_A, G_B) = K \cdot ubf(G_A, G_B)$. For example, $lbf(K, G_{days}, G_{hours}) = ubf(K, G_{days}, G_{hours}) = K \cdot 24$.

Derivation 3.3 *G_A is finer than G_B:* If G_A is finer than G_B, then the lower and upper bound factors can be calculated using the formulas:

$$lbf_i(N, G_A, G_B) = \max_{K \in \mathbf{Z}} \{K \mid N \geq ubf(K, G_B, G_A)\} \tag{3.5}$$

$$ubf_i(N, G_A, G_B) = \min_{K \in \mathbf{Z}} \{K \mid N \leq lbf(K, G_B, G_A)\} \tag{3.6}$$

■

Example 3.5 To illustrate the formulas in Derivation 3.3, suppose one wants to find the number of months in 45 *days*. Then:

$$\begin{aligned} lbf_i(45, G_{day}, G_{month}) &= \max_{K \in \mathbf{Z}} \{K \mid 45 \geq ubf(K, G_{month}, G_{day})\} = 1 \\ ubf_i(45, G_{day}, G_{month}) &= \min_{K \in \mathbf{Z}} \{K \mid 45 \leq lbf(K, G_{month}, G_{day})\} = 2 \end{aligned}$$

Hence, the number of months in 45 *days* is $1 \sim 2$. □

Note that it is not necessary that K be an integer. It can be a real number as well, in which case the amount of indeterminacy in finding the number of months in 45 *days* is reduced. Thus, the formulas in Derivation 3.3 become:

$$lbf_r(R, G_A, G_B) = \max_{K \in \mathbf{R}^+} \{K \mid R \geq ubf(K, G_B, G_A)\} \tag{3.7}$$

$$ubf_r(R, G_A, G_B) = \min_{K \in \mathbf{R}^+} \{K \mid R \leq lbf(K, G_B, G_A)\} \tag{3.8}$$

Example 3.6 It is known that the number of days in 1 *month* is $28 \sim 31$ and the number of days in 2 *months* is $59 \sim 62$. Therefore, it can be reasonably concluded that for $1 \leq K \leq 2$:

$$\begin{aligned} lbf(K, G_{month}, G_{day}) &= 28 + (59 - 28) \cdot (K - 1) = 31 \cdot K - 3 \\ ubf(K, G_{month}, G_{day}) &= 31 + (62 - 31) \cdot (K - 1) = 31 \cdot K \end{aligned}$$

$$\begin{aligned} lbf_r(45, G_{day}, G_{month}) &= \max_{K \in \mathbf{R}^+} \{K \mid 45 \geq ubf(K, G_{month}, G_{day})\} \\ &= \max_{K \in \mathbf{R}^+} \{K \mid 45 \geq 31 \cdot K\} = 45/31 = 1.45 \\ ubf_r(45, G_{day}, G_{month}) &= \min_{K \in \mathbf{R}^+} \{K \mid 45 \leq lbf(K, G_{month}, G_{day})\} \\ &= \min_{K \in \mathbf{R}^+} \{K \mid 45 \leq 31 \cdot K - 3\} = 48/31 = 1.55 \end{aligned}$$

In this case the number of months in 45 *days* is $1.45 \sim 1.55$, quite a contrast from what was obtained for K as an integer. □

In TSQL2 [Sno95b], a calendar has a specification file which provides regular and irregular mappings between granularities. It is not clear however, how these mappings are derived. In this section, detailed derivation procedures for the $lbf(G_A, G_B)$ and $ubf(G_A, G_B)$ functions which represent regular and irregular mappings between any two granularities in a calendar were provided. Section 3.3.2.1 shows how the $lbf(G_A, G_B)$ and $ubf(G_A, G_B)$ functions are used in the conversion of unanchored temporal primitives to a given calendric granularity.

3.2.4 Mapping to TIGUKAT

In this section the calendar model described in Sections 3.2.1–3.2.3 is incorporated into the TIGUKAT object model. Types relevant to the representation of temporal information are depicted in Figures 3.2 and 3.3, along with their subtyping relationships. Likewise, operations defined in Sections 3.2.1–3.2.3 have corresponding TIGUKAT behaviors. These behaviors (along with their signatures) are given in Tables 3.1 and 3.2.

The type `T_calendar` models different kinds of calendars. It is a direct subtype of the `T_object` type as shown in Figure 3.2. Behaviors defined on `T_calendar` are shown in Table 3.1.

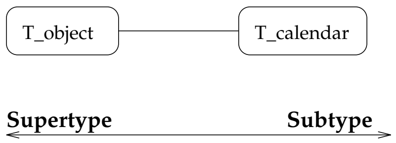


Figure 3.2: The calendar type.

Behavior *B_name* returns the name of a calendar e.g., *Gregorian*, *Academic*. *B_origin* returns the origin of the calendar in terms of a span. *B_calGranularities* returns a totally ordered collection of the calendric granularities of the calendar. For example, *B_calGranularities* of the Gregorian calendar shown in Figure A.1 returns $\{G_{Year}, G_{Month}, G_{Day}, G_{Hour}\}$. Finally, behavior *B_convFunctions* returns a list of the conversion functions described in Section 3.2.2. The behaviors *B_origin*, *B_calGranularities*, and *B_convFunctions* correspond to the properties *P_origin*, *P_calGranularities*, and *P_functions* of the `T_calendar` type in the temporal framework.

`T_calendar`	*B_name*:	`T_string`
	B_origin:	`T_span`
	B_calGranularities:	T_orderedColl⟨T_calGranularity⟩
	B_convFunctions:	T_list⟨T_function⟩

Table 3.1: Behaviors defined on calendars.

Since a calendric granularity in this thesis is a special kind of a determinate span, the type `T_calGranularity` is defined as a subtype of `T_discreteSpan` (described later in Section 3.3.2.5) as shown in Figure 3.3. The `T_calGranularity` is a new type that does not exist in the temporal framework inheritance hierarchy shown in Figure 2.14. It is introduced in the TIGUKAT temporal model to model the notion of calendric granularity described in Section 3.2.1. This illustrates the extensibility of the framework in that existing framework types can be subtyped with new types that are required to accomodate new features in a temporal model.

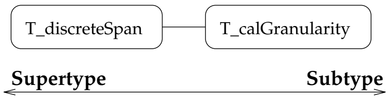


Figure 3.3: Calendric Granularity types.

Instances of `T_calGranularity` represent the different kinds of calendric granularities, e.g., *year*, *hour*, *semester*. Behaviors on calendric granularities are shown in Table 3.2.

`T_calGranularity`	*B_calendar*:	`T_calendar`
	B_similarCalGran:	T_collection⟨T_calGranularity⟩
	B_calElements:	T_list⟨T_calElement⟩
	B_lowerBound:	`T_calGranularity`
	B_upperBound:	`T_calGranularity`
	B_exactlyConvertibleTo:	`T_calGranularity` → `T_boolean`
	B_i-lbf:	`T_integer` → `T_calGranularity` → `T_integer`
	B_r-lbf:	`T_real` → `T_calGranularity` → `T_real`
	B_i-ubf:	`T_integer` → `T_calGranularity` → `T_integer`
	B_r-ubf:	`T_real` → `T_calGranularity` → `T_real`

Table 3.2: Behaviors defined on calendric granularity.

Behavior *B_calendar* in `T_calGranularity` returns the calendar which the calendric granularity belongs to. Behavior *B_similarCalGran* returns the set of calendric granularities that have similar duration as a particular calendric granularity. Behavior *B_calElements* returns an ordered collection of the calendric elements of a calendric granularity. For example *B_calElements* applied on the calendric granularity *semester* returns $\langle Fall, Winter, Spring, Summer\rangle$. The *B_lowerBound* and *B_upperBound* behaviors are refined accordingly to return a calendric granularity as the lower and upper bound of a calendric granularity. Since the calendric granularity is determinate, these behaviors return the same value. The behavior *B_exactlyConvertibleTo* checks if a calendric granularity is exactly convertible to another calendric granularity. For example, $G_{day}\cdot$ *B_exactlyConvertibleTo*(G_{hour}) returns `True`, while $G_{month}\cdot$ *B_exactlyConvertibleTo*(G_{day}) returns `False`. `T_calGranularity` also defines new behaviors, *B_i-lbf*, *B_r-lbf*, *B_i-ubf* and *B_r-ubf*. *B_i-lbf* and *B_i-ubf* return the lower and upper bound factors (see Section 3.2.3) of two calendric granularities with integer coefficients. Similarly, *B_r-lbf* and *B_r-ubf* return the lower and upper bound factors of two calendric granularities with real coefficients. For example, $G_{month}\cdot$ *B_i-lbf*$(1, G_{day})$ returns 28 while $G_{month}\cdot$ *B_i-ubf*$(1, G_{day})$ returns 31.

3.3 Temporal Structure

In this section, the problem of modeling and managing, within an ODBMS, the different anchored and unanchored primitives of the temporal framework (shown in Figure 2.3) is addressed. Many applications require support for (a) unanchored temporal primitives that are specified in different (for example, 3 *months*, 150 *seconds*) and mixed granularities (for example, 2 *hours and* 20 *minutes*), and (b) anchored temporal primitives that are specified in different granularities (for example, 1990, *October* 1993, 15 *June* 1996). Supporting anchored and unanchored temporal primitives with different granularities necessitates the proper handling of granularity mismatches in operations between temporal primitives with different granularities. This usually requires converting a temporal primitive from one granularity to another. Although there have been various recent proposals that handle multiple granularities [CC87, WJL91, WJS93, WBBJ97, BP85, MPB92, MMCR92, Sno95b], the focus has been on representing anchored temporal primitives that are specified in different granularities. Granularity conversions are given for anchored temporal primitives only. However, supporting unanchored temporal primitives with different granularities is equally important, and the issues that arise therein must be addressed. The real-world example from clinical medicine presented in Section 3.3.1 motivates this claim.

3.3.1 Motivation

In this section, a clinical example related to a patient with cardiological problems, particularly related to the widely known problem of diagnosing and following up unstable angina [BMJ94] is presented. Unstable angina[5] is a transitory clinical syndrome usually associated with an increased duration and/or intensity of symptoms related to coronary artery disease; risk of cardiac death and myocardial infarction increase. In this situation it is important to consider both the time when the symptoms (like chest pain) began and the time duration of these symptoms.

Consider the following sentences which are related to information contained in the cardiological medical record of a patient:

$S1$. The patient suffered from chest pain at rest for 2 hours and 55 minutes on 13 December 1995.

$S2$. The patient presented an episode of acute chest pain on 29 January 1996 from 13:20:15 to 13:56:23.

$S3$. The patient has been admitted to an Intensive Care Unit from 21:00 29 January 1996, and he has undergone intensive medical management for 36 hours.

[5]Formally known as angina pectoris. A clinical syndrome typically characterized by a deep, poorly localized chest or arm discomfort that is reproducibly associated with physical exertion or emotional stress and relieved promptly by rest or sublingual nitroglycerine. The discomfort of angina is often hard for patients to describe, and many patients do not consider it to be "pain." In most, but not all patients, these symptoms reflect myocardial ischemia resulting from significant underlying coronary artery disease [BMJ94].

$S4$. On 15 February 1996 the patient had myocardial infarction.

$S5$. At 3 pm 12 April 1996 the patient presented a new episode of chest pain of 7 minutes and 35 seconds during a soft exertion.

$S6$. From December 1994 to April 1996 the patient took aspirin.

$S7$. From 30 January 1996 the patient had to take a thrombolytic therapy for 38 months.

It can be observed from the above sentences that there are many different granularities for time instants (days in $S1$, $S4$, and $S7$; seconds in $S2$; minutes in $S3$; hours in $S5$; months in $S6$) and different and mixed granularities for time spans (hours and minutes in $S1$; hours in $S3$; minutes and seconds in $S5$; months in $S7$). Moreover, in a single sentence there may be time instants and time spans having heterogeneous granularities. For example, in $S3$ the time instant at which the patient is admitted is specified at the granularity of minutes, while the time duration of the medical management that the patient undertook is specified at the granularity of hours.

In addition to the different and mixed granularities in the patient-related information, the definition of unstable angina itself involves time spans given at different granularities. Unstable angina is, in fact, defined as: (1) symptoms of angina at rest, for more than 20 minutes, or (2) new onset, within two months, of exertional angina, involving marked limitations of ordinary physical activity, or (3) increasing angina within two months from the initial presentation, or (4) post- myocardial infarction angina, i.e., angina occurring from 1 to 60 days after an acute myocardial infarction [BMJ94].

In a clinical setting, one should be able to derive some extra information from the stored sentences about the patient. For example:

1. What is the time span between the myocardial infarction and the last episode of chest pain?

   To derive this, the elapsed time (which is a time span) between the time instants 15 *February* 1996 and 3*pm* 12 *April* 1996 (see sentences $S4$ and $S5$) needs to be computed.

2. What is the global span of the symptoms of angina?

   To answer this question, the elapsed time between the time instants 13:20:15 29 *January* 1996 and 13:56:23 29 *January* 1996 has to be added to the time spans 7 *minutes and* 35 *seconds*, and 2 *hours and* 55 *minutes* (see sentences $S1$, $S2$, and $S5$).

3. When did the patient finish the intensive medical management and what is the time span between the end of the intensive medical management and the onset of the new angina episode?

   To answer the first part of the question, the time span 36 *hours* has to be added to the time instant 21:00 29 *January* 1996. The elapsed time between the resulting time instant and the time instant 3*pm* 12 *April* 1996 gives the answer to the second part of the question (see sentences $S3$ and $S5$).

4. Was the patient taking aspirin when the past episode of chest pain happened?

   The answer to this question depends on what interpretation is given to the time instants *December* 1994 and *April* 1996. If the interpretation is that the patient took aspirin from *sometime* in *December* 1994 to *sometime* in *April* 1996, then a definite answer to the question cannot be given. However, if the time instants *December* 1994 and *April* 1996 are interpreted to mean the entire specified months, then *December* 1994 means the entire period between 00:00:00 1 *December* 1994 and 23:59:59 31 *December* 1994. Similarly, *April* 1996 means the entire period between 00:00:00 1 *April* 1996 and 23:59:59 30 *April* 1996. In this case a definite answer can be given that the patient was taking aspirin when the episode of chest pain happened (see sentences $S5$ and $S6$).

5. When will the thrombolytic therapy end?

   In this case the time span 38 *months* has to be added to the time instant 30 *January* 1996 (see sentence $S7$).

These questions substantiate the need for a temporal DBMS to provide the means for (a) representing and storing time instants with different granularities, and time spans with different and mixed granularities, (b) handling granularity mismatches in operations between temporal primitives with different granularities, (c) converting a temporal primitive from one granularity to another, and (d) considering different interpretations for time instants. The following sections show how these issues can be supported in a temporal ODBMS.

3.3.2 Unanchored Temporal Primitives

In Section 2.3.1.1, a time span was identified as being an unanchored, relative duration of time. Since a calendric granularity is a unit time span in the TIGUKAT temporal model, calendric granularities can be used to construct time spans. For example, the time span of 36 *hours* which represents the duration of intensive medical management the patient underwent (see sentence $S3$ in Section 3.3.1), is obtained as $36 \cdot G_{hour}$. A time span of 2 *hours and* 55 *minutes*, which represents the duration of chest pain the patient suffered (see sentence $S1$ in Section 3.3.1), can be obtained as $2 \cdot G_{hour} + 55 \cdot G_{minute}$. In general, a time span is made up of mixed calendric granularities and is defined as a finite sum:

Definition 3.5 *Discrete Determinate span:*

$$S_{discr} = \sum_{i=1}^{N} (K_i \cdot G_i) \tag{3.9}$$

where K_i is an integer coefficient of G_i, which is a distinct calendric granularity in the calendar. ■

Definition 3.6 *Continuous Determinate span:*

$$S_{cont} = \sum_{i=1}^{N} (R_i \cdot G_i) \tag{3.10}$$

where R_i is a real coefficient of G_i, which is a distinct calendric granularity in the calendar. ■

Basically, S_{discr} and S_{cont} are summations of distinct calendric granularities over a given calendar. S_{cont} is a generalization of S_{discr} for the case of real coefficients. In a temporal model where time spans with different calendric granularities are supported, one should be able to convert a time span to a given calendric granularity. This conversion process, together with the semantics of operations on time spans with mixed granularities, are discussed in the following sections.

3.3.2.1 Conversion of Time Spans

The first question that comes to mind is whether it is always possible to convert a time span from a coarser to a finer calendric granularity without loss of information. The answer, perhaps surprisingly, is negative. To illustrate this point, consider the following: the conversion of the time span 1 *hour* to the calendric granularity of minutes is exact and will result in the time span of 60 *minutes*. However, the conversion of the time span 1 *month* to the finer calendric granularity of days cannot possibly be an exact one. Should the resulting time span be 31, 30, 29 or 28 days? One cannot tell unless one knows which month is involved. Since a time span is *unanchored* this information is not available. One could convert 1 *month* to the indeterminate span 28 *days* $\sim$ 31 *days* but in this case the conversion is not exact and some information is lost. Therefore, the following observation is made:

Observation 3.1 The set of all calendric granularities is not totally ordered with respect to the binary relation "exactly convertible to." □

The conversion of a determinate time span to any given calendric granularity G_A is now defined.

Definition 3.7 *Discrete time span conversion:* The conversion of a time span of the form depicted in Definition 3.5 to a calendric granularity G_A results in a time span[6].

with lower bound

$$\lfloor \sum_{i=1}^{N} L_i \rfloor \cdot G_A \tag{3.11}$$

[6]Note that conversion of a time span to any calendric granularity may be exact or inexact. In the former case, the lower and upper bounds of the resulting time span are identical, which signifies that the time span is determinate. In case of an inexact conversion, the resulting time span will be indeterminate.

and upper bound

$$\lceil \sum_{i=1}^{N} U_i \rceil \cdot G_A \tag{3.12}$$

where

$$L_i = lbf_r(K_i, G_i, G_A) \text{ and } U_i = ubf_r(K_i, G_i, G_A) \tag{3.13}$$

■

The definition of converting a continuous time span to a given calendric granularity is similar to Definition 3.7. The following examples illustrate exact and inexact time span conversions.

Example 3.7 To convert the duration of the chest pain in sentence $S1$ (see Section 3.3.1), which is the discrete time span 2 *hours and* 55 *minutes*, to a time span in the calendric granularity of minutes (G_{minute}), the duration is first represented in the form given in Definition 3.5: $2 \cdot G_{hour} + 55 \cdot G_{minute}$. In this span, $K_1 = 2, K_2 = 55, G_1 = G_{hour}, G_2 = G_{minute}$. Since G_2 is already in minutes, it is sufficient to convert the time span $2 \cdot G_{hour}$ to minutes, and then add the resulting time span to $55 \cdot G_{minute}$.

$$\begin{array}{lcl@{\qquad}lcl}
L_1 & = & lbf(K_1, G_1, G_{minute}) & U_1 & = & ubf(K_1, G_1, G_{minute}) \\
 & = & lbf(2, G_{hour}, G_{minute}) & & = & ubf(2, G_{hour}, G_{minute}) \\
 & = & 2 \cdot lbf(G_{hour}, G_{minute}) & & = & 2 \cdot ubf(G_{hour}, G_{minute}) \\
 & = & 2 \cdot 60 & & = & 2 \cdot 60 \\
 & = & 120 & & = & 120 \\
 & & & & & \\
L_2 & = & 55 & U_2 & = & 55
\end{array}$$

$$\begin{array}{rcl@{\quad}rcl}
\textit{lower bound} & = & \lfloor L_1 + L_2 \rfloor \cdot G_{minute} & \textit{upper bound} & = & \lceil U_1 + U_2 \rceil \cdot G_{minute} \\
 & = & 175 \cdot G_{minute} & & = & 175 \cdot G_{minute}
\end{array}$$

Hence, the result of the conversion is the determinate discrete time span time span $175 \cdot G_{minute}$ (175 *minutes*). This is an example of an exact time span conversion. The next example shows an inexact time span conversion. □

Example 3.8 To convert the discrete time span 2 *months and* 45 *hours* to a time span in the calendric granularity of days (G_{day}), 2 *months and* 45 *hours* is first represented in the form given in Definition 3.5: $2 \cdot G_{month} + 45 \cdot G_{hour}$. In this span, $K_1 = 2, K_2 = 45, G_1 = G_{month}, G_2 = G_{hour}$. Formula (3.13) is used to compute L_1, L_2, U_1, U_2:

$$
\begin{aligned}
L_1 &= lbf(K_1, G_1, G_{day}) \\
&= lbf(2, G_{month}, G_{day}) \\
&= 59
\end{aligned}
\qquad
\begin{aligned}
U_1 &= ubf(K_1, G_1, G_{day}) \\
&= ubf(2, G_{month}, G_{day}) \\
&= 62
\end{aligned}
$$

$$
\begin{aligned}
L_2 &= lbf(K_2, G_2, G_{day}) \\
&= lbf(45, G_{hour}, G_{day}) \\
&= max\{K \mid 45 \geq ubf(K, G_{day}, G_{hour})\} \\
&= max\{K \mid 45 \geq K \cdot 24\} \\
&= 1.875
\end{aligned}
\qquad
\begin{aligned}
U_2 &= ubf(K_2, G_2, G_{day}) \\
&= ubf(45, G_{hour}, G_{day}) \\
&= min\{K \mid 45 \leq lbf(K, G_{day}, G_{hour})\} \\
&= min\{K \mid 45 \leq K \cdot 24\} \\
&= 1.875
\end{aligned}
$$

$lbf(K, G_{month}, G_{day})$, $lbf(K, G_{day}, G_{hour})$, $ubf(K, G_{month}, G_{day})$, and $ubf(K, G_{day}, G_{hour})$ are calculated from the conversion functions in the Gregorian calendar. Lastly, the lower and upper boundary of the resulting time span are computed according to formulas (3.11) and (3.12), respectively:

$$
\begin{aligned}
lower\ bound &= \lfloor L_1 + L_2 \rfloor \cdot G_{day} \\
&= \lfloor 59 + 1.875 \rfloor \cdot G_{day} \\
&= 60 \cdot G_{day}
\end{aligned}
\qquad
\begin{aligned}
upper\ bound &= \lceil U_1 + U_2 \rceil \cdot G_{day} \\
&= \lceil 62 + 1.875 \rceil \cdot G_{day} \\
&= 64 \cdot G_{day}
\end{aligned}
$$

Hence, the result of the conversion is the indeterminate discrete time span 60 *days* $\sim$ 64 *days*.
□

3.3.2.2 Canonical Forms for Time Spans

In addition to the set of granularities $G_1, \ldots, G_N$ and conversion functions discussed earlier, each calendar also implicitly defines the relation *exactly convertible to* between its granularities. G_i is *exactly convertible to* G_j iff $ubf(k, G_i, G_j) = lbf(k, G_i, G_j) = k \cdot C$, where C is a natural number. Note that exact convertibility is a partial order on granularities which is a suborder of magnitude ordering. If G_i is exactly convertible to G_j, then $G_i = C \cdot G_j$, where C is a natural number. Since discrete determinate time spans have the form $S = \sum_{i=1}^{N} K_i G_i$, where K_i are integer numbers, the presence of the exact conversion rules implies the existence of different forms of a time span. For example, 2 *hour* 55 *minutes* and 175 *minutes* are different forms of the same time span S'. To adhere as much as possible to human readability and user intuition, it is usually desirable to represent time spans in some canonical form. For example, when the time span 1 *hour* 30 *minutes* is added to the time span 35 *minutes*, the user would expect the time span 2 *hours* 05 *minutes* rather than the time span 1 *hour* 65 *minutes*. In this section, canonical forms for time spans are defined. First, *representations* for time spans are defined.

Definition 3.8 *Span Representation:* The N-tuple $r = \langle a_i \rangle_{i=1}^{N}$ (where a_i are integer numbers and N is the number of calendric granularities in a calendar) is called a *representation* of a span S (denoted $r \in Rep(S)$) iff $S = \sum_{i=1}^{N} a_i G_i$. ■

Example 3.9 Assume that the Gregorian calendar has the calendric granularities *year*, *month*, *day*, *hour*, *minute* and *second*. Then 2 *hour* 55 *minutes* and 175 *minutes* which are two forms of S' have the representations $r_1 = \langle 0,0,0,2,55,0 \rangle$ and $r_2 = \langle 0,0,0,0,175,0 \rangle$, respectively. □

Span representations will be used to define a *canonical form* for a time span. In order to do that, the notion of a *strictly non-negative span* is introduced.

Definition 3.9 *Strictly Non-Negative Span:* A span S is a *strictly non-negative span* (denoted $S >^+ 0$) iff $\exists r = \langle a_i \rangle_{i=1}^N \in Rep(S) : a_i \geq 0$ for $i = 1, \ldots, N$. ■

Example 3.10 The time span 2 *hour* 55 *minutes* is strictly non-negative while the time spans 1 *week* − 10 *days* and 1 *month* − 30 *days* are not strictly non-negative since no positive representations of either of them exist. In the first time span, although 1 *week* can be converted to *days* exactly, the resulting span −3 *days* does not have a positive representation. In the second time span, no positive representations are possible since 1 *month* does not have an exact conversion to *days*. □

Another definition that needs to be defined for a canonical form is a dominancy relation between span representations. The dominancy relation is in fact a lexicographical order on span representations, which is used in determining the canonical representation of a span..

Definition 3.10 *Dominancy:* A representation $r = \langle a_i \rangle_{i=1}^N$ *dominates* another representation $r' = \langle b_i \rangle_{i=1}^N$ (denoted $r \succ r'$), $r, r' \in Rep(S)$, iff $\exists k : a_k > b_k \wedge a_i = b_i$ for $i = 1, \ldots, (k-1)$. ■

Example 3.11 $r_1 = \langle 0,0,0,2,55,0 \rangle \succ r_2 = \langle 0,0,0,0,175,0 \rangle$. □

Having defined strictly non-negative spans and dominancy, the *canonical representation* and the *canonical form* for strictly non-negative spans[7] can now be defined.

Definition 3.11 *Canonical Representation:* A representation $r = \langle a_i \rangle_{i=1}^N \in Rep(S)$ is *the canonical representation* of span $S >^+ 0$ iff $a_i \geq 0$ for $i = 1, \ldots, N \wedge \forall r' \in Rep(S) : r \succ r' \vee r = r'$. ■

Example 3.12 r_1, i.e., $\langle 0,0,0,2,55,0 \rangle$ is the canonical representation of the time span S'. □

Observation 3.2 Every strictly non-negative span has one and only one canonical representation. □

The canonical representation is the best representation of a given strictly non-negative span.

Definition 3.12 *Canonical Form:* A strictly non-negative span $S = \sum_{i=1}^N a_i \cdot G_i$ is in *canonical form* iff $r = \langle a_i \rangle_{i=1}^N$ is the canonical representation of S. ■

Example 3.13 The canonical form for the time span S' is 2 *hour* 55 *minutes*. □

[7]Strictly non-positive spans can be defined similarly and the canonical form can also be defined for them.

3.3.2.3 Operations Between Time Spans

In this section the semantics of arithmetic and comparison operations between time spans are given and the questions posed in Section 3.3.1 are answered.

Arithmetic Operations Between Time Spans

As described earlier, a time span is represented as a summation of different calendric granularities. In this section we elaborate on the arithmetic operations between time spans using various examples. The semantics of adding (subtracting) two time spans is to add (subtract) the components which have the same calendric granularity, concatenate the remaining components to the resulting time span, and reduce the resulting time span to canonical form as described in Section 3.3.2.2.

Example 3.14

1. $(5\ years + 4\ months) + 2\ years \rightarrow (7\ years + 4\ months)$
2. $(5\ years + 4\ months) + 15\ days \rightarrow (5\ years + 4\ months + 15\ days)$

□

Similar semantics hold true for addition (subtraction) of determinate time spans and indeterminate time spans. The following example shows the global duration of the symptoms of angina, described in sentences $S1$ and $S5$ for the patient considered in the motivating example presented in Section 3.3.1.

Example 3.15

$$\begin{aligned}&(2\ hours + 55\ minutes) + (7\ minutes + 35\ seconds)\\&\rightarrow (2\ hours + 62\ minutes + 35\ seconds)\\&\rightarrow (3\ hours + 2\ minutes + 35\ seconds)\end{aligned}$$

□

It can be noted from the above example that the global duration of the two angina episodes is converted to its canonical form by the addition operation.

Subtraction leads to the notion of negative spans. In this work, both positive and negative spans are allowed. Positive spans have the semantics of forward duration in time, while negative spans have the semantics of backward duration in time. Allowing positive and negative spans enables one to carry out the subtraction operation between spans of different calendric granularities which could result in either a positive or negative span, for example, $1\ month - 30\ days$.

Comparison Operations Between Time Spans

The semantics of comparing two time spans is to first convert each time span to the finest granularity that exists between the two time spans, and then carry out the comparison. The following example illustrates the various combinations that could occur:

Example 3.16

1. $(1\ hour + 30\ minutes) = 90\ minutes$?
   $\Leftrightarrow 90\ minutes = 90\ minutes$
   $\Leftrightarrow$ `True`

2. $1\ month > 30\ days$?
   $\Leftrightarrow (28\ days \sim 31\ days) > 30\ days$
   $\Leftrightarrow$ `Unknown`

□

It is noted from the above example that time spans which *overlap* (or even *meet* each other) cannot be compared. This follows from Observation 3.1 (see Section 3.2.3) which states that calendric granularities are partially ordered with respect to the binary relation "exactly convertible to." The next example compares the duration of the first symptom of angina (sentence $S1$) with the duration ($20\ minutes$) defined for establishing unstable angina (see the motivating example in Section 3.3.1).

Example 3.17

$(2\ hours + 55\ minutes) > 20\ minutes$?
$\Leftrightarrow 175\ minutes > 20\ minutes$
$\Leftrightarrow$ `True`

Therefore the patient is identified as suffering from unstable angina. □

3.3.2.4 Related Work

In this section the approach presented in this work of representing and operating on unanchored time durations (time spans) is compared to that of Lorentzos [Lor94] and TSQL2 [Sno95b]. Since a time span is independent of any time instant or time interval due to its relative nature, granularity conversions in the context of anchored temporal primitives cannot be used for unanchored temporal primitives. Hence, none of the temporal models [CC87, WJL91, WJS93, MPB92, MMCR92, Sno95b, WBBJ97] can support the unanchored temporal information needs of an application like the clinical example given in Section 3.3.1.

Although the work of Lorentzos [Lor94] does not explicitly deal with temporal granularity, it proposes a scheme for representing and operating on non-metric types. Mixed granularity time durations, with separate fields for their composite parts (e.g., hours, minutes, seconds) are one example of a non-metric data type. These can be represented as elements of sets of composite numbers which provide conversion relationships (mappings) between the composite fields. However, only exact (regular) mappings are discussed. The representation does not provide inexact (irregular) mappings. Therefore time durations with composite parts having granularities of months and days cannot be modeled. In the approach presented in this work, a time span is simply a summation of calendric granularities. Both exact and inexact mappings between granularities are provided (using the $lbf(G_A, G_B)$ and $ubf(G_A, G_B)$ functions). This allows time durations to be converted to any given calendric granularity.

The conversion of a time duration to a particular granularity is possible in [Lor94]. However, the target granularity is restricted to be one of the granularities of the composite parts of the time duration. For example, if the time duration is 2 hours, 50 minutes, 30 seconds, then the time duration can be converted to hours, minutes, or seconds. Such a restriction is not enforced in this work. A time duration can be converted to any desired granularity in the calendar. The conversion process of the time duration 2 *months and* 45 *hours* to a time duration in the granularity of days is shown in Example 3.8.

In [Lor94], addition between time durations is also possible. However, the operands have to be *addition compatible*. If S_1 and S_2 are time durations, then they are addition compatible if S_2 consists of at most as many composite parts as S_1, and for these composite parts, the granularities should be the same. For example, the time durations with composite granularities (days,hours,minutes,seconds), (hours,minutes,seconds), (minutes,seconds), and (seconds) are addition compatible, and thus can be added to each other. The approach presented in this thesis is more general in that time durations do not have to be addition compatible. The components of the time durations which have the same calendric granularity are simply added to each other, and the remaining components are concatenated to the resulting time span, as shown in Section 3.3.2.3.

In TSQL2 [Sno95b], time spans (durations) which have mixed granularities cannot be represented [Sno96]. For example, the duration of the chest pain in sentence $S1$ (see Section 3.3.1) would have to be represented in hours or in minutes. Since a time span is a summation of distinct granularities in this thesis, representing symptom durations with mixed granularities is straightforward. The representation of mixed granularity time spans in this work is also more general than that used in SQL-92 in that time spans are not restricted to only *year-month* or *day-time* combinations.

A time span in TSQL2 is necessarily indeterminate at both coarser and finer granularities. This is because a granularity is modeled as an anchored partitioning of the timeline, whereas a time span in unanchored. Therefore, all time span conversions in TSQL2 are treated as inexact, resulting in indeterminate time spans. In the TIGUKAT temporal model, a time span conversion can be exact or inexact. Consider the simple conversion of the time span 1 *hour* to the granularity of minutes. In TSQL2, this conversion results in the indeterminate span $1 \sim 119$ *minutes* – an indeterminacy of 120 minutes. In this thesis however, the conversion is exact and results in the determinate span 60 *minutes*, which is what is expected in reality.

Operations involving time spans in TSQL2 could give rise to ambiguities and even incorrect results. Consider the addition of the time span 1 *hour* to the time span 40 *minutes* in TSQL2. There are two semantics defined: left-operand (coarser granularity) semantics and finer granularity semantics. In left-operand (coarser granularity) semantics, this addition can result in two different time spans:

$$
\begin{aligned}
1.\ 1\ hour + 40\ minutes &\rightarrow 1\ hour + scale(40\ minutes) \\
&\rightarrow 1\ hour + (0 \sim 1\ hour) \\
&\rightarrow 1 \sim 2\ hours \\
2.\ 1\ hour + 40\ minutes &\rightarrow 1\ hour + cast(40\ minutes) \\
&\rightarrow 1\ hour + cast(0 \sim 1\ hour) \\
&\rightarrow 1\ hour + 0\ hour \\
&\rightarrow 1\ hour
\end{aligned}
$$

The first operation *scales* the time span of 40 *minutes* to the granularity of hours (granularity of the left operand), which results in the indeterminate time span $0 \sim 1$ *hour*. In the second operation, the time span of 40 *minutes* is *cast* to the granularity of hours. The cast operation first scales the time span 40 *minutes* to the granularity of hours which result in the indeterminate time span $0 \sim 1$ *hour* from which the first component, 0 *hour*, is arbitrarily chosen.

In finer granularity semantics, this addition can result in two different time spans:

$$
\begin{aligned}
1.\ 1\ hour + 40\ minutes &\rightarrow scale(1\ hour) + 40\ minutes \\
&\rightarrow (1 \sim 119\ minutes) + 40\ minutes \\
&\rightarrow 41 \sim 159\ minutes \\
2.\ 1\ hour + 40\ minutes &\rightarrow cast(1\ hour) + 40\ minutes \\
&\rightarrow cast(1 \sim 119\ minutes) + 40\ minutes \\
&\rightarrow 1\ minute + 40\ minutes \\
&\rightarrow 41\ minutes
\end{aligned}
$$

The first operation *scales* the time span of 1 *hour* to the granularity of minutes (the finer granularity of the operands), which results in the indeterminate time span $1 \sim 119$ *minutes*. In the second operation, the time span of 1 *hour* is *cast* to the granularity of minutes resulting in the time span 1 *minute*.

In both cases, the addition operation yields results which are counter-intuitive to what a user actually expects, since some information is lost in the conversion process. Indeed neither semantics gives the desired result of 100 *minutes* or 1 *hour and* 40 *minutes*. In this work, the resulting time span for the addition of 1 *hour* to 40 *minutes* is 1 *hour and* 40 *minutes*.

Since a time span is represented as a summation of different calendric granularities, the semantics of arithmetic operations between time spans of different calendric granularities in this thesis exactly model what is intuitively expected in the real-world.

Comparison of time spans of different granularities in TSQL2 can also lead to incorrect results. Consider the comparison of the time span 30 *minutes* with the time span 1 *hour* in TSQL2 [Dyr96], using left-operand semantics or finer granularity semantics:

$$
\begin{aligned}
&30\ minutes > 1\ hour\ ? \\
&\Leftrightarrow 30\ minutes > cast(1\ \sim\ 119\ minutes) \\
&\Leftrightarrow 30\ minutes > 1\ minute \\
&\Leftrightarrow \texttt{True}!
\end{aligned}
$$

The time span 1 *hour* is first converted to the granularity of the leftmost operand. Since a time span is indeterminate at any finer or coarser granularity in TSQL2, the conversion of 1 *hour* to the granularity of minutes yields the indeterminate time span 1 $\sim$ 119 *minutes*. The cast operation then converts this to a determinate time span by arbitrarily choosing the lower bound. This leads to comparing the time span 30 *minutes* to the time span 1 *minute*, and subsequently returning `True` which is the opposite of what is expected. In the approach presented in this thesis, the time span 1 *hour* would be converted exactly to the time span 60 *minutes*, and the comparison would then return `False`.

3.3.2.5 Mapping to TIGUKAT

The TIGUKAT types corresponding to the span types in the temporal framework are shown in Figure 3.4. The various behaviors on time spans together with their signatures are shown in Table 3.3.

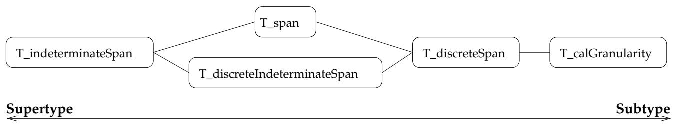


Figure 3.4: Span types.

The type `T_indeterminateSpan` models continuous indeterminate time spans, and corresponds to the `T_indetContSpan` type in Figure 2.3. Behaviors defined on `T_indeterminateSpan` include *B_lessthan* and *B_greaterthan* which model the comparison operations on time spans. Behaviors *B_add* and *B_subtract* allow continuous determinate spans to be added to and subtracted from continuous indeterminate spans, respectively. The unary behaviors, *B_lowerBound* and *B_upperBound* return the lower and upper boundaries (which are continuous determinate spans) of a continuous indeterminate time span, respectively.

`T_indeterminateSpan` has two direct subtypes: `T_discreteIndeterminateSpan` and `T_span`. The latter corresponds to the notion of a *continuous determinate span*. This subtyping relationship has the following justification: Every continuous determinate span can be treated as an indeterminate one (with identical lower and upper bounds) and every discrete span can be treated as a continuous one. `T_discreteIndeterminateSpan` corresponds to the `T_indetDiscSpan` type of the temporal framework, while `T_span` corresponds to the `T_detContSpan` of the temporal framework.

T_indeterminateSpan	*B_lessthan*:	T_indeterminateSpan → T_boolean
	B_greaterthan:	T_indeterminateSpan → T_boolean
	B_add:	T_span → T_indeterminateSpan
	B_subtract:	T_span → T_indeterminateSpan
	B_lowerBound:	T_span
	B_upperBound:	T_span
T_discreteIndeterminateSpan	*B_add*:	T_discreteSpan → T_discreteIndeterminateSpan
	B_subtract:	T_discreteSpan → T_discreteIndeterminateSpan
	B_lowerBound:	T_discreteSpan
	B_upperBound:	T_discreteSpan
T_span	*B_add*:	T_span → T_span
	B_subtract:	T_span → T_span
	B_calGranularities:	T_collection⟨T_calGranularity⟩
	B_coefficient:	T_calGranularity → T_real
	B_multiply:	T_real → T_span
	B_divide:	T_real → T_span
	B_convertTo:	T_calGranularity → T_indeterminateSpan
T_discreteSpan	*B_add*:	T_discreteSpan → T_discreteSpan
	B_subtract:	T_discreteSpan → T_discreteSpan
	B_coefficient:	T_calGranularity → T_integer
	B_multiply:	T_integer → T_discreteSpan
	B_succ:	T_span
	B_pred:	T_span

Table 3.3: Behaviors defined on time spans.

In `T_discreteIndeterminateSpan`, the behaviors *B_add* and *B_subtract* take a discrete determinate span as an argument and return a discrete indeterminate span as the result. Furthermore, the unary behaviors *B_lowerBound* and *B_upperBound* are refined to return a discrete determinate span.

Behaviors *B_add* and *B_subtract* are refined in `T_span` to take a continuous determinate span as an argument and return a continuous determinate span as the result. Behaviors *B_calGranularities*, *B_coefficient*, *B_multiply*, *B_divide* and *B_convertTo* in `T_span` are used in the conversion process of a time span to a specific calendric granularity as shown in Section 3.3.2.1. *B_calGranularities* returns a collection of calendric granularities in a time span. For example, the behavior application $(1\ month + 5\ days) \cdot$ *B_calGranularities* returns $\{G_{day}, G_{month}\}$. The behavior *B_coefficient* returns the (real) coefficient of a time span given a specific calendric granularity. For example, $(1\ month + 5\ days) \cdot$ *B_coefficient*(G_{day}) returns 5.0. Behaviors *B_multiply* and *B_divide* are basically used in the conversion process. The *B_convertTo* behavior is derived from the rest of the behaviors in `T_span` and essentially converts a determinate time span to an indeterminate time span with the specified calendric granularity.

The type `T_discreteSpan`, corresponding to the `T_detDiscSpan` of the temporal framework, is defined as a subtype of the `T_discreteIndeterminateSpan` and `T_span` types described above. Behaviors *B_add* and *B_subtract* are refined in `T_discreteSpan` to take a discrete determinate span as an argument and return a discrete determinate as a result. Behavior *B_coefficient* is refined to return the integer coefficient of a discrete time span and the *B_multiply* behavior is refined to multiply an integer by a discrete time span. Behaviors *B_succ* and *B_pred* are defined in `T_discreteSpan` to return the next or previous discrete time span of a particular discrete time span. For example, $(2\ months + 45\ hours)\cdot$ *B_succ* returns the time span $3\ months + 46\ hours$ while $(2\ months + 45\ hours)\cdot$ *B_pred* returns the time span $1\ month + 44\ hours$.

3.3.3 Anchored Temporal Primitives

To complete the puzzle on temporal granularity, anchored temporal primitives with different granularities are also supported in the TIGUKAT temporal model. Consider the following 3 sentences:

1. The spaceship will be launched on 15 January 1995.

2. The Family Day holiday in Alberta is on 17 February.

3. John was born on 12 May 1965.

It can be noticed from the above sentences that the event in each case takes place *on a certain day*. The temporal primitive associated with all three events is traditionally called a *time instant* (*granule*), which is an anchored temporal primitive. However, time instants can be subjected to different interpretations. In particular, three possible interpretations of a time instant are proposed:

1. A time instant which refers to the *beginning* of the period it denotes. For example in the first sentence, what is meant is that the spaceship will be launched precisely at the beginning of the day, signifying the *start* of the time instant 15 January 1995.

2. A time instant which refers to the *whole* period it denotes. For example in the second sentence, what is meant is that the *whole* of the time instant 17 February is a holiday.

3. A time instant which refers to *sometime* in the period it denotes. This is perhaps the most commonly used time instant in "real-world" temporal measurements. For example in the third sentence, the birth took place *sometime* during the time instant 12 May 1965.

Basically, only the time instant which refers to the beginning of the period it denotes is a "genuine" instant. The other two times are special kinds of time intervals that are called time instants only due to tradition. The time which refers to the whole period it denotes is analogous to a *determinate interval*, signifying that the event took place during the entire period. Similarly, the time which refers to sometime in the period it denotes is analogous to an *indeterminate interval*, signifying that the event took place sometime during the period, although exactly when it took place is unknown. Since a time interval is represented by two anchored time instants, the presentation in this section focusses on time instants. The following sub-sections show how these "time instants" are mapped to their corresponding time intervals. The sub-sections also show how time instants are converted to different granularities, and describes the various operations that involve time instants specified in different granularities. These include comparison of time instants, elapsed time between two time instants, and arithmetic operations that involve time spans and time instants.

3.3.3.1 Representation of Time Instants

Figure 3.5 shows the structural representation of a time instant. Every time instant belongs to a specific calendar and is composed of calendric elements which belong to different calendric granularities of the same calendar. Table 3.4 gives examples of time instants, the

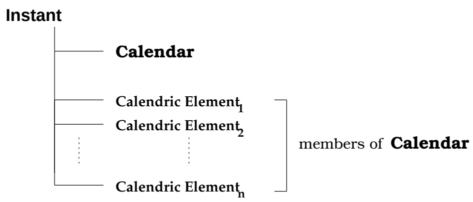


Figure 3.5: Structural representation of a time instant.

calendar they belong to, the calendric elements they are composed of and the respective calendric granularities.

Instant	Calendar	Calendric Elements	Calendric Granularities
15 June 1995	Gregorian	15 June 1995	*Day* *Month* *Year*
Fall 1995	Academic	Fall 1995	*Semester* *AcademicYear*

Table 3.4: Examples of time instants.

Three possible interpretations of a time instant can be identified:

- *Beginning Instant* (I_{beg}). This type of instant refers to the *beginning* of the period it denotes. Examples of beginning instants would be the time of space launches, start of exams, process start and end times in real-time systems, etc. Therefore the time instants 1995_{beg}, *January* 1995_{beg}, and 1 *January* 1995_{beg} are equivalent and refer to the beginning of the year 1995.

- *Determinate Instant* (I_{det}). This type of instant refers to the *whole* period it denotes. The time instants denoting national holidays are examples of determinate instants. For example, Victoria Day (a national holiday in Canada) occurs on 24 May each year. This means that the whole day of 24 May is a holiday. In this case, 24 May_{det} is a determinate instant.

- *Indeterminate Instant* (I_{indet}). This type of instant refers to *sometime* in the period it denotes, and is perhaps the most commonly used time instant in "real-world" temporal measurements. For example birth dates are usually specified in the granularity of days. In reality however, the person is born sometime on that day. Another example would be the times of clinical events. In sentence $S4$ of the motivating example, 15 February 1996, the time at which the patient had a myocardial infarction, would be represented as 15 $February$ 1996_{indet} (which means *the patient had a myocardial infarction some time on that day*).

Essentially, a determinate (indeterminate) time instant behaves like a determinate (indeterminate) interval whose lower and upper bounds are beginning time instants. For example, the time instant 5 $February$ 1997_{det} is analogous to the interval [5 $February$ 1997_{beg}, 6 $February$ 1997_{beg}).

Every determinate (indeterminate) time instant has a granularity (G_i) associated with it. This granularity determines the mapping of the given determinate (indeterminate) time instant I_{det} (I_{indet}) to the domain of beginning time instants. The mapping is defined as follows:

$$I_{det} \mapsto [I_{beg}, I_{beg} + G_i)$$
$$I_{indet} \mapsto [I_{beg} \sim I_{beg} + G_i)$$

Here I_{beg} denotes the counterpart of I_{det} and I_{indet} in the domain of beginning time instants. This is exemplified by the mapping of the discrete determinate instant 5 $January$ 1997_{det} to the interval [5 $February$ 1997_{beg}, 6 $February$ 1997_{beg}). In this case $G_i = G_{days} = 1$ *day*. The upper bound of the resulting interval is defined to be open to ensure that different time instants with the same granularity do not overlap. Table 3.5 gives examples of a beginning, determinate, and indeterminate time instant at different granularities.

Granularity	Beginning Instant	Determinate Instant	Indeterminate Instant
Year	1995_{beg}	$1995_{det} \mapsto [1995_{beg}, 1996_{beg})$	$1995_{indet} \mapsto [1995_{beg} \sim 1996_{beg})$
Month	$January$ 1995_{beg}	[$January$ 1995_{beg}, $January$ 1996_{beg})	[$January$ 1995_{beg} $\sim$ $January$ 1996_{beg})
Day	1 $January$ 1995_{beg}	[1 $January$ 1995_{beg}, 1 $January$ 1996_{beg})	[1 $January$ 1995_{beg} $\sim$ 1 $January$ 1996_{beg})

Table 3.5: Conversion of time instants to finer granularities.

A beginning time instant is determinate at all finer and coarser granularities. A determinate time instant is also determinate at all finer granularities. This is because a determinate instant is first mapped to a determinate interval whose bounds are beginning time instants. This interval is then determinate at all finer granularities. An indeterminate instant is indeterminate *at the granularity it is defined*. It is mapped to an indeterminate interval whose bounds are beginning time instants. At all finer levels of granularity, the bounds of the indeterminate interval are simply replaced by equivalent ones at that granularity. Since determinate and indeterminate time instants can be mapped to the domain of beginning time instants, the rest of this section concentrates on beginning time instants. Without losing generality, in the rest of the thesis, a beginning time instant is referred simply as a time instant.

3.3.3.2 Operations on Time Instants

As with time spans, instants can be compared with each other, and subtracted from one another to find the elapsed time between them. Additionally, a time span can be added to, or subtracted from, a time instant to return another time instant.

Comparison Between Time Instants

Let $I^1_{G_A} = (i_1, \ldots, i_m)$ and $I^2_{G_B} = (i^{'}_1, \ldots, i^{'}_n)$ be two time instants, with finest granularities G_A and G_B, respectively, where $i_1, \ldots, i_m$ are ordinal numbers of the calendric elements of $I^1_{G_A}$ and $i^{'}_1, \ldots, i^{'}_n$ are ordinal numbers of the calendric elements of $I^2_{G_B}$, respectively. It can be assumed without loss of generality that $m \geq n$. The following algorithm checks if $I^1_{G_A} \leq I^2_{G_B}$:

Algorithm 3.1 *Comparison of time instants:*

$Compare_1(I^1_{G_A}, I^2_{G_B})$
$I^1_{G_A} = (i_1, \ldots, i_m)$
$I^2_{G_B} = (i^{'}_1, \ldots, i^{'}_n)$
{
  $I^2_{G_B} := (i^{'}_1, \ldots, i^{'}_n, \underbrace{1, \ldots, 1}_{m-n});$
  for j **from** 1 **to** m {
    if $(i_j > i^{'}_j)$
      return `False`
  }
  return `True`
}

The algorithm basically compares the time instants by comparing each of their calendric elements. The instant $I^2_{G_B}$ is adjusted by adding the calendric element with the ordinal number 1[8] until its finest granularity is the same as that of $I^1_{G_A}$. This is reasonable because a time instant refers to the beginning of the time period it denotes.

Example 3.18 This example gives several time instants and the ordinal values of their respective calendric elements. The comparison algorithm is then illustrated by comparing different time instants.

[8]Calendric elements are counted starting from 1. Thus, in this work hours are 1-24 rather than 0-23. The same with minutes and seconds. This is an internal representation. The data will still be entered and printed "normally" in its external representation.

5 *June* 1990 $\equiv (1990, 6, 5)$; 15 *June* 1990 $\equiv (1990, 6, 15)$; *June* 1990 $\equiv (1990, 6)$; 1990 $\equiv (1990)$. Then, 5 *June* 1990 $<$ 15 *June* 1990 because $(1990, 6, 5) < (1990, 6, 15)$; *June* 1990 $<$ 15 *June* 1990 because $(1990, 6) \equiv (1990, 6, 1)$, and $(1990, 6, 1) < (1990, 6, 15)$; 1990 $<$ 15 *June* 1990 because $(1990) \equiv (1990, 1, 1)$, and $(1990, 1, 1) < (1990, 6, 1)$. □

Elapsed Time Between Time Instants

Let $(i_1, \ldots, i_n)$ and $(i'_1, \ldots, i'_n)$ be two time instants belonging to the same calendar. Then:

$$\texttt{Elapsed}((i_1, \ldots, i_n), (i'_1, \ldots, i'_n)) = \sum_{j=1}^{n} (K_j \cdot G_j), \text{where } K_j = i'_j - i_j$$

The following examples illustrate the various cases that can take place:

Example 3.19 `Elapsed`((13 *hour* 20 *min* 15 *sec* 29 *January* 1996), (13 *hour* 56 *min* 23 *sec* 29 *January* 1996)) $\Rightarrow$ (26 *minutes* 8 *seconds*)
This is the simplest case in which both instants have the same finest granularity. The calendric elements of the first time instant are simply subtracted from the corresponding calendric elements of the second time instant. In this example the time span (duration) of the acute chest pain in sentence $S2$ (see Section 3.3.1) for the generic patient was evaluated. Adding this time span to the result in Example 3.15 enables one to determine the global span of the symptoms of angina (see question 2 in Section 3.3.1). □

Example 3.20 `Elapsed`((15 *February* 1996), (16 *hour* 12 *April* 1996)) $\Rightarrow$
`Elapsed`((1 *hour* 15 *February* 1996), (16 *hour* 12 *April* 1996)) $\Rightarrow$ (1 *month* 27 *days* 15 *hours*)
Here, the finest calendric granularity of 15 *February* 1996 is coarser than that of 16 *hour* 12 *April* 1996. Thus, 15 *February* 1996 is first replaced by the time instant 1 *hour* 15 *February* 1996, its equivalent time instant with the finest granularity of *hour*. The elapsed time between 1 *hour* 15 *February* 1996 and 16 *hour* 12 *April* 1996 is then calculated as shown in the previous example. In this example the time span between the myocardial infarction and the last episode of angina (see sentences $S4$ and $S5$, and question 1 of the motivating example in Section 3.3.1) was evaluated. □

From the above two examples it can be noted that the `Elapsed` function returns a time span which comprises of all participating granularities in the two time instants. It may be desirable, however, to return a time span in some specified granularity or set of granularities. In this case, the `Elapsed` function can be extended to accept a third argument, namely, the granularities of the resulting span. If such an additional argument is omitted, the `Elapsed` function reverts to the default behavior described in the above two examples.

Operations Between Spans and Time Instants

In performing arithmetic operations that involve spans and time instants, there are two cases to consider:

- If the finest calendric granularity of the span is coarser than or the same as the finest calendric granularity of the instant, then each component of the span is simply added to the corresponding calendric element of the time instant.

  Example 3.21 Adding the time span 38 *months* to the time instant 30 *January* 1996, in order to know when the thrombolytic therapy (see sentence $S7$ and question 5 in Section 3.3.1) will finish, results in the time instant 30 *March* 1999. □

- If the finest calendric granularity of the span is finer than the finest calendric granularity of the time instant, then the time instant is first replaced by an equivalent time instant whose finest granularity is the same as that of the span, and the addition is carried out.

  Example 3.22 To know when the the first episode of chest pain ended (see sentence $S1$ in Section 3.3.1), the time span 2 *hours and* 55 *minutes* has to be added to the time instant 13 *December* 1995. In this case the time instant is first replaced by its equivalent time instant 1 *hour* 1 *min* 13 *December* 1995. The addition of the time span to this time instant results in the time instant 3 *hour* 56 *min* 13 *December* 1995[9] □

3.3.3.3 Related Work

Most of the research on temporal relational models has concentrated on modeling temporal information with a single underlying granularity. There have been some recent proposals, however, that handle multiple granularities.

Clifford and Rao [CC87] introduce a general structure for time domains called a *temporal universe* which consists of a totally ordered set of granularities. Operations are defined on a temporal universe, which basically convert different *anchored* times to a (common) finer granularity before carrying out the operation. Wiederhold et al., [WJL91] also examine the issue of multiple granularities. An algebra is described that allows the conversion of event times to an interval representation. This involves converting the coarser granularity to the finer granularity in light of the semantics of the time varying domains. [WJS93] extend this work by providing semantics for moving up and down a granularity lattice. In [BP85] the issues of absolute, relative, imprecise, and periodic times are discussed. Multiple granularities are supported for each time. Operands (which are anchored) in operations involving mixed granularities are converted to the coarser granularity to avoid indeterminacy. In a more recent work [MPB92], the existence of a minimum underlying granularity (quantum of time) to which time is mapped, is assumed. Montanari et al., [MMCR92] examined the issue of multiple granularities, but considered exact granularity conversions only. Corsetti et al., [CMR91] deal with different time granularities in specifications of real-time systems.

[9]Note that this time instant is given in the internal representation. The external representation of the time instant would be 2 *hour* 55 *min* 13 *December* 1995.

In the above works, granularities are treated as *anchored* partitionings on the timeline whereas in this work, granularities are unanchored since they are modeled as unit spans. Hence, while *both* anchored and unanchored granularity conversions can be performed in this work, the proposals above consider *only* anchored granularity conversions. Furthermore, none of the models, with the exception of [MPB92], explicitly address the notion of indeterminacy.

TSQL2 [Sno95b] treats all instants as indeterminate at finer granularities. In contrast, the treatment of time instants in this work depends on the interpretation given to the time instant. This is illustrated in Table 3.5. In this thesis, indeterminacy in time instants is not in the conversion to finer granularities, but it is in the interpretation of the time instant. Furthermore, even when indeterminacy arises (in the case of indeterminate time instants), it is at the granularity at which the time instant is defined. This is due to the mapping of indeterminate instants to intervals (that was defined in Section 3.3.3.1 which have beginning instants as their lower and upper bounds.

In TSQL2, the semantics of arithmetic operations is not explicitly supported. More specifically, the semantics of arithmetic operations which involve time spans and time instants are left to the calendar as calendar specific operations. These include $instant + span$, $span + instant$, and $instant - instant$. This is partly due to ambiguities which arise from these operations. Consider the operation $1\ month + 15\ June$ 1995 in TSQL2. Here, according to the [left operand] semantics of TSQL2, the following would happen:

$$\begin{aligned} &1\ month + 15\ June\ 1995 \\ &\qquad\qquad \rightarrow 1\ month + June\ 1995 \\ &\qquad\qquad \rightarrow July\ 1995 \end{aligned}$$

The time instant 15 $June$ 1995 is first scaled to the coarser granularity of $month$ (which is the granularity of the left operand), resulting in the time instant $June$ 1995. The addition operation then results in the time instant $July$ 1995, which is not what one would intuitively expect. In this work (see Section 3.3.3.2), the addition operation would return the expected time instant 15 $July$ 1995. Basically, the semantics of arithmetic operations is not explicitly supported in TSQL2. It is left to the calendar. Therefore, if $month + DATE = DATE$ is not supported by the calendar, then the operation $1\ month + 15\ June$ 1995 in TSQL2 will result in $July$ 1995 [Sno96]

3.3.3.4 Mapping to TIGUKAT

The TIGUKAT types and behaviors for anchored temporal primitives are similar to the corresponding concrete anchored primitive types shown in Figure 2.3. For example, the type `T_instant` is defined in TIGUKAT an an abstract type of all time instants. The behaviors defined in TIGUKAT on the type `T_instant` are shown in Table 3.6. These include the comparison behaviors *B_less*, *B_greater*, *B_leq* and *B_geq* (these are essentially the $<$, $>$, $\leq$ and $\geq$ operators, respectively), the *B_elapsed* behavior which returns the elapsed

`T_instant`	*B_less*:	`T_instant` → `T_boolean`
	B_greater:	`T_instant` → `T_boolean`
	B_leq:	`T_instant` → `T_boolean`
	B_geq:	`T_instant` → `T_boolean`
	B_elapsed:	`T_instant` → `T_span`
	B_add:	`T_span` → `T_instant`
	B_subtract:	`T_span` → `T_instant`
	B_calendar:	`T_calendar`
	B_calElements:	T_list⟨T_calElement⟩

Table 3.6: Behaviors defined on time instants.

time (duration) between two time instants, and the *B_add* and *B_subtract* which are used in arithmetic operations between time instants and time spans. Behavior *B_calendar* returns the calendar which the instant belongs to and behavior *B_calElements* returns a list of the calendric elements in a time instant. For example, *B_calElements* applied to the instant June 15, 1995 returns the list $(1995, 6, 15)$. Other types and behaviors are similar to those given in Figure 2.3.

There is also one additional instant type `T_specialInstant` defined as a subtype of `T_instant`. The only instances of this type are $-\infty$ and $+\infty$. These instances are used as minimum and maximum bounds of a timeline as will be discussed in Section 3.4.

3.3.4 Implementation Issues

The formulae (3.3) and (3.4) for $lbf(K, G_A, G_B)$ and $ubf(K, G_A, G_B)$ (see Derivation 3.2) are computationally expensive. However, they are not designed for direct computation. These formulae are just mathematical definitions. Any approximation of these formulae will suffice, and such approximations for the most common conversions can be chosen at the time when the calendar is defined. Another technique that can be used to make computations less expensive would be to simplify these formulae since they allow for many simplifications once a set of particular calendric functions is chosen. As an example, consider the Gregorian calendar. In this calendar,

$$\begin{aligned} f^{\text{year}}(y) &= 12 \text{ (months)} \\ f^{\text{month}}(y, m) &= \begin{cases} 30 & \text{if } m \text{ is } 3,\ 5,\ 8, \text{ or } 10 \\ 31 & \text{if } m \text{ is } 0,\ 2,\ 4,\ 6,\ 7,\ 9, \text{ or } 11 \\ 28 & \text{if } m = 1 \text{ and } y \text{ is not leap} \\ 29 & \text{if } m = 1 \text{ and } y \text{ is leap} \end{cases} \quad \text{(days)} \\ f^{\text{day}}(y, m, d) &= 24 \text{ (hours)} \end{aligned}$$

where y is leap when $y \bmod 400 = 0 \vee y \bmod 4 = 0 \wedge y \bmod 100 \neq 0$.

Consider the conversions from years to months, from years to days, and from months to days. Then,

$$\begin{aligned}
f^{\text{year}\to\text{month}}(y) &= f^{\text{year}}(y) = 12 \\
f^{\text{year}\to\text{day}}(y) &= \begin{cases} 366 & \text{if } y \text{ is leap} \\ 365 & \text{otherwise} \end{cases} \\
f^{\text{month}\to\text{day}}(y, m) &= f^{\text{month}}(y, m)
\end{aligned}$$

Then, using formulae (3.3) and (3.4) it is found that

$$\begin{aligned}
lbf(K, G_{\text{year}}, G_{\text{month}}) &= \min_y \{ \sum_{0 \le dist_{\text{year}}(y', y) \le K-1} f^{\text{year}\to\text{month}}(y') \} \\
&= \min_y \{12K\} \\
&= 12K \\
ubf(K, G_{\text{year}}, G_{\text{month}}) &= 12K \\
lbf(K, G_{\text{year}}, G_{\text{day}}) &= \min_y \{365(y+K) + \lfloor (y+K)/4 \rfloor - \lfloor (y+K)/100 \rfloor + \lfloor (y+K)/400 \rfloor \\
&\quad - 365y - \lfloor y/4 \rfloor + \lfloor y/100 \rfloor - \lfloor y/400 \rfloor \} \\
&\ge 365K + \lfloor K/4 \rfloor - \lfloor (K+96)/100 \rfloor \\
ubf(K, G_{\text{year}}, G_{\text{day}}) &\le 365K + \lfloor (K+3)/4 \rfloor
\end{aligned}$$

The above lbf and ubf bounds can be used instead of exact formulae. These bounds are easily computable and introduce an error that is less than a day per century. Analogous methods can be used to find computationally cheap approximations for conversion of months to days; however, to obtain reasonable approximations, values for small K ($K \le 48$) have to be tabulated. Let $g_{min}(K)$ and $g_{max}(K)$ be such tabulations. Then:

$$\begin{aligned}
lbf(K, G_{\text{month}}, G_{\text{day}}) &= g_{min}(K \bmod 48) + lbf(\lfloor K/48 \rfloor \cdot 4, G_{\text{year}}, G_{\text{day}}) \\
ubf(K, G_{\text{month}}, G_{\text{day}}) &= g_{max}(K \bmod 48) + ubf(\lfloor K/48 \rfloor \cdot 4, G_{\text{year}}, G_{\text{day}})
\end{aligned}$$

Using these formulae one can now make fast and quite precise conversions. For example, the number of days (d) in 100 months according to the above formulae is $120 + 2921 = 3041 \le d \le 3044 = 122 + 2922$, which is the correct estimate. The simplistic approach where the number of months is not taken into account when the coefficients are computed would give $28 * 100 = 2800 \le d \le 3100 = 31 * 100$, which is an error of more than 8%, or 200 days.

3.4 Temporal Order

In the TIGUKAT temporal model, the notion of a temporal order (defined in Section 2.3.1.3) is enhanced by introducing the concept of a *timeline*. A timeline represents an axis over which time can be perceived in an ordered manner. Basically, timelines are used to give an order to the timestamps in histories (described in Section 3.5).

Definition 3.13 *Timeline ($\mathcal{T_L}$):* A timeline $\mathcal{T_L}$ is a triplet $\langle \mathcal{O}, \mathcal{I}, \mathbf{L\_}\mathcal{T} \rangle$, where $\mathcal{O}$ is the temporal order of $\mathcal{T_L}$, $\mathcal{I}$ is the defining time interval of $\mathcal{T_L}$, and $\mathbf{L\_}\mathcal{T}$ is a collection of the timestamps (time intervals) which belong to $\mathcal{T_L}$. ■

The different temporal orders of a timeline can be classified as being *linear* or *branching* (described in Section 2.3.1.3).

Definition 3.14 *Sub-linear Order:* Let t_i and t_j be anchored timestamps. Then,
$\forall t_i t_j (t_i \texttt{ overlaps } t_j \vee t_i \texttt{ precedes } t_j \vee t_j \texttt{ precedes } t_i)$ ■

Definition 3.15 *Linear Order:* Let t_i and t_j be anchored timestamps. Then,
$\forall t_i t_j (t_i \texttt{ precedes } t_j \vee t_j \texttt{ precedes } t_i \vee t_i \texttt{ equals } t_j)$ ■

In a sub-linear order, timestamps are allowed to overlap each other while in a linear order, timestamps strictly follow or precede each other, i.e., they do not overlap.

In a branching order, time is linear in the past up to a certain point, at which it branches out into alternate futures.

Definition 3.16 *Branching Order:* Let t_i, t_j and t_k be anchored timestamps. Then,
$\forall t_i t_j t_k ((t_j \texttt{ precedes } t_i \wedge t_k \texttt{ precedes } t_i) \rightarrow (t_j \texttt{ precedes } t_k \vee t_j \texttt{ overlaps } t_k \vee t_k \texttt{ precedes } t_j))$
■

The branching order defined above is a forward (in time) branching order. It ensures the two predecessors of a given time are comparable. Without loss of generality, in the rest of this chapter timelines with a linear order will be referred to as *linear timelines* while those with a branching order will be referred to as *branching timelines*.

Definition 3.17 *Defining time interval ($\mathcal{I}$):* A defining time interval $\mathcal{I} = [t_s, t_e]$ is a time interval over which a timeline $\mathcal{T_L}$ is defined. t_s is a time instant denoting the start time of $\mathcal{T_L}$ and t_e is a time instant denoting the end time of $\mathcal{T_L}$. ■

A timeline is comprised of one defining time interval $\mathcal{I}$ which essentially determines the size of a timeline. Two constants (which are essentially instances of `T_specialInstant`) $-\infty$ and $+\infty$ are defined to be the lower (t_s) and upper (t_e) bounds of the longest defining time interval time of which a timeline could be comprised.

A timeline may then contain any number of additional time intervals and time instants with the restriction that each of them lies within the defining time interval $\mathcal{I}$. Over the lifetime of the timeline, $\mathcal{I}$ remains the same, but other additional time intervals and time instants may be added, deleted or modified. For example, consider a timeline with an $\mathcal{I}$ $[08:00\ January\ 1\ 1993,\ +\infty)$. Time intervals or time instants (which could be modeling the history of a particular object) such as $[January\ 15\ 1993,\ February\ 20\ 1995]$, $March\ 27\ 1995,\ 00:$

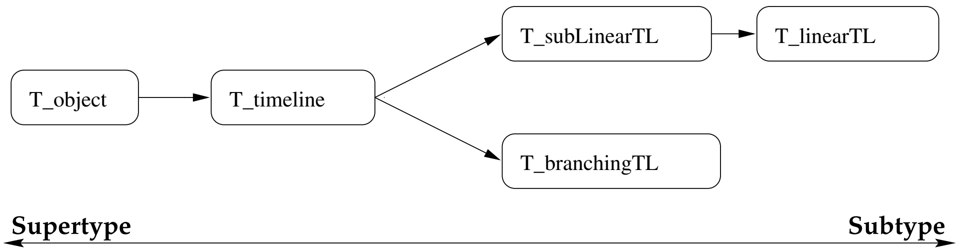


Figure 3.6: The timelines type hierarchy.

$00:03\ May$ 25 1995 can now be added to the timeline. These time intervals and instants form the collection $\mathbf{L}\_\mathcal{T}$.

Since $\mathbf{L}\_\mathcal{T}$ can consist of time intervals or time instants or both, both *homogeneous* and *heterogeneous* timelines can be defined. This provides additional flexibility in defining histories of various activities and objects because some activities occur at moments in time, while others occur over a period of time. Figure 3.6 shows the type hierarchy of the types used to model the different kinds of timelines described in this section. The abstract type `T_timeline` (corresponding to the `T_temporalOrder` type shown in Figure 2.7) is first defined as a supertype of all linear and branching timelines, and defines the following behaviors:

$$\begin{aligned} B\_definingTimeInterval: &\quad \texttt{T\_interval} \\ B\_timeIntervals: &\quad \texttt{T\_collection} < \texttt{T\_interval} > \end{aligned}$$

B_definingTimeInterval returns the defining time interval of a timeline, while *B_timeIntervals* returns a collection of time intervals that have a certain temporal order (this behavior is similar to the *P_temporalPrimitives* property defined on the `T_temporalOrder` type). The rest of the types shown in Figure 3.6 are instantiations of the corresponding subtypes of `T_temporalOrder` of the temporal framework.

3.5 Temporal History

One requirement of a temporal model is an ability to adequately represent and manage histories of real-world events. In this section the types and behaviors in TIGUKAT that can be used to model histories of real-world events are described. These are then used to uniformly model valid and transaction histories in TIGUKAT.

3.5.1 Real-World Event Histories

The approach of modeling histories in TIGUKAT temporal model makes use of the availability of parametric types in the underlying object model. This allows one to model histories that adhere more closely to the real-world.

The type T_history of the temporal framework is used as an abstract type of all histories modeled in TIGUKAT. Temporal histories of real-world objects whose type is T_X are then represented in the TIGUKAT temporal model as objects of the T_history⟨T_X⟩ type, which is a subtype of T_history (see Figure 3.7). For example, suppose a behavior *B_salary* is defined in the T_employee type. Now, to keep track of the changes in salary of employees, *B_salary* would return an object of type T_history⟨T_real⟩ which would consist of the different salary objects of a particular employee and their associated time periods.

A temporal history consists of objects and their associated timestamps (time intervals or time instants). One way of modeling a temporal history would be to define a behavior that returns a collection of <**timestamp**, **object**> pairs. However, instead of structurally representing a temporal history in this manner, a behavioral approach is used by defining the notion of a *timestamped object*. A timestamped object knows its timestamp (time interval or time instant) and its associated value at (during) the timestamp. A temporal history is made up of such objects. The following behaviors, similar to the properties defined on T_history of the temporal framework, are defined on the T_history type in TIGUKAT:

$$\begin{array}{rl} B\_history: & \texttt{T\_collection}\langle\texttt{T\_timeStampedObject}\langle\texttt{T\_X}\rangle\rangle \\ B\_timeline: & \texttt{T\_timeline} \\ B\_insert: & \texttt{T\_X}, \texttt{T\_anchPrim} \rightarrow \\ B\_remove: & \texttt{T\_X}, \texttt{T\_anchPrim} \rightarrow \\ B\_validObjects: & \texttt{T\_anchPrim} \rightarrow \texttt{T\_collection}\langle\texttt{T\_timeStampedObject}\langle\texttt{T\_X}\rangle\rangle \\ B\_validObject: & \texttt{T\_anchPrim} \rightarrow \texttt{T\_timeStampedObject}\langle\texttt{T\_X}\rangle \end{array}$$

Behavior *B_history* returns the set (collection) of all timestamped objects that comprise the history. A history object also knows the timeline it is associated with and this timeline is returned by the behavior *B_timeline*. The timeline basically orders the timestamps of timestamped objects (see Section 3.4). The *B_insert* behavior accepts an object and an anchored timestamp as input and creates a timestamped object that is inserted into the history. Behavior *B_remove* drops a given object from the history at a specified anchored timestamp. The *B_validObjects* behavior allows the user to get the objects in the history that were valid at (during) a given anchored timestamp. Behavior *B_validObject* is derived from *B_validObjects* to return the timestamped object that exists at a given time instant.

Each timestamped object is an instance of the T_timeStampedObject⟨T_X⟩ type. This type represents objects and their corresponding timestamps. Behaviors *B_value* and *B_timeStamp* defined on T_timeStampedObject return the value and the timestamp (time interval or time instant) of a timestamped object, respectively.

$$\begin{array}{rl} B\_value: & \texttt{T\_X} \\ B\_timeStamp: & \texttt{T\_anchPrim} \end{array}$$

3.5.2 Valid and Transaction Time Histories

To represent valid, transaction, and event time histories in the TIGUKAT temporal model, temporality is associated with class objects. The T_temporalClass<T_X> type is introduced as a subtype of the primitive type T_class<T_X> to manage temporal information of objects. An instance of T_temporalClass<T_X> (or any of its subtypes) is called a *temporal class*, and so objects belonging to a temporal class are called *temporal objects*.

TIGUKAT treats everything as objects; consequently classes are objects too. It therefore makes sense to distinguish between the notions of *class* temporality and *object* temporality. An object that is not a class can be either temporal or non-temporal whereas a class object may be temporal as a class or temporal as an object[10].

Definition 3.18 *Temporality of Objects:* An object o is temporal as an object if and only if the class of o is a temporal class. ■

Definition 3.19 *Temporality of Classes:* A class object o is temporal as a class if and only if its type is a subtype of T_temporalClass. ■

These definitions imply that temporality of objects in TIGUKAT is not orthogonal to their class. If a class is temporal, then all of its members are temporal; if a class is non-temporal, then none of its members is temporal.

Type T_temporalClass defines additional functionality for representing the semantics of the temporality of objects. It allows its instances (which are temporal classes) to maintain histories of their constituent objects. The T_temporalClass type defines three behaviors to model valid, transaction, and event time histories independently of one another as follows:

$$\begin{aligned} B\_validHistory: &\quad \texttt{T\_history} < \texttt{T\_X} > \\ B\_transHistory: &\quad \texttt{T\_history} < \texttt{T\_X} > \\ B\_eventHistory: &\quad \texttt{T\_history} < \texttt{T\_X} > \end{aligned}$$

Behaviors *B_validHistory*, *B_transHistory*, and *B_eventHistory* return the valid, transaction, and event time history of an object belonging to the type T_X, respectively. Modeling the different kinds of histories using behaviors conforms to the behavioral nature of the TIGUKAT object model. Instead of using the temporal framework types that model the different kinds of histories, histories in TIGUKAT are modeled using behaviors defined on T_temporalClass.

The approach of separating valid and transaction times into two behaviors is in contrast to other temporal object models (for example, [RS91]) where they are structurally modeled together. This usually requires an object value to have corresponding entries for *both* the

[10] The same word (*temporal*) is used for both these notions since it is usually clear from the context which one of the two is being referred to. In the rest of this section, unless otherwise specified, *temporal class* will mean the instances of the class are temporal.

valid and transaction times. The separation in modeling of valid and transaction times is more intuitive and gives substantial flexibility as it allows different types of object databases to be defined as per application needs [Sno87]:

- *Rollback* or *transaction-time* object databases can be modeled using the *B_transHistory* behavior. This facilitates the state of the object database to be seen *as of* some particular time.

- *Historical* object databases can be modeled using the *B_validHistory* behavior. This shows the time when the stored historical information was valid.

- *Temporal* object databases encompass the functionalities of rollback and historical object databases and are, therefore, modeled using both the *B_validHistory* and *B_transHistory* behaviors.

Example 3.23 Suppose the hospital staff wants to maintain a record of the time when operations in the hospital took place. In this case, the class of all operations (**C_operation**) will be a temporal class with type `T_temporalClass<T_operation>`. **C_operation** will then consist of temporal objects having the semantics of different operations and the times they took place. The following behavior returns the history of different operations that took place in the hospital:

$$\textbf{C\_operation}.B\_validHistory$$

Applying the behavior *B_history* to this object returns a collection consisting of timestamped operations. If one were also interested in the times when operations in the hospital were entered in the object database, then one would simply use the *B_transHistory* behavior to access these. □

Figure 3.7 shows the type hierarchy of the history types described in this section.

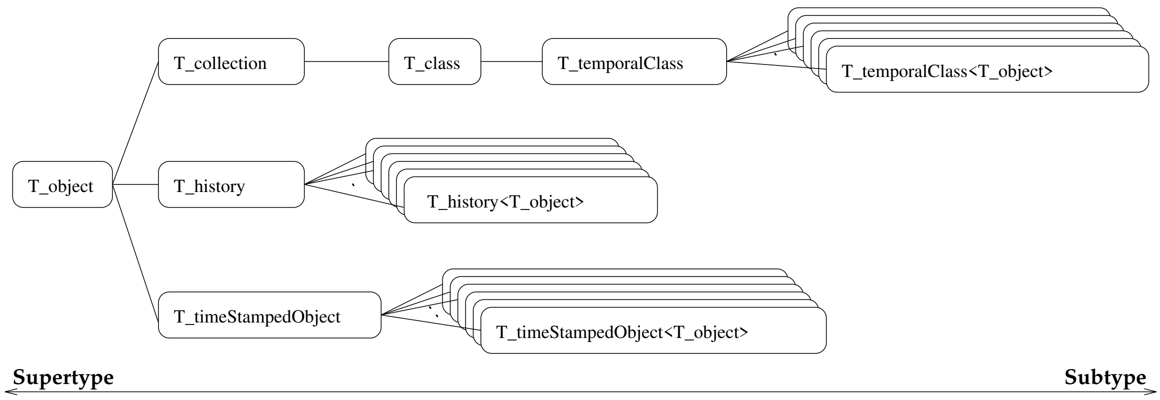


Figure 3.7: The temporal histories type hierarchy.

3.6 A Medical Trial Object Database

In this section, a pharmacoeconomics study (briefly described in Example 2.4) is described which makes use of the timeline and history features of the TIGUKAT temporal model that were presented in Sections 3.4 and 3.5. The rest of the section shows these features can be used to represent, store, analyze, and reason about the components of the study.

3.6.1 Medical Trials in Pharmacoeconomics

One of the methods used in the field of pharmacoeconomics is cost-effectiveness analysis of different treatments [JJGB92]. This method has been used in the pharmacoeconomic analysis of treatments for illnesses such as pneumonia, asthma, chest infection, etc. In these trials, the group of patients suffering from the illness of interest are divided into sub-groups. Each of these sub-groups is administered one of the different treatments under investigation. Pharmacoeconomic analysis has so far mainly focussed on the comparison of different drugs used for treating the same illness. During the course of the trial, an object database of information is maintained which will be used in the cost-effectiveness analysis of each treatment. To illustrate, consider a trial comparing two different antibiotic treatments. The information required would include:

- *Antibiotic Related Costs* – The different antibiotics used and the time period during which they were applied. For each antibiotic, the costs related to its use are also recorded. These include:
  - Acquisition cost
  - Preparation and administration costs
  - Laboratory monitoring costs
  - Cost of treating adverse effects
- *Infection Related Costs* – These include:
  - Laboratory monitoring tests (for example, microbiology blood tests and radiology tests), the times the patient took the tests, and the cost incurred for each test
  - Health care used (this includes emergency room visits, hospital bed costs, etc.) and related costs

In addition to this information, a medical trial has various temporal modeling requirements. These include:

- A branching model of time in which histories of alternate treatments of a medical trial could be represented, and subsequently analyzed for their cost-effectiveness.
- The different kinds of medication and the time periods during which they were administered during the course of a particular treatment in the medical trial.

- The time periods during which different treatments in the medical trial were administered.
- Whether a patient in the current medical trial underwent a similar medical trial in the past.
- The different blood tests a patient took while undergoing a particular treatment.

The following sections show how a medical trial can effectively be modeled in the TIGUKAT temporal ODBMS. Analysts working in pharmacoeconomics can then use the temporal ODBMS to retrieve components relevant to their line of investigation to aid their pharmacoeconomic analyses.

3.6.2 Medical Trial Types and Behaviors

This section describes all the necessary types and behaviors which are needed to model a pharmacoeconomic trial (such as that described in Section 3.6.1) in the TIGUKAT temporal model. Figure 3.8 shows the types used to model the various components of a medical trial. The behaviors of the types are given in Table 3.7.

The `T_medicalTrial` type is introduced to represent different alternative treatments in a medical trial. For example, treatmentA and treatmentB described in Example 2.4 are objects of type `T_medicalTrial` and represent the different treatments used in a medical trial. Since a medical trial is comprised of alternate treatments which take place during a time period, its semantics is best captured by a branching timeline as was shown in Example 2.4. The *B_timeline* behavior defined on `T_medicalTrial` returns the timeline that is associated with the medical trial as is depicted in Figure 2.10. Each treatment then has the same timeline. A treatment also has a collection of patients and antibiotics. To model these, `T_medicalTrial` defines the behaviors *B_patients* and *B_antibiotics*, respectively. For a given treatment, the *B_treatmentPeriod* behavior returns the time period during which the treatment took place. The illness for which the medical trial is being undertaken is given by the behavior *B_illness*.

B_antibiotics returns a collection of timestamped antibiotics. Each member of such a collection has an associated timestamp which returns the time period during which the antibiotic was administered. In addition to the timestamp, each member also has an antibiotic object whose type is `T_antibiotic`. This type defines several behaviors which returns the various costs (outlined in Section 3.6.1) associated with an antibiotic. These behaviors are shown in Table 3.7.

The type `T_patient` represents the patients undergoing the medical trial. In the course of a treatment, a patient takes various blood tests and radiology tests. These are represented by the type `T_test`. Behaviors *B_bloodTests* and *B_radiologyTests* are defined on `T_patient` to return the history of tests taken by a patient as shown in Table 3.7. Using histories to model a patient's tests enables one to look up the times when particular tests were taken.

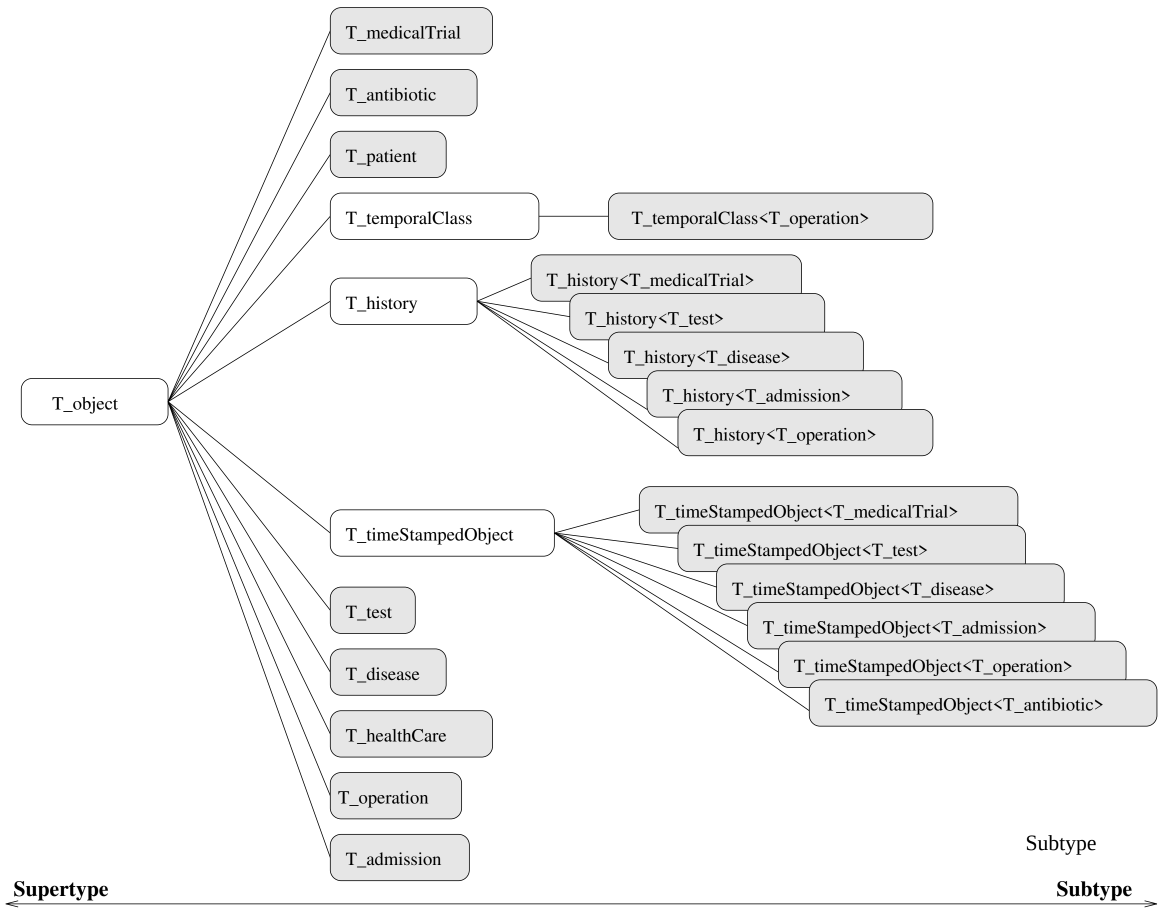


Figure 3.8: The type hierarchy for a medical trial.

Type `T_test` defines the *B_cost* behavior which returns the cost associated with a particular treatment.

The behavior *B_diseases* defined on `T_patient` returns the disease history of a patient. It enables one to find out what diseases a patient has and at what times they were diagnosed. The *B_medicalTrials* behavior gives the history of all the different medical trials the patient has gone through up to the current date. Finally, the *B_healthCare* behavior returns the health care object of a patient.

The `T_healthCare` type represents the prescribed medical treatment of a patient. Behavior *B_bedCost* returns the bed cost of a patient during the period of a given treatment of a medical trial. *B_emergencyVisits* returns the number of emergency visits of a patient during the period of a given treatment of a medical trial. Finally, behaviors *B_admissions* and *B_operations* give the hospital admission and operation histories of a patient, respectively. Using histories to model a patient's hospital admissions enables one to easily determine when the patient was admitted and when he/she was discharged, and the number of times the patient has been admitted to the hospital. Similarly, modeling a patient's operations using histories gives us the start and end times of each operation, and the number of operations.

Type	Signatures	
T_medicalTrial	*B_timeline*:	T_branchingTL
	B_antibiotics:	T_collection⟨T_timeStampedObject⟨T_antibiotic⟩⟩
	B_patients:	T_collection⟨T_patient⟩
	B_treatmentPeriod:	T_interval
	B_illness:	T_disease
T_antibiotic	*B_acquisitionCost*:	T_real
	B_prepAdmCost:	T_real
	B_labCost:	T_real
T_patient	*B_bloodTests*:	T_history⟨T_test⟩
	B_radiologyTests:	T_history⟨T_test⟩
	B_diseases:	T_history⟨T_disease⟩
	B_medicalTrials:	T_history⟨T_medicalTrial⟩
	B_healthCare:	T_healthCare
T_test	*B_cost*:	T_real
T_healthCare	*B_bedCost*:	T_medicalTrial → T_real
	B_emergencyVisits:	T_medicalTrial → T_integer
	B_admissions:	T_history⟨T_admission⟩
	B_operations:	T_history⟨T_operation⟩

Table 3.7: The medical trial types and behaviors.

3.6.3 A Medical Trial Instance

Having defined the types and behaviors used to model a medical trial, Figure 3.9 shows a pictorial view of one of the treatments (treatmentA) in a medical trial. The pictorial view shows how, starting from treatmentA, various components of a treatment (as outlined in Section 3.6.1) can be reached using the behaviors defined in Table 3.7.

Figure 3.9 shows that treatmentA has a branching timeline medicalTrialBranchingTL which is shared by all other treatments. During the course of treatmentA, two antibiotics, antibioticA and antibioticB were used. antibioticA was administered during the time interval [January 1, 1995, April 1, 1995) and was followed by antibioticB which was administered for the time period [April 1, 1995, July 1, 1995). During the time period they were used, antibioticA incurred an acquisition cost of 45.0, a laboratory cost of 24.0, and a preparation and administration cost of 53.50, while antibioticB incurred costs of 55.0, 37.25, and 77.50, respectively.

patient1 was one of the patients who went through treatmentA. The previous medical trials that patient1 went through are given by medicalTrialHistory. Similarly, patient1's history of diseases is given by diseaseHistory. The various monitoring tests taken by patient1 are given by radiologyTestHistory and bloodTestHistory. Figure 3.9 shows that patient1 had three blood tests done; hematology1 and microbiology were done on January 15, 1995 and incurred costs of 64.50 and 43.50, respectively. hematology2 was done on February 20, 1995 and incurred a cost of 70.0.

Finally, patient1HealthCare gives the prescribed medical treatment of patient1. Using the hospital bed during the course of treatmentA incurred a cost of 550.0. patient1 had 2 visits to the emergency room while undergoing treatmentA. A history of patient1's hospital admissions is given by admissionHistory while the different operations he went through are given by operationHistory.

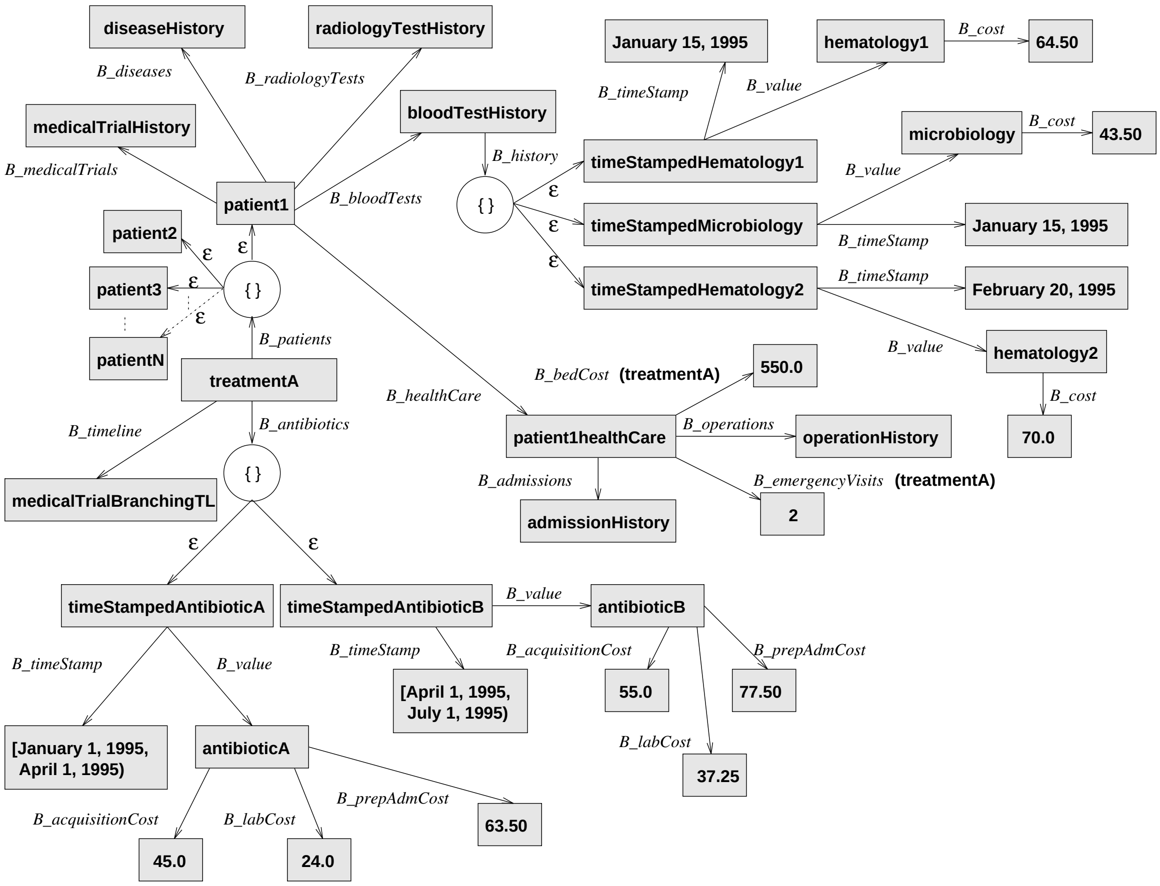


Figure 3.9: A pictorial representation of the components of a medical trial.

Researchers working on the pharmacoeconomic analysis of different treatments can now use the objects, types and behaviors shown in Figure 3.9, Figure 3.8 and Table 3.7 to retrieve costs of antibiotics, monitoring tests, hospital bed, etc. and the time period during which they occurred. They can then use the retrieved information to carry out comparisons of the costs and effectiveness of alternative treatments. The following section shows how queries can be constructed in TIGUKAT to retrieve the various components of a medical trial.

3.6.4 Example Queries

3.6.4.1 The TIGUKAT Query Language

The TIGUKAT query model is a direct extension to the object model. It is defined by type and behavioral extensions to the primitive model. The languages for the query model include a complete calculus, and equivalent object algebra and a SQL-like user language [PLÖS93]. This sub-section briefly discusses the TIGUKAT Query Language (TQL) and demonstrates how it can be used to access objects of a medical trial. TQL[11] is based on the SQL paradigm [Dat87] and its semantics is defined in terms of the object calculus. Hence,

[11]TQL was developed before the release of OQL [Cat94]. It is quite similar to OQL in structure.

every statement of the language corresponds to an equivalent object calculus expression. The basic query statement of TQL is the *select statement* which operates on a set of input collections, and returns a new collection as the result:

select $<$ *object variable list* $>$
[**into** $<$ *collection name* $>$]
from $<$ *range variable list* $>$
[**where** $<$ *boolean formula* $>$]

The *select clause* in this statement identifies the objects to be returned in a new collection. There can be one or more object variables with different formats (constant, variables, path expressions or index variables) in this clause. They correspond to free variables in object calculus formulas. The *into clause* declares a reference to a new collection. If the *into clause* is not specified, a new collection is created; however, there is no reference to it. The *from clause* declares the ranges of object variables in the *select* and *where* clauses. Every object variable can range over either an existing collection, or a collection returned as a result of a subquery, where a subquery can be either given explicitly, or as a reference to a query object. The *where clause* defines a boolean formula that must be satisfied by objects returned by a query. Two additional predicates are added to TQL boolean formulas to represent existential and universal quantification. The existential quantifier is expressed by the *exists predicate* which is *true* if the referenced collection is not empty. The universal quantifier is expressed by the *forAll predicate* which is *true* if for every element in every collection in the specified range variable list, the given boolean formula is satisfied.

Having described TQL, the next section shows how temporal objects can uniformly be queried using behavior applications without changing any of the basic constructs of TQL.

3.6.4.2 Query Examples

To make the queries easier to read, assume that treatmentA exists in the object database and is a member of the **C_medicalTrial** class. That is, the presence of the "*treatmentA* **in C_medicalTrial**" construct is assumed in the *from clause* of each query.

Example 3.24 Which antibiotics were administered in treatment A and during what times?
select $timeStampedAntibiotic.B\_value$, $timeStampedAntibiotic.B\_timeStamp$
from $timeStampedAntibiotic$ **in** treatmentA.$B\_antibiotics$
This query simply goes through the collection of timestamped antibiotics of treatmentA and returns the antibiotic and timestamp associated with each timestamped antibiotic in the collection. As seen in Figure 3.9, this query would return a collection containing antibioticA, antibioticB and their associated timestamps (time intervals). □

Example 3.25 What was the total cost of using antibiotic A?
select $antibioticACost$

from $antibioticACost$ **in C_real**, $timeStampedAntibiotic$ **in** treatmentA.$B\_antibiotics$,
$antibioticA$ **in C_antibiotic**
where $(antibioticA = timeStampedAntibiotic.B\_value)$ **and**
$(antibioticACost = antibioticA.B\_acquisitionCost.B\_add(antibioticA.B\_labCost).$
$B\_add(antibioticA.B\_prepAdmCost))$
In this query, the acquisition, laboratory, and preparation and administration costs of antibiotic A are added to return the total cost, which would be 132.50 according to Figure 3.9.
□

Example 3.26 Which antibiotics had an acquisition cost of more than $50?
select $timeStampedAntibiotic.B\_value$
from $timeStampedAntibiotic$ **in** treatmentA.$B\_antibiotics$
where $timeStampedAntibiotic.B\_value.B\_acquisitionCost > 50$
This query goes through the collection of timestamped antibiotics of treatmentA and returns the collection of antibiotics whose acquisition cost is greater than 50. From Figure 3.9, it can be seen that a collection containing only antibioticB is returned as a result of this query.
□

Example 3.27 Which blood tests did patient 1 take while antibiotic A was being administered, and what were the associated costs?
select $timeStampedbloodTest.B\_value, timeStampedbloodTest.B\_value.B\_cost$
from $patient1$ **in** treatmentA.$B\_patients$, $timeStampedbloodTest$ **in** $patient1.B\_bloodTests.$
$B\_history$, $timeStampedAntibiotic$ **in** treatmentA.$B\_antibiotics$
where $(antibioticA = timeStampedAntibiotic.B\_value)$ **and**
$(timeStampedbloodTest.B\_timeStamp.B\_within(timeStampedAntibiotic.B\_timeStamp))$
This query returns patient1's blood tests (with their associated costs) whose timestamps fell within the time interval during which antibioticA was being used. From Figure 3.9, it can be seen that these blood tests were hematology1, microbiology, and hematology2. □

Example 3.28 What was the time period during which treatment A took place?
select treatmentA.$B\_treatmentPeriod$
This query returns a collection consisting of the time interval during which treatmentA took place. Assuming treatmentA ends when no more antibiotics are administered, an alternative approach would be to construct the query such that it goes through the collection of the timestamped antibiotics of treatmentA and takes the union of the timestamps associated with each antibiotic in the collection. Then, for the instance given in Figure 3.9, the query would return the time interval [January 1, 1995, July 1, 1995). □

Example 3.29 How many alternative treatments took place in the medical trial in which treatment A was administered?
select treatmentA.$B\_timeline.B\_branches.B\_cardinality$

Since treatmentA has a branching timeline associated with it, the number of alternative treatments in the medical trial is simply the number of branches of the branching timeline. This is obtained by taking the cardinality of the collection returned as a result of the treatmentA.*B_timeline*.*B_branches* behavior application. If the medical trial consisted of treatment A and treatment B, then for the instance in Figure 3.9, this query returns a collection containing 2. □

Example 3.30 Has patient 1 undergone a medical trial for the same illness before?
select $timeStampedMedicalTrial.B\_timeStamp$
from $patient1$ **in** treatmentA.$B\_patients$, $timeStampedMedicalTrial$ **in**
$patient1.B\_medicalTrials.B\_history$
where $timeStampedMedicalTrial.B\_value.B\_illness.B\_equal$(treatmentA.*B_illness*)
This query goes through patient1's medical trial history and checks if there exists an illness which is the same as the one for which treatmentA is being administered. □

Example 3.31 How many patients went through treatment A?
select treatmentA.$B\_patients.B\_cardinality$
The number of patients in treatmentA is the cardinality of the collection returned as a result of the treatmentA.*B_patients* behavior application. □

Chapter 4

Schema Evolution

In this chapter[1], the issues of schema evolution and temporal object models are addressed. These two issues are generally considered to be orthogonal and are handled independently. However, many ODBMS applications require both. For example:

- The results reported in [Sjø93] illustrate the extent to which schema changes occur in real-world database applications such as health care management systems. Such systems also require a means to represent, store, and retrieve the temporal information in clinical data [KFT91, DM94, CPP95].
- The engineering and design oriented application domains (e.g., CAD, software design process) require incremental design and experimentation [KBCG90, GTC$^+$90]. This usually leads to frequent changes to the schema over time which need to be retained as historical records of the design process.

Given that the applications supported by ODBMSs need support for incremental development and experimentation with changing and evolving schema, a temporal domain is a natural means for managing changes in schema and ensuring consistency of the system. The result is a uniform treatment of schema evolution and temporal support for many ODBMS applications that require both.

A typical schema change can affect many aspects of a system. There are two fundamental problems to consider:

1. **Semantics of Change**. The effects of the schema change on the overall way in which the system organizes information (i.e., the effects on the schema). The traditional approach to solving this problem is to define a set of invariants that must be preserved over schema modifications.
2. **Change Propagation**. The effects of the schema change on the consistency of the underlying objects (i.e., the propagation of the schema changes to the existing object

[1]Contents of this chapter are published as [GSÖP97] and [GSÖP98].

instances). The traditional approach of solving this is to *coerce* objects to coincide with the new definition of the schema.

In this chapter, a method for managing schema changes and propagating the changes to underlying instances by exploiting the functionality of the TIGUKAT temporal object model is presented. The approach described in this chapter is conducted within the context of the TIGUKAT temporal ODBMS that was described in Chapter 3. However, the results reported here extend to any ODBMS that uses time to model evolution histories of objects. The issue of semantics of schema change is addressed in Section 4.1 and the issue of schema change propagation is addressed in Section 4.2. Numerous approaches that have been proposed to handle schema evolution are examined in detail in Sections 4.1.2 and 4.2.2.

4.1 Semantics of Schema Change

4.1.1 Overview

In this section, the necessary modifications that could occur on the schema are described, and a treatment of managing the implications triggered by the modifications is presented. The schema changes considered include adding a behavior to a type, dropping a behavior from a type, changing the implementation of a behavior for a particular type, and adding a supertype or subtype relationship between two types. By defining appropriate behaviors on the meta-architecture, the evolution of schema is supported. That is, changes to the schema involve updating the history of certain behaviors. For example, adding a new behavior to a type changes the history of the type's interface to include the new behavior. The old interface of the type is maintained and can be accessed through TIGUKAT query language language features that allow behavior applications to be qualified by a time reference point. Similarly, the subtype relationship behavior is defined to be temporal and, therefore, the structure of the type lattice can be reconstructed at any time of interest. The TIGUKAT query language gives designers a practical way of accessing temporal information in their experimental and incremental design phases.

4.1.2 Related Work

The issue of schema evolution has been an area of active research in the context of ODBMSs [BKKK87, KC88, PS87, NR89]. In many of the previous work, the usual approach is to define a set of invariants that must be preserved over schema modifications in order to ensure consistency of the system. The Orion [BKKK87, KC88] model is the first system to introduce the invariants and rules approach as a more structured way of describing schema evolution in ODBMSs. Orion defines a complete set of invariants and a set of accompanying rules for maintaining the invariants over schema changes. The work of Smith and Smith [SS77] on aggregation and generalization sets the stage for defining invariants when subtypes and supertypes are involved. Changes to schema in previous works are *corrective* in that once the schema definitions are changed, the old definitions of the schema are no longer traceable. In TIGUKAT, a set of invariants similar to those given in [BKKK87] is defined. However, changes to the schema are not corrective. The provision of time in TIGUKAT establishes a natural foundation for keeping track of the changes to the schema. This allows applications, such as CAD, to trace their design over time and make revisions, if necessary.

In handling temporal information, temporal object models (for example, [RS91, SC91, WD92, KS92, CITB92, BFG97]) have focussed on managing the evolution of real-world entities. The implicit assumption in these models is that the schema of the object database is static and remains unchanged during the lifespan of the object database. More specifically, the evolution of schema objects (i.e., types, behaviors, etc) is considered to be orthogonal to the temporal model. However, given the kinds of applications that an ODBMS is expected to support, the underlying temporal domain in the TIGUKAT temporal model has been exploited to support schema evolution.

In the context of relational temporal models, Ariav [Ari91] examines the implications of allowing data structures to evolve over time in a temporal data model, identifies the problems involved, and establishes a platform for their discussion. McKenzie and Snodgrass [MS90] develop an algebraic language to handle schema evolution. The language includes functions that help track the schema that existed at a particular time. Schema definitions can be added, modified, or deleted. Apart from the addition and removal of attributes, the nature of the modifications to the schema and their implications are not demonstrated. Roddick [Rod91] investigates the incorporation of temporal support within the meta-database to accommodate schema evolution. In [Rod92], SQL/SE, an SQL extension that is capable of handling schema evolution in relational database systems is proposed using the ideas presented in [Rod91]. The approach used in the TIGUKAT temporal object model is similar in the sense that temporal support of real-world objects is extended in a uniform manner to schema objects, and then used to support schema evolution. Some of the ideas in [Rod91, Rod92, Rod95] have been carried forward in the design of the TSQL2 temporal query language [Sno95b].

Skarra and Zdonik [SZ86, SZ87] define a framework within the Encore object model for versioning types as a support mechanism for changing type definitions. A type is organized as a set of individual versions. This is known as the *version set* of the type. Every change to a type definition results in the generation of a new version of the type. Since a change to a type can also affect its subtypes, new versions of the subtypes may also be generated. This approach provides fine granularity control over schema changes, but may lead to inefficiencies due to the creation of a new version of the versioned part of an object every time a single attribute changes its value. In TIGUKAT, any changes in type definitions involve changing the history of certain behaviors to reflect the changes. For example, adding a new behavior to a type changes the history of the type's interface to include the new behavior. The old interface of the type is still accessible at a time before the change was made. This alleviates the need of creating new versions of a type each time a type changes.

4.1.3 Schema Related Changes

There are different kinds of objects modeled by TIGUKAT, some of which are classified as schema objects. These objects fall into one of the following categories: *type, class, behavior, function,* and *collection*. There are three kinds of operations that can be performed on schema objects: *add, drop* and *modify*. Table 4.1 shows the combinations between the various schema object categories and the different kinds of operations that can be performed in TIGUKAT [Pet94, PÖ97]. The **bold** entries represent combinations that implicate schema changes while the *emphasized* entries denote non-schema changes.

	Operation		
Objects	Add (A)	Drop (D)	Modify (M)
Type (T)	**subtyping**	**type deletion**	**add behavior**(AB) **drop behavior**(DB) **add supertype link**(ASL) **drop supertype link**(DSL)
Class (C)	**class creation**	**class deletion**	*extent change*
Behavior (B)	*behavior definition*	**behavior deletion**	**change association**(CA)
Function (F)	*function definition*	**function deletion**	*implementation change*
Collection (L)	**collection creation**	**collection deletion**	*extent change*

Table 4.1: Classification of schema changes.

In the context of a temporal model, *adding* refers to creating the object and beginning its history, *dropping* refers to terminating the history of an object, and *modifying* refers to updating the history of the schema object. Since type-related changes form the basis of most other schema changes, the modifications that affect the type schema objects are described. Type modification (depicted at the intersection of the M column and T row in Table 4.1) includes several kinds of type changes. They are separated into changes in the behaviors of a type (depicted as **MT-AB** and **MT-DB** in Table 4.1) and changes in the relationships between types (depicted as **MT-ASL** and **MT-DSL** in Table 4.1). Invariants for maintaining the semantics of schema modifications in TIGUKAT are described in [Pet94, PÖ97]. The invariants are used to gauge the consistency of a schema change in that the invariants must be satisfied both before and after a schema change is performed.

The meta-model of TIGUKAT is uniformly represented within the object model itself, providing reflective capabilities [PÖ93]. One result of this uniform approach is that types are objects and they have a type (called `T_type`) that defines their behaviors. `T_type` defines behaviors to access a type's interface (*B_interface*), its subtypes (*B_subtypes*), its supertypes (*B_supertypes*), plus many others that are not relevant for the scope of this chapter. Since types are objects with well-defined behaviors, the approach of keeping track of the changes to a type is the same as that for keeping track of the changes to objects discussed in Chapter 3. This is one of the major advantages of the uniformity of the object model. The semantics of the changes to a type are discussed in the following sections.

4.1.4 Changing Behaviors of a Type

Every type has an *interface* which is a collection of behaviors that are applicable to the objects of that type. A type's interface can be classified into two disjoint subsets:

1. the collection of *native* behaviors which are those behaviors defined by the type and are not defined on any of its supertypes;

2. the collection of *inherited* behaviors which are those behaviors defined natively by some supertype and inherited by the type.

There are three behaviors defined on `T_type` to return the various components of a type's interface: *B_native* returns the collection of native behaviors, *B_inherited* returns the inherited behaviors and *B_interface* returns the entire interface of the type.

Types can evolve in different ways. One aspect of a type that can change over time is the behaviors in its interface (i.e., adding or deleting behaviors). To keep track of this aspect of a type's evolution, histories of interface changes are defined by extending the interface behaviors with time-varying properties. The definition of the extended behaviors are as follows:

$$\begin{aligned} B\_native &: \texttt{T\_history}\langle\texttt{T\_collection}\langle\texttt{T\_behavior}\rangle\rangle \\ B\_inherited &: \texttt{T\_history}\langle\texttt{T\_collection}\langle\texttt{T\_behavior}\rangle\rangle \\ B\_interface &: \texttt{T\_history}\langle\texttt{T\_collection}\langle\texttt{T\_behavior}\rangle\rangle \end{aligned}$$

Each behavior now returns a collection of a collections of timestamped behaviors. Adding a new behavior to a type changes the history of the type's interface to include the new behavior. The old interface of the type is still accessible at a time before the change was made.

Note that separate histories for each of these behaviors need not be explicitly maintained. For example, in an implementation one can choose to maintain only the native behaviors of a type. The entire interface of a type can be derived by unioning the native behaviors of all the supertypes of the type. The inherited behaviors can be derived by taking the difference of the interface and the native behaviors of the type. As another alternative, one may choose to maintain the interface of a type and derive the native and inherited behaviors. In this approach, the native behaviors of a type can be derived by unioning the interfaces of the direct supertypes and subtracting the result from the interface of the type. The inherited behaviors can be derived in the same way as above.

With the time-varying interface extensions, one can determine the various aspects of a type's interface at any time of interest. For example, Figure 4.1 shows the history of the entire interface for the type `T_person`.

At time t_0, behaviors *B_name*, *B_birthDate*, and *B_age* are defined on `T_person` and the initial history of `T_person`'s interface is $\{<t_0, \{B\_name, B\_birthDate, B\_age\}>\}$. At time t_5, *B_spouse* is added to `T_person`. To reflect this change, the interface history is updated to $\{<t_0, \{B\_name, B\_birthDate, B\_age\}>, <t_5, \{B\_name, B\_birthDate, B\_age, B\_spouse\}>$

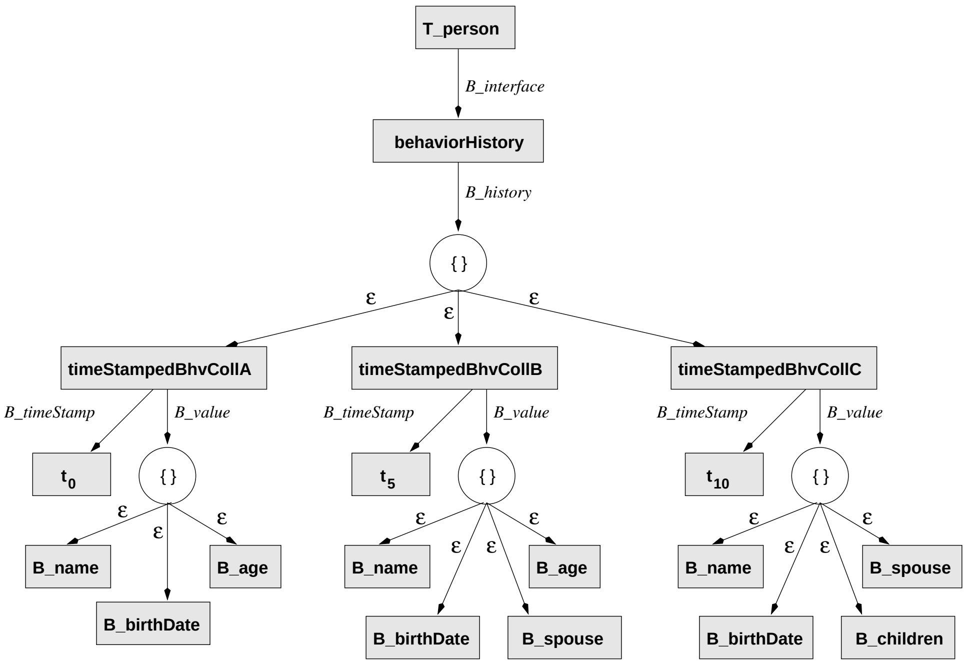


Figure 4.1: Interface history of type `T_person`.

}. This shows that between t_0 and t_5 only behaviors *B_name*, *B_birthDate*, and *B_age* are defined and at t_5 behaviors *B_name*, *B_birthDate*, *B_age*, *B_spouse* exist. Next, at time t_{10}, behavior *B_age* is dropped from type `T_person` and at the same time behavior *B_children* is added. The final history of the interface of `T_person` after this change is $\{<t_0, \{B\_name, B\_birthDate, B\_age\}>, <t_5, \{B\_name, B\_birthDate, B\_age, B\_spouse\}>$ $, <t_{10}, \{B\_name, B\_birthDate, B\_spouse, B\_children\}>\}$[2]. The native and inherited behaviors would contain similar histories. Using this information, one can reconstruct the interface of a type at any time of interest. For example, at time t_3 the interface of type `T_person` was $\{B\_name, B\_birthDate, B\_age\}$, at time t_5 it was $\{B\_name, B\_birthDate, B\_age,$ $B\_spouse\}$, and at time t_{10} (*now*) it is $\{B\_name, B\_birthDate, B\_spouse, B\_children\}$.

The behavioral changes to types include the **MT-AB** and **MT-DB** entries of Table 4.1. These changes affect various aspects of the schema and have to be properly managed to ensure consistency of the schema.

Modify Type - Add Behavior (MT-AB). This change adds a native behavior b to a type T at time t. The **MT-AB** change has the following effects:

- The histories of the native and interface behaviors of type T need to be updated. The $T.B\_native.B\_insert(b, t)$ and $T.B\_interface.B\_insert(b, t)$ behavior applications perform this update. For example, the behavior application

[2]Note that in Figure 4.1 objects that are repeated in the timestamped collections are actually the same object. For example, the *B_name* object in all three timestamped collections is the same object. It is shown three times in the figure for clarity.

`T_person`.$B\_interface.B\_insert(B\_spouse, t_5)$ updates the interface history of the type `T_person` when behavior *B_spouse* is added to `T_person` at time t_5.

- The implementation history of behavior b needs to be updated to associate it with some function f. This is achieved by the $b.B\_implementation.B\_insert(f, t)$ behavior application (details on implementation histories of behaviors are given in Section 4.1.5). For example, if the function associated with behavior *B_spouse* is the stored function s_{spouse}, then the implementation history of *B_spouse* is updated using the $B\_spouse.B\_implementation.B\_insert(s_{spouse}, t_5)$ behavior application.
- The history of inherited and interface behaviors of all subtypes of type T needs to be adjusted. That is, $\forall T^{'} \mid T^{'}$ subtype-of T,

$$T^{'}.B\_inherited.B\_insert(b, t) \text{ and } T^{'}.B\_interface.B\_insert(b, t)$$

  For example, the histories of inherited and interface behaviors of types `T_employee` and `T_patient` (see Figure 3.1) need to be adjusted to reflect the addition of behavior *B_spouse* in type `T_person` at time t_5. For `T_employee`, this is accomplished by the behavior applications `T_employee`.$B\_interface.B\_insert(B\_spouse, t_5)$ and `T_employee`.$B\_inherited.B\_insert(B\_spouse, t_5)$. Similar behavior applications are carried out for `T_patient`.

Modify Type - Drop Behavior (MT-DB). This change drops a native behavior b from a type T at time t. When a behavior is dropped, its native definition is propagated to the subtypes unless the behavior is inherited by the subtype through some other chain. In this way, as with the supertypes, the subtypes of a type also retain their original behaviors. Thus, only the single type involved in the operation actually drops the behavior and the overall interface of the subtypes and supertypes are not affected by the change. Many behavior inheritance semantics are possible. One such semantics is that when a native behavior is dropped from a type, all subtypes retain that behavior. This means that if another supertype of the subtype defines this behavior, there is no change. Otherwise, the behavior in the subtype moves from the inherited set to the native set. This is the semantics that is modeled in this thesis. If any other behavior inheritance semantics are used, appropriate changes can easily be made to the temporal histories. The **MT-DB** change has the following effects:

- The native behaviors history of type T changes. The behavior application $T.B\_native.B\_remove(b, t)$ performs this update. For example, the behavior application `T_person`.$B\_native.B\_remove(B\_age, t_{10})$ updates the history of native behaviors of `T_person` when the behavior *B_age* is dropped from type `T_person`.
- The native and inherited behavior histories of the subtypes of T (possibly) change. For example, the `T_employee`.$B\_native.B\_insert(B\_age, t_{10})$ and

T_employee.$B\_inherited.B\_remove(B\_age, t_{10})$ behavior applications add behavior *B_age* to the native behaviors of T_employee, and drop behavior *B_age* from the inherited behaviors of T_employee respectively, when *B_age* is dropped from T_person at t_{10}. This is because *B_age* is not inherited by T_employee through any other chain. If *B_age* was inherited by T_employee from some other supertype, nothing would change. Similar behavior applications are carried out for type T_patient.

4.1.5 Changing Implementations of Behaviors

Each behavior defined on a type has a particular implementation for that type. The *B_implementation* behavior defined on T_behavior is applied to a behavior, accepts a type as an argument and returns the implementation (function) of the receiver behavior for the given type. In order to model the aspect of schema evolution that deals with changing the implementations of behaviors on types, the history of implementation changes is maintained by extending the *B_implementation* behavior with time-varying properties. The definition of the extended behavior is as follows:

$$B\_implementation : \texttt{T\_type} \rightarrow \texttt{T\_history}\langle\texttt{T\_function}\rangle$$

With this behavior one can determine the implementation of a behavior defined on a type at any time of interest. For example, Figure 4.2 shows the history of the implementations for behavior *B_name* on type T_person.

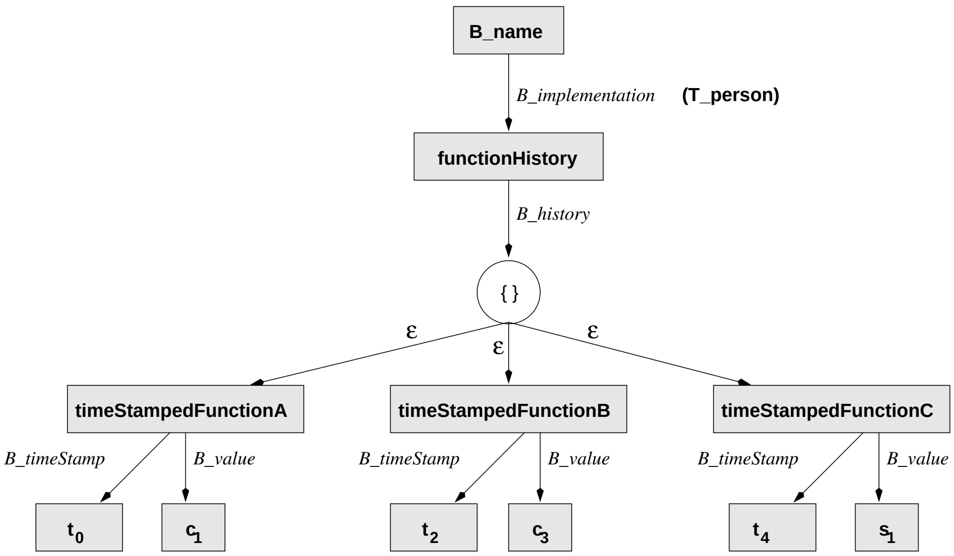


Figure 4.2: Implementation history of behavior *B_name* on type T_person.

In Figure 4.2, c_i denotes a computed function, s_i to denote a stored function (computed and stored functions are described in more detail in Section 4.2.3) . At time t_2, the implementation of *B_name* changed from the computed function c_1 to the computed function c_3.

At time t_4, the implementation of *B_name* changed from the computed function c_3 to the stored function s_1. All these changes are reflected in the implementation history of behavior *B_name*, which is $\{<t_0, c_1>, <t_2, c_3>, <t_4, s_1>\}$.

Using the results of this section and Section 4.1.4, we can reconstruct the behaviors, their implementations and the object representations[3] for any type at any time t. For example, the interface of type `T_person` at time t_3 is given by the behavior application `T_person`.$[t_3]$*B_interface* which results in $\{B\_name, B\_birthDate, B\_name\}$, as shown in Figure 4.1. The syntax $o.[t]b$ is used to denote the application of behavior b to object o at time t. The implementation of *B_name* at time t_3 is given by the behavior application *B_name*.$[t_3]$*B_implementation*(`T_person`) which is c_3, as shown in Figure 4.2.

If the binding of a behavior to a function changes in a type, one semantics is that the bindings of that behavior in the subtypes are unaffected. That is, there is no implementation inheritance. This is the semantics modeled in this thesis. If implementation inheritance is desired, it can easily be modeled by temporal histories similarly to behavioral inheritance.

4.1.6 Changing Subtype/Supertypes of a Type

The changes in a type's interface, described in Section 4.1.4, is one aspect in which a type evolves. Another aspect of a type that can change over time is the relationships between types. These include adding a direct supertype link and dropping a direct supertype link. The *B_supertypes* and *B_subtypes* behaviors defined on `T_type` return the direct supertypes and subtypes of the receiver type, respectively. In order to model the structure of the type lattice through time, we define histories of supertype and subtype changes of a type by extending the *B_supertypes* and *B_subtypes* behaviors with time-varying properties:

$$\begin{aligned} B\_supertypes &: \texttt{T\_history}\langle\texttt{T\_collection}\langle\texttt{T\_type}\rangle\rangle \\ B\_subtypes &: \texttt{T\_history}\langle\texttt{T\_collection}\langle\texttt{T\_type}\rangle\rangle \end{aligned}$$

Using the *B_supertypes* and *B_subtypes* behaviors, one can reconstruct the structure of a type's supertype and subtype lattice at any time of interest. To facilitate this, the derived behaviors *B_superlattice* and *B_sublattice* are defined on `T_type`:

$$\begin{aligned} B\_superlattice &: \texttt{T\_history}\langle\texttt{T\_poset}\langle\texttt{T\_type}\rangle\rangle \\ B\_sublattice &: \texttt{T\_history}\langle\texttt{T\_poset}\langle\texttt{T\_type}\rangle\rangle \end{aligned}$$

The behavior *B_superlattice* is derived by recursively applying *B_supertypes* until `T_object` is reached, while the behavior *B_sublattice* is derived by recursively applying *B_subtypes* until `T_null` is reached. In both cases, the intermediate results are partially ordered. Figure 4.3 shows the supertype lattice history for type `T_employee`.

[3] Stored functions associated with behaviors allow one to reconstruct object representations (i.e., states of objects) for any type at any time t. This is useful in propagating changes to the underlying object instances as shown in Section 4.2.

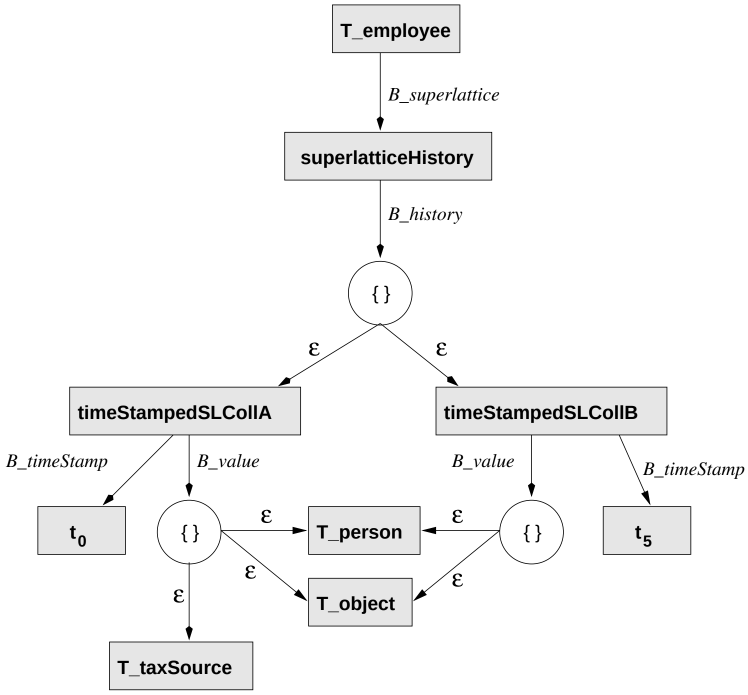


Figure 4.3: Supertype lattice history for type `T_employee`.

At time t_0, the superlattice history of type `T_employee` includes the types `T_person`, `T_taxSource`, and `T_object`. At time t_5, the supertype link between `T_employee` and `T_taxSource` is dropped. To reflect this change, the superlattice history of `T_employee` is updated to $\{<t_0, \{$`T_person`, `T_taxSource`, `T_object`$\}>, <t_5, \{$`T_person`, `T_object`$\}>\}$.

The relationships between types include the **MT-ASL** and **MT-DSL** entries of Table 4.1. Similar to the behavioral changes to types discussed in Section 4.1.4, the relationships between types affect various aspects of the schema and have to be properly managed to ensure consistency of the schema.

Modify Type - Add Supertype Link (MT-ASL). Add a type, say S, as a direct supertype of another type, say T at time t. The **MT-ASL** change has the following effects:

- The history of the collection of supertypes of type T is updated. The behavior application $T.B\_supertypes.B\_insert(S, t)$ performs this update. The history of the super-lattice of T is adjusted accordingly. For example, adding the supertype link between `T_employee` and `T_taxSource` at t_0 necessitates an update to the history of supertypes for `T_employee`. This is done by the behavior application `T_employee`.$B\_supertypes.B\_insert($`T_taxSource`,$t_0)$. The history of the direct supertypes of `T_employee` would then be $\{<t_0, \{$`T_taxSource`$\}>\}$.
- The history of the collection of subtypes of type S is updated. The behavior application $S.B\_subtypes.B\_insert(T, t)$ performs this update. The history of the

sub-lattice of S is adjusted accordingly. In this case, the history of the collection of subtypes of `T_taxSource` has to be updated. This is done by the behavior application `T_taxSource`.*B_subtypes*.*B_insert*(`T_employee`,t_0). The history of the direct subtypes of `T_taxSource` would then be $\{<t_0, \{$`T_employee`$\}>\}$.

- The behaviors of S are inherited by T and all the subtypes of T. Therefore, the inherited behavior history of T and all subtypes of T is adjusted. The current behaviors of S are inherited by T and all subtypes of T, and timestamped with t. That is, $\forall b \in S.B\_interface.B\_history.B\_last$, and $\forall T' \mid T'$ subtype-of T, $T'.B\_inherited.B\_insert(b, t)$. Behavior *B_last* returns the collection of behaviors that are currently valid from the interface history of S. Assume that `T_taxSource` has the behavior *B_taxBracket* defined at t_0. *B_taxBracket* then has to be added to the history of inherited behaviors of `T_employee`. This is done by the behavior application `T_employee`.*B_inherited*.*B_insert*(*B_taxBracket*,t_0). The history of the inherited behaviors would then be $\{<t_0, \{B\_name, B\_birthDate, B\_age, B\_taxBracket\}>\}$. Behaviors *B_name*,*B_birthDate*,*B_age* are inherited from type `T_person` (see Figure 4.1), while behavior *B_taxBracket* is inherited from type `T_taxSource`.

Modify Type - Drop Supertype Link (MT-DSL). Drop a direct supertype link between two types (a direct supertype link to `T_object` cannot be dropped) at time t. Consider types T and S where S is the direct supertype of T. Then, removing the direct supertype link between T and S at time t has the following effects:

- Adjust the history of supertypes of T and the history of subtypes of S. For example, dropping the supertype link between `T_employee` and `T_taxSource` at t_5 requires updating the history of supertypes of `T_employee` and history of subtypes of `T_taxSource`. This is carried out using the behavior applications `T_employee`.*B_supertypes*.*B_remove*(`T_taxSource`,t_5) and `T_taxSource`.*B_subtypes*. *B_remove*(`T_employee`,t_5).
- The **MT-ASL** operation is carried out from T to every supertype of S, unless T is linked to the supertype through another chain. This operation is not required when the supertype link between `T_employee` and `T_taxSource` is dropped because `T_employee` is linked to the supertype of `T_taxSource` (`T_object`) through `T_person`.
- The **MT-ASL** operation is carried out from each subtype of T to S, unless the subtype is linked to S through another chain. This operation requires adding a supertype link between `T_null` and `T_taxSource`.
- The native behaviors of S are dropped from the interface of T. That is, the history of inherited behaviors of T is adjusted. This means the behavior *B_taxBracket*, defined natively on `T_taxSource`, has to be dropped from the history of inherited

behaviors of `T_employee`. The `T_employee`.*B_inherited*.*B_remove*(*B_taxBracket*,t_5) behavior application performs this operation.

4.1.7 Queries

In this section several queries are constructed using the TIGUKAT query language (TQL) [PLÖS93] (described in Section 3.6.4.1) to retrieve schema objects at any time in their evolution histories. This gives software designers a temporal user interface which provides a practical way of accessing temporal information in their experimental and incremental design phases. TQL incorporates reflective temporal access in that it can be used to retrieve both objects, and schema objects in a uniform manner. Hence, TQL does not differentiate between queries (which query objects) and meta-queries (which query schema objects).

Example 4.1 Return the time when the behavior *B_children* was added to the type `T_person`.
select *b*.*B_timestamp*
from *b* **in** `T_person`.*B_interface*.*B_history*
where *B_children* **in** *b*.*B_value*
The result of this query would be the time t_{10} as seen in Figure 4.1. □

Example 4.2 Return the types that define behaviors *B_age* and *B_taxBracket* as part of their interface.
select *T*
from *T* **in** **C_type**
where (*b*1 **in** *T*.*B_interface*.*B_history* **and** *B_age* **in** *b*1.*B_value*) **or**
(*b*2 **in** *T*.*B_interface*.*B_history* **and** *B_taxBracket* **in** *b*2.*B_value*)
This query would return the types `T_person`, `T_taxSource`, `T_employee`, and `T_null`. The type `T_person` defines behavior *B_age* natively (see Figure 4.1), while the type `T_taxSource` defines behavior *B_taxBracket* natively. The behaviors *B_age* and *B_taxBracket* are inherited by types `T_employee` and `T_null` since they are subtypes of `T_person` and `T_taxSource` as shown in Figure 3.1. □

Example 4.3 Return the implementation of behavior *B_age* in type `T_person` at time t_1.
select *i*.*B_value*
from *i* **in** *B_age*.*B_implementation*(`T_person`).*B_history*
where *i*.*B_timestamp*.*B_lessthaneqto*(t_1)
The behavior *B_lessthaneqto* is defined on type `T_timeStamp` and checks if the receiver timestamp is less than or equal to the argument timestamp. The result of the query is the computed function c_1 as shown in Figure 4.4. □

Example 4.4 Return the super-lattice of type `T_employee` at time t_3.
select *r*.*B_value*
from *r* **in** `T_employee`.*B_super-lattice*.*B_history*

where $r.B\_timestamp.B\_lessthaneqto(t_3)$
The super-lattice of `T_employee` at t_3 consists of the types `T_person`, `T_taxSource`, and `T_object`. This is shown in Figure 4.3. □

Example 4.5 Return the types that define behavior *B_age* with the same implementation as one of their supertypes.
select T
from T **in C_type**, S **in** $T.B\_supertypes.B\_history$,
i **in** $B\_age.B\_implementation(T).B\_history$,
j **in** $B\_age.B\_implementation(S.B\_value).B\_history$
where b **in** $S.B\_value.B\_interface.B\_history$ **and** B_age **in** $b.B\_value$ **and**
$i.B\_value = j.B\_value$ **and** $i.B\_timestamp = j.B\_timestamp$
This query would return the types `T_employee`, `T_patient`, and `T_null`, assuming the implementation of behavior *B_age* is not changed when it is inherited by these types. □

4.2 Semantics of Change Propagation

4.2.1 Overview

In this thesis, change propagation is performed lazily. When a behavior is re-associated with a different implementation, the old implementation is maintained and the change is recorded in the implementation history of the behavior. When a behavior is applied to an object at a particular time, it is coerced to the changed implementation, if it has not already been updated. The old implementations of the behavior are still maintained by the implementation history. Thus, one can still run historical queries on objects. Several algorithms are presented that dynamically fetch the correct implementation (function/method) of a behavior and apply it to the correct representation of an object during a behavior application process at a given time.

A novel characteristic of the change propagation approach presented in this thesis is that the granularity of object coercion is based on individual behaviors. That is, individual behaviors defined on the type of an object can be coerced to a new definition for that object when the object is accessed, leaving the other behaviors to retain their old definitions. Converting one behavior at a time gives more flexibility, since implementations are associated with individual behaviors. If an implementation of a behavior changes, one does not need to coerce the rest of the behaviors in that type. Since behaviors are used to model attributes, the same is true for structural changes. Furthermore, a historical record of the coerced behaviors is maintained for each object so that older definitions of the behaviors can still be accessed for each object. Complete object coercion takes place when all behaviors of a type have been accessed. This is in contrast to other models where an object is converted in its entirety to a changed type [PS87, BKKK87, FMZ+95]. This means that for every behavior modification that takes place in a type, default conversion functions are defined for all unmodified behaviors in that type. Furthermore, the old information of the object is lost. Since the model used to handle change propagation in this thesis is time based, the old information of the object is available so even if objects are coerced to a changed type, historical queries can be run by giving an appropriate time point sometime in the past history of the object.

4.2.2 Related Work

In addition to modifications to schema, a system must define how schema changes are reflected in the instances. In order for the instances to remain meaningful, either the relevant instances must be coerced into the new definition of the schema or a new version of the schema must be created leaving the old version intact. Three main approaches have been identified and employed in the past. *Immediate* (*conversion*) and *deferred* (*lazy, screening*) propagate changes to the instances - only at different times - while *filtering* is a solution for versioning which attempts to maintain the semantic differences between versions of schema. A fourth approach is to combine the above three methods into a *hybrid* model. The various techniques are summarized below.

- **Immediate:** Each schema change initiates an immediate conversion of all objects affected by the change. This approach causes delays during the modification of schema, but no delays are incurred during access to objects. GemStone [PS87] and O_2 [FMZ+95] systems report the use of immediate conversion for schema change propagation. In O_2, immediate conversion is implemented using the algorithm defined for deferred conversion.

- **Deferred:** Schema changes generate a conversion program that is capable of converting objects into the new representation. The conversion is not immediate; but is delayed until an instance of the modified schema is accessed. Object access is monitored and whenever an object is accessed, the conversion program is invoked, if necessary, to convert the object into the new definition. The conversion programs resulting from multiple independent changes to a type are composed, meaning access to an object may invoke the execution of multiple conversion programs where each one handles a certain change to the schema. Deferred conversion causes delays during object access. ORION [BKKK87] uses this approach and OTGen [LH90] uses it for database reorganization. In O_2 [FMZ94, FMZ+95], implementation strategies are defined for conversion functions implemented as deferred database updates.

- **Filtering:** In the filtering approach, changes are never propagated to the instances. Instead, objects become instances of particular versions of the schema. When the schema is changed, the old objects remain with the old version of the schema and new objects are created as instances of the new one. The filters define the consistency between the old and new schema versions and handle the problems associated with behaviors written according to one version accessing objects of a different version. Error handlers are one example of filters. They can be defined on each version of the schema to trap inconsistent access and produce error and warning messages. The Encore model [SZ86] uses type versioning with error handlers as a filtering mechanism. The Avance [BH89] system adopts a similar approach to Encore. Exception handlers are defined as filters to cope with mismatches between different versions. Both Encore and Avance use emulation to present old instances as if they are new ones. It is not possible to associate additional storage with existing attributes since all objects are strictly connected to the version in which they were created. As such additional attributes would necessarily be read-only and have a fixed, default value. This problem is remedied in CLOSQL [MS92] where objects are allowed to dynamically change the class version with which they are connected. Each attribute of an object has update and backdate functions (provided by the user) for converting objects into different formats. However, the overhead of the conversion process and the added responsibility on the user are quite significant in CLOSQL.

- **Hybrid:** A hybrid approach combines two or more of the above methods. GemStone mentions an effort to incorporate a hybrid approach, but currently we are unaware of such a system implementation. In Sherpa [NR89], schema changes are propagated to instances through conversion or screening, which is selected by the user. However, only the conversion approach is discussed. Change propagation is assisted by the notion of *relevant classes*. A *relevant class* is a semantically consistent partial definition of a complete class and is bound to the class. A relevant class is similar to a type version in [SZ86] and a complete class resembles a version set.

Although numerous approaches have been proposed for propagating different schema changes to object instances, the schema change that involves changing the implementation of a behavior, and how it affects the underlying object structure has not been addressed comprehensively. In this thesis, a deferred approach that uses a finer grained filtering based on behavior histories is used as the underlying mechanism for behavior implementation change propagation. The approach also allows for immediate behavior coercion to reflect the changed schema. This makes it feasible for the system to take a more active role by using deferred object coercion as the default and switching to immediate object coercion whenever the system is idle.

In systems that use immediate or deferred object coercion, the entire object must be converted upon coercion and in the systems that don't define versions of schema, the old state of the object is lost. The approach in this thesis differs in that the granularity of object coercion is based on individual behaviors. That is, an individual behavior of an object's type can be coerced to a new definition for that object, leaving the other behaviors to retain their own definition. Furthermore, a historical record of the coerced behaviors is maintained for each object so one can access the older definitions of the behaviors for each object. Complete object conversion takes place only if all behaviors defined in the type of the object have been coerced. This results in considerable savings of work.

4.2.3 Changing Implementations of Behaviors

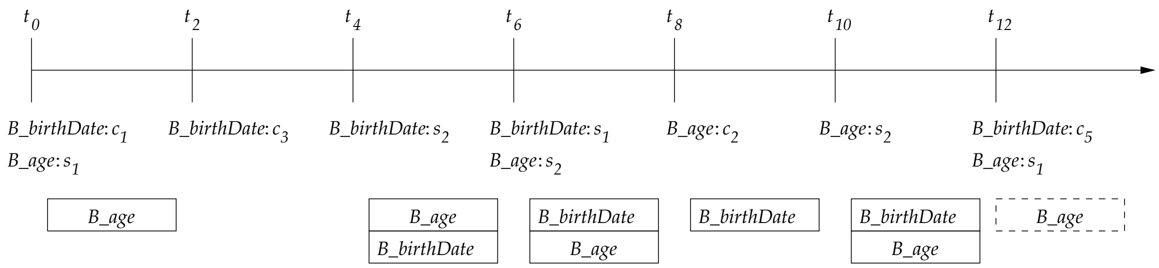
There are two kinds of implementations for behaviors [Pet94]. A *computed function* consists of runtime calls to executable code and a *stored function* is a reference to an existing object in the object database. Thus a behavior with a computed function implementation can be considered an abstraction of a method in classical object models, whereas a behavior with a stored function implementation is an abstraction of an attribute (with getter and setter functions). The valid implementation changes for behaviors are shown in Table 4.2. The notation $computed_i$ (c_i) and $stored_i$ (s_i) refer to computed and stored functions respectively. The subscripts i and j are used to denote distinct functions. The term *undefined* is for the case when the behavior is undefined. The combinations $computed_i$ to $computed_i$ and $stored_i$ to $stored_i$ (which imply changes to the function code) are not included in the table because these do not reflect changes in function association. The *emphasized* entries represent user-level changes (i.e., by the schema designer) and the **bold** entry is a system-level change for reorganizing the internal representation of objects.

	Old Implementation	New Implementation
CC	$computed_i$	$computed_j$
CS	$computed_i$	$stored_j$
SS	$\mathbf{stored}_i$	$\mathbf{stored}_j$
SC	$stored_i$	$computed_j$
US	*undefined*	$stored_j$
UC	*undefined*	$computed_j$

Table 4.2: Valid implementation changes of a behavior in a type.

With the *B_implementation* behavior (defined in Section 4.1.5) we can determine the implementation of a behavior defined on a type at any time of interest. For example, Figure 4.4 shows the history of the implementations for behaviors *B_birthDate* and *B_age* on type `T_person`. A timeline representation and the result of *B_birthDate*.*B_implementation*(`T_person`). *B_history* and *B_age*.*B_implementation*(`T_person`).*B_history* are shown. The implementation histories of *B_birthDate* and *B_age* return a collection of timestamped function objects. The value of each timestamped function is a computed or stored function. The timestamp of each timestamped function denotes the time interval during which the particular implementation is valid. The interface history of `T_person` is also shown for clarity. The *B_interface* behavior is defined in `T_type` and returns a history of the evolution of behaviors in a type. Each timestamped object in the history consists of a collection of behaviors that are valid during the associated time interval.

In the timeline representation, $B\_X{:}c_i$ or $B\_X{:}s_i$ denotes the association of a computed or stored function with behavior *B_X*. Moreover, for stored functions the subscript i refers to a location (e.g., a slot number) in an object representation that the stored function accesses. Each association is valid at a certain time t and remains valid until it is modified or removed. An object representation (i.e., the state of an object) consists of a number of slots for holding information carried by the object. The representations of objects at different times according to the stored functions associated with behaviors at those times are depicted by the boxes labeled with behaviors. For example, between times t_4 and t_6, the object representation consists of two slots – the first slot is for the stored implementation of behavior *B_age* and the second is for *B_birthDate*. Between times t_8 and t_{10}, the object representation consists of only one slot which is for *B_birthDate*, since during this interval, *B_age* is associated with the computed function, c_2.



Implementation history of behavior *B_birthDate* for type T_person:
$\{<[t_0, t_2), c_1>, <[t_2, t_4), c_3>, <[t_4, t_6), s_2>, <[t_6, t_{12}), s_1>, <[t_{12}, now], c_5>\}$

Implementation history of behavior *B_age* for type T_person:
$\{<[t_0, t_6), s_1>, <[t_6, t_8), s_2>, <[t_8, t_{10}), c_2>, <[t_{10}, t_{12}), s_2>, <[t_{12}, now], s_1>\}$

Interface history of type T_person:
$\{<[t_0, now], \{B\_birthDate, B\_age\}>\}$

Figure 4.4: Implementation histories of behaviors *B_birthDate* and *B_age* for type `T_person` and object representations.

Figure 4.4 is used to describe how the implementation changes in Table 4.2 are maintained by implementation histories. Prior to time t_0 both behaviors are undefined and at time t_0, *B_age* is defined as stored (US) and *B_birthDate* is defined as computed (UC). At time t_2, the implementation of *B_birthDate* changes from the computed function c_1 to the computed function c_3 (CC). At time t_4, the implementation of *B_birthDate* changes from the computed function c_3 to the stored function s_2 (CS). At time t_6, the implementation of *B_birthDate* changes from the stored function s_2 to the stored function s_1 (SS) and *B_age* changes from s_1 to s_2 (SS). At time t_8, the implementation of *B_age* changes from the stored function s_2 to the computed function c_2 (SC).

Note that at time t_{12} the behavior *B_birthDate* changes from the stored behavior s_1 to the computed behavior c_5. Since all object representations at time t_{12} require only one slot, the change to *B_birthDate* forces a change to *B_age* so that at time t_{12} behavior *B_age* accesses slot one instead of slot two. Furthermore, the implicit implementation change of

B_age is from a stored function to a stored function (SS) which is a system managed change and therefore is transparent to the user. The implicit implementation change of *B_age* is reflected in its history by the two entries $<[t_{10}, t_{12}), s_2>$ and $<[t_{12}, now], s_1>$. In general, the slots of an object representation are reorganized (i.e., an implicit change occurs) whenever a stored to computed implementation change removes a slot other than the last slot of an object's representation. The system can also rearrange slots as part of an implementation change, necessitating internal system organization as at t_6.

Using the results of this section, one can reconstruct the implementations of behaviors, and the object representations for any type at any time t. The implementation of *B_birthDate* at time t_7 (where $t_6 < t_7 < t_8$) is given by *B_birthDate*.$[t_7]$*B_implementation* (`T_person`) which is s_1. Similarly, the implementation of *B_age* at time t_7 is given by the behavior application *B_age*.$[t_7]$*B_implementation*(`T_person`) which is s_2. Since there are two stored functions, this implies a two slot representation for objects at time t_7. That is, *B_birthDate* accesses slot one using stored function s_1 and *B_age* accesses slot two using stored function s_2.

4.2.4 Change Propagation

The behaviors that are applicable to an object at creation is the set of behaviors that are defined on its type. The implementations of these behaviors are those which exist in the implementation histories for the type at creation time (which can be obtained by means of the *B_created* behavior defined on `T_object`).

When changes occur to the type definition and behavior implementations, they do not immediately get propagated to the instances. Instead, the old version of the schema is maintained and the change is recorded in the proper behavior histories (as described in Sections 4.1.4 and 4.1.5) The propagation of changes to the instances is delayed until the instances are accessed. This occurs when a behavior is applied to an object. At that point in time, the behavior is coerced to reflect the implementation changes that have occurred on the behavior since the last behavior application. These changes are recorded in the *B_changes* behavior which is defined in `T_type`. The signature for *B_changes* is as follows:

$$B\_changes : \texttt{T\_list}\langle\texttt{T\_timeStamp}, \texttt{T\_behavior}\rangle$$

The result of *B_changes* is a list of (*timestamp, behavior*) pairs. Each pair denotes the time at which the implementation for the behavior has changed. The *B_changes* list is used by the behavior dispatch routine (defined in Section 4.2.5) to determine the most recent coercion time of the behavior that is applied to an object. The time is used as a reference point for finding an appropriate implementation of the behavior.

A novel characteristic of the model presented in this thesis is that the basic unit of object coercion is individual behaviors. More specifically, objects from the older schema are coerced to the newer schema one behavior at a time. Thus, portions of an object (i.e., some behaviors) may correspond to older schema, while other portions correspond to newer schema.

In order to model the representations of an object over time (resulting from changes to its structure), the `T_history` mechanism that is described in Chapter 3 is used. For example, type $\texttt{T\_person} = \texttt{T\_history}\langle\texttt{T\_person}'\rangle$ is created to maintain the representations of a person over time. Therefore, if joe is an object of type `T_person`, then joe represents the history of its different structural changes over time. The value of each timestamped object in an object o of type $\texttt{T\_X} = \texttt{T\_history}\langle\texttt{T\_X}'\rangle$ is called a *representation object* of o, and is of type $\texttt{T\_X}'$.

In this thesis the notation $\texttt{T\_history}\langle\texttt{T\_X}'\rangle$ is used to denote a type whose schema changes are recorded. However, an actual user of the ODBMS would simply use the notation `T_X` and indicate at type creation time that the schema changes should be recorded for this type. The ODBMS would then create `T_X` as $\texttt{T\_history}\langle\texttt{T\_X}'\rangle$ and the user would never deal directly with $\texttt{T\_X}'$. However, the notation $\texttt{T\_history}\langle\texttt{T\_X}'\rangle$ will continue to be used in this thesis to show how the model and algorithms for schema changes can be defined using only the existing TIGUKAT temporal model and without introducing any new concepts. The user does not actually see or use `T_history`.

Whenever a change to the representation of an object occurs due to coercion of one of the behaviors of its base type[4], the change is recorded by updating the history of its structural changes. Thus, an object of type $\texttt{T\_history}\langle\texttt{T\_X}'\rangle$ is generic in the sense that it consists of all its representation objects over time. This is called the *generic instance* of the object. The default representation object of a generic instance is the most current representation object in the history of its structural changes. The individual representation objects in the history denote how the object existed at certain times in the past. Each of these representation objects is called a *structural instance* of the object and has type $\texttt{T\_X}'$. In essence the changes list of the type $\texttt{T\_X}'$ and the objects of type $\texttt{T\_history}\langle\texttt{T\_X}'\rangle$ (potentially) "grow" with each behavior application.

Example 4.6 Consider Figure 4.5, which contains the object joe created as an instance of type $\texttt{T\_history}\langle\texttt{T\_person}'\rangle$. Assuming no behavior application has occurred, the figure shows the created time and the representation objects of joe. It also shows the changes list of `T_person`[5]. The notation $o@t_i$ is used to denote the structural instance of an object o at time t_i. Object joe is created at time t_0. The default properties and implementations for this object are those that exist at time t_0, namely, $B\_birthDate$:c_1 and B_age:s_1 (see Figure 4.4). There are no entries in the changes list of `T_person` since no coercion of any behaviors of $\texttt{T\_person}'$ has taken place yet. Therefore, joe has only one structural instance $joe@t_0$, the representation object that existed at the creation time of joe.

Now suppose joe is accessed at time t_7 through the behavior application joe.$[t_7]B\_birthDate$. The $B\_birthDate$ behavior is coerced to a version at t_7, and joe is updated. These changes

[4]The base type of an object o of type `T_history<T_X'>` is the type `T_X'`.

[5]The changes list of `T_person`, is actually computed from the base type as $\texttt{T\_person}'.B\_changes$.

$$\begin{array}{rcl} \mathsf{joe}.B\_created & = & t_0 \\ \mathsf{joe}.B\_history & = & \{<[t_0, now], joe@t_0>\} \\ \mathtt{T\_person}'.B\_changes & = & \{\ \} \end{array}$$

Figure 4.5: Initial representation of joe and changes list of T_person.

are shown in Figure 4.6.

$$\begin{array}{rcl} \mathsf{joe}.B\_created & = & t_0 \\ \mathsf{joe}.B\_history & = & \{<[t_0, t_4), joe@t_0>, <[t_4, t_6), joe@t_4>, <[t_6, now], joe@t_6>\} \\ \mathtt{T\_person}'.B\_changes & = & \{<t_0, B\_birthDate>, <t_2, B\_birthDate>, <t_4, B\_birthDate>, \\ & & <t_6, B\_birthDate>\} \end{array}$$

Figure 4.6: The representation objects of joe and the changes list of T_person after behavior application of *B_birthDate* at time t_7.

Since this is the first behavior application of *B_birthDate* on object joe, the *B_changes* list of T_person is updated with the times of all implementation changes that took place on behavior *B_birthDate* prior to time t_7. From Figure 4.4 we see that these times are t_0, t_2, t_4, and t_6. The behavior coercions at times t_4 and t_6 lead to changes in the representation of object joe. At t_4, the implementation of *B_birthDate* changes from a computed to a stored function and at t_6, the implementation changes from a stored to a stored function. These changes in structural representation are recorded in joe as shown in Figure 4.6. Note that changes to *B_age* are not yet recorded since deferred coercion is used and *B_age* has not yet been applied at t_7. □

4.2.5 Temporal Behavior Dispatch

Having established the mechanism for maintaining the histories of the implementations of behaviors for a type, and the representations of objects, the behavior dispatch process that occurs when some behavior b is applied to an object o at given time t is illustrated in this section. This application is denoted as $o.[t]b$. The time component is optional and if left out the current time *now* is assumed.

Figure 4.7 provides an overview of the dispatch process. Detailed explanations of the various steps are given in the sections that follow. In general, a dispatch mechanism takes a type and a behavior and returns the function associated with the behavior for the given type [HS97]. In this thesis, the dispatch mechanism is extended to take a third argument which is time.

A behavior application is first checked for temporal validity. It is considered valid if the object o exists at time t and behavior b is defined in the interface of o's base type at time t. An invalid behavior application produces an error and this is the only place an error can occur. This is a good feature because errors are caught early in the dispatch process. After

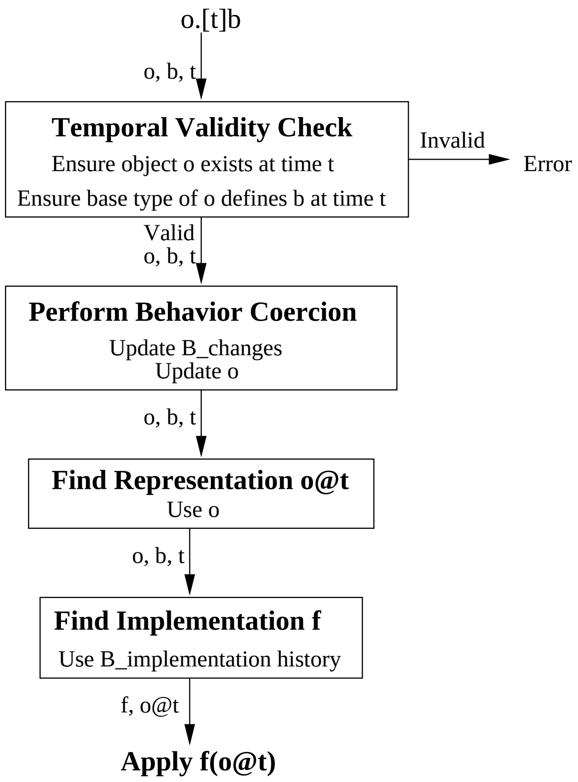


Figure 4.7: Dispatch process for applying a behavior b to an object o at time t.

the validity check, all behaviors will execute even in the presence of schema changes.

For a valid application, the *B_changes* list of the base type of o is updated. A search is made in *b.B_implementation* for implementation changes that took place before or at the same time as t. The *B_changes* list of the base type of o is then updated with all implementation changes that have not yet been recorded in *B_changes*. Object o is then updated if neccssary.

The appropriate representation object $o@t$ of o, and the appropriate implementation f of b for the base type of o at time t are then retrieved by indexing into o and the *B_implementation* history of b, respectively. Finally, function f is applied to the representation object $o@t$.

4.2.5.1 Dispatch Semantics

In order for a behavior application to be valid, object o must exist at time t and behavior b must be defined in the interface of the base type of o at time t. The temporal validity check algorithm, Algorithm 4.1, performs this test in the form of a logical expression.

Algorithm 4.1 *TemporalValidity:*

Input: An object o, a behavior b and a time t

Output: True if the application is valid, false otherwise

Procedure:

return

$$(t.B\_within(o.B\_lifespan(o.B\_mapsto.B\_classof.B\_shallowExtent)) \tag{4.1}$$
$$\wedge\ \exists x(x \in o.B\_mapsto.B\_baseType.B\_interface.B\_history \tag{4.2}$$
$$\wedge\ t.B\_within(x.B\_timeStamp) \tag{4.3}$$
$$\wedge\ b \in x.B\_value)) \tag{4.4}$$

The first part of the expression (4.1) checks that o exists at time t by testing whether time t lies within[6] the lifespan of o in the class of its associated type. In the second part of the expression, *o.B_mapsto.B_baseType.B_interface.B_history* returns the interface history for the base type of object o. This history is searched for an entry x that satisfies (4.3), which checks that time t lies within the timestamp of entry x, and (4.4), which checks that behavior b is part of the collection of behaviors defined in the interface of the type at this time. If all conditions are satisfied, the behavior application is valid.

If the validity test is satisfied, the next step is to coerce behavior b based on the implementation changes that took place prior to time t. Algorithm 4.2 performs this operation.

Algorithm 4.2 *Coerce:*

Input: An object o, a behavior b and a time t

Procedure:

$$o.B\_mapsto.B\_baseType.B\_updateChanges(t, b.B\_implementation(o.B\_mapsto.B\_baseType)) \tag{4.5}$$
$$o.B\_updateRep(t, b.B\_implementation(o.B\_mapsto.B\_baseType)) \tag{4.6}$$

In step (4.5), *B_changes* list of the base type of o is updated with the implementation changes that took place on behavior b at or before time t. The *B_updateChanges* behavior, defined on the `T_type` type, performs this update by taking t and *B_implementation* of b as arguments. It searches *B_implementation* of b for implementation changes that took place before or at the same time as t and updates the *B_changes* list with all implementation changes that have not yet been recorded in *B_changes*. For example, the behavior application `T_person`$'.B\_updateChanges(t_7, B\_birthDate.B\_implementation($`T_person`$'))$ updates the *B_changes* list of the base type of `joe` (`T_person`$'$) during the behavior application `joe`.$[t_7]B\_birthDate$. The updated *B_changes* list is shown in Figure 4.6. The object o is then updated if neccessary (4.6). The *B_updateRep* behavior, defined on the type of o, performs this update. For each behavior implementation change at time t_i that leads to a

[6]The *B_within* behavior is defined on `T_anchPrim` and checks whether one timestamp is within another timestamp.

change in the representation of o, *B_updateRep* updates o with the appropriate representation object with respect to time t_i and the time interval during which it was valid. The behaviors applicable to the representation object are those which exist in the interface of its type at t_i. The implementations of these behaviors are those which exist in the implementation histories for the type at t_i. The stored functions at t_i determine the initial state of the representation object.

Algorithm 4.3 performs the simple task of returning the appropriate representation object of o at time t.

Algorithm 4.3 *Representation:*

Input: An object o and time t

Output: An object with its representation at time t

Procedure:
    return $o.B\_validObject(t).B\_value$

The appropriate implementation f of b for the base type of o at time t is then retrieved from the *B_implementation* history of b. Algorithm 4.4 finds and returns this implementation.

Algorithm 4.4 *Implementation:*

Input: An object o, a behavior b and a time t

Output: The function that implements behavior b for object o at time t

Procedure:
    return $b.B\_implementation(o.B\_mapsto.B\_baseType).B\_validObject(t).B\_value$

A final step of the dispatch mechanism is the execution of the function returned from Algorithm 4.4 to the representation object returned by Algorithm 4.3. The *B_execute* behavior is used on functions to accomplish this. The relationships between all the algorithms are shown in Algorithm 4.5.

Algorithm 4.5 *Dispatch:*

Input: An object o, a behavior b and a time t

Output: An object resulting from the application $o.[t]b$

Procedure:

if *TemporalValidity(o,b,t)* **then**
    $Coerce(o, b, t)$
    $o@t \leftarrow Representation(o, t)$
    $f \leftarrow Implementation(o, b, t)$
    $f.B\_execute(o@t)$
else
    INVALID: object o does not exist at time t
        or behavior b not defined in the interface of o's base type at time t

4.2.5.2 Dispatch Examples

For the following examples, consider Figure 4.8, which extends the timeline of type `T_person` in Figure 4.4 by adding a behavior *B_spouse* with the computed implementation c_6 at time t_{14} and dropping the behavior *B_age* at time t_{16}. Note that an object representation will not change by adding behavior *B_spouse* and the representations will be empty after behavior *B_age* is dropped. For this example, $now > t_{16}$.

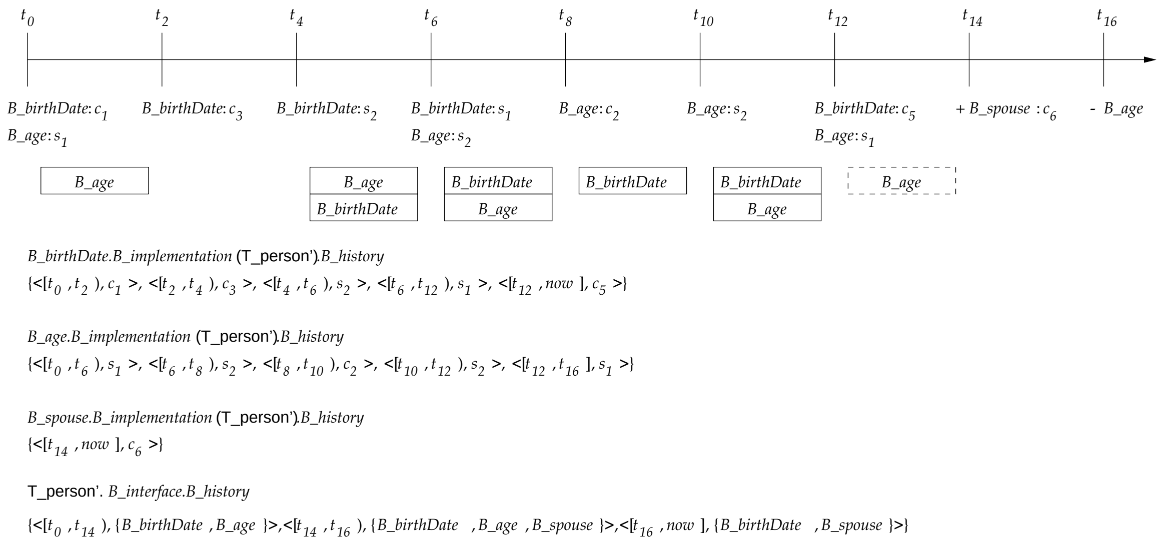


Figure 4.8: Example showing effects on implementation histories of first adding and then dropping a behavior.

Several example behavior applications using time are presented to show how the dispatch process is followed in order to determine the proper implementation and state instance that are appropriate at the given time of interest. We assume the behavior applications take place in chronological order.

Example 4.7 Behavior application joe.$[t_7]B\_birthDate$ (assuming no previous behavior application has taken place)

Validity: Object joe was created at time t_0 and exists at time *now*. Therefore, the lifespan of joe is the time interval $[t_0, now]$. Since t_7 in within this interval (i.e., lifespan), the object part of the behavior application is valid. The base type of joe is `T_person`$'$. The interface of `T_person`$'$ at time t_7 is $\{B\_birthDate, B\_age\}$. Since *B_birthDate* is part of this interface, the behavior part of the application is valid and thus the validity test is satisfied.

Coerce: The next step is to update the *B_changes* list of the base type of o and the representation history of o. These updates are performed by the behavior applications `T_person`$'.B\_updateChanges(t_7, B\_birthDate.B\_implementation($`T_person`$'))$ and joe.$B\_updateRep(t_7, B\_birthDate.B\_implementation(\texttt{T\_person}'))$, respectively. The updated *B_changes* list and representation history of o is shown in Figure 4.6.

Representation: The behavior application joe.$B\_validObject(t_7).B\_value$ returns joe@t_6, which is the appropriate representation object of joe at time t_7 (see Figure 4.6).

Implementation: The behavior application $B\_birthDate.B\_implementation(\texttt{T\_person}').B\_validObject(t_7).B\_value$ returns the appropriate implementation of *B_birthDate* for type `T_person` at time t_7, which is the stored function s_1.

Dispatch: To complete the dispatch of the behavior, the stored function s_1 is executed using the representation object joe@t_6 as an argument. This will access the first slot of the representation of joe at t_6. The represenation of joe at t_7 is the same as the one at t_6, so the behavior application accesses the appropriate birthdate slot of joe at t_7.

□

Example 4.8 Behavior application joe.$[t_3]B\_birthDate$
The validity test is satisfied. The *B_birthDate* has already been coerced to the implementations at times t_0 and t_2 since the entries $<t_0, B\_birthDate>$ and $<t_2, B\_birthDate>$ exist in the changes list of `T_person`. Therefore, *B_changes* and o remain unchanged. The representation object at t_3 is joe@t_0 (see Figure 4.6) and the implementation chosen at t_3 is the computed function c_3. The function c_3 is then applied to joe@t_0. □

Example 4.9 Behavior application joe.$[t_{12}]B\_birthDate$
The validity test is satisfied. In Algorithm 4.2, the implementation change of *B_birthDate* at time t_{12} is recorded in the *B_changes* list, and the representation history for object joe also changes since the implementation for *B_birthDate* changes from a stored function (time t_6) to a computed function (time t_{12}). Figure 4.9 shows the changes in *B_changes* and the representation history of joe.

$$\begin{array}{rcl}
\text{joe}.B_created & = & t_0 \\
\text{joe}.B_history & = & \{<[t_0,t_4), joe@t_0>, <[t_4,t_6), joe@t_4>, <[t_6,t_{12}), joe@t_6>, \\
& & <[t_{12}, now], joe@t_{12}>\} \\
\texttt{T_person}'.B_changes & = & \{<t_0, B_birthDate>, <t_2, B_birthDate>, <t_4, B_birthDate>, \\
& & <t_6, B_birthDate>, <t_{12}, B_birthDate>\}
\end{array}$$

Figure 4.9: The representation objects of joe and the changes list of `T_person` after behavior application of *B_birthDate* at time t_{12}.

The appropriate implementation for *B_birthDate* at t_{12}, which is the computed function c_5, is then applied to joe@t_{12}, which is the representation object of joe at t_{12}. □

Example 4.10 Behavior application joe.$[t_{10}]B\_age$
Now suppose a different behavior (*B_age*) is applied to object joe. The validity test is satisfied. The *B_changes* list is updated with the times of all implementation changes that took place on behavior *B_age* prior to time t_{10}. From Figure 4.8 we see that these times are t_0, t_6, t_8, and t_{10}. The behavior coercions at all these times lead to changes in the structural representation of object joe. At t_0, the implementation of *B_age* changed from undefined to stored (UC), at t_6 the implementation changed from stored to stored (SS), at t_8 the implementation changed from stored to computed (SC), and at t_{10} the implementation changed from computed to stored (CS). The new value of *B_changes* and the structural representation history of joe are shown in Figure 4.10.

$$\begin{array}{rcl} \text{joe}.B\_created & = & t_0 \\ \text{joe}.B\_history & = & \{<[t_0,t_4), joe@t_0>, <[t_4,t_6), joe@t_4>, <[t_6,t_8), joe@t_6>, \\ & & <[t_8,t_{10}), joe@t_8>, <[t_{10},t_{12}), joe@t_{10}>, <[t_{12}, now], joe@t_{12}>\} \\ \texttt{T\_person}'.B\_changes & = & \{<t_0, B\_birthDate>, <t_0, B\_age>, <t_2, B\_birthDate>, \\ & & <t_4, B\_birthDate>, <t_6, B\_birthDate>, <t_6, B\_age>, \\ & & <t_8, B\_age>, <t_{10}, B\_age>, <t_{12}, B\_birthDate>\} \end{array}$$

Figure 4.10: The representation objects of joe and the changes list of `T_person` after behavior application of *B_age* at time t_{10}.

Having updated the *B_changes* list and o, the representation object $joe@t_{10}$, and the implementation s_2 are returned from Algorithms 4.3 and 4.4 at time t_{10}. We can now apply s_2 to $joe@t_{10}$. □

Example 4.11 Behavior application joe.$[now]B\_age$
This fails the validity test because behavior *B_age* is not part of the interface of `T_person′` at time now. □

Example 4.12 Behavior application joe.$[t_{15}]B\_spouse$
The validity test is satisfied. An appropriate entry $<t_{14}, B\_spouse>$ is added to the changes list of `T_person`. The representation history of joe remains unchanged since the implementation change is from an undefined function to a computed one (UC). The representation object at time t_{14} is $joe@t_{12}$ (see Figure 4.10) and the implementation is the computed function c_6. The function c_6 is then applied to $joe@t_{12}$. □

Example 4.13 Behavior application jane.$[t_7]B\_age$
Suppose the object jane was created at time t_6. The validity test is satisfied. The changes list of `T_person` remains unchanged since it has already been updated with the implementation changes of *B_age* prior to t_7 (see Figure 4.10). The structural representation history of jane however, is updated to reflect the behavior coercions that took place at or after jane was created and before or at t_7. This is shown in Figure 4.11.

The time t_7 is used to find the appropriate representation object for jane and the correct implementation of *B_age* for type `T_person′`. The representation object chosen is $jane@t_6$

$$
\begin{array}{lcl}
\mathsf{jane}.\mathit{B\_created} & = & t_6 \\
\mathsf{jane}.\mathit{B\_history} & = & \{<[t_6, now], jane@t_6>\}
\end{array}
$$

Figure 4.11: The representation objects of jane after behavior application of *B_age* at time t_7.

and the implementation returned is the stored function s_2. This function is then applied to $jane@t_6$. □

4.2.6 Immediate Object Conversions

The temporal infrastructure proposed in this thesis is sufficiently powerful to support coercion approaches other than deferred coercion. In this section, it is shown how the immediate object coercion approach of schema change propagation can be implemented using the model presented in Sections 4.2.3-4.2.5. In immediate object coercion, changes are immediately propagated to the instances. This would mean that each time the implementation of a behavior changes, the behavior is coerced to the newer implementation at that time and the structural representations of all objects of that type are updated, if necessary. These changes are recorded in the *B_implementation* and *B_changes* behaviors, respectively. Figure 4.12 shows the changes list for `T_person` and the representation history of object joe when immediate object coercion is used for the behavior implementation changes shown in Figure 4.8.

$$
\begin{array}{rcl}
\mathsf{joe}.\mathit{B\_created} & = & t_0 \\
\mathsf{joe}.\mathit{B\_history} & = & \{<[t_0, t_4), joe@t_0>, <[t_4, t_6), joe@t_4>, <[t_6, t_8), joe@t_6>, \\
& & <[t_8, t_{10}), joe@t_8>, <[t_{10}, t_{12}), joe@t_{10}>, <[t_{12}, t_{16}), joe@t_{12}>, \\
& & <[t_{16}, now], joe@t_{16}>\} \\
\mathsf{jane}.\mathit{B\_created} & = & t_6 \\
\mathsf{jane}.\mathit{B\_history} & = & \{<[t_6, t_8), jane@t_6>, <[t_8, t_{10}), jane@t_8>, <[t_{10}, t_{12}), jane@t_{10}>, \\
& & <[t_{12}, t_{16}), jane@t_{12}>, <[t_{16}, now], jane@t_{16}>\} \\
\mathtt{T\_person}'.\mathit{B\_changes} & = & \{<t_0, B\_birthDate>, <t_0, B\_age>, <t_2, B\_birthDate>, \\
& & <t_4, B\_birthDate>, <t_6, B\_birthDate>, <t_6, B\_age>, \\
& & <t_8, B\_age>, <t_{10}, B\_age>, <t_{12}, B\_birthDate>, \\
& & <t_{12}, B\_age>, <t_{14}, B\_spouse>\}
\end{array}
$$

Figure 4.12: The representation objects of joe and jane, and the changes list of `T_person` for immediate object coercion.

The *B_changes* behavior for `T_person′` shows that each time the implementation of a behavior changes after the object was created, the behavior is coerced to the newer implementation since immediate object coercion is used. For example, after joe is created, the implementation of behavior *B_birthDate* changes at times t_2, t_4, t_6, and t_{12} (see Figure 4.8). Subsequently, *B_birthDate* is also coerced to the newer implementations at these times. This is shown in the changes list of `T_person`. The representation history of an object is only updated when a change to the representation of an object occurs due to the coercion of one of its behaviors. For example, although the behavior *B_birthDate* is coerced to a newer implementation at time t_2, the representation of joe is unaffected since the implementation is changed from one computed function to another computed function (see Figure 4.8). Therefore, joe is unchanged at t_2. A similar situation occurs at t_{14} for joe and jane when behavior *B_spouse* is added to type `T_person'`.

With immediate coercion, if a behavior implementation change at time t for a type T necessitates an update of the representation of an object, the change is recorded in the representation histories of *all objects* of type T that exist at time t. This is exemplified in Figure 4.12 where the tuples in representation histories of objects joe and jane (of type `T_person`) are updated at the same time after jane was created (from t_8 to *now*).

In the immediate coercion approach, Algorithm 4.2 is carried out at the time of behavior implementation change, and not during a behavior application process as was the case in deferred coercion. The only difference to the dispatch algorithm is that invocation of coerce is not necessary. The example below shows how the dispatch process is followed when immediate object coercion is used for the behavior application given in Example 4.7.

Example 4.14 Behavior application joe.$[t_7]B\_birthDate$
The validity test is satisfied. The appropriate representation object for joe at time t_7 is $joe@t_6$, while the appropriate implementation of *B_birthDate* for type `T_person'` is the stored function s_1. s_1 can now be applied to $joe@t_6$. The function and representation are correct for joe since the implementation of behavior *B_birthDate* changed at time t_6 for this object, and *B_birthDate* was coerced to the new version at the same time. □

From the above example, it is noted that the function and the representation object obtained for joe using immediate object coercion are the same as those obtained in Example 4.7, in which deferred object coercion was used. The function chosen in both cases is the stored function s_1, and the representation object chosen for joe in both cases is $joe@t_6$. This equivalence of deferred and immediate object coercion strategies is neccessary.

Chapter 5

Conclusions

5.1 Summary and Contributions

The first and most important result of this thesis is the definition of an object-oriented temporal framework which supports the diverse notions of time under a single infrastructure. The framework is expressive in that it can be used to accommodate the temporal needs of different real-world applications, and also reflect different temporal object models that have been reported in the literature. Using the object-oriented type system to structure the design space of temporal object models and identify the dependencies within and among the design dimensions helps simplify the presentation of the otherwise complex domain of time. The framework is extensible in that additional temporal features can be added as long as the relationships between the design dimensions are maintained. The focus in this thesis is on the unified provision of temporal features which can be used by temporal object models according to their temporal needs. Once these are in place, the model can then define other object-oriented features to support its application domain. The temporal framework also provides a means of comparing temporal objects models according to the design dimensions identified in Section 2.3.1. This helps identify the strengths and weaknesses of the different models.

The fundamental contributions of the temporal framework are the following:

1. A design space for temporal models. This includes the design dimensions, their temporal features, and the dependencies within and among the design dimensions.

2. The capability of accommodating different applications that require temporal support.

3. The capability of representing various existing temporal object models.

4. The capability of analyzing and comparing different temporal object models based on the design dimensions

The second important result described in this thesis is the design and development of a temporal object model that implements the temporal framework and provides concrete and consistent semantics for the different temporal features necessary for their coexistence. The work on the temporal object model is conducted within the context of the TIGUKAT system [ÖPS+95]. The behavioral and uniform features of the TIGUKAT model are exploited in order to incorporate time so as to accommodate different applications that require varying temporal support. Consequently, the TIGUKAT temporal object model consists of an extensible set of primitive time types with a rich set of behaviors to consistently and uniformly model the diverse features of time. In [LGÖS97], the temporal model is used to manage temporal relationships which is inherent in multimedia data such as video, while in [GÖS97a], the temporal model is used to store and retrieve historical information commonly found in clinical trials.

The contributions and novelty of the TIGUKAT temporal object model are the following:

1. The model provides a novel approach to the treatment of granularity in temporal data. Granularities are modeled as unit unanchored temporal primitives. Granularities are accommodated within the context of calendars and granularity conversions are presented and discussed in terms of unanchored durations of time, unlike other works where the emphasis has only been on granularity conversions with respect to anchored temporal data.

2. The model also provides a means to represent both anchored and unanchored temporal primitives at different and mixed granularities and gives the semantics of operations on these primitives.

3. The model supports multiple calendars and provides means for granularity conversions and operations across calendars.

4. Both discrete and continuous domains of time are supported. This is in contrast to previous work which deals with only a single time domain that is usually discrete.

5. Different types of orderings (linear, branching) are supported, unlike most temporal object models that support only linear time.

6. Since the TIGUKAT model is behavioral, different dimensions of time (e.g., valid and transaction time dimensions) are represented using separate behaviors in contrast to other works in which they are structurally combined in a single behavior.

The provision of time in the TIGUKAT object model establishes a platform from which temporality can be used to investigate advanced database features such as schema evolution. In this thesis a uniform treatment of schema evolution using the TIGUKAT temporal object model is presented. Schema evolution is managed by exploiting the functionality of the TIGUKAT temporal object model. This constitutes the third major contribution of this thesis.

Using time to maintain and manage schema changes gives substantial flexibility in the software design process. Given that the applications supported by ODBMSs need support for incremental development and experimentation with changing and evolving schema, a temporal domain is a natural means for managing changes in schema and ensuring consistency of the system. The evolution history of the interface of types, which includes the inherited and native behaviors of each type, describes the semantics of types through time. Using the interface histories, the interface of a type can be reconstructed at any time of interest. The evolution histories of the supertype and subtype links of types describe the structure of the lattice through time. Using these histories, the structure of the lattice can be reconstructed at any time of interest. The implementation histories of behaviors give the implementations of behaviors on types at any time of interest. From these, one can reconstruct the representation of objects by examining the stored functions associated with behaviors at a given time. The TIGUKAT query language gives designers a practical way of accessing temporal information in their experimental and incremental design phases.

The strategy for managing schema evolution for ODBMSs in this thesis is characterized by four novel concepts:

1. The schema evolution strategy is based on a uniform temporal object model. Consequently, no special concepts are introduced for modeling schema changes which are expressed as changes to the behaviors defined on types. These changes are tracked using the same `T_history` mechanism that is used for modeling temporal changes to non-schema objects in the database.

2. In addition to schema changes, the strategy supports lossless recording of these changes, allowing historical queries.

3. The strategy supports both deferred (lazy) object update semantics and immediate object update semantics using the same basic algorithms. Only the application time of the algorithms is changed to produce the desired update semantics.

4. The granularity of schema changes is finer than traditional approaches that require a complete type change every time a single behavior changes. We handle behavior changes individually. This approach has two distinct advantages depending on whether deferred or immediate update semantics are used. If deferred update semantics are used, the finer granularity results in an even "lazier" update semantics, since when a behavior is applied to an object, only part of the object's structure needs to be updated, to reflect changes for only that particular behavior. Updates resulting from other behavior changes are delayed until they are needed by other behavior applications. If immediate update semantics are used, then the update can be done more quickly since the system knows that changes to the affected type are localized to the single behavior that was just changed. Since the major drawback of immediate update semantics is the speed of update, this is important.

Support for historical queries potentially has a profound effect on ODBMS behavior dispatch. In traditional behavior dispatch, each behavior on a type is bound to a single function (implementation) and dispatch is a mapping of behavior-type pairs to functions. With recorded schema evolution, each behavior may be bound to a different function at different times. Therefore, the dispatch process must map a three-tuple (behavior, type, time) to a function. Unfortunately the domain of the temporal argument is very large compared to the domain of all behaviors or all types so standard dispatch techniques do not work very well. This thesis provides a temporal dispatch algorithm to demonstrate that no new concepts need to be added to the schema evolution model to solve the temporal dispatch problem.

The fourth contribution of this thesis is a toolkit which allows users/temporal model designers to interact with the framework at a high level and generate specific framework instances for their own applications.

5.2 Future Research

A number of interesting future research avenues can be outlined from the work presented in this thesis. An interesting first step would be to build query semantics on top of the present framework. This will involve addressing issues such as: how the choices of different design dimensions affect the query semantics; what kind of query constructs are needed; what properties should be provided; how are these properties used, to name a few.

Section 3.3.4 showed that it is possible to establish computationally inexpensive yet quite precise formulae for lower and upper bound coefficients in the granularity conversion process. Currently, the derivation of these formulae has to be done by a database administrator; their automatic derivation would be an interesting topic for future research. In handling indeterminacy, it would useful to consider modality, multiple-valued logics, and probabilistic approaches [DS93, MPB92, CPP96] for dealing with uncertainty in relationships.

To overcome the corrective nature of schema evolution, the concept of *schema versioning* in ODBMSs has been proposed [SZ86, SZ87, KC88, ALP91, MS92, MS93]. In most of these systems, a change to a schema object may result in a new *version* of the schema object, or the schema in general. However, schema changes are usually of a finer granularity than definable versions. This implies that not every schema change should necessarily result in a new version. Rather, one should be able to define a version during any stable period in the evolutionary history of the schema. Within a particular version, the evolution of the schema should be traceable. For example, in an engineering design application many components of an overall design may go through several modifications in order to produce a released product. Furthermore, each intermediate version of the component may have certain properties that need to be retained as a historical record of that particular component (the different versions may have been used in other products). The inter-connection of the various versions of components also gives rise to versions of the overall design. The resulting designs may be part of others and so on. The contention of this thesis is that schema evolution using temporal modeling sets the stage for full-fledged version control.

The schema evolution policies reported in this thesis can be used as a basis for version control in ODBMSs.

Bibliography

[All84] J. F. Allen. Towards a General Theory of Action and Time. *Artificial Intelligence*, 23(123):123–154, July 1984.

[ALP91] J. Andany, M. Leonard, and C. Palisser. Management of Schema Evolution in Databases. In *Proc. 17th Int'l Conf. on Very Large Data bases*, pages 161–170, September 1991.

[Ari86] G. Ariav. A Temporally Oriented Data Model. *ACM Transactions on Database Systems*, 11(4):499–527, December 1986.

[Ari91] G. Ariav. Temporally oriented data definitions: Managing schema evolution in temporally oriented databases. *Data & Knowledge Engineering*, (6):451–467, 1991.

[ATGL96] A-R. Adl-Tabatabai, T. Gross, and G-Y. Lueh. Code Reuse in an Optimizing Compiler. In *Proc. of the Int'l Conf on Object-Oriented Programming: Systems, Languages, and Applications - OOPSLA '96*, pages 51–68, October 1996.

[BFG97] E. Bertino, E. Ferrari, and G. Guerrini. T_Chimera - A Temporal Object-Oriented Data Model. *Theory and Practice of Object Systems*, 3(2):103–125, 1997.

[BH89] A. Bjørnerstedt and C. Hültén. Version Control in an Object-Oriented Architecture. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 18, pages 451–485. Addison Wesley, September 1989.

[BKKK87] J. Banerjee, W. Kim, H-J. Kim, and H.F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 311–322, May 1987.

[BKP86] H. Barringer, R. Kuiper, and A. Pnueli. A Really Abstract Concurrent Model and its Temporal Logic. In *Proc. of the 13th ACM Symposium on Principles of Programming Languages*, pages 173–183, 1986.

[BMJ94] E. Braunwald, D.B. Mark, and R.H. Jones. Diagnosing and Managing Unstable Angina - Quick Reference Guide for Clinicians. (10. AHCPR Publication No. 94-0603), May 1994.

[Böh95] M.H. Böhlen. Temporal Database System Implementations. *ACM SIGMOD Record*, 24(4):53–60, December 1995.

[BP85] F. Barbic and B. Pernici. Time Modeling in Office Information Systems. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 51–62, May 1985.

[Cat94] R. Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, 1994.

[CC87] J. Clifford and A. Crocker. The Historical Relational Data Model (HRDM) and Algebra Based on Lifespans. In *Proc. 3rd Int'l. Conf. on Data Engineering*, pages 528–537, February 1987.

[CG93] T.S. Cheng and S.K. Gadia. An Object-Oriented Model for Temporal Databases. In *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, pages N1–N19, June 1993.

[Cho94] J. Chomicki. Temporal Query Languages: A Survey. In D. Gabbay and H. Ohlbach, editors, *Proceedings of the International Conference on Temporal Logic*, pages 506–534. Lecture Notes in Computer Science, Vol. 827, Springer Verlag, July 1994.

[CITB92] W.W. Chu, I.T. Ieong, R.K. Taira, and C.M. Breant. A Temporal Evolutionary Object-Oriented Data Model and Its Query Language for Medical Image Management. In *Proc. 18th Int'l Conf. on Very Large Data Bases*, pages 53–64, August 1992.

[CJR87] R.H. Campbell, G.M. Johnston, and V.F. Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *Operating Systems Review*, 21(3):9–17, 1987.

[CK94] S. Chakravarthy and S-K. Kim. Resolution of Time Concepts in Temporal Databases. *Information Sciences*, 80(1-2):91–125, September 1994.

[CMR91] E. Corsetti, A. Montanari, and E. Ratto. Dealing with Different Time Granularities in Formal Specifications of Real-Time Systems. *The Journal of Real-Time Systems*, 3(2):191–215, 1991.

[Cod70] E.F. Codd. A Relational Model for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[CPP95] C. Combi, F. Pinciroli, and G. Pozzi. Managing Different Time Granularities of Clinical Information by an Interval-Based Temporal Data Model. *Methods of Information in Medicine*, 34(5):458–474, 1995.

[CPP96] C. Combi, F. Pinciroli, and G. Pozzi. Managing Time Granularity of Narrative Clinical Information: The Temporal Data Model TIME-NESIS. In L. Chittaro, S. Goodwin, H. Hamilton, and A. Montanari, editors, *Third International Workshop on Temporal Representation and Reasoning (TIME'96)*, pages 88–93. IEEE Computer Society Press, 1996.

[CR88] J. Clifford and A. Rao. A Simple, General Structure for Temporal Domains. In C. Rolland, F. Bodart, and M. Leonard, editors, *Temporal Aspects in Information Systems*, pages 17–30. North-Holland, 1988.

[CS93] R. Chandra and A. Segev. Managing Temporal Financial Data in an Extensible Database. In *Proc. 19th Int'l Conf. on Very Large Data Bases*, pages 302–313, August 1993.

[CSS94] R. Chandra, A. Segev, and M. Stonebraker. Implementing Calendars and Temporal Rules in Next-Generation Databases. In *Proc. 10th Int'l. Conf. on Data Engineering*, pages 264–273, February 1994.

[Dat87] C.J. Date. *A Guide to SQL Standard.* Addison Wesley, 1987.

[DDS94] W. Dreyer, A.K. Dittrich, and D. Schmidt. An Object-Oriented Data Model for a Time Series Management System. In *Proc. 7th International Working Conference on Scientific and Statistical Database Management*, pages 186–195, September 1994.

[DM94] A.K. Das and M.A. Musen. A Temporal Query System for Protocol-Directed Decision Support. *Methods of Information in Medicine*, 33:358–370, 1994.

[DR90] N. Dershowitz and E.M. Reingold. Calendrical calculations. *Software – Practice & Experience*, 20(9):899–928, September 1990.

[DS93] C.E. Dyreson and R.T. Snodgrass. Valid-time Indeterminacy. In *Proc. 9th Int'l. Conf. on Data Engineering*, pages 335–343, April 1993.

[DSS94] C.E. Dyreson, M.D. Soo, and R.T. Snodgrass. The TSQL2 Data Model for Time. A TSQL Commentary. September 1994.

[DW92] U. Dayal and G. Wuu. A Uniform Approach to Processing Temporal Queries. In *Proc. 8th Int'l. Conf. on Data Engineering*, pages 407–418, August 1992.

[Dyr96] C. Dyreson. June 1996. Private correspondence.

[EGS93] O. Etzion, A. Gal, and A. Segev. Temporal Active Databases. In *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, June 1993.

[Flo91] R. Flowerdew. *Geographical Information Systems*. John Wiley and Sons, 1991. Volume 1.

[FMZ94] F. Ferrandina, T. Meyer, and R. Zicari. Implementing Lazy Database Updates for an Object Database System. In *Proc. 20th Int'l Conf. on Very Large Data Bases*, pages 261–272, September 1994.

[FMZ+95] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec. Schema and Database Evolution in the O_2 Object Database System. In *Proc. 21st Int'l Conf. on Very Large Data Bases*, pages 170–181, September 1995.

[Gad88] S. Gadia. A Homogeneous Relational Model and Query Languages for Temporal Databases. *ACM Transactions on Database Systems*, 13(4):418–448, December 1988.

[GLÖS97] I.A. Goralwalla, Yuri Leontiev, M.T. Özsu, and Duane Szafron. Modeling Temporal Primitives: Back to Basics. In *Proc. Sixth Int'l. Conf. on Information and Knowledge Management*, pages 24–31, November 1997.

[GÖ93] I.A. Goralwalla and M.T. Özsu. Temporal Extensions to a Uniform Behavioral Object Model. In *Proc. 12th Int'l Conf. on the Entity Relationship Approach (ER'93)*, pages 115–127, December 1993.

[Gor92] I. Goralwalla. An Implementation of a Temporal Relational Database Management System. Master's thesis, Bilkent University, Ankara, Turkey, 1992.

[GÖS97a] I.A. Goralwalla, M.T. Özsu, and D. Szafron. Modeling Medical Trials in Pharmacoeconomics using a Temporal Object Model. *Computers in Biology and Medicine - Special Issue on Time-Oriented Systems in Medicine*, 27(5):369 – 387, 1997.

[GÖS97b] I.A. Goralwalla, M.T. Özsu, and Duane Szafron. A Framework for Temporal Data Models: Exploiting Object-Oriented Technology. In R. Ege, M. Singh, and B. Meyer, editors, *Twenty-third International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS USA '97)*, pages 16–30. IEEE Computer Society, July 1997.

[GÖS98] I.A. Goralwalla, M.T. Özsu, and D. Szafron. An Object-Oriented Framework for Temporal Data Models. In O. Etzion, S. Jajodia, and S. Sripada, editors, *Temporal Databases: Research and Practice*. Springer Verlag, 1998. To appear.

[GSÖP97] I.A. Goralwalla, D. Szafron, M.T. Özsu, and R.J. Peters. Managing Schema Evolution using a Temporal Object Model. In *Proc. 16th International Conference on Conceptual Modeling (ER'97)*, pages 71–84, November 1997. Proceedings published as Lecture Notes in Computer Science, David Embley and Robert Goldstein (eds.), Springer-Verlag, 1997.

[GSÖP98] I.A. Goralwalla, D. Szafron, M.T. Özsu, and R.J. Peters. A Temporal Approach to Managing Schema Evolution in Object Database Systems. *Data & Knowledge Engineering*, November 1998. To appear.

[GTC+90] S. Gibbs, D.C. Tsichritzis, E. Casais, O.M. Nierstrasz, and X. Pintado. Class Management for Software Communities. *Communications of the ACM*, 33(9):90–103, September 1990.

[GTÖ95] I.A. Goralwalla, A.U. Tansel, and M.T. Özsu. Experimenting with Temporal Relational Databases. In *Proc. Fourth Int'l. Conf. on Information and Knowledge Management*, pages 296–303, October 1995.

[HJE95] H. Hüni, R. Johnson, and R. Engel. A Framework for Network Protocol Software. In *Proc. of the Int'l Conf on Object-Oriented Programming: Systems, Languages, and Applications - OOPSLA '95*, pages 358–369, 1995.

[HKOS96] W.H. Harrison, H. Kilov, H.L. Ossher, and I. Simmonds. From Dynamic Supertypes to Subjects: a Natural way to Specify and Develop Systems. *IBM Systems Journal*, 35(2):244–256, 1996.

[HS97] W. Holst and D. Szafron. A General Framework For Inheritance Management and Method Dispatch in Object-Oriented Languages. In *Proc. European Conference on Object-Oriented Programming (ECOOP'97)*, pages 276–301, 1997.

[JF88] R.E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[JJGB92] L.M. Jolicoeur, A.J. Jones-Grizzle, and J.G. Boyer. Guidelines for performing a pharmacoeconomic analysis. *American Journal of Hospital Pharmacy*, 49:1741–1747, July 1992.

[KBCG90] W. Kim, J. Banerjee, H.T. Chou, and J.F. Garza. Object-oriented database support for CAD. *Computer Aided Design*, 22(8):469–479, 1990.

[KC88] W. Kim and H-J. Chou. Versions of Schema for Object-Oriented Databases. In *Proc. 14th Int'l Conf. on Very Large Data Bases*, pages 148–159, 1988.

[KFT91] M.G. Kahn, L.M. Fagan, and S. Tu. Extensions to the Time-Oriented Database Model to Support Temporal Reasoning in Medical Expert Systems. *Methods of Information in Medicine*, 30:4–14, 1991.

[KGBW90] W. Kim, J.F. Garza, N. Ballou, and D. Wolek. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, March 1990.

[KKR90] P.C. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint Query Languages. In *Proc. of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 299–313, April 1990.

[Kli93] N. Kline. An Update of the Temporal Database Bibliography. *ACM SIGMOD Record*, 22(4):66–80, December 1993.

[KP88] G.E. Krasner and S.T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, pages 26–49, August-September 1988.

[KS92] W. Kafer and H. Schoning. Realizing a Temporal Complex-Object Data Model. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 266–275, June 1992.

[LEW96] J.Y. Lee, R. Elmasri, and J. Won. Specification of Calendars and Time Series for Temporal Databases. In *Proc. 15th International Conference on Conceptual Modeling (ER'96)*, pages 341–356, October 1996. Proceedings published as Lecture Notes in Computer Science, Volume 1157, Bernhard Thalheim (editor), Springer-Verlag, 1996.

[LGÖS97] J.Z. Li, I.A. Goralwalla, M.T. Özsu, and Duane Szafron. Modeling Video Temporal Relationships in an Object Database Management System. In *SPIE Proceedings of Multimedia Computing and Networking (MMCN97)*, pages 80–91, February 1997.

[LH90] B.S. Lerner and A.N. Habermann. Beyond Schema Evolution to Database Reorganization. In *ECOOP/OOPSLA '90 Proceedings*, pages 67–76, October 1990.

[LJ88] N. Lorentzos and R. Johnson. Extending Relational Algebra to Manipulate Temporal Data. *Information Systems*, 13(3):289–296, 1988.

[Lor94] N. Lorentzos. DBMS Support for Non-Metric Measuring Systems. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):945–953, December 1994.

[MMCR92] A. Montanari, E. Maim, E. Ciapessoni, and E. Ratto. Dealing with Time Granularity in Event Calculus. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 702–712, June 1992.

[MPB92] R. Maiocchi, B. Pernici, and F. Barbic. Automatic Deduction of Temporal Information. *ACM Transactions on Database Systems*, 17(4):647–688, 1992.

[MS90] E. McKenzie and R. Snodgrass. Schema evolution and the relational algebra. *Information Systems*, 15(2):207–232, 1990.

[MS92] S.R. Monk and I. Sommerville. A Model for Versioning of Classes in Object-Oriented Databases . In *10th British National Conference on Databases (BNCOD '92), Aberdeen, Scotland July 1992*, pages 42–58, July 1992.

[MS93] S. Monk and I. Sommerville. Schema Evolution in OODBs using Class Versioning. *ACM SIGMOD Record*, 22(3):16–22, September 1993.

[NA89] S.B. Navathe and R. Ahmed. A Temporal Relational Model and a Query Language. *Information Sciences*, 49:147–175, 1989.

[NR89] G.T. Nguyen and D. Rieu. Schema evolution in object-oriented database systems. *Data & Knowledge Engineering*, 4:43–67, 1989.

[ÖPS+95] M.T. Özsu, R.J. Peters, D. Szafron, B. Irani, A. Lipka, and A. Munoz. TIGUKAT: A Uniform Behavioral Objectbase Management System. *The VLDB Journal*, 4:100–147, August 1995.

[ÖS95] G. Özsoyoğlu and R. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, August 1995.

[Pea94] N. Pissinou and et. al. Towards an Infrastructure for Temporal Databases: Report of an Invitational ARPA/NSF Workshop. *ACM SIGMOD Record*, 23(1):35–51, March 1994.

[Pet94] R.J. Peters. *TIGUKAT: A Uniform Behavioral Objectbase Management System.* PhD thesis, University of Alberta, 1994.

[PLL96] M.J. Perez-Luque and T.D.C. Little. A Temporal Reference Framework for Multimedia Synchronization. *IEEE Journal on Selected Areas in Communications*, 14(1):36–51, January 1996.

[PLÖS93] R.J. Peters, A. Lipka, M.T. Özsu, and D. Szafron. An Extensible Query Model and Its Languages for a Uniform Behavioral Object Management System. In *Proc. Second Int'l. Conf. on Information and Knowledge Management*, pages 403–412, November 1993.

[PM92] N. Pissinou and K. Makki. A Framework for Temporal Object Databases. In *Proc. First Int'l. Conf. on Information and Knowledge Management*, pages 86–97, November 1992.

[PÖ93] R.J. Peters and M.T. Özsu. Reflection in a Uniform Behavioral Object Model. In *Proc. 12th Int'l Conf. on the Entity Relationship Approach (ER'93)*, pages 37–49, December 1993.

[PÖ97] R.J. Peters and M.T. Özsu. An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems. *ACM Transactions on Database Systems*, 22(1):75–114, March 1997.

[PS87] D.J. Penney and J. Stein. Class Modification in the GemStone Object-Oriented DBMS. In *Proc. of the Int'l Conf on Object-Oriented Programming: Systems, Languages, and Applications*, pages 111–117, October 1987.

[Rev90] P.Z. Revesz. A Closed Form for Datalog Queries with Integer Order. In *International Conference on Database Theory*, pages 187–201, 1990.

[Rod91] J.F. Roddick. Dynamically Changing Schemas within Database Models. *Australian Computer Journal*, 23(3):105–109, 1991.

[Rod92] J.F. Roddick. SQL/SE- A Query Language Extension for Databases Supporting Schema Evolution. *ACM SIGMOD Record*, 21(3):10–16, 1992.

[Rod95] J.F. Roddick. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology*, 37(7):383–393, 1995.

[RS91] E. Rose and A. Segev. TOODM - A Temporal Object-Oriented Data Model with Temporal Constraints. In *Proc. 10th Int'l Conf. on the Entity Relationship Approach*, pages 205–229, October 1991.

[RS93a] E. Rose and A. Segev. TOOA - A Temporal Object-Oriented Algebra. In *Proceedings of the European Conference on Object-Oriented Programming*, July 1993.

[RS93b] E. Rose and A. Segev. TOOSQL - A Temporal Object-Oriented Query Language. In *Proc. 12th Int'l Conf. on the Entity Relationship Approach (ER'93)*, pages 128–138, December 1993.

[SA85] R. Snodgrass and I. Ahn. A Taxonomy of Time in Databases. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 236–246, May 1985.

[Sar90] N. Sarda. Extensions to SQL for Historical Databases. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):220–230, June 1990.

[SC91] S.Y.W. Su and H.M. Chen. A Temporal Knowledge Representation Model OSAM*/T and its Query Language OQL/T. In *Proc. 17th Int'l Conf. on Very Large Data bases*, pages 431–442, 1991.

[Sci94] E. Sciore. Versioning and Configuration Management in an Object-Oriented Data Model. *The VLDB Journal*, 3:77–106, 1994.

[Sjø93] Dag Sjøberg. Quantifying Schema Evolution. *Information and Software Technology*, 35(1):35–44, January 1993.

[SJS95] A. Segev, C.S. Jensen, and R.T. Snodgrass. Report on the 1995 Intl. Workshop on Temporal Databases. *ACM SIGMOD Record*, 24(4):46–52, 1995.

[Sno86] R. Snodgrass. Research Concerning Time in Databases: Project Summaries. *ACM SIGMOD Record*, 15(4), December 1986.

[Sno87] R.T. Snodgrass. The Temporal Query Language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[Sno90] R. Snodgrass. Temporal Databases: Status and Research Directions. *ACM SIGMOD Record*, 19(4):83–89, December 1990.

[Sno92] R.T. Snodgrass. Temporal Databases. In *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*, pages 22–64. Springer-Verlag, LNCS 639, 1992.

[Sno95a] R. Snodgrass. Temporal Object-Oriented Databases: A Critical Comparison. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability and Beyond*, pages 386–408. Addison-Wesley/ACM Press, 1995.

[Sno95b] R. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

[Sno96] R. Snodgrass. May 1996. Private correspondence.

[Soo91] M.D. Soo. Bibliography on Temporal Databases. *ACM SIGMOD Record*, 20(1):14–23, 1991.

[SRH90] M. Stonebraker, L.A. Rowe, and M. Hirohama. The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March 1990.

[SS77] J.M. Smith and D.C.P. Smith. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems*, 2(2):105–133, 1977.

[SS88] R. Stam and R. Snodgrass. A Bibliography on Temporal Databases1. *IEEE Database Engineering*, 7(4):231–239, December 1988.

[SZ86] A.H. Skarra and S.B. Zdonik. The Management of Changing Types in an Object-Oriented Database. In *Proc. of the Int'l Conf on Object-Oriented Programming: Systems, Languages, and Applications*, pages 483–495, September 1986.

[SZ87] A.H. Skarra and S.B. Zdonik. Type Evolution in an Object-Oriented Database. In *Research Directions in Object-Oriented Programming*, pages 393–415. M.I.T. Press, 1987.

[Tal94] Taligent, Inc., Cupertino, CA. *Building Object-Oriented Frameworks*, 1994. White Paper.

[Tan86] A. Tansel. Adding Time Dimension to Relational Model and Extending Relational Algebra. *Information Systems*, 13(4):343–355, 1986.

[TCG+93] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings, 1993.

[TK96] V.J. Tsotras and A. Kumar. Temporal Database Bibliography Update. *ACM SIGMOD Record*, 25(1):41–51, March 1996.

[VL90] J.M. Vlissides and M.A. Linton. Unidraw: A Framework for Building Domain-Specific Graphical Editors. *ACM Transactions on Information Systems*, 8(3):237–268, July 1990.

[WBBJ97] X.S. Wang, C. Bettini, A. Brodsky, and S. Jajodia. Logical Design for Temporal Databases with Multiple Granularities. *ACM Transactions on Database Systems*, 22:115–170, June 1997.

[WD92] G. Wuu and U. Dayal. A Uniform Model for Temporal Object-Oriented Databases. In *Proc. 8th Int'l. Conf. on Data Engineering*, pages 584–593, Tempe, USA, February 1992.

[WJL91] G. Wiederhold, S. Jajodia, and W. Litwin. Dealing with Granularity of Time in Temporal Databases. In R. Andersen, J.A. Bubenko Jr., and A. Solvberg, editors, *Advanced Information Systems Engineering, 3rd Int'l Conference CAiSE '91*, pages 124–140. Springer-Verlag, 1991.

[WJS93] X.S. Wang, S. Jajodia, and V. Subrahmanian. Temporal Modules: An Approach Toward Temporal Databases. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 227–236, 1993.

[WLH90] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris Architecture and Implementation. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):63–75, March 1990.

Appendix A

Multiple Calendar Support

In many applications [CS93, CSS94, DDS94], it is desirable to have multiple calendars that have different calendric granularities. One way to meet this requirement is to provide system support for multiple calendars with a large number of calendric granularities. However, this has high overhead, and inevitably, there will be applications that will need calendars and calendric granularities beyond a reasonable set provided by the system. It is, therefore, important for the model to be extensible and not limited to a predefined set of calendars and calendric granularities. In this appendix, extensions of the work presented in Section 3.2-3.3 are given when multiple calendars are involved. Only those sections that need to be extended are described.

A.1 Calendars

A.1.1 Calendric Granularities

Each calendric granularity in a calendar has a reference to a set of similar calendric granularities belonging to different calendars, hereafter referred to as Set_{G_A}, associated with it. Set_{G_A} contains calendric granularities which have the same time duration as G_A. For example, the calendric granularities G_{month} and $G_{academicMonth}$ have references to the set $\{month, academicMonth\}$. More specifically, $Set_{G_{month}} = Set_{G_{academicMonth}} = \{month, academicMonth\}$. Set_{G_A} is utilized when a calendric granularity of one calendar needs to be converted to a calendric granularity with the same time duration but belonging to a different calendar. For example, the span 2 *months* is equivalent to the span 2 *academicMonths* when converted to the calendric granularity of *academicMonths*.

Example A.1 Figure A.1 shows the hierarchical calendric structures of two real-world calendars namely, the Gregorian and Academic calendars. We use these two example calendars in the following discussion. □

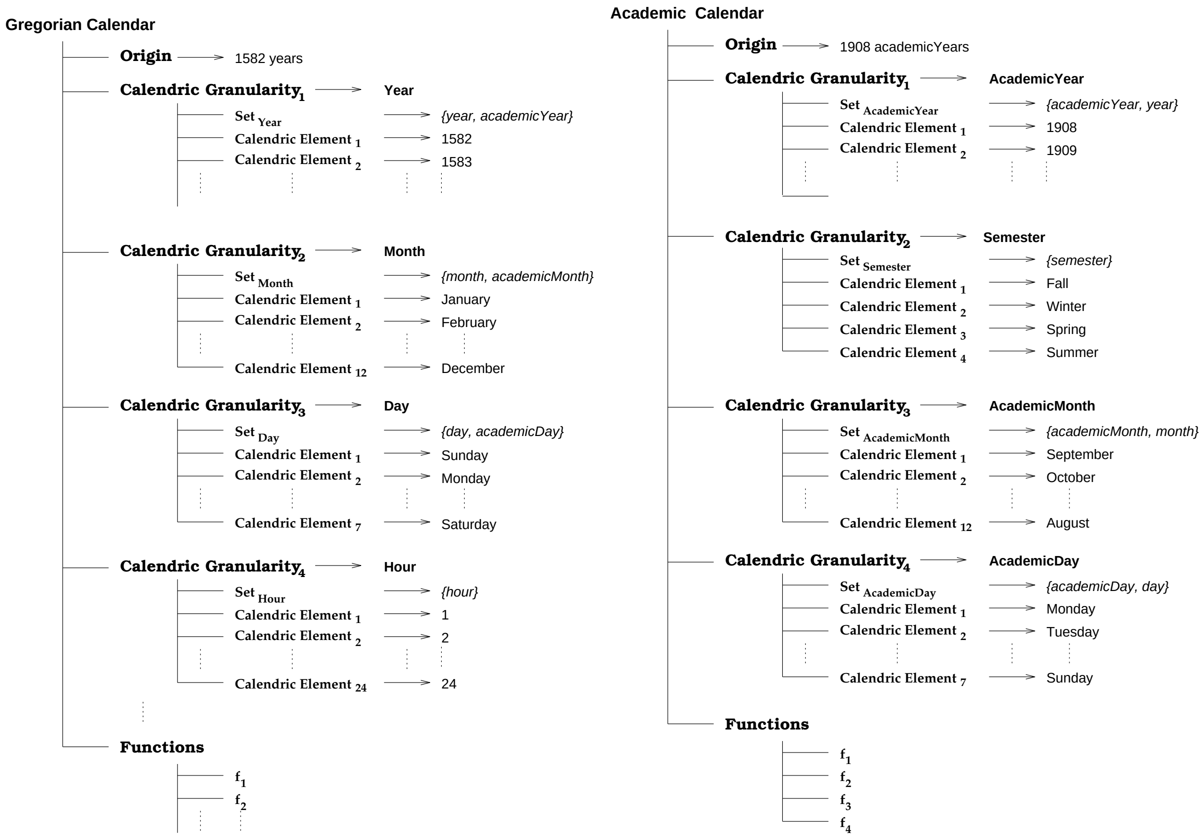


Figure A.1: The Gregorian and Academic calendric structures.

A.1.2 Conversions between Calendric Granularities

In Section 3.2.3 the derivations for $lbf(K, G_A, G_B)$ and $ubf(K, G_A, G_B)$ when G_A and G_B belong to the same calendar were considered. In this section, the case when G_A and G_B belong to different calendars is considered.

Derivation A.1 *G_A and G_B belong to different calendars:*

Let G_A and G_B belong to calendars C_1 and C_2, respectively. The following procedure derives $lbf(K, G_A, G_B)$ and $ubf(K, G_A, G_B)$ for both K as an integer coefficient and K as a real coefficient:

if $\exists G'_A, G'_B \mid G'_A \in Set_{G_A} \wedge G'_B \in Set_{G_B} \wedge G'_A \in C' \wedge G'_B \in C'$
{
  if G'_A is coarser than G'_B
    Derive $lbf(K, G'_A, G'_B)$ and $ubf(K, G'_A, G'_B)$ using Derivation 3.2
  else if G'_A is finer than G'_B
    Derive $lbf(K, G'_A, G'_B)$ and $ubf(K, G'_A, G'_B)$ using Derivation 3.3
  Use $lbf(K, G'_A, G'_B)$ for $lbf(K, G_A, G_B)$ and $ubf(K, G'_A, G'_B)$ for $ubf(K, G_A, G_B)$
}

else

$lbf(K, G_A, G_B)$ and $ubf(K, G_A, G_B)$ have to be explicitly specified

Since Set_{G_A} and Set_{G_B} contain calendric granularities which have the same time duration as G_A and G_B, respectively, the above procedure first checks whether there exists in Set_{G_A} and Set_{G_B} calendric granularities which belong to the same calendar. If such calendric granularities exist, then they are used instead of G_A and G_B in the derivations of $lbf(K, G_A, G_B)$ and $ubf(K, G_A, G_B)$. If no calendric granularities exist in Set_{G_A} and Set_{G_B} belonging to the same calendar, the $lbf(K, G_A, G_B)$ and $ubf(K, G_A, G_B)$ have to be explicitly provided. ■

Example A.2 Suppose we want to calculate $lbf(G_{businessMonth}, G_{year})$ where $G_{businessMonth}$ belongs to the Business calendar and G_{year} belongs to the Gregorian calendar. Suppose also that $Set_{G_{businessMonth}} = \{G_{businessMonth}, G_{academicMonth}\}$ and $Set_{G_{year}} = \{G_{year}, G_{academicYear}\}$, where $G_{academicMonth}$ and $G_{academicYear}$ belong to the Academic calendar. Then,
$lbf(G_{businessMonth}, G_{year}) \equiv lbf(G_{academicMonth}, G_{year}) \equiv lbf(G_{academicMonth}, G_{academicYear})$.
□

A.2 Unanchored Temporal Primitives

A.2.1 Representation of Time Spans

In general, a time span is made up of different calendric granularities, possibly belonging to different calendars.

Definition A.1 *Discrete Determinate span:*

$$S_{discr} = \sum_{i=1}^{N} \sum_{j=1}^{M} (K_j^{C_i} \cdot G_j^{C_i}) \tag{A.1}$$

where $K_j^{C_i}$ is an integer coefficient of $G_j^{C_i}$, which is a distinct calendric granularity in calendar C_i. ■

Definition A.2 *Continuous Determinate span:*

$$S_{cont} = \sum_{i=1}^{N} \sum_{j=1}^{M} (R_j^{C_i} \cdot G_j^{C_i}) \tag{A.2}$$

where $R_j^{C_i}$ is a real coefficient of $G_j^{C_i}$, which is a distinct calendric granularity in calendar C_i. ■

Basically, S_{discr} and S_{cont} are summations of distinct calendric granularities over different calendars.

A.2.2 Conversion of Time Spans

Definition A.3 *Discrete span conversion:* The conversion of a span of the form depicted in formula (A.1) to a calendric granularity G_A results in a span with lower bound

$$\lfloor \sum_{j=1}^{N} \sum_{i=1}^{M} L_i^{C_j} \rfloor \cdot G_A \tag{A.3}$$

and upper bound

$$\lceil \sum_{j=1}^{N} \sum_{i=1}^{M} U_i^{C_j} \rceil \cdot G_A \tag{A.4}$$

where

$$L_i^{C_j} = lbf_r(K_i^{C_j}, G_i^{C_j}, G_A) \tag{A.5}$$

and

$$U_i^{C_j} = ubf_r(K_i^{C_j}, G_i^{C_j}, G_A) \tag{A.6}$$

■

Definition A.4 *Continuous span conversion:* The conversion of a span of the form depicted in formula (A.2) to a calendric granularity G_A results in an span with lower bound

$$\sum_{j=1}^{N} \sum_{i=1}^{M} L_i^{C_j} \cdot G_A \tag{A.7}$$

and upper bound

$$\sum_{j=1}^{N} \sum_{i=1}^{M} U_i^{C_j} \cdot G_A \tag{A.8}$$

where

$$L_i^{C_j} = lbf_r(K_i^{C_j}, G_i^{C_j}, G_A) \tag{A.9}$$

and

$$U_i^{C_j} = ubf_r(K_i^{C_j}, G_i^{C_j}, G_A) \tag{A.10}$$

■

Example A.3 To illustrate the conversion described above, the discrete time span 2 *months and* 45 *hours and* 3 *academicYears* is converted to a discrete indeterminate span in the calendric granularity of days (G_{days}). First the given span is represented in the form given in formula (A.1):

$$2 \cdot G_{months} + 45 \cdot G_{hours} + 3 \cdot G_{academicYears}$$

This span has calendric granularities from two calendars, the Gregorian and Academic calendars. G_{months} and G_{hours} are members of the Gregorian calendar (C_1), while $G_{academicYears}$ is a member of the Academic calendar (C_2). Additionally, $K_1^{C_1} = 2$, $K_2^{C_1} = 45, K_1^{C_2} = 3, G_1^{C_1} = G_{months}$, $G_2^{C_1} = G_{hours}, G_1^{C_2} = G_{academicYears}$. The formulas (A.5) and (A.6) are now used to compute $L_1^{C_1}, L_2^{C_1}, L_1^{C_2}, U_1^{C_1}, U_2^{C_1}$, $U_1^{C_2}$. The calculations of $L_1^{C_1}, U_1^{C_1}, L_2^{C_1}$, and $U_2^{C_1}$ are the same as those given for L_1, L_2, U_1, U_2 in Section 3.3.2.1, Example 3.8. Therefore, it is only necessary to show how $L_1^{C_2}$ and $U_1^{C_2}$ are calculated.

$$
\begin{aligned}
L_1^{C_2} &= lbf(K_1^{C_2}, G_1^{C_2}, G_{days}) \\
&= lbf(3, G_{academicYears}, G_{days}) \\
&= lbf(3, G_{years}, G_{days}) \\
&= 1095 \\
U_1^{C_2} &= ubf(K_1^{C_2}, G_1^{C_2}, G_{days}) \\
&= ubf(3, G_{academicYears}, G_{days}) \\
&= ubf(3, G_{years}, G_{days}) \\
&= 1096
\end{aligned}
$$

In deriving $lbf(K, G_{academicYears}, G_{days})$ and $ubf(K, G_{academicYears}, G_{days})$, since $G_{academicYears}$ and G_{days} belong to different calendars, Derivation A.1 is used. It can be noted that in $Set_{G_{AcademicYear}}$ and $Set_{G_{day}}$ there exist calendric granularities G_{year} and G_{day} which belong to the same calendar (the Gregorian calendar). Therefore, $lbf(K, G_{academicYears}, G_{days})$ is equivalent to $lbf(K, G_{years}, G_{days})$ which is then calculated from the conversion functions in the Gregorian calendar. The same holds true for $ubf(K, G_{academicYears}, G_{days})$. If $Set_{G_{AcademicYear}}$ and $Set_{G_{day}}$ did not have calendric granularities belonging to the same calendar, then $lbf(K, G_{academicYears}, G_{days})$ and $ubf(K, G_{academicYears}, G_{days})$ would have to be explicitly specified.

Lastly, the lower and upper boundary of the resulting indeterminate span are computed according to formulas (A.3) and (A.4), respectively:

$$
\begin{aligned}
lower\ bound &= \lfloor L_1^{C_1} + L_2^{C_1} + L_1^{C_2} \rfloor \cdot G_{days} \\
&= \lfloor 59 + 1.875 + 1095 \rfloor \cdot G_{days} \\
&= 1155 \cdot G_{days} \\
upper\ bound &= \lceil U_1^{C_1} + U_2^{C_1} + U_1^{C_2} \rceil \cdot G_{days} \\
&= \lceil 62 + 1.875 + 1096 \rceil \cdot G_{days} \\
&= 1160 \cdot G_{days}
\end{aligned}
$$

Hence, the result of the conversion is the indeterminate discrete time span $1155 \sim 1160\, days$.
□

A.2.3 Operations between Time Spans

The semantics of adding (subtracting) two spans is to add (subtract) the components which have the same calendric granularity and concatenate the remaining components to the resulting span. For example, assume we have two calendars, the Gregorian calendar with calendric granularities *year*, *month*, and *day*, and the Academic calendar with calendric granularities *academicYear* and *semester*. Suppose also that $Set_{G_{year}} = \{G_{year}, G_{academicYear}\}$.

Example A.4

1. $(5\ years{+}4\ months){+}2\ academicYears \rightarrow (5\ years{+}4\ months{+}\ 2\ academicYears) \equiv 4\ months + 7\ years \equiv 4\ months + 7\ academicYears$

2. $(5\ years + 4\ months + 2\ academicYears) + (2\ academicYears + 1\ semester) \rightarrow (5\ years + 4\ months + 4\ academicYears + 1\ semester) \equiv 4\ months + 1\ semester + 9\ academicYears \equiv 4\ months + 1\ semester + 9\ years$

□

It is worth mentioning that mathematical operations between spans could result in spans which are composed of calendric granularities belonging to different calendars. In such a case, if human understandability becomes an issue, the span can be converted to a single calendric granularity using the conversion procedure described in Section A.2.2.

A.3 Anchored Temporal Primitives

Operands in the operations between time instants may belong to different calendars. Similarly, operations between spans and time instants may involve spans composed of calendric granularities belonging to different calendars. Therefore, to carry out operations on time instants, it may be necessary to convert time instants from one calendar to another. In the following section, detailed conversion functions which enable an instant to be converted from one calendar to another are given.

A.3.1 Conversion of Time Instants

To convert a time instant from one calendar to another, the time instant is first mapped to a real value on the global time axis (G_{tl}). This value is then mapped to a time instant in the calendar of interest. Therefore, functions are defined to convert a time instant to its respective value on G_{tl}, and inverse functions are defined to convert a value on G_{tl} to an instant in a particular calendar. To simplify the description, the functions for a simple calendar are first given and then generalized for any given calendar.

Derivation A.2 *Mapping a time instant to G_{tl} – Simple calendar:* Let C be a calendar with the calendric granularities *year*, *month* and *day*. The following functions are defined in C:

$$\begin{aligned} f_C^1(y) &\rightarrow N_{months} \\ f_C^2(y,m) &\rightarrow N_{days} \\ f_C^3(y,m,d) &\rightarrow R \end{aligned}$$

where y, m, and d are ordinal values of calendric elements in the calendric granularities *year*, *month*, and *day*, respectively. Since a time instant is represented in terms of the calendric elements of a calendar, we can write any time instant in C using the ordinal values of calendric elements as (y, m, d). For example, the time instant September 12, 1995 is written as $(1995, 9, 12)$.

Now, to map the time instant (y, m, d) to G_{tl} the R value for all days up to year y is first calculated. This is given by the summation:

$$\sum_{a_1=1}^{y-1} \sum_{a_2=1}^{f_C^1(a_1)} \sum_{a_3=1}^{f_C^2(a_1,a_2)} f_C^3(a_1, a_2, a_3) \tag{A.11}$$

Formula A.11 calculates the R value up to year y by summing R for every day, in every month, of every year up to year y. Next, the R value for all days in all months up to month m in year y is calculated:

$$\sum_{a_1=1}^{m-1} \sum_{a_2=1}^{f_C^2(y,a_1)} f_C^3(y, a_1, a_2) \tag{A.12}$$

This formula calculates the R value in year y by summing R for every day, in every month up to month m of year y. Lastly, the R value for all days up to day d in month m is calculated:

$$\sum_{a_1=1}^{d-1} f_C^3(y, m, a_1) \tag{A.13}$$

Formula A.13 calculates the R value in month m by summing R for every day up to day d in month m of year y. The R value corresponding to the time instant (y, m, d) is then obtained by summing Formulas A.11, A.12, and A.13 to the origin of calendar C. Consider now the mapping of a time instant belonging to any general calendar to G_{tl}. ∎

Derivation A.3 *Mapping a time instant to G_{tl} – General calendar:* Let C be a calendar with origin $\mathcal{O}_C$, and conversion functions $f_C^1(i_1), f_C^2(i_1, i_2), \ldots, f_C^n(i_1, i_2, \ldots, i_n)$ (see Definition 3.2 in Section 3.2.2). Additionally, let $(i_1, \ldots, i_n)$ be a time instant in C. Then, $R(i_1, \ldots, i_n)$, the R value for the time instant $(i_1, \ldots, i_n)$ is given by:

$$\begin{aligned} R(i_1, \ldots, i_n) = \mathcal{O}_C + \\ \sum_{k=1}^{n} \left(\sum_{a_k=1}^{i_k-1} \sum_{a_{k+1}=1}^{f_C^k(i_1,\ldots,i_{k-1},a_k)} \cdots \sum_{a_n=1}^{f_C^{n-1}(i_1,\ldots,i_{k-1},a_k,\ldots,a_{n-1})} f_C^n(i_1, \ldots, i_{k-1}, a_k, \ldots, a_n) \right) \end{aligned} \tag{A.14}$$

Formula A.14 first calculates the R value of the time instant up to the calendric element i_n followed by the R value in i_n, up to the calendric element i_{n-1}. This procedure is repeated up to the finest calendric granularity, i.e., up to calendric element i_1. The following derivation shows how a real value on G_{tl} is converted to a time instant in any given calendar. ∎

Derivation A.4 *Mapping a real value from G_{tl} to a time instant – Simple Calendar:* Let C be a calendar with origin $\mathcal{O}_C$, calendric granularities *year*, *month*, and *day*, and r be a real value on G_{tl}. Then the following formulas calculate a time instant in C corresponding to r:

$$\begin{aligned}
Y &= \max_{y \in \mathbf{Z}} \{y \mid R(y, 1, 1) \leq r - \mathcal{O}_C\} \\
M &= \max_{m \in \mathbf{Z}} \{m \mid R(Y, m, 1) \leq r - \mathcal{O}_C\} \\
D &= \max_{d \in \mathbf{Z}} \{d \mid R(Y, M, d) \leq r - \mathcal{O}_C\}
\end{aligned}$$

The above formulas first find the maximum year Y which when mapped to G_{tl} gives a real value which is less than or equal to $r - \mathcal{O}_C$. The trick here is to vary the year value (y) in $R(y, m, d)$ (Formula A.14) and keep the month (m) and day (d) values constant at 1. After having found Y, the maximum month in year Y which when mapped to G_{tl} gives a real value which is less than or equal to $r - \mathcal{O}_C$ is then calculated. In this case, the year value in $R(y, m, d)$ is kept fixed at Y, the month value is changed, and the day value is kept constant at 1. Finally, the maximum day in year Y and month M which when mapped to G_{tl} gives a real value which is less than or equal to $r - \mathcal{O}_C$ is calculated. ∎

Derivation A.5 *Mapping a real value from G_{tl} to a time instant – General Calendar:* Let C be a calendar with origin $\mathcal{O}_C$, and calendric granularities $G_1, G_2, \ldots, G_n$, where G_1 is the coarsest calendric granularity and G_n is the finest calendric granularity. Additionally, let r be a real value on G_{tl}. Then the following formulas calculate a time instant $(i_1, \ldots, i_n)$ in C corresponding to r:

$$\begin{aligned}
i_1 &= \max_{a \in \mathbf{Z}} \{a \mid R(a, \underbrace{1, \ldots, 1}_{n-1}) \leq r - \mathcal{O}_C\} \\
&\vdots \\
i_k &= \max_{a \in \mathbf{Z}} \{a \mid R(i_1, i_2, \ldots, i_{k-1}, a, \underbrace{1, \ldots, 1}_{n-k}) \leq r - \mathcal{O}_C\} \\
&\vdots \\
i_n &= \max_{a \in \mathbf{Z}} \{a \mid R(i_1, i_2, \ldots, i_{n-1}, a) \leq r - \mathcal{O}_C\}
\end{aligned}$$

∎

Having defined the conversion functions necessary to convert a time instant from one calendar to another, in the following sections, algorithms are given that show how the operations on time instants are carried out when the operands belong to different calendars.

A.3.2 Comparison between Time Instants

Let $(i_1, \ldots, i_n)$ and $(i_1^{'}, \ldots, i_m^{'})$ be two time instants belonging to calendars C_1 and C_2, respectively. The algorithm to compare $(i_1, \ldots, i_n)$ and $(i_1^{'}, \ldots, i_m^{'})$ is:

Algorithm A.1 *Comparison of instants belonging to different calendars:*

$$
\begin{array}{l}
r_1 := R(i_1, \ldots, i_n) \\
r_2 := R(i_1^{'}, \ldots, i_m^{'}) \\
\texttt{Compare } r_1 \text{ and } r_2
\end{array}
$$

Algorithm A.1 makes use of the global time axis which provides a homogeneous underlying platform on which time instants can be mapped. The two time instants are first converted to their respective real values on the global time axis using Derivation A.3. These real values are then compared.

A.3.3 Elapsed Time between Time Instants

Let $(i_1, \ldots, i_n)$ and $(i_1^{'}, \ldots, i_m^{'})$ be two time instants belonging to calendars C_1 and C_2, respectively. The algorithm to find the elapsed time between $(i_1, \ldots, i_n)$ and $(i_1^{'}, \ldots, i_m^{'})$ is:

Algorithm A.2 *Elapsed time between instants belonging to different calendars:*

$$
\begin{array}{l}
\texttt{Convert } (i_1^{'}, \ldots, i_m^{'}) \text{ to } (i_1^{''}, \ldots, i_n^{''}) \text{ using Derivations A.3 and A.5} \\
S^N := \texttt{Elapsed}((i_1, \ldots, i_n), (i_1^{''}, \ldots, i_n^{''}))
\end{array}
$$

The algorithm first converts the time instant $(i_1^{'}, \ldots, i_m^{'})$ to its equivalent counterpart in calendar C_1, $(i_1^{''}, \ldots, i_n^{''})$ using Derivations A.3 and A.5. It then finds the elapsed time between $(i_1, \ldots, i_n)$ and $(i_1^{''}, \ldots, i_n^{''})$. The result is a time span, S^N, which is of the form shown in formulas (A.1) or (A.2).

A.3.4 Operations between Spans and Time Instants

Consider the situation when the calendric granularities of the span do not belong to the same calendar as the instant. Let S be a span of the form:

$$
\begin{aligned}
S &= \sum_{i=1}^{N} \sum_{j=1}^{M} (K_j^{C_i} \cdot G_j^{C_i}) \\
&= \sum_{j=1}^{M} (K_j^{C_1} \cdot G_j^{C_1}) + \ldots + \sum_{j=1}^{M} (K_j^{C_N} \cdot G_j^{C_N}) \\
&= S_{C_1} + S_{C_2} + \ldots + S_{C_N}
\end{aligned}
$$

Basically S_{C_i} is a span composed of calendric granularities belonging to calendar C_i. The algorithm for adding span S to a time instant I_{C_A} (a time instant belonging to calendar C_A) is as follows:

Algorithm A.3 *Addition of a span to an instant:*

$I'_{C_A} := I_{C_A}$
repeat for $i := 1$ *to* N
    `Convert` I'_{C_A} to I_{C_i} using Derivations A.3 and A.5
    $I'_{C_i} := S_{C_i} + I_{C_i}$
    `Convert` I'_{C_i} to $I^i_{C_A}$ using Derivations A.3 and A.5
    $I'_{C_A} := I^i_{C_A}$
return I'_{C_A}

For each span component, S_{C_i}, the algorithm converts the time instant belonging to calendar C_A to a corresponding time instant in calendar C_i, adds it to S_{C_i} and converts the resulting instant back to an instant of calendar C_A. The algorithm for subtracting span S from a time instant I_{C_A} is similar. That is, $I_{C_A} - S = I_{C_A} + (-S)$.