Experience Using TSQL2 in a Natural Language Interface

I. Androutsopoulos, G.D. Ritchie
Department of Artificial Intelligence, University of Edinburgh
Scotland, U.K.

P. Thanisch
Department of Computer Science, University of Edinburgh
Scotland, U.K.

Abstract

Attempting to use TSQL2 in a natural language interface for temporal databases has focused our attention on certain points in the semantics of TSQL2 where we believe that clarifications or modifications are needed. It has also led us to some extensions to TSQL2 that seem generally useful, regardless of natural language issues.

1 Introduction

For many years, there have been interfaces to databases which allowed the user to frame queries in natural language, such as English (see [2], [4], [10] for reviews). Until recently, the databases involved have mainly been conventional "snapshot" database systems, that do not facilitate the handling of temporal information. We are attempting to build an experimental natural language interface for temporal databases, using methods similar to those already in use for natural language interfaces to non-temporal databases (see [3] for a description of our project). In particular, our architecture has an English query parsed into a syntactic structure and converted into a formal expression (in a logic-like language) encoding the relevant aspects of its meaning. This expression is then translated into TSQL2 [13], a temporal extension of SQL-92 [9]. The evaluation of the TSQL2 query against the temporal database supplies the answer to the original English query. Unfortunately, we are not aware of any available system that can evaluate TSQL2 queries. Hence, our natural language interface simply translates English questions to TSQL2 queries, without evaluating the resulting TSQL2 queries.

We have studied the evolving definitions of TSQL2 very closely. The aim of constructing a natural language interface has concentrated our scrutiny in two ways: in order to prove that the translation from our logic-like language to TSQL2 was correct, we had to think very rigorously about the semantics of TSQL2; also, because our TSQL2 statements are created dynamically by a program (instead of statically by a human user), we had to be very sure of what was and was not possible within TSQL2. It has become apparent as a result of these deliberations that the current definitions of TSQL2, particularly the informal outlines of its semantics, are still in need of some development. There are several inclarities and inconsistencies in the available TSQL2 documents, and a few points which we believe could be defined in a more appropriate way. We also devised some extensions to the current TSQL2 definition, that seem generally useful, regardless of natural language issues.

Some parts of this paper may be based on misunderstandings of the available TSQL2 definition. We note that although [13] defines the syntax of TSQL2 rigorously, the semantics of the language is often left to the intuition of the reader, or to the TSQL2 commentaries; these do not always agree with [13]. Our confusion may serve as pointers to sections in the TSQL2 documentation whose clarity could be improved.

Although we propose clarifying or modifying the TSQL2 definition, we do not question the value of TSQL2 as an extension of SQL-92. On the contrary, the built-in temporal support of TSQL2 (especially built-in temporal functions like `CONTAINS`, `BEGIN`, `INTERSECT`, and the `PERIOD` data-type, none of which is supported by SQL-92) has proven valuable in our project, and it has greatly simplified the task of formulating the translation from the logic-like language to a database query language.

This paper assumes that the reader is familiar with SQL-92. Section 2 provides an introduction to the aspects of TSQL2 on which our criticisms will be focused. Section 3 discusses the points in the definition of TSQL2 where the semantics of TSQL2 is unclear or where we think that the semantics of TSQL2 should be modified. Section 4 describes our extensions to TSQL2, and section 5 summarises.

2 TSQL2 basics

Temporal ontology

TSQL2 assumes that time in the real world is linear, and that it consists of *instants* (time-points). TSQL2 remains agnostic on whether time in the real world is continuous, dense or discrete (see [7]).

In TSQL2 an instant can only be specified with respect to a particular *granularity*. For example, at the granularity of years, one can specify that an instant falls within the granule (year) 1995. At the granularity of days, one can specify that the instant falls within the granule (day) March 2nd 1995. TSQL2 supports a variety of granularities (e.g. years, months, days, seconds), and application-specific granularities can also be defined (e.g. academic semesters; see [6]).

The TSQL2 granularities form a lattice that shows how the granularities relate to each other (e.g. that the granularity of months is finer than the granularity of years, that a year-granule contains twelve month-granules, etc.). The finest available granularity is that of *chronons*. Depending on the database application, a chronon could correspond to a day, a second, a nano-second, etc. A granule (even at the level of chronons) is much bigger than an instant, and there is at least one instant within each granule. Hence, an instant can never be specified precisely in TSQL2. One can only specify the containing granule.

A *period* is a convex set of instants. ("Convex set" of instants means that if the set contains two instants i_1 and i_2, then it also contains all the instants between i_1 and i_2.) To specify a period, it is enough to specify the period's earliest and latest instants. In TSQL2 a period can only be specified with respect to a particular granularity. One determines the granules where the earliest and the latest instants of the period are located. For example, at the granularity of years, one can specify a period by determining that the earliest instant of the period falls within the year-granule 1991, and that the latest instant of the period falls within the year-granule 1995. (According to section 2.4 of [7], both the earliest and the latest instant of a period must be specified with respect to the same granularity.) A period can be represented as a convex set of granules of some granularity. ("Convex set of granules" means that if the set contains two granules g_1 and g_2 of some granularity G, then it also contains all the granules of G between g_1 and g_2.) For example, at the granularity of years a period that starts at some instant in 1991 and ends at some instant in 1995 can be represented as the set that contains the year-granules 1991, 1992, 1993, 1994, and 1995.

In TSQL2 the term *interval* is used to denote a duration. TSQL2 intervals are expressed as numbers of granules. For example, one can specify an interval (duration) of 5 year-granules, or an interval of 2 minute-granules, etc.

Relations

Apart from the ordinary relations of the traditional relational model, which are called *snapshot relations* in TSQL2, TSQL2 provides *valid-time relations*, *transaction-time relations*, and *bitemporal relations*. Transaction-time relations and bitemporal relations are ignored in this paper, as they have not been used in our project.

Valid-time relations are further divided into *state valid-time relations* and *event valid-time relations*. (This and the following paragraphs are based on our interpretation of [12]. The distinction between state and event relations also applies to bitemporal relations.) In all valid-time relations, there is a special unnamed attribute, the *implicit attribute*, the values of which are called *time-stamps*. (The other attributes are called *explicit attributes*.) Each tuple of a valid-time relation represents a situation in the world. The tuple's time-stamp shows when the situation was/is/will be true in the world. Each time-stamp is a set of granules of some granularity. The same granularity has to be used in all the time-stamps of a valid-time relation. (Section 5.2 of [8] assumes that time-stamps are always sets of *chronons*. We do not see how this is compatible with TSQL2's multiple granularities. Instead, we follow section 3.2 of [5], and assume that each time-stamp is a set of granules, not necessarily a set of chronons.)

The exact semantics of the time-stamps of a valid-time relation depend on whether the relation is a state or an event one. Let us assume that `manager_of` is a state valid-time relation of the following form, and that the time-stamps of `manager_of` are specified at the granularity of years.

manager_of		
name	department	
K.Prince	sales	{1987, 1988, 1989, 1992, 1993, 1994}
J.Pappas	personnel	{1987, 1988, 1989, 1990}
...	...	...

The semantics in `manager_of` above is that K.Prince was the manager of the sales department all the time from some instant in 1987 to some instant in 1989, and all the time from some instant in 1992 to some instant in 1994. To interpret the time-stamp of a state valid-time relation, one first forms the maximal subsets of the time-stamp, such that each subset is convex. In the case of the tuple for K.Prince and sales, the maximal subsets are {1987, 1988, 1989} and {1992, 1993, 1994}. The periods represented by the maximal subsets are called *maximal periods* of the time-stamp. (The reader is reminded that periods can be represented as convex sets of granules). The semantics of a time-stamp in a state valid-time relation is that the situation represented by the corresponding tuple is true at all the instants of the maximal periods of the time-stamp.

In the case of event valid-time relations, the semantics of each time-stamp is that the corresponding situation is true at some instant within each granule of the time-stamp. Let us assume that `become_employed` is an event valid-time relation of the following form, and that the time-stamps of `become_employed` are specified at the granularity of years:

become_employed	
name	
K.Prince	{1987, 1992}
J.Pappas	{1987}
T.Smith	{1991,1992}
...	...

The semantics of `become_employed` above is that, for example, K.Prince became employed at some instant in 1987, and then again at some instant in 1992. Unlike tuples of state valid-time relations, the tuple for T.Smith above does not mean that T.Smith was becoming employed all the time from an instant in 1991 to an instant in 1992. It simply means that T.Smith became employed at some instant in 1991, and again at some instant in 1992.

The distinction between state valid-time relations and event valid-time relations shows up when some TSQL2 functions are used. For example, when the TSQL2 function `FIRST` is applied to the time-stamp of a state valid-time relation, it returns the set of granules that represents the earliest maximal period of the time-stamp. In contrast, when `FIRST` is applied to the time-stamp of an event valid-time relation, it returns the earliest granule of the time-stamp.

The time-stamps of state valid-time relations are called *temporal elements*, while the time-stamps of event valid-time relations are called *instant sets*. (According to this definition, the term "instant set" is unfortunate, since if an instant set is a time-stamp, then it is a set of granules, not a set of instants.) Actually, the TSQL2 documents often try to distinguish between the terms "temporal element" and "instant set" by saying that temporal elements are unions of periods, while instant sets are simply sets of instants (see for example section 2.1 of [12]). In our view, this definition is problematic. A union of periods in the real world is a union of convex sets of instants, i.e. simply a set of instants. (The union of two convex sets of instants is not necessarily a convex set of instants.) Hence, one cannot distinguish between unions of periods and simple sets of instants. Perhaps when the TSQL2 documents refer to "unions of periods" they mean unions of convex sets of granules (a period can be represented as a convex set of granules), and when they refer to sets of instants they mean sets of granules (an instant can be represented as a granule). Even in this case, however, a union of convex sets of granules is simply a set of granules. (The union of two convex sets of granules is not necessarily a convex set of granules.) Hence, again one cannot distinguish between unions of convex sets of granules and simple sets of granules. In our view, both temporal elements and instant sets are sets of granules. The term "temporal element" simply means that the set of granules is used as a a time-stamp in a state valid-time relation, while the term "instant set" means that the set of granules is used as time-stamp in an event valid-time relation (and as discussed above, the semantics of the time-stamps depend on whether the time-stamps are used in state valid-time relations or in event valid-time relations).

In BCDM, the conceptual data model of TSQL2, a relation can never contain *value equivalent* tuples, i.e. tuples that contain the same values in their explicit attributes (see section 5.2 of [8]). TSQL2, however, provides *partitioning units*, special keywords that can be used in TSQL2 queries, and that cause a relation to be viewed within a query as if it contained value-equivalent tuples. For example, the `(PERIOD)` partitioning unit would cause `manager_of` to be viewed as if each tuple were time-stamped by a set of granules representing a single maximal period of the corresponding time-stamp in the unpartitioned `manager_of`:

manager_of partitioned with (PERIOD)		
name	department	
K.Prince	sales	{1987, 1988, 1989}
K.Prince	sales	{1992, 1993, 1994}
J.Pappas	personnel	{1987, 1988, 1989, 1990}
...	...	...

TSQL2 also supports a partitioning unit called `(INSTANT)`; this will be discussed in section 4.3.

Literals

TSQL2 provides three kinds of *temporal literals*: literals that specify *instants*, literals that specify *periods*, and literals that specify *intervals* (see [14] and [15]). For example, the literal `TIMESTAMP 'March 3, 1995' DAY` specifies an instant at the granularity of days that falls within March 3rd 1995. Instants can also be specified by `DATE` and `TIME` literals. The status of `DATE` and `TIME` literals in TSQL2 is unclear. The assumption seems to be that `DATE` literals are syntactic sugar for `TIMESTAMP` literals at the granularity of days. For example, `DATE 'March 3, 1995'` is the same as `TIMESTAMP 'March 3, 1995' DAY`. `TIME` literals seem to be intended to specify an instant within a day, but without determining that day (e.g. `TIME '08:54:32'`). `DATE` and `TIME` literals will be discussed further in section 3.1.

TSQL2 supports closed-closed, closed-open, open-closed, and open-open period literals (see [13], section 10.2, additional syntax rule 11, and additional general rule 5). For example, `PERIOD '[March 3, 1995 - March 20, 1995]' DAY` is a closed-closed period literal that specifies a period at the granularity of days, that starts at some instant on March 3rd 1995, and ends at some instant on March 20th 1995. `PERIOD '(March 3, 1995 - March 20, 1995]' DAY` is an open-closed period literal that specifies a period at the granularity of days. In this case, the period starts at some instant on March 4th 1995 (the day after March 3rd 1995), and ends at some instant on March 20th 1995. Closed-open and open-open period literals have similar meanings. Finally, to give an example of an interval literal, `INTERVAL '5' DAY` specifies an interval 5 days long.

In the rest of this paper, we adopt a notation similar to TSQL2 period literals to refer informally to periods specified at some granularity. For example, we write [June 1993 – May 1994] to refer to the period specified by the TSQL2 literal `PERIOD '[June 1993 - May 1994]' MONTH`.

Calendars

TSQL2 supports multiple *calendars*. Roughly speaking, a TSQL2 calendar describes a system that people use to measure time (e.g. Gregorian calendar, Lunar calendar, etc.). TSQL2 calendars also specify the strings that are allowed to appear within the quotes of temporal literals (e.g. `DATE 'December 31, 1991'`, `DATE 'Ramadam 1, 1872'`), the meanings assigned to these strings, and the available granularities (e.g. days, weeks, academic semesters, etc.). According to section 3 of [11], TSQL2 calendars are defined by the database administrator, the DBMS vendor, or third parties. Users can then select the calendar they wish to use, or even use combinations of calendars.

`SCALE` *and* `CAST`

TSQL2 provides two functions, `SCALE` and `CAST`, that can be used to change the granularity of TSQL2 temporal expressions; see sections 4.5 – 4.11 of [6]. (Section 11.16 of [13] does not allow `SCALE` to be used with expressions that specify instant sets. We see no reason for this restriction, and propose adding an extra syntax rule in section 11.16 of [13] to remove this restriction. The extra rule would be very similar to the rule for `SCALE` in section 11.14 of [13].) `CAST` can also be used to convert non-temporal expressions from one data-type to another; see section 6.3 of [9]. `CAST` was already supported in SQL-92, but TSQL2 has enhanced its functionality. `SCALE` is a new function introduced in TSQL2.

The following examples illustrate the behaviour of `SCALE` in the case of instant-specifying expressions. The examples are a modified version of the examples in section 4.5 of [6]. We use "=" as a shorthand for "is equivalent to".

```
SCALE(TIMESTAMP 'May 6, 1994' DAY AS YEAR) =
   TIMESTAMP '1994' YEAR
SCALE(TIMESTAMP 'May 6, 1994' DAY AS MONTH) =
   TIMESTAMP 'May 1994' MONTH
SCALE(TIMESTAMP 'May 6, 1994' DAY AS DAY) =
   TIMESTAMP 'May 6, 1994' DAY
SCALE(TIMESTAMP 'May 6, 1994' DAY AS MINUTE) =
   TIMESTAMP 'May 6, 1994 00:00 ~ May 6, 1994 23:59' MINUTE
```

In the last example above, `TIMESTAMP 'May 6, 1994 00:00 ~ May 6, 1994 23:59' MINUTE` is an *indeterminate* instant-specifying literal, that reflects the fact that we do not know the exact minute of May 6th 1994 where the instant should be placed (the instant could be placed at any minute of that day).

The behaviour of `SCALE` and `CAST` can be explained in terms of the granules or sets of granules that correspond to the temporal expressions to which `SCALE` or `CAST` are applied.

In the case of instant-specifying expressions, when converting from a finer to a coarser granularity (e.g. `DAY` to `YEAR`), `SCALE` maps the granule g that represents the instant at the old granularity to the granule g' of the new granularity that contains g. For example, in the first case above the day-granule May 6th 1994 is mapped to the year-granule 1994. When converting from a coarser to a finer granularity (e.g. from `DAY` to `MINUTE`) using `SCALE`, the granule g that represents the instant at the old granularity is mapped to an indeterminate granule g' ranging over all the granules of the new granularity that are contained within g. For example, in the last case above the day-granule May 6th 1994 is mapped to an indeterminate minute-granule that ranges over all the minute-granules of May 6th 1994. In contrast, when converting from a coarser to a finer granularity using `CAST`, the granule g that represents the instant at the old granularity is mapped to the *earliest* granule of the new granularity that is contained within g. For example, below the day-granule May 6th 1994 is mapped to the minute-granule May 6th 1994 00:00.

```
CAST(TIMESTAMP 'May 6, 1994' DAY AS MINUTE) =
   TIMESTAMP 'May 6, 1994 00:00' MINUTE
```

When converting from a finer to a coarser (or to the same) granularity (e.g. `DAY` to `MONTH`), `CAST` behaves in the same way as `SCALE`.

The behaviour of `SCALE` and `CAST` in the case of period-specifying expressions is similar. The granules that specify the earliest and latest instants of the period are mapped to granules of the new granularity as in the case of instant-specifying expressions. The following examples illustrate the behaviour of `SCALE` and `CAST` with periods.

```
CAST(PERIOD '[1991 - 1995]' YEAR AS DAY) =
   PERIOD '[Jan 1, 1991 - Jan 1, 1995]' DAY
SCALE(PERIOD '[1991 - 1995]' YEAR AS DAY) =
   PERIOD '[Jan 1, 1991 ~ Dec 31, 1991 -
            Jan 1, 1995 ~ Dec 31, 1995]' DAY
```

The behaviour of `CAST` in the case of period-specifying expressions will be discussed further in section 3.2.

[6] does not provide much information on the behaviour of `SCALE` and `CAST` with expressions that refer to time-stamps of valid-time relations (instant sets or temporal elements). We assume that in the case of instant sets, each granule of the instant set is mapped to a granule of the new granularity, as in the case of instant-specifying expressions. For example, assuming that the set of day-granules {May 5 1994, August 13 1994, March 20 1995} is used as an instant-set, when converting from `DAY` to `YEAR`, both `SCALE` and `CAST` would generate the set of year-granules {1994, 1995}. Notice that the first two day-granules are merged into one single year-granule.

In the case of temporal elements, we assume that first each maximal period of the temporal element is converted individually to the new granularity, as when `SCALE` and `CAST` are used with period-specifying expressions. Then, the sets of granules that represent the maximal periods at the new granularity are formed, and the resulting temporal element is the union of these sets. The resulting temporal element may have fewer maximal periods than the original one. Let us consider the temporal element whose maximal periods are [May 1991 – July 1991], [Nov 1992 – March 1993], and [March 1995 – May 1995]. When converting to the granularity of years, both `SCALE` and `CAST` would map the first maximal period to [1991 – 1991] (the period starts at some instant in 1991 and ends at some instant in the same year), the second maximal period to [1992 – 1993], and the third one to [1995 – 1995]. These periods are represented by the sets of year-granules {1991}, {1992, 1993}, and {1995} respectively. The resulting temporal element is the union of these sets of year-granules, i.e.

{1991,1992,1993,1995}, and it has only two maximal periods: [1991 – 1993] and [1995 – 1995].

3 Clarifications and modifications in the semantics of TSQL2

3.1 Granularities of temporal literals

As mentioned in section 2, TSQL2 provides `DATE` literals, `TIME` literals, `TIMESTAMP` literals, `PERIOD` literals, and `INTERVAL` literals. These literals specify instants, periods, or intervals at some granularity. In the case of `INTERVAL` literals, the granularity is always specified explicitly within the literal. For example, in `INTERVAL '5' DAY` the granularity is `DAY`. In `TIME`, `TIMESTAMP`, and `PERIOD` literals the granularity may or may not be specified explicitly within the literal. For example, both `TIMESTAMP '10:00am Jan 1, 1994' MINUTE` and `TIMESTAMP '10:00am Jan 1, 1994'` are valid TSQL2 literals. Using the terminology of section 10.2 of [13], `TIME`, `TIMESTAMP`, and `PERIOD` literals may or may not contain "precision" specifications. These precision specifications in effect determine the granularities of the literals. (Actually, the rule for `TIME` literals in section 10.2 of [13] does not allow precision specifications in `TIME` literals. We assume that this is an unintended omission, since additional general rule 1 of the same section refers to the precision specifications of `TIME` literals.) In `DATE` literals, the granularity can never be specified explicitly (see the syntax rules in section 10.2 of [13]). For example, `DATE 'Jan 1, 1994' DAY` is *not* a valid TSQL2 literal.

The various TSQL2 documents do not agree on how the granularities of `TIME`, `TIMESTAMP`, `PERIOD`, and `DATE` literals are determined when the granularities are not specified explicitly within the literals:

Section 4.2 of [6] assumes that in these cases the granularity is decided by the TSQL2 calendar that interprets the literals. This seems to agree with the discussion in section 3.3 of [14], where an example is given showing how a `TIMESTAMP` literal with no explicitly defined granularity is passed to all the available TSQL2 calendars, with each calendar that manages to interpret it returning a granularity for the literal.

In contrast, additional general rules 1 and 2 in section 10.2 of [13] specify that if the granularity is not defined explicitly within a `TIME` or a `TIMESTAMP` literal, then the granularity is `SECOND`. Section 10.2 of [13] does not specify what happens in the case of `DATE` and `PERIOD` literals when the granularity is not defined explicitly by the literals.

Finally, although this is not stated in [14], section 3.2 of [14] (which contains some discussion on declaring attributes of type `DATE`) seems to suggest that a `DATE` literal is simply syntactic sugar for a `TIMESTAMP` literal that specifies explicitly the granularity `DAY`.

Our proposal is that in the case of `DATE` literals, the underlying granularity should always be `DAY`. In the case of `TIME`, `TIMESTAMP`, and `PERIOD` literals, we propose adopting the view of [6]. That is, if the literals do not specify explicitly granularities, the granularities of the literals should be determined by the calendar that interprets each literal. This seems to allow greater flexibility than the approach whereby for each one of these types of literals a default calendar-independent granularity is specified. (As mentioned above, the latter approach is partially adopted in section 10.2 of [13], where a calendar-independent default granularity, `SECOND`, is assumed for `TIME` and `TIMESTAMP` literals.) The approach of [6], that we propose for adoption in the case of `TIME`, `TIMESTAMP`, and `PERIOD` literals, allows each calendar to choose the most natural granularity for each temporal literal. For example, the calendar interpreting `TIMESTAMP '10:00am Jan 1, 1994'` is free to assign to the literal the granularity `MINUTE`, which is the most reasonable choice. In contrast, if the approach of section 10.2 of [13] is adopted, `TIMESTAMP '10:00am Jan 1, 1994'` will be assigned the default calendar-independent granularity `SECOND`. A conversion mechanism, similar to `CAST` or `SCALE` would have to be invoked, that would cause `TIMESTAMP '10:00am Jan 1, 1994'` to be treated as equivalent to `TIMESTAMP '10:00:00am Jan 1, 1994' SECOND` (in the case of `CAST`), or as equivalent to `TIMESTAMP '10:00:00am Jan 1, 1994 ~ 10:00:59am Jan 1, 1994' SECOND` (in the case of `SCALE`).

There are some outstanding questions concerning what happens in the case of literals like `TIMESTAMP '10:00am Jan 1, 1994' SECOND`, or `TIMESTAMP '10:00am Jan 1, 1994' DAY`, where the explicitly defined granularity (`SECOND` or `DAY`) differs from the granularity that would be returned by the interpreting TSQL2 calendar if there were no explicit granularity (assuming that the interpreting calendar would return the most reasonable granularity, as discussed above, i.e. `MINUTE`). Do such literals cause errors? Or are the literals converted to the explicitly specified granularity (e.g. `TIMESTAMP '10:00am Jan 1, 1994' DAY` is treated as equivalent to `TIMESTAMP 'Jan 1, 1994' DAY`, ignoring the "`10:00am`")? If the literals are converted to the explicitly specified granularity, how is the conversion carried out? Is it carried out invoking the mechanism of `CAST`, or is it carried out invoking the mechanism of `SCALE`? If the conversion is carried out using the `CAST` mechanism, then `TIMESTAMP '10:00am Jan 1, 1994' SECOND` would be treated as equivalent to `TIMESTAMP '10:00:00am Jan 1, 1994' SECOND`. If it is carried out using the `SCALE` mechanism, then `TIMESTAMP '10:00am Jan 1, 1994' SECOND` would be treated as equivalent to `TIMESTAMP '10:00:00am Jan 1, 1994 ~ 10:00:59am Jan 1, 1994' SECOND`.

We propose that when literals specify explicitly a granularity other than the granularity that would be returned by the interpreting calendar if no granularity were specified, the literals should be converted to the specified granularity (i.e. no errors should arise), provided that the calendar that interprets the quoted part of the literal (e.g. `'10:00am Jan 1, 1994'`) also supports the specified granularity (e.g. `DAY`). Otherwise, an error should occur. The exact conversion mechanism (`CAST` or `SCALE`) to be used when converting a literal should be determined by the calendar that interprets the literal.

3.2 Casting periods from coarser to finer granularities

As discussed in section 2, the following:

```
CAST(PERIOD '[1993 - 1994]' YEAR AS DAY)
```

generates a period specified at the granularity of days, which is the same as the period specified by:

```
PERIOD '[Jan 1, 1993 - Jan 1, 1994]' DAY
```

The TSQL2 view is that in the case of periods like the one specified by `PERIOD '[1993 - 1994]' YEAR`, the exact day on which the period starts or ends is not known (see sections 3.2 and 3.3 of [6]). The period could start on any day in 1993, and it could end on any day in 1994. When converting from `YEAR` to `DAY` using `SCALE`, this ignorance of the exact starting and ending days is expressed by generating an indeterminate period. For example,

```
SCALE(PERIOD '[1993 - 1994]' YEAR AS DAY)
```

produces the same indeterminate period as the one specified by:

```
PERIOD '[Jan 1, 1993 ~ Dec 31, 1993 - Jan 1, 1994 ~ Dec 31, 1994]' DAY
```

`CAST`, however, is not allowed to generate indeterminate results. Hence, it has to choose a particular starting day from 1993, and a particular ending day from 1994. According to the current TSQL2 definition, `CAST` chooses the first day of 1993 and the first day of 1994, i.e. it chooses the earliest possible granules of the new granularity for both the left and the right boundaries of the original period.

This way, however, casting a period from a coarser to a finer granularity causes the boundaries of the original period to be treated asymmetrically. In the example above where `PERIOD '[1993 - 1994]' YEAR` is cast to `DAY`, `CAST` has chosen to include in the result the whole left boundary of the original period (the whole 1993). In contrast, the right boundary of the original period is excluded almost completely (only the first day of 1994 has been included). This asymmetrical behaviour seems unreasonable to us. It would be more consistent to (a) include both the whole of 1993 and the whole of 1994 in the result, or (b) include only the last day of 1993 and the first day of 1994. We propose adopting (a). That is, when casting a period from a coarser to a finer granularity, the resulting period should start at the earliest granule of the new granularity that falls within the original left boundary, and it should end at the latest granule of the new granularity that falls within the original right boundary. For example,

```
CAST(PERIOD '[1993 - 1994]' YEAR AS DAY)
```

would produce a period that is the same as the period specified by:

```
PERIOD '[Jan 1, 1993 - Dec 31, 1994]' DAY
```

Apart from treating both boundaries of the original period in a symmetrical manner, our proposal is also more intuitive. We believe that when a user specifies a period by using, say, `PERIOD '[1993 - 1994]' YEAR`, in most cases he/she means that the period contains the *whole* of 1993 and the *whole* of 1994. Hence, when converting from `YEAR` to `DAY`, the most reasonable result is [Jan 1, 1993 – Dec 31, 1994], i.e. the result that `CAST` would produce according to our proposal.

3.3 State and event relations

As explained in section 2, TSQL2 distinguishes between state valid-time relations and event valid-time relations. It is unclear how one can check whether a valid-time relation is a state or an event one. For valid-time relations stored directly in the database (tables of the database), the `VALID_TIME` attribute of the system table `TABLES` can be consulted. (This attribute shows whether a table in the database is event or state; see [1], and section 18.1 of [13].) In the case, however, of valid-time relations computed by `SELECT` statements (relations not stored in the database), there does not seem to be any TSQL2 mechanism to find out whether the relations are state or event. In fact, in many cases it unclear whether the computed relations are supposed to be state or event. Consider the statement:

```
SELECT *
VALID PERIOD '[1990 - 1995]' YEAR
FROM table1, table2
```

where `table1` is a state valid-time relation, and `table2` is an event valid-time relation. (The `VALID` clause specifies that all the tuples in the computed relation are time-stamped by the set of granules that represents the period [1990 – 1995].) The TSQL2 documents do not state clearly whether the result is supposed to be a state or an event valid-time relation.

4 Extensions to TSQL2

4.1 Coarser/finer granularity semantics by default

When the arguments of a temporal operator are specified at different granularities, TSQL2 by default enforces *left argument granularity semantics* (see section 4.9 of [6], and additional syntax rule 7 in section 16.1 of [13]). This means that in effect `CAST` or `SCALE` operators are inserted to convert the granularities of all the arguments to the granularity of the first argument. The user can specify whether this default conversion should be carried out by inserting `CAST`s or `SCALE`s, using the statements `SET CAST AS DEFAULT` or `SET SCALE AS DEFAULT` respectively (see section 4.9 of [6], and section 16.1 of [13]). For example, if a `SET CAST AS DEFAULT` statement is in force, then:

```
INTERSECT(PERIOD '[May 1994 - July 1994]' MONTH,
          PERIOD '[April 20, 1994 - June 5, 1994]' DAY)
```

is treated as equivalent to:

```
INTERSECT(PERIOD '[May 1994 - July 1994]' MONTH,
          CAST(PERIOD '[April 20, 1994 - June 5, 1994]' DAY AS MONTH))
```

i.e. the result is a period at the granularity of months, that is the same as the period specified by the literal `PERIOD '[May 1994 - June 1994]' MONTH`. (Here, the TSQL2 function `INTERSECT` returns the period that corresponds to the overlap of the periods specified by the two arguments; see section 3.3 of [15].) The `SET CAST` and `SET SCALE` statements can also be used to cast or scale all temporal expressions to a particular granularity. For example, `SET CAST MINUTE` causes all temporal expressions to be cast to the granularity of minutes (see section 4.10 of [6], and section 16.1 of [13]).

Section 4.9 of [6] comments that although left argument granularity semantics is enforced by default, other granularity semantics (e.g. finest/coarsest granularity semantics, i.e. converting all the arguments to the granularity of the argument that is specified at the finest/coarsest granularity) can be effected by explicitly inserting cast or scale operators. For example, in the INTERSECT example above, finest granularity semantics can be effected by using:

```
INTERSECT(CAST(PERIOD '[May 1994 - June 1994]' MONTH AS DAY),
          PERIOD '[April 20, 1994 - May 5, 1994]' DAY)
```

This comment of [6] is certainly true in cases where the granularities of the various expressions are known. However, in programs that generate TSQL2 code dynamically (and our natural language interface is an example of such a program), it is often difficult to determine the granularities of the various expressions. For example, a program may utilise a generic SELECT pattern, that when given two expressions arg_1 and arg_2 that specify two valid-time relations r_1 and r_2, computes a new valid-time relation r. Each tuple t in r derives from a tuple t_1 in r_1 and a tuple t_2 in r_2, such that t_1 and t_2 agree on an attribute called key, and t contains all the explicit attribute values of t_1 and t_2. The time-stamp of t is the intersection of the sets of granules that time-stamp t_1 and t_2, with the intersection computed with *coarsest granularity semantics*. That is, if the time-stamp of t_1 is a set of day-granules, and the time-stamp of t_2 is a set of year-granules, then t_1 should be converted to the granularity of years before computing the intersection. (Since any time-stamp conversion will be from a finer to the coarsest granularity, it does not matter whether the conversions will be carried out using SCALE or CAST.) The problem is that the granularities of the time-stamps of t_1 and t_2 are not known in advance.

One could try to formulate the SELECT pattern as follows. (The VALID clause specifies the time-stamp of each tuple in the resulting relation. VALID(t1) and VALID(t2) return the time-stamps of t_1 and t_2 respectively.)

```
SELECT t1.*, t2.*
VALID INTERSECT(VALID(t1), VALID(t2))
FROM arg1 AS t1, arg2 AS t2
WHERE t1.key = t2.key
```

The expressions arg_1 and arg_2 would be computed at run-time. For example, they could be table names chosen at run-time, or embedded SELECT statements generated by other parts of the program. Similar patterns are used in our natural language interface. (We note that section 3.3 of [15] allows the INTERSECT function to be used only with expressions specifying periods, not with expressions referring to time-stamps, i.e. not with temporal elements or instant sets. Sections 11.12, 11.14, and 11.16 of [13], however, indicate that INTERSECT can be used with both of its arguments specifying periods, or with both of its arguments referring to temporal elements, or with both of its arguments referring to instant sets. We follow [13].)

Unfortunately, assuming TSQL2 defaults, the pattern above would not always produce the correct results, because TSQL2 by default enforces left argument granularity semantics. This means that the intersection would always be computed at the granularity of the time-stamps of r_1, which is not guaranteed to be the coarsest one. It is also not possible to solve the problem by inserting CAST or SCALE operators in the INTERSECT of the VALID clause, because we do not know the granularities of VALID(t1) and VALID(t2), and we do not know which one of these granularities is the coarsest one. Neither is it possible to solve the problem by using a statement like `SET CAST MINUTE`, because this would cause the intersection to be computed at the granularity of minutes, which is not guaranteed to be the coarsest of the granularities of `VALID(t1)` and `VALID(t2)`.

To solve problems of this kind, we propose enhancing the repertoire of `SET CAST` and `SET SCALE`, so that finest or coarsest granularity semantics can be enforced by default using statements of the form: `SET CAST AS COARSEST`, `SET CAST AS FINEST`, `SET SCALE AS COARSEST`, and `SET SCALE AS FINEST`. Using the statement `SET CAST AS COARSEST` at the beginning of the TSQL2 session would cause the `SELECT` pattern above to produce the correct results.

4.2 Period constructors

Although TSQL2 allows closed-closed, closed-open, open-closed, and open-open period literals (see section 2), there is only one period constructor, `PERIOD(`$expr_1$`,` $expr_2$`)`, that can only be used to construct periods by specifying them in their closed-closed form (see section 11.12 of [13], and sections 3.1.3 and 3.3 of [15]). The arguments of the `PERIOD` constructor, $expr_1$ and $expr_2$, are always instant-specifying expressions. `PERIOD(`$expr_1$`,`$expr_2$`)` constructs the period whose earliest instant is the one specified by $expr_1$, and whose latest instant is the one specified by $expr_2$. Intuitively, `PERIOD(`$expr_1$`,`$expr_2$`)` constructs the period $[expr_1, expr_2]$. There are cases, however, where the period that has to be constructed is $(expr_1, expr_2]$, or $[expr_1, expr_2)$, or $(expr_1, expr_2)$.

Let us imagine a job centre that uses a state valid-time relation, called `employment`, to store information showing when in the past the persons registered at the centre were employed. Assuming that the granularity of the time-stamps of `employment` is `DAY`, `employment` could have the following form. (For simplicity, in the remainder of this paper when showing a state valid-time relation, we often show the sets of maximal periods of the time-stamps of the relation, instead of the time-stamps themselves. For example, below we write {[May 1994 – July 1994], [Nov 1994 – Jan 1995]} instead of the set of granules {May 1994, June 1994, July 1994, Nov 1994, Dec 1994, Jan 1995}.)

employment		
id	name	
10345	T.Smith	{ [May 1994 - July 1994], [Nov 1994 - Jan 1995] }
10478	J.Adams	{ [July 1992 - May 1994] }
...	...	...

Let us assume that we want to compute a new state valid-time relation, that associates with each person in `employment` a time-stamp of only one maximal period, with the maximal period being $(month_1, month_2]$, where $month_1$ is the last month during which the person was employed, and $month_2$ is the present month. For example, if the current month is March 1995, the time-stamp for T.Smith in the new relation should have only one maximal period, the period (Jan 1995 – March 1995], i.e. the time-stamp should be the set of granules {Feb 1995, March 1995}. For simplicity, let us also assume that no person in `employment` was employed during the current month. The new relation would show the last periods of unemployment of the persons registered at the centre.

Using the following TSQL2 query would not compute the correct relation.

```
(1)  SELECT t.*
     VALID PERIOD(END(LAST(VALID(t))), TIMESTAMP 'now')
     FROM employment AS t
```

In (1), for each tuple of `employment`, `LAST(VALID(t))` returns the set of granules that represents the latest maximal period of the time-stamp of the tuple, and `END(LAST(VALID(t)))` returns the latest granule of that set. Assuming again that the current month is March 1995, in the resulting relation the time-stamp of T.Smith will be {Jan 1995, Feb 1995, March 1995}, while we want it to be {Feb 1995, March 1995}. In other words, the (single) maximal period of the new time-stamp of T.Smith will be [Jan 1995 – March 1995], while we want it to be (Jan 1995 – March 1995], i.e. [Feb 1995 – March 1995].

Of course, in cases like the example above, where the granularities of the expressions that are used as arguments of `PERIOD` are known, one can still produce the required result using interval arithmetic. In our example, the granularity of the first argument of `PERIOD` is known to be `MONTH` (the same as the granularity of the time-stamps of `employment`). Instead of (1) one could use (2), that would yield the correct result:

```
(2)  SELECT t.*
     VALID PERIOD(END(LAST(VALID(t))) + INTERVAL '1' MONTH,
                  TIMESTAMP 'now')
     FROM employment AS t
```

If, however, we did not know the granularity of the expression that is used as the first argument of `PERIOD`, then we would not be able to deduce whether we should add to the first argument of `PERIOD` an interval one `MONTH` long, or one `DAY` long, or one `MINUTE` long, etc. As mentioned in section 4.1, in programs that generate TSQL2 code dynamically, like for example our natural language interface, it is not always possible to predict the granularities of the various expressions.

In our natural language interface project we have added an extra argument to the `PERIOD` function, with possible values `'(]'`, `'[)'`, `'()'`, and `'[]'`. If $expr_1$ and $expr_2$ specify instants at the granularity G, and $expr_1$ specifies an instant located within a granule g_1, and $expr_2$ specifies an instant located within a granule g_2, `PERIOD(`$expr_1$`, `$expr_2$`, '(]')` constructs the period whose earliest instant is located within the granule immediately after g_1 in G, and whose latest instant is located within g_2 (intuitively the result is the period $(g_1, g_2]$). Similarly, `PERIOD(`$expr_1$`, `$expr_2$`, '()')` constructs the period whose earliest instant is located within the granule immediately after g_1 in G, and whose latest instant is located within the granule immediately before g_2 in G (intuitively the result is the period (g_1, g_2)). The semantics of `PERIOD(`$expr_1$`, `$expr_2$`, '[)')` and `PERIOD(`$expr_1$`, `$expr_2$`, '[]')` are similar. To preserve compatibility with the current TSQL2 definition, `PERIOD(`$exrp_1$`,`$expr_2$`)` is the same as `PERIOD(`$exrp_1$`,`$expr_2$`,'[]')`.

Instead of adding an extra argument to the `PERIOD` function, an alternative solution (suggested to us by Hugh Darwen and Mike Sykes) would be to introduce two new functions, `PREVIOUS` and `NEXT`. `PREVIOUS(`$expr$`)` would return the granule immediately before $expr$, and `NEXT(`$expr$`)` the granule immediately after $expr$. For example, instead of using `PERIOD(`$expr_1$`, `$expr_2$`, '(]')`, one could use `PERIOD(NEXT(`$expr_1$`), `$expr_2$`)`.

4.3 Calendric tables

Let us imagine a chemical company with many sites. Various technical problems arise occasionally at the sites of the company, and specialised company technicians (called "visiting technicians") are sent from the headquarters to the sites to help solve the problems. The company uses a state valid-time relation, called `technician_visits`, to keep track of the time each visiting technician has spent at each site. The time-stamps of `technician_visits` are specified at the granularity of minutes. `technician_visits` may have the following form. (We show the sets of maximal periods of the time-stamps rather than the time-stamps.)

technician_visits		
technician	site	
T.Smith	Glasgow Central	{ [9:00am Jan 1, 1995 - 10:20am Jan 1,1995], [4:30pm Jan 2, 1995 - 6:10pm Jan 2, 1995], [11:30pm Feb 6, 1995 - 1:40am Feb 7, 1995] }
T.Smith	Edinburgh Leith	{ [2:15pm Dec 28, 1994 - 2:55pm Dec 28, 1994] }
J.Adams	Glasgow Central	{ [4:00am Jan 7, 1995 - 9:45am Jan 7, 1995], [4:40pm Feb 10, 1995 - 6:35pm Feb 10, 1995] }
J.Adams	North Sea	{ [10:00am May 25, 1994 - 5:30pm June 4, 1994] }
...	...	...

Given a table like `technician_visits`, there does not seem to be any easy way to express questions like the following in TSQL2:

(3) Report any visiting technician who was at some site on a Sunday in 1994.
(4) Report any visiting technician who was at Glasgow Central (for some time) on every Monday in 1994.

Similar problems arise in questions involving *"every February"*, *"some summer"*, etc. The only way, for example, to express (3) seems to be to use a TSQL2 query that mentions explicitly all the 1994 Sundays, as in (5). (The reader is reminded that `(PERIOD)` is a partitioning unit; see section 2. It is unclear whether or not an expression referring to a temporal element may be used as an argument of `OVERLAPS`. Section 3.6.2 of [15] does not allow it. Table 2 in section 13.4 of [13] allows it, but additional syntax rule 3 of the same section seems to forbid it. If an expression referring to a temporal element *can* be used as an argument of `OVERLAPS`, then the `(PERIOD)` partitioning unit in (5) and in other similar `SELECT` statements of this section can be omitted.)

(5)
```
SELECT DISTINCT SNAPSHOT t.technician
FROM technician_visits(PERIOD) AS t
WHERE VALID(t) OVERLAPS
        PERIOD '[00:00 Jan 2, 1994 - 23:59 Jan 2, 1994]'
   OR VALID(t) OVERLAPS
        PERIOD '[00:00 Jan 9, 1994 - 23:59 Jan 9, 1994]'
   OR VALID(t) OVERLAPS
        PERIOD '[00:00 Jan 16, 1994 - 23:59 Jan 16, 1994]'
   OR ...
```

(5) reports the `technician` value of any tuple t in `technician_visits`, such that a maximal period of the time-stamp of t overlaps a 1994 Sunday-period.

Alternatively, one could try to express (3) using the TSQL2 `EXTRACT` function. In SQL-92, `EXTRACT` can be used to extract a field from a datetime (instant-specifying) expression or from an interval expression (see section 5.9.1 of [9]). For example, in SQL-92 the expression `EXTRACT(YEAR FROM DATE '1992-06-01')` returns the numeric value 1992. TSQL2 generalises `EXTRACT`, so that a granularity can be extracted from a datetime or interval expression (see section 3.5 of [14]). It is unclear exactly what can be extracted from an instant-specifying or interval-specifying expression in TSQL2, and exactly what the result of the extraction is supposed to be. Let us assume, however, that `EXTRACT` can be used to find out the name of the day that corresponds to an instant-specifying expression, and that the result of `EXTRACT` is an integer showing the name of the day (1 for Monday, 2 for Tuesday, etc.; according to section 5.9.1 of [9], the result of `EXTRACT` is a number). Then one could try to express (3) using a `SELECT` statement of the form:

(6)
```
SELECT DISTINCT SNAPSHOT t.technician
FROM technician_visits(INSTANT) AS t
WHERE EXTRACT(DAYNAME FROM VALID(t)) = 7
```

The `(INSTANT)` partitioning unit causes a valid-time relation to be viewed as if it were allowed to contain value-equivalent tuples, with each tuple time-stamped by a single granule. For each tuple t' in the unpartitioned form of the relation, if t' is time-stamped by the set of granules $\{g_1, g_2, g_3, \ldots, g_n\}$, there are n tuples in the partitioned form of the relation, one time-stamped by the granule g_1, one time-stamped by g_2, etc. These n tuples in the partitioned form of the relation are all value-equivalent to the original tuple t'. The granules in the partitioned form of the relation are interpreted as specifying instants. For example, `(INSTANT)` causes `technician_visits` to be viewed as if it had the following form.

technician_visits		
technician	site	
T.Smith	Glasgow Central	9:00am Jan 1, 1995
T.Smith	Glasgow Central	9:01am Jan 1, 1995
T.Smith	Glasgow Central	9:02am Jan 1, 1995
...	...	...
T.Smith	Glasgow Central	1:40am Feb 7, 1995
T.Smith	Edinburgh Leith	2:15pm Dec 28, 1994
...	...	...

Unfortunately, the `(INSTANT)` partitioning unit can only be used with event tables (see [13], section 11.3, additional syntax rule 7, and [12]), not with state tables like `technician_visits`. Even if partitioning state tables with `(INSTANT)` were allowed, it seems there would still be no easy way to express questions like (4) (universal quantification over Mondays).

TSQL2 already supports multiple calendars. As mentioned in section 2, a TSQL2 calendar describes a system that people use to measure time. TSQL2 calendars also specify the strings that can appear within the quotes of temporal literals, the meanings of these strings, and the available granularities. We propose that a TSQL2 calendar should also be allowed to define *calendric tables.* Calendric tables behave like ordinary tables stored in the database, except that they are defined by the creator of the TSQL2 calendar, and they cannot be updated. (We have not examined which mechanisms TSQL2 would have to provide to allow calendar creators to define calendric tables. It is also unclear exactly how TSQL2 calendars are created. For example there does not seem to be any `CREATE CALENDAR` TSQL2 command.)

The exact purpose and contents of each calendric table are left to the calendar creator. We assume, however, that a calendric table provides information about the time-measuring system described by the TSQL2 calendar that defines the calendric table. For example, in the case of the Gregorian calendar, the calendar creator may have defined a calendric table listing the Monday-periods, the Tuesday-periods, etc. Other calendric tables may list the periods that correspond to the various seasons (spring-periods, summer-periods, etc.), the periods that correspond to the various months (January-periods, February-periods, etc.), special days (e.g. Easter days), etc.

Let us assume that a state valid-time calendric table `&year_month_day` is available. (To distinguish calendric tables from ordinary tables, the names of calendric tables start with an ampersand.) We assume that the time-stamps of our calendric table are specified at the granularity of chronons. `&year_month_day` has the form shown below. (We show the sets of maximal periods of the time-stamps, instead of the time-stamps.)

&year_month_day				
year	month	day_num	day_name	
...	...	...	...	...
1994	September	2	Friday	$\{[c_{n_1} - c_{n_2}]\}$
1994	September	3	Saturday	$\{[c_{n_3} - c_{n_4}]\}$
...	...	...	...	...
1995	February	9	Thursday	$\{[c_{n_5} - c_{n_6}]\}$
1995	February	10	Friday	$\{[c_{n_7} - c_{n_8}]\}$
...	...	...	...	...

The calendric table above means that the period that starts at the chronon c_{n_1} and ends at the chronon c_{n_2} is a Friday, the second day of September 1994; that the period that starts at the chronon c_{n_3} and ends at the chronon c_{n_4} is a Saturday, the third day of September 1994, etc. Of course, the cardinality of `&year_month_day` will be very large, though it will not be infinite (since time in TSQL2 is bounded; see [7]). It is important, however, to realise that although `&year_month_day` behaves like a table stored in the database, it does not need to be physically present in the database. Its tuples could be computed dynamically, whenever they are needed, using some algorithm specified by the TSQL2 calendar.

Assuming that the calendric table `&year_month_day` is available, (3) can be expressed in TSQL2 as follows:

(7)
```
SELECT DISTINCT SNAPSHOT t.technician
FROM technician_visits(PERIOD) AS t, &year_month_day(PERIOD) AS d
WHERE d.day_name = 'Sunday'
  AND d.year = 1994
  AND VALID(t) OVERLAPS VALID(d)
```

(7) reports the `technician` value of any tuple t in `technician_visits`, such that a maximal period of the time-stamp of t overlaps a 1994 Sunday-period. Similarly, the answer to (4) can be found using the following TSQL2 query:

(8)
```
SELECT DISTINCT SNAPSHOT t1.technician
FROM technician_visits AS t1
WHERE NOT EXISTS
    ( SELECT *
      FROM &year_month_day(PERIOD) AS d
      WHERE d.day_name = 'Monday'
        AND d.year = 1994
        AND NOT EXISTS
          ( SELECT *
            FROM technician_visits(PERIOD) AS t2
            WHERE t2.technician = t1.technician
              AND t2.site = 'Glasgow Central'
              AND VALID(t2) OVERLAPS VALID(d)  ))
```

Intuitively, the TSQL2 query above means: *"Report any visiting technician, for which there is no 1994 Monday on which the technician did not visit Glasgow Central."*.

Questions involving counts like (9) can also be formulated easily, provided that the appropriate calendric tables are available. For example, the question:

(9) On how many Sundays was J.Adams at Glasgow Central in 1994?

can be expressed as:

(10)
```
SELECT SNAPSHOT COUNT(DISTINCT d.*)
FROM &year_month_day(PERIOD) AS d, technician_visits(PERIOD) AS t
WHERE d.year = 1994
  AND d.day_name = 'Sunday'
  AND VALID(t) OVERLAPS VALID(d)
  AND t.technician = 'J.Adams'
  AND t.site = 'Glasgow Central'
```

The TSQL2 query above counts the 1994 Sunday-periods during which J.Adams was at Glasgow Central. We cannot see how questions like (9) could be expressed without using calendric tables.

5 Conclusions

Although the built-in temporal support of TSQL2 has proven valuable in our experimental natural language interface, there are some points in the definition of TSQL2 where we think that the semantics of the language should be clarified or modified. These points have to do with the granularities of temporal literals, the behaviour of `CAST` when converting period-specifying expressions from coarser to finer granularities, and the distinction between state and event relations. Some extra features were added to TSQL2 during our project, namely a statement to enforce coarser or finer granularity semantics by default, an enhanced period constructor, and calendric tables. These extra features seem generally useful, regardless of natural language issues.

Acknowledgements

The authors wish to thank Hugh Darwen and Mike Sykes for comments on an earlier version of this paper. This paper was written while the first author was a postgraduate student at the University of Edinburgh under the supervision of the other two authors. The first author wishes to thank the Greek State Scholarships Foundation for funding his studies in Edinburgh.

Contact addresses

Correspondence concerning this paper may be addressed to: I. Androutsopoulos, Department of Artificial Intelligence, University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, Scotland, U.K.; G.D. Ritchie, Department of Artificial Intelligence, University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, Scotland, U.K.; P. Thanisch, Department of Computer Science, University of Edinburgh, King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, Scotland, U.K.

References

[1] I. Ahn. System Tables for TSQL2. TSQL2 commentary, available by anonymous `ftp` from `cs.arizona.edu`, September 1994.

[2] I. Androutsopoulos, G.D. Ritchie, and P. Thanisch. Natural Language Interfaces to Databases – An Introduction. Research paper 709, Department of Artificial Intelligence, University of Edinburgh, 1994. To appear in the *Journal of Natural Language Engineering*, Cambridge University Press.

[3] I. Androutsopoulos, G.D. Ritchie, and P. Thanisch. A Framework for Natural Language Interfaces to Temporal Databases. Research Paper 734, Department of Artificial Intelligence, University of Edinburgh, 1995.

[4] A. Copestake and K. Sparck Jones. Natural Language Interfaces to Databases. *The Knowledge Engineering Review*, 5(4):225–249, 1990.

[5] C.E. Dyreson and R.T. Snodgrass. A Timestamp Representation for TSQL2. TSQL2 commentary, available by anonymous `ftp` from `cs.arizona.edu`, September 1994.

[6] C.E. Dyreson and R.T. Snodgrass. Temporal Granularity in TSQL2. TSQL2 commentary, available by anonymous `ftp` from `cs.arizona.edu`, September 1994.

[7] C.E. Dyreson, M.D. Soo, and R.T. Snodgrass. The TSQL2 Data Model for Time. TSQL2 commentary, available by anonymous `ftp` from `cs.arizona.edu`, September 1994.

[8] C.S. Jensen, R.T. Snodgrass, and M.D. Soo. The TSQL2 Data Model. TSQL2 commentary, available by anonymous `ftp` from `cs.arizona.edu`, September 1994.

[9] J. Melton and A.R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[10] C.R. Perrault and B.J. Grosz. Natural Language Interfaces. In H.E. Shrobe, editor, *Exploring Artificial Intelligence*, pages 133–172. Morgan Kaufmann Publishers Inc., San Mateo, California, 1988.

[11] R.T. Snodgrass. A TSQL2 Tutorial. Available by anonymous `ftp` from `cs.arizona.edu`, September 1994.

[12] R.T. Snodgrass. Event Tables in TSQL2. TSQL2 commentary, available by anonymous `ftp` from `cs.arizona.edu`, September 1994.

[13] R.T. Snodgrass, I. Ahn, G. Ariav, D. Batory, J. Clifford, C.E. Dyreson, R. Elmasri, F. Grandi, Jensen C.S., W. Kaefer, N. Kline, K. Kulkarni, T.Y.C. Leung, N. Lorentzos, J.F. Roddick, A. Segev, M.D. Soo, and S.M. Sripada. TSQL2 Language Specification. Available by anonymous `ftp` from `cs.arizona.edu`, September 1994. An earlier version of this document has appeared in *SIGMOD Record*, 23(1):65–86, March 1994.

[14] R.T. Snodgrass and M. Soo. Supporting Multiple Calendars in TSQL2: An Overview. TSQL2 commentary, available by anonymous `ftp` from `cs.arizona.edu`, September 1994.

[15] M. Soo and R. Snodgrass. User-defined Time in TSQL2. TSQL2 commentary, available by anonymous `ftp` from `cs.arizona.edu`, September 1994.