SIS – Data Entry Language User's Manual

Version 2.2

Institute of Computer Science

Foundation for Research and Technology - Hellas

TABLE OF CONTENTS

1. Overview of the SIS data entry language (Telos)

1.1 General information

The data entry language of the SIS is derived from the Telos knowledge representation language. The detailed design of the original Telos language is in [1]. For the SIS only the data model was kept and all AI features removed. Extensions were made mainly to handle efficiently automatic generation of update statements. The purpose of this section is to give an overview of the features of the Telos knowledge representation language as implemented in the SIS, the syntax of the system and details on how to run it. The implementation is done completely in C++. The description of the current implementation is in [4].

The Telos language representational framework is object-oriented. Telos objects are grouped into individuals (entities, concepts, nodes) and attributes (relationships, attribute links). It provides three structuring principles, namely the classification (inverse instantiation,) specialization (inverse generalization) and the aggregation (inverse decomposition). It does not distinguish between schema and data. Schema changes may be done without loss of data at any time by simple data entry statements.

1.2 Changed features from the original Telos design

Dropped features

1. Assertional language.
2. Inference mechanism.

The inference mechanism of the query language and the assertion language have been dropped due to their inefficiency.

Additional features

1. Update mechanism (RETELL statements). The update mechanism allows changing any part of the stored description of an object without affecting other parts, as long as no inconsistency with other descriptions is caused.
2. Enhanced name scope mechanism. This name scope mechanism allows to enter entities of software descriptions e.g. with exactly the symbols appearing in the source code without causing name conflicts. *(not yet implemented in this version*)
3. Procedure triggers mechanism. These triggers are host language (C++) invocations attached to individual or attribute classes. The insertion or deletion of objects from the class triggers them. These will replace the integrity constrains of the old system. (*not yet implemented in this version*)
4. Context mechanism. A context is a part of the knowledge base, which can be queried and updated separately from other contexts. Different contexts may have overlapping objects. Each context has its own local name space (symbol table) so one can provide new names for objects existing in other contexts. (*not yet implemented in this version*)

1.3 General Concepts of the Datamodel

The built-in Classes

The SIS base contains some built-in classes, which cannot be changed by the user. They constitute the initial population, and any data entered by the user must directly or indirectly be related to these. Built-in classes are given in this chapter by *italics*.

All data a user can enter are grouped into objects. These are all instances of *Object*. Each object has a logical name as identifier. Logical names are ASCII strings, currently limited to 95 characters. Internally the names are mapped to system identifiers (SYSID) for efficiency, which are hidden to the user. Objects in the SIS base are partial descriptions of corresponding entities, concepts, relations or notions in the real world. The logical names are thought to be as close as possible to the natural language expression by which we could characterize the corresponding thing in real world. They must be unique however within their scope.

Among the objects, we distinguish individuals and attributes. Individuals correspond to real things or sets or sets of sets etc. of these. Attributes correspond to the relations among objects (and not the object they relate to) or sets or sets of sets etc. of relations. Individuals are instances of *Individual*. They have no internal structure. Attributes are instances of *Attribute*. They are objects in their own right. Any object must be an individual or an attribute. Hence

Individual ***isA*** *Telos_Object*

Attribute ***isA*** *Telos_Object*

One individual together with a set of attributes and the objects they relate to constitute a structured object in the common sense (e.g. as a C++ class instance or a C++ class). There exist no cardinality constraints on the set of attributes at the moment. Such constraints (as e.g. hold for a C++ class instance) are of minor importance for descriptive purposes.

Classes are understood as a set of "instances" which have a logical name as identifier. These instances may be classes again. Instances of a class must be declared explicitly. There is no automatic classification. Classes may have no instances. Classes a user can define are instances of *Class*. It is not allowed to define classes, which mix individuals and attributes. Hence, *Class* is partitioned into *IndividualClass* and *AttributeClass* and :

Telos_Class ***isA*** *Telos_Object*

IndividualClass ***isA*** *Telos_Class*

IndividualClass ***isA*** *Individual*

AttributeClass ***isA*** *Telos_Class*

AttributeClass ***isA*** *Attribute*

The hereto-defined built-in classes cannot be directly instantiated by the user. They can only be related by user defined attributes. Classes users can directly instantiate follow now.

Simple individuals or attributes, i.e. which are not classes, are tokens. Tokens are instances of *Token*. They are said to have the "token instantiation level". Classes, which have exclusively tokens as instances are simple classes. They are instances of *S_Class* and said to have "simple class instantiation level". Classes, which have exclusively simple classes as instances are metaclasses. They are instances of *M1_Class* and said to have "metaclass instantiation level". We continue this scheme deliberately and thus form the **Instantiation Hierarchy** of Telos (see Figure 1). Currently these are supported until *M4_Class*, but in practice it is hard to find something above meta_meta_meta level which makes sense. These "level classes" constitute a partitioning of the objects orthogonal to the individual-attribute partitioning. A user can directly instantiate the intersections of one level class with

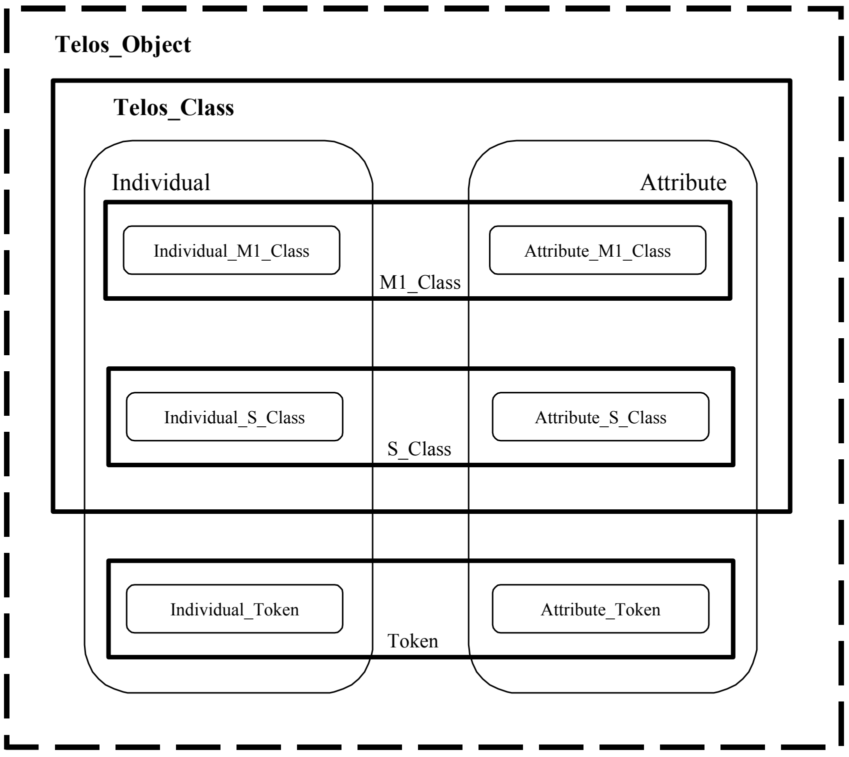

Figure 1 Instantiation Hierarchy of Telos

either *Individual* or *Attribute*, together with some user defined classes or not, and nothing else.

Hence any data entry statement must refer two built-in classes, e.g.

`TELL Individual Person in S_Class end Person`

These intersections are also built-in classes. Only for sake of readability their names are not used in the data entry statement. Hence, in the above example we refer to the class *Individual_S_Class* which is the intersection of *Individual* and *S_Class*:

Person **instance** of *Individual_S_Class*

and

Individual_S_Class ***isA*** *Individual*

Individual_S_Class ***isA*** *S_Class*

Individual_S_Class ***isA*** *IndividualClass*

S_Class ***isA*** *Telos_Class*

All other intersections are named in the same way as *Individual_S_Class* with an underscore between the two names. There are four built-in individual simple classes for primitive values:

Telos_Integer Telos_Real Telos_String Telos_Time

Primitive values cannot be created or deleted. They can only be referenced. Remind that an attribute is understood as a relation to an object, a primitive value for instance, and not as the value itself. The identity of a primitive value is its value. The instance relationship to the corresponding class is detected automatically. All built-in classes can be related by user defined attributes.

Internal representation of SIS base objects

Every individual in the SIS base, except primitive values, consists of the following single and necessary values

- SYSID - Internal identifier
- Sys_name - logical name
- Sys_class - built-in class it is instance of

and two sets

- IN_set - user defined classes it is an instance of
- ISA_set - user defined superclasses

Every attribute in the SIS base consists of the following single and necessary values

- SYSID - Internal identifier
- Sys_name - logical name
- Sys_class - built-in class it is instance of
- Sys_from - the relating object
- Sys_to - the related object

and two sets, which both may be empty

- IN_set - user defined classes it is an instance of
- ISA_set - user defined superclasses

All fields above, except Sys_class, are implemented by direct bidirectional linkage. In a semantic network, there is no preference of query direction. There is no need for building user defined secondary indexes. The direction (Sys_from -> Sys_to) of an attribute is purely linguistic. It should correspond to the name of the attribute. An attribute "father_of" may be defined inversely as "son_of" for instance.

The IN_set and ISA_set implement multiple instantiation and multiple inheritance. Multiple instantiation is useful for classification purposes. A software object with the name "MyFifo" may be instance of "C++Class" and "FIFO_Implementation" for instance. Both sets may be empty. Isa relations are not supported at token level.

Structuring with Attributes

The logical name (Sys_name) of an attribute is called also "label" in the following. The classes, an attribute is instance of, are called also the "categories" of the attribute. An attribute with empty IN_set expresses a non-classified relation. The label expresses its semantics only. In an implicit declaration (see below), it is declared under the category - keyword "attribute".

An attribute class relates two classes, Sys_from and Sys_to. Besides setmembership, as mentioned above, all instances of an attribute class must relate objects, which are instances of the class Sys_from, to objects, which are instances of the class Sys_to (Figure 2). The object p is an instance of the object P. The attribute x of object p is instance of the attribute X of the object P, the attribute X is an attribute category for attribute x. The object q is an instance of the object Q. All three instance relations may also be inherited ones (see below).

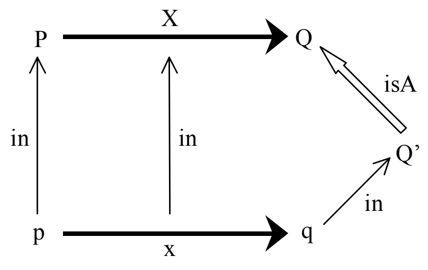

Figure 2

This rule expresses the common structuring mechanism of object-oriented datamodels. An equivalent C++ example would be:

```
class P {
        class Q *X;
       }   p;
     class Q q;
     p.X = &q;
```

In Telos, p.X may be a set. The set element, which relates p to q, has label x. At token level, the label may be omitted (see below). In Telos, p.X may be empty. An attribute x may also be instance of more than one category. Categories in Telos play an explanatory role, not a formatting role. The objects related by an attribute may be of different level (compare C++ static class member). The level of the attribute itself must be equal (default) or less to the minimum of the levels of the Sys_from and Sys_to object.

General rules related to Specialization:

Isa relations (specialization) must be declared explicitly by the user. They assume a subset relationship between the corresponding classes, even if these have no instances. IsA relations must be non-cyclic. They are transitive. Any two classes related by an isA relation must belong to the same instantiation level. If a class P is a specialization of a class Q and Q is a specialization of a class R then P is regarded as a specialization of R (inherited isA relation, Figure 3).

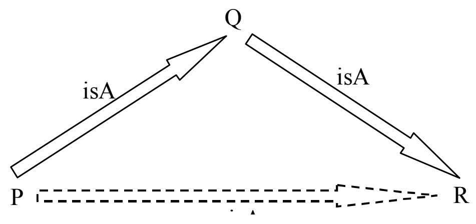

Figure 3

If a class P is a specialization of a class Q and p is an instance of P then p is regarded as an instance of Q as well (inherited instance relation, Figure 4).

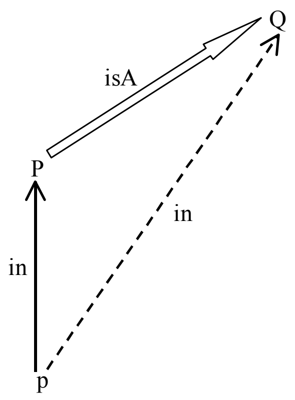

Figure 4

If an attribute class L1 is a specialization of an attribute class L2, both Sys_from of L1 must be a specialization of Sys_from of L2, and Sys_to of L1 must be a specialization of Sys_to of L2. In this case L1 "overwrites" L2 for all specializations of L1's Sys_from. This means, no instance of L1's Sys_from may have an attribute, which is explicit (non-inherited) instance of L2. If a class P is a specialization of class Q and they have both an attribute class with the same Sys_name L, but different attribute values (To_object), lets say V1 and V2 respectively, then both attribute classes L become isA related, and the class V1 must be isA to class V2 (Figure 5). All three isA relations may also be inherited ones (see above).

The datamodel supports strict multiple inheritance along the edges of the graph determined by the relevant isA relations.

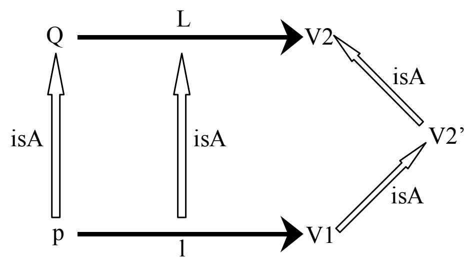

Figure 5

2. The TELL statement

The TELL statement of the Telos language provides the ability to enter objects, classes of objects, attributes and classes of attributes in to the database (SIS) and also to define hierarchy relations (ISA, INSTANCE) between objects, classes of objects, attributes and classes of attributes. Telos provides two alternative ways for associating objects with attributes:

1. explicit definition using the Attribute declaration,
2. implicit definition within the Individual declaration.

Explicit definition allows defining all possible fields of an attribute (see above). Implicit definition assigns the individual declared explicitly as Sys_from to all implicitly defined attributes. IsA relations and attributes of attributes cannot be declared implicitly. The level of an implicitly defined attribute is the default (see above).

2.1 Names of attributes

The only name scope mechanism implemented up to now, deals with attribute labels. Labels must be unique among all attributes with the same Sys_from value. Identical labels along an isA path of their Sys_from values designate an isA relation of the corresponding attributes in the same order (see above). Token level attributes may have no labels (nonnamed attributes). Their identity is regarded to be the combination of the names of their Sys_from, Sys_to, and IN_set (categories) values. It is an error to declare two nonnamed attributes with the same identity. Sometimes a simple name is not enough to reference an attribute class due to multiple inheritance or instantiation (figure 6).

$$\begin{tikzcd}
V & & U \arrow[ll, "X"', ""{name=XU}] & P \arrow[rr, "X", ""{name=XP}] & & Q \\
v \arrow[u, "in"] & & O \arrow[ll, "z", ""{name=Z}] \arrow[u, "in"] \arrow[ur, "in"] \arrow[rrr, "y"', ""{name=Y}] & & & q \arrow[u, "in"]
\arrow[from=Z, to=XU, "in"]
\arrow[from=Y, to=XP, "in"]
\end{tikzcd}$$

Figure 6

We have two attribute classes with Sys_name X. The attribute y is instance of X starting at Class P, and the attribute z is instance of X starting at Class U. To solve this name conflict we extend the Sys_name X with the keyword "from" by the Sys_name of the From_object. e.g

```
TELL Individual O in S_Class, P, U
          with X from P { here X starting at P is used  }
               y :q
          with X from U { here X starting at U is used  }
               z :v
end O
```

In the absence of the "from" clauses, the system would abort the input transaction and report a proper error message about the cause of the detected ambiguity.

2.2 Examples

This list of examples demonstrates all the possible features of the Telos Tell subsystem. Every Tell session consists of Transactions. Each transaction starts with BEGINTRANSACTION and ends with ENDTRANSACTION and has Tell operations for each Telos object. Within one transactions, the Tell statements are commutative. Comments in Telos code are included in {}. Nested comments are supported.

The following Tell transaction demonstrates how someone can define Telos objects.

```
BEGINTRANSACTION

TELL Individual Persons in M1_Class
     with attribute
          familyRelation : Persons
     end Persons

TELL Individual Person in S_Class, Persons
```

```
     with familyRelation
          fatherOf  : Person;
          motherOf  : Person
     end Person

TELL Individual LegalIdentity in S_Class
     end LegalIdentity

TELL Individual ResIdentity in S_Class
     end ResIdentity

TELL Individual Researcher in S_Class isA Person
     with attribute
          identity  : ResIdentity
     end Researcher

TELL Individual Citizen in S_Class isA Person
     with attribute
          identity  : LegalIdentity
     end Citizen

TELL Individual Status in S_Class
     end Status

{ This defines an attribute class from class  Person to
class  Status with attribute label  status }

TELL Attribute status
               From : Person
               To   : Status
          in S_Class
     end status

TELL Individual studentStatus in Token, Status
     end studentStatus

TELL Individual identity1 in Token, LegalIdentity
     end identity1

{  george  has  the  attribute  category  status  which  is
inherited   from class Person. The attribute identity is
the one from Citizen, that is stated explicitly because
Researcher has the same attribute as well, this is a case
of multiple inheritance. }

TELL Individual george in Token , Researcher, Citizen
     with fatherOf
          myFather  : mike
     with status
                    : studentStatus;
          secStatus : employeeStatus
     with identity from Citizen
                    : identity1
     end george

TELL Individual employeeStatus in Token, Status
     end employeeStatus

{ Person can have its own instances in addition to the
ones of its   subclasses }

TELL Individual mike in Token, Person
     end mike

ENDTRANSACTION
```

The following example demonstrates the use of the TELOS primitive types (i.e. Telos_Real, Telos_Integer, Telos_String, Telos_Time).

```
BEGINTRANSACTION

TELL Individual Researcher in S_Class
     with attribute
          name           : Telos_String;
          salary         : Telos_Integer;
          height         : Telos_Real;
          birth_date     : Telos_Time
     end Researcher

TELL Individual researcher1 in Token , Researcher
     with salary
          CSIsalary      : 100000
     with height
                         :1.85
     with name
                         :"george"
     with birth_date
                         :[1974 March 6]
     end researcher1

ENDTRANSACTION
```

How to write a Telos_String : A Telos_String is a string of characters and is declared between the symbols " and " (like: "george"). Inside the string special characters can be written by using the symbol "\" (like: "\n" for newline, "\0", "\t", "\r", "\b", "\f", all these special symbols have the same meaning as the relevant C or C++ special characters) or the symbol "\" followed by the octal escape sequence of the character. If the string is too long, using the symbol "\" the string can be continued to the next line skipping the blank spaces which are at the beginning of the line.

How to write a Telos_Time : Telos adopts an interval-based time model. There is a set of expressions that can be used in order to declare a Telos_Time. These expressions follow the prototype of Art and Architecture Thesaurus. They are declared between the symbols [and] (like: [1974 March 6]) and can be grouped as follows:

Date expressions: These expressions have the format [Year Month Day]. One can declare only the Year, the Year and the Month or the whole date. Months must be designated verbaly rather than numericaly. If it is the case that the date is not fully declared, (i.e. only the year is given) the interval representation of this declaration is the minimum interval that can fully contain the given information. For example if the declaration [1974] is used, its internal representation will be the interval with bounds 1974/1/1 and 1974/12/31 respectively. This interval is the minimum one that contains every date within 1974. The following examples demonstrate the use of date expressions:

[1974 March 6]	(with interval representation (1974/3/6, 1974/3/6))
[1974 March]	(with interval representation (1974/3/1, 1974/3/31))
[1974]	(with interval representation (1974/1/1, 1974/12/31))

The abbreviated form of the era designation "Before Common Era", BCE, in full capitals and with no periods, is used for all dates before the year 1 (i.e. [1453 BCE]).

Decade expressions: These expressions declare the desired decade either absolutely (i.e. [decade of 1970]) or relatively (i.e. [first decade of 19th century]). The internal representation of these expressions is again the minimum interval that contains the declared decade. The following examples demonstrate the use of decade expressions:

Absolute declaration:

Declaration	Interval Representation
[Decade of 1970]	(1970/1/1, 1979/12/31)
[Decade of 1970 BCE]	(1979/1/1, 1970/12/31)

Relative declaration:

Declaration	Interval Representation
[First decade of 20th century]	(1900/1/1, 1909/12/31)
[First decade of 20th century BCE]	(1909/1/1, 1900/12/31)

For the relative declaration the keywords (first/second/third/ .../ninth/last) are used.

There is an exception in the correspondence between the decade expressions and their interval representation. For the first decade first century CE (the first decade first century BCE) the interval representation begins at the year 1 (ends at year 1). For example the expression [First decade of 1st century] is represented as (1/1/1, 9/12/31). This is so, because we do not expect year zero as a legal year.

Century expressions: These expressions have the format [(Number) century] (i.e. [19th century]). Here are some century expression examples:

Declaration	Interval Representation
[1st century]	(1/1/1, 999/12/31)
[2nd century BCE]	(199/1/1, 100/12/31)
[16th Century]	(1500/1/1, 1599/12/31)

There is an exception in the correspondance between the century expressions and their interval representation. For the first century CE (the first century BCE) the interval representation begins at the year 1 (ends at year 1). For example the expression [1st century BCE] is represented as (999/1/1, 1/12/31).

Period Expressions: These expressions declare a time period either absolutely or relatively. The absolute period expressions declare the begining and the ending of the time period explicitly. The begining and the ending expressions can be any of the expressions mentioned above (date, decade e.t.c) and are separated with a dash. Their interval representation has lower bound the lower bound of the begining expression and upper bound the upper bound of the ending expression. The following examples demonstrate the use of absolute period expressions:

Declaration	Interval Representation
[16th century - decade of 1970]	(1500/1/1, 1979/12/31)
[14th century BCE - 1300 August CE]	(1399/1/1, 1300/8/31)
[second decade of 1400 - 3rd century BCE]	(1419/1/1, 200/12/31)

The BCE designation should appear in the ending of the period expression if both the beginning and the ending expressions are before the year 1. If only the beginning expression is before the year 1, then the BCE designation is used for the first expression while the CE (common era) expression is used for the last one (as can be seen in the examples above).

A time period can be declared relatively as well. Relative period expressions have the following formats.

- [Early/Mid/Late (number) century]. The interval representation for each of the keywords early, mid and late is the interval (0/1/1, 40/12/31), (30/1/1, 70/12/31) and (60/1/1, 99/12/31) respectively. This means that for the declaration [mid 16th century] the interval that represents the given information is (1530/1/1, 1570/12/31).
- [1st/2nd half (number) century]. These expressions correspond to the intervals (0/1/1, 60/12/31) and (40/1/1, 99/12/31) respectively (i.e. the declaration [1st half 16th century] corresponds to the interval (1500/1/1, 1560/12/31)). Note that the duration of the period assigned to each interval is 60 years rather than 50, as someone would expect. This is so, because these periods represent uncertainty periods, thus their boundaries should overlap (one cannot determine exactly when the first half ends and when the second begins).
- [1st/2nd/3rd/4th quarter (number) century]. For this expressions the intervals (0/1/1, 27/12/31), (25/1/1, 52/12/31), (50/1/1, 77/12/31) and (75/1/1, 99/12/31) are assigned respectively (i.e. the interval (1525/1/1, 1552/12/31) is assigned to the declaration [2nd quarter 16th century]).

All the keywords can be written with whatever combination of upper and lower case letters. So the declaration [Decade of 1970] can be written [decAde OF 1970] or [DECADE of 1970].

The following example demonstrates the use of **`from`**-clauses. It causes an abort to the transaction due to the fact that the token *identity1* is not an instance of the class *LegalIdentity* although it should because there exist a **`From`** clause in the declaration of token *george* specifying that the entity associated with it via the *identity* attribute must be an instance of this class.

```
BEGINTRANSACTION

TELL Individual Researcher in S_Class
     with attribute
          identity  : ResIdentity
     end Researcher

TELL Individual Citizen in S_Class
     with attribute
          identity  : LegalIdentity
     end Citizen

TELL Individual LegalIdentity in S_Class
     end LegalIdentity

TELL Individual ResIdentity in S_Class
     end ResIdentity

TELL Individual george in Token , Researcher, Citizen
with identity from Citizen
          : identity1
     end george

TELL Individual identity1 in Token, ResIdentity
     end identity1

ENDTRANSACTION
```

The following example demonstrates the creation of Isa-relations between explicitly declared attribute classes.

```
BEGINTRANSACTION

TELL Individual ResIdentity in S_Class
     isA PersonIdentity
     end ResIdentity

TELL Individual Person in S_Class
     end Person

TELL Individual Authority in S_Class
     end Authority

TELL Attribute identity
               from: Person
               to: PersonIdentity
     in S_Class
     with  attribute
               certifiedBy: Authority
     end identity

TELL Individual Researcher in S_Class isA Person
     end Researcher

TELL Individual AcadAuthority in S_Class isA Authority
     end AcadAuthority

TELL Attribute resIdentity
               from: Researcher
               to: ResIdentity
     in S_Class isA identity from Person
     with  attribute
               certifiedBy: AcadAuthority
     end resIdentity

TELL Individual PersonIdentity in S_Class
     end PersonIdentity

ENDTRANSACTION
```

The following example demonstrates the explicit declaration of token attributes.

```
BEGINTRANSACTION

TELL Individual Person in S_Class
     end Person

TELL Individual Authority in S_Class
     end Authority

TELL Attribute identity
               from: Person
               to: PersonIdentity
     in S_Class
          with  attribute
               certifiedBy: Authority
     end identity

TELL Individual PersonIdentity in S_Class
     end PersonIdentity

TELL Individual george in Token, Person
     end george

TELL Individual identity1 in Token, PersonIdentity
     end identity1

TELL Individual authority1 in Token, Authority
     end authority1

TELL Attribute myIdentity
```

```
              from: george
              to: identity1
     in Token, identity
          with certifiedBy
              :authority1
     end myIdentity
```

ENDTRANSACTION

2.3 SIS EXAMPLES

In the following we give some examples from the SIS base [3] built in FORTH. Objects at each level essentially *model* the objects in their lower level.

This allows us to include a meta-model in the SIS base for the several description models that are used for requirements, design, implementation etc. This requires three levels.

- At the intermediate level, the entities and relationships relevant to each model are defined.
- At the top level, these are grouped into classes and given the appropriate attributes. Thus, the meta-class is defined that will allow addition of new models should this be deemed necessary.
- At the lowest level, we have actual descriptions as entities and relationships are combined into constructs meaningful according to the model.

The above can be seen through the following telos definitions.

```
TELL Individual DescriptionModel in M1_Class
     with attribute
          entities       : Entity;
          connections    : Connection;
          constructs     : Construct;
     end DescriptionModel

TELL Individual Entity in M1_Class
     end Entity

TELL Individual Connection in M1_Class
     end Connection

TELL Individual Construct in M1_Class
     end Construct
```

ORM is now defined as a model, ie. an instance of the meta-model

```
TELL Individual ORM in S_Class, DescriptionModel
     with entities  : ORMRole;
                    : ORMState
     with connections
                    : RoleToRole;
                    : StateToState
     with constructs
                    : RoleTransitionRule;
                    : StateTransitionTule
     end ORM
```

At this point we enumerate the entities for the model. Below is one of them

```
TELL Individual ORMRole in S_Class, Entity
     with attribute
          property  : Property
     end ORMRole
```

When we instantiate the description, we are able to declare particular roles

```
TELL Individual CarDriver in Token, ORMRole
          .
          .
          .
     end CarDriver
```

Let us suppose that we intend to represent a program in the UNIX environment, which is implemented as a shell script. The program has three views.

1. As an entity in the file system. The attributes of this entity are related with the fact that it is seen as a file.
2. As a functional entity, ie. a program. Functional characteristics are of interest here.
3. As a UNIX script. The structure in terms of UNIX command invocations is the principal information associated with a unix script.

The three views correspond to three telos classes.

```
TELL Individual File in S_Class
     with attribute
          fileSystem     : FileSystem;
          address        : DiskAddress;
     with necessary
          filename       : FileName
     end File

TELL Individual Program in S_Class
     with attribute
          signature      : Signature;
          domain         : Domain;
          comments       : Telos_String
     with necessary
          name           : ProgramName;
     end Program
```

In the following we allow for sequencing information in a script. Information is supplied in pairs, but only one program is *necessary* in order to allow for a simple script with only one program

```
TELL Individual CommandSequence in S_Class
     with attribute
          nextCommand    : Program
     with necessary
          prevCommand    : Program;
     end CommandSequence

TELL Individual Script in S_Class
     with attribute
          signature       : Signature
     with necessary
          commandSequence : CommandSequence
     end Script
```

Assume that we want to print all the filenames under a directory in alphabetical order, regardless of the subdirectory they are in. This needs a lower level script that visits a subdirectory, lists all names and returns to the parent directory. This script, invoked through a `foreach i (*)` command produces a flat listing of all file and directory names, which can be subsequently piped to `sort`.

Let us call this program `allFiles`, store it in `/usr/etc`, and use the programs `cd` and `ls` to implement it.

```
TELL Individual prog1 in Token, Program
     with name
                    : cd
     with signature
                    : '$i'
     end prog1

TELL Individual prog2 in Token, Program
     with name
                    : ls
     end prog2

TELL Individual prog3 in Token, Program
     with name
                    : cd
     with signature
                    : '..'
     end prog3

TELL Individual allFiles1 in Token, CommandSequence
     with prevCommand
                    : prog1
     with nextCommand
                    : prog2
     end allFiles1

TELL Individual allFiles2 in Token, CommandSequence
     with prevCommand
                    : prog2
     with nextCommand
                    : prog3
     end allFiles2

TELL Individual allFiles in Token, File, Program, Script
     with commandSequence
                    : allFiles1;
                    : allFiles2
     with name
                    : 'allfiles'
     with filename
                    : /usr/etc/allFiles
     with domain
                    : Utilities
     end allFiles
```

2.4 A grammar for the TELL statement

For the syntax description, we use the following conventions:

- [] denotes optional phrases,
- {} is written if the phrases enclosed in the brackets may be repeated arbitrarily often,
- | denotes alternative phrases,
- terminal symbols are enclosed in quotes.

TELL_statement::= "**TELL**" Individual_or_Attribute_definition
in_clause [isA_clause] [with_clause]
"**end**" [identifier]

Individual_or_Attribute_definition::= "**Individual**" identifier
| "**Attribute**" identifier
Component_section

Component_section::= "**from**" "**:**" identifier
"**to**" "**:**" identifier

in_clause::= "**in**" Built_in_class {"**,**" identifier}

isA_clause::= "**isA**" identifier {"**,**" identifier} /* Tokens can not have */
/* Superclasses */

with_clause::= "**with**" attributes {attributes}

attributes::= categories attributes

categories::= identifier {"**,**" identifier}

attributes::= attribute {"**;**" attribute}

attribute::= identifier : identifier
| : identifier

```
Built_in_class::=    "Token"             /* At this statement we define the instance */
                   | "S_Class"           /* level of each object    */
                   | "M1_Class"
                   | "M2_Class"
                   | "M3_Class"
                   | "M4_Class"

identifier::=    {letter} ({letter} | {digit}| _ | ("`"{letter}) | ("`"{digit}))*       |
                 "(" any character except SPACE TAB PARENTHENSIS ")"
```

3. The RETELL statement

Up to now we have used the TELL statement to define objects their hierarchies and their relations in SIS base. With the RETELL statement we will be able to make updates in SIS base. The RETELL statement collects the following features.

- Syntax of RETELL is as close as possible to that of the TELL statement.
- You can redefine with the RETELL statement anything you can define with the TELL statement, except the assignment to the built-in system class (Sys_class). So when we use RETELL we can change the Sys_name, Sys_from and Sys_to values. We cannot delete them, because they are necessary. The RETELL statement understands such a request as deletion of the object itself. We can add or delete elements of the IN_set and ISA_set sets.
- No redundant information is required. The RETELL statement needs only information to identify the object or the set of objects to be changed, added, deleted and the corresponding action to be taken. This feature helps the user to avoid queries.
- Apply several changes to an object with in one RETELL. The consistency of the resulting changes is controlled at the end of the RETELL statement. There is no dependency on intermediate states.
- You may have more than one RETELL statements for the same object in the same transaction. In that case the updates, of each RETELL statement are executed in order the parser finds them. Inside a RETELL statement all update operations are commutative. You cannot update an object defined first time in the same transaction.

3.1 The RETELL grammar

A grammar for RETELL statement

```
RETELL_statement::    "RETELL"  current_object [Component_section]
                                [hierarchies_relations]
                      "end" [object]
                    | "RETELL" Individual_or_Attribute [hierarchies_relations]
                      "end" [object]

Individual_or_Attribute::=    "Individual" current_object S_IN_Clause
```

```
                          | "Attribute" current_object Component_section
                                                   S_IN_Clause

Component_section::=  "from" ":" object
                      "to" ":" object

hierarchies_relations::=  {clause}

clause::=     IN_Clause
            | ISA_Clause
            | WITH_Clause

S_IN_Clause::=       "in" Built_in_class "," object

IN_Clause::=         "in" object "," object

ISA_Clause::=        "isA" object "," object

WITH_Clause::=       "with" List_of_attribute_classes List_of_attributes

List_of_classes::=   object "," object

List_of_attributes::= object {";" object}

Built_in_class::=    "Token"         /* At this statement we define the instance */
                   | "S_Class"       /* level of each object    */
                   | "M1_Class"
                   | "M2_Class"
                   | "M3_Class"
                   |  "M4_Class"

object::=          |                 /* Special Addition for a nonnamed attribute */
                   |   identifier                    /* General Addition */
                   |   identifier "#"                         /* General Deletion */
                   |   identifier "@" identifier     /* General Change */
                   |   "attof" identifier {"," identifier}
                         /* Special Addition for a attribute defined  */
                          /* by its categories */
                   |   "attof" identifier {"," identifier} "#"
                         /* Special Rename of a attribute defined by its categories */
                         /* to a nonnamed attribute */
                   |   "attof" identifier {"," identifier} "@" identifier
                         /* Special Change of a attribute name defined by its  */
                         /* categories to physical name */
```

In fact the RETELL grammar is more complicated, but this simple form will help the reader to understand how the RETELL works.

3.2 How RETELL works

As you can see in the RETELL grammar there are two formats for a RETELL statement. When you use the first and simpler RETELL statement you want to update an object that already exists in SIS base from a previous transaction.

The second RETELL statement is closer to a TELL statement, this RETELL statement is used when you want to update an object, but you do not know if that object exists in SIS base. In that case it is necessary to define the type of object ("Individual" or "Attribute") and the instance level (every object must be instance of one Builtin Class). If the object exists in SIS base, parser checks if the type and the instance level in the RETELL statement are the same with the type and the instance level that object already has in SIS base. If there are not the same there is a conflict error.

The <object> in the RETELL grammar may be the name of any object in the SIS base. With <object> we can define both the name of an object and the particular update operation (Add,Delete, change) between the <current_object> and the <object>.

When we use it in <IN_Clause> we update the instance hierarchies, or when we use it in <ISA_Clause> we update ISA hierarchies between <current_object> and <object>.

Update operations can be additions, deletions, and changes. A change is defined as a conditional deletion followed by an addition, depending on the existence of the information to be deleted. First all updates are collected and divided into two sets: the deletions and additions. Then all deletions are executed on the base of the state previous to the current RETELL statement, leading to an inconsistent intermediate state. Then all additions are executed leading to the final consistent database state.

In <WITH_Clause> we update the attributes that belong to the <current_object> by listing them in <List_of_attributes>. In <List_of_attributes> we can both define attributes and update operations on these attributes. Update operations can be additions, deletions, and changes. A change is defined as a conditional deletion followed by an addition, depending on the existence of the information to be deleted. In the execution of a RETELL <WITH_Clause>, first all updates are collected and divided into two sets: the deletions and additions. Then all deletions are executed on the base of the state previous to the current RETELL statement, leading to a inconsistent intermediate state. Then all additions are executed leading to the final consistent database state. This state is used for updating instance hierarchies between any element of the set, which is defined in <List_of_attributes> and the attribute classes, which are defined in <List_of_classes>.

3.3 How we define an object

As you can see with <object> we can refer an object or an attribute. When we use the <object> to refer an object, <object> is the name of object referred. If we need to refer to an attribute, <object> has the form "name from From_object". e.g Recall the previous example in figure 6.

1 " U " is an <object>

2 " P " is an <object>

3 " X From U " is an <object>

4 " X From P " is an <object>

5 " z From O " is an <object>

Sometimes you can avoid referencing the From_object. e.g If object O (in figure 6) is only an instance of P not an instance of U You can omit the from_clause when we referencing to attribute X . So

6 " X " is an <object>

When we refer an attribute, we use two <object>'s, the Label and the To_object, and attribute has the form "Label: To_object".

Because the Label of an attribute is unique within the From_object, we can avoid To_object so attribute may also have the form "Label: ".

If we want to refer a no name attribute we write “: To_object".

Sometimes it's desirable to refer an attribute or a set of attributes only with its class name and its To_object. In this case the attribute has this format " attof class1, class2...: To_object". So you can not have more than one attributes with the same set of classes and the same To_object.

If we want to refer all attributes which they starts from "From_object" we write " attof attribute:". Because "class attribute" is the union of all attribute classes in SIS base.

If we want to refer an attribute or a set of attributes only with it's class name we write " attof class1, class2... :".

If we want to refer an attribute or a set of attributes only with it's To_object we write " attof attribute: To_object".

Because all "To_object" are individual objects in SIS base we use the keyword "individual" to present any possible "To_object". e.g

```
"attof class1, class2 From Object_A : individual"
```

We have the same result with

```
"attof class1, class2 From Object_A :"
```

If we delete the To_object the attribute is deleted. The following example will illustrate that :

Let's make the assumption that "Maria", "Nick", "Tom", "John" are all objects in SIS base and object "Maria" has the following relations:

"Maria has_father Nick"
"Maria has_husband Tom"
"Maria lives_with Tom"
"Maria has_lover John"

Relation "has_father" and "has_husband" are instance of class "Family_relations", and relation "has_lover" with relation "has_husband" are instance of class "Sex_relations", relation "lives_with" is an instance of class "Social_relations".

```
BEGINTRANSACTION

TELL Individual Maria in Token, Women_Class
```

```
    with Family_relations
             has_father      : Nick
    with Social_relations
             lives_with      : Tom
    with Sex_relations
             has_lover       : John
    with Sex_relations, Family_relations
             has_husband     : Tom
    end Maria

ENDTRANSACTION
```

With attribute " `has_father : Nick` " we refer the attribute {has_father}.

With attribute " `has_father :`" we also refer the attribute {has_father}.

With attribute " `attof attribute :` "
or " `attof attribute : individual`"
or " `: individual`"
we refer the set of all the attributes {has_father, lives_with, has_lover, has_husband}.

With attribute " `attof attribute : Nick`" we refer the attribute {has_father}.

With attribute " `attof attribute : Tom`" we refer the set of attributes {has_husband, lives_with}.

With attribute "`attof Family_relations : Nick`" we refer the attribute {has_father}.

With attribute "`attof Family_relations :`"
or "`attof Family_relations : individual`"
we refer the set of attributes {has_father, has_husband}.

With attribute "`attof Family_relations, Sex_relations :`"
we refer the attribute {has_husband}.

With attribute "`attof Family_relations, Sex_relations : Tom`"
we also refer the attribute {has_husband}.

But attribute "`attof Family_relations, Sex_relations :John`" does not exist in SIS base, because there is no relation between "Maria" and "John" as an instance either of "Family_relations" or "Sex_relations".

3.4 How we define an operation on a particular object

Up to now we have seen how we define every object or attribute or set of attributes in SIS base with the <object>. Recall that every object in SIS base has single and set values, these values contains object ids from SIS base. RETELL designed to update these values. All potential uses of RETELL are to Add, Delete or Change the contains of these values. According to these three functions of a RETELL statement (Add, Delete, Change) when we refer to an object with <object> we also define and the operation between <object> and the particular single or set values.

To accomplish that we use three notations for <object>

1. <object>
   Means ADD the Object, if object does not exist. if object exists and has the same form this operation has no effect. If object exists but has a different form, e.g different To_object (" has_friend : John" the old form, " has_friend : Tom" the new form) it is error, because Telos parser does not know if you really want to change the old form of Object.
2. <object> #
   Means DELETE the Object, if object exists. if object does not exist this operation has no effect.
3. <object1> @ <object2>
   Means Change object1 to object2 if object1 exists. if object1 does not exist this operation has no effect.

3.5 Examples

To become more familiar with the RETELL statement we present the following examples:

If you want to ADD an instance relation of Object_A to CLASS_A

```
RETELL Object_A in CLASS_A end
```

If you want to ADD an instance relation of Object_A to CLASS_A,CLASS_B

```
RETELL Object_A in CLASS_A, CLASS_B end
```

If you want to DELETE an instance relation of Object_A to CLASS_C

```
RETELL Object_A in CLASS_C # end
```

If you want to ADD an instance relation of Object_A to CLASS_A, CLASS_B and DELETE an instance relation of Object_A to CLASS_C

```
RETELL  Object_A in CLASS_A, CLASS_B, CLASS_C # end
```

If you want to REDIRECT an instance relation of Object_A to CLASS_A to CLASS_B

```
RETELL Object_A in CLASS_A @ CLASS_B end
```

Be careful!!! This statement is different from this

```
RETELL Object_A in CLASS_A #, CLASS_B end
```

The first statement checks if instance relation of Object_A to CLASS_A can be replaced by instance relation of Object_A to CLASS_B. The second statement just delete instance relation of Object_A to CLASS_A and add an instance relation of Object_A to CLASS_B.

If you want to ADD an isA relation of CLASS_A to CLASS_B

```
RETELL CLASS_A isA CLASS_B end
```

If you want to ADD an instance relation of CLASS_A to MCLASS_M and an isA relation of CLASS_A to CLASS_B

```
RETELL CLASS_A in MCLASS_M isA CLASS_B end
```

If you want to ADD an instance relation of CLASS_A to MCLASS_M and DELETE an isA relation of CLASS_A to CLASS_B

```
RETELL CLASS_A in MCLASS_M isA CLASS_B # end
```

If you want to ADD an instance relation of CLASS_A to MCLASS_M and an isA relation of CLASS_A to CLASS_B and REDIRECT an isA relation of CLASS_A to CLASS_C to CLASS_D

```
RETELL CLASS_A in MCLASS_M isA CLASS_B, CLASS_C @
CLASS_D
end
```

If you want to ADD the attribute " has_father : Nick " to object "Maria"

```
 RETELL Maria
    with Family_relations
         has_father : Nick
    end
```

The attribute " has_father : Nick" is an instance of object "Family_relations"

If you want to ADD the attributes " has_father : Nick ", " has_husband : Tom", "lives_with : Tom", "has_lover : John" to object "Maria"

```
RETELL Maria
    with Family_relations
              has_father     : Nick;
              has_husband    : Tom
    with Social_relations
              lives_with     : Tom
    with Sex_relations
              has_husband    : Tom;
              has_lover      : John
    end
```

If you want to ADD a nonname attribute which is an instance to "Family_relations" and pointing to "Nick", and the attribute "has_husband" which is also an instance to "Family_relations"

```
RETELL Maria
    with Family_relations
                             : Nick;
              has_husband    : Tom
```

```
          end
```

If you want to make all attributes which they start from "Maria" and are instances of "Family_relations" to be instances of "Social_relations"

```
     RETELL Maria
          with Social_relations
               attof Family_relations :
          end
```

This example is the previous example with one more constrain, attributes owes point to "Nick"

```
     RETELL Maria
          with Social_relations
               attof Family_relations : Nick
          end
```

If you want to make all attributes which they start from "Maria" and are both instances of "Family_relations" and "Sex_relations", " to be instances of "Social_relations"

```
     RETELL Maria
          with Social_relations
               attof Family_relations, Sex_relations :
          end
```

This example is the previous example with one more addition, the attribute " has_friend : George "

```
     RETELL Maria
          with Social_relations
               attof Family_relations, Sex_relations :;
                    has_friend  : George
          end
```

If you want to delete all attributes which they start from "Maria" and are instances of "Family_relations" and pointing to "Nick", and add the attribute "has_lover : John"

```
     RETELL Maria
          with attribute
               attof Family_relations   : Nick #;
                    has_lover           : John
     end
```

In previous example, we also want to delete all attributes that starts from "Maria" and pointing to "Tom"

```
     RETELL Maria
          with attribute
               attof Family_relations   : Nick # ;
               attof attribute          : Tom # ;
                    has_lover           : John
     end
```

If you want to redirect the attribute "has_friend : George" to pointing to "John"

```
     RETELL Maria
          with attribute
               has_friend   : George @ John
     end
```

If you want to Change attribute "has_friend : George" to " has_boy_friend : John"

```
     RETELL Maria
          with attribute
               has_friend @ has_boy_friend :George @ John
          end
```

If you to change attribute "has_friend" to " has_boy_friend" and don't want to change (or you don't now) the To_object

```
     RETELL Maria
          with attribute
               has_friend  @  has_boy_friend  :
     end
```

If you want to change the attribute with name "has_friend " to attribute "has_boy_friend " which pointing to "John"

```
     RETELL Maria
     with attribute
          has_friend  @  has_boy_friend      :  John
     end
```

If you want to redirect all attributes which pointing to "George" pointing to "John"

```
     RETELL Maria
          with attribute
               attof attribute : George @ John
     end
```

If you want to make all attributes which are instances of "Sex_relations" pointing to "John", and replace the attribute "has_father" (if exist) with attribute "has_grand_father", and add the attribute "has_friend : George"

```
     RETELL Maria
          with attribute
               attof Sex_relations : individual @ John;
                    has_father @ has_grand_father :;
                    has_friend  : George
     end
```

In previous example we use the Keyword "individual" to present any object but attribute in SIS base.

If you want to delete all attributes which are instances of "Sex_relations" and to add new attributes under this category

```
RETELL Maria
     with Sex_relations
          attof     Sex_relations : # ;
                    new_lover : George
end
```

Now let's have a full example with RETELL

```
RETELL Maria in Person,woman @ young_woman
     with Family_relations,
          Social_relations @ Sex_relations
               has_friend  : George;
          attof Sex_relations : individual @ John;
               has_husband # :
     end
```

This RETELL statement adds an instance relation to object "Maria" pointing to object class "Person". If an instance relation exists pointing from object "Maria" to object "woman" this relation is redirected pointing to object "young_woman". Then RETELL deletes the attribute "has_husband" from object "Maria", adds the attribute " has_friend : George" and then redirects all attributes which are instances of class "Sex_relations" pointing to object "John". Then all attributes that are referenced in <WITH_CLAUSE> and have not been deleted will be instance of class "Family_relations" and if some of them are instance of class "Social_relations" their instance relations to "Social_relations" will be redirected pointing to "Sex_relations".

Final in the RETELL statement we may have more than one <IN_Clause> or <ISA_Clause> or <WITH_CLAUSE>. You can mix TELL and RETELL statements in one Transaction. e.g

```
BEGINTRANSACTION

     TELL ....

     RETELL ....

     TELL ....
          .
          .
          .
     RETELL ...

     TELL ...

ENDTRANSACTION
```

4. Appendix A - Reserved Keywords

any_category
attof
Attribute
attribute
AttributeClass
Attribute_M1_Class
Attribute_M2_Class
Attribute_M3_Class
Attribute_M4_Class
Attribute_S_Class
Attribute_Token
BEGINTTANSACTION
Cin
CisA
Class
components
end
ENDTRANSACTION
from
in
Individual
IndividualClass
IndividualClass
Individual_M1_Class
Individual_M2_Class
Individual_M3_Class
Individual_M4_Class
Individual_S_Class
Individual_Token
isA
label
M1_Class
M2_Class
M3_Class
M4_Class
OmegaClass
Proposition
RETELL
S_Class
TELL
Telos_Class
Telos_Integer
Telos_Object
Telos_Real
Telos_String
Telos_Time
to
Token
with

5. Appendix B - Changes from previous versions

5.1 Changes fron version 1.3 to 1.3.1

This manual was updated to include the description of the Telos_Time primitive type declaration and handling.

5.2 Changes from version 1.3.1 to version 2.0

None. The manual version-numbering follows the code version-numbering.

5.3 Changes from version 2.0 to version 2.1

The following Telos_Time period expressions changed from
[a'/b' half (number) century] to [1st/2nd half (number) century]
[a'/b'/c'/d' quarter (number) century] to [1st/2nd/3rd/4th quarter (number) century].

6. References

[1] M. Koubarakis, J. Mylopoulos, M. Stanley and A. Borgida, *Telos Features and Formalization*, Institute of Computer Science Forth, Technical Report FORTH/CSI/TR/1989/018, Febr. 1989 also University of Toronto, Computer Science Dept., Technical Report KRR-TR-89-4, Febr. 1989

[2] T. Topaloglou and M. Koubarakis, *Implementation of Telos: Problems and Solutions*, University of Toronto, Computer Science Dept., Technical Report KRR-TR-89-8, May 1989.

[3] ITHACA.FORTH.91.E2.#4, *SIB Contents' Manual*

[4] ITHACA.FORTH.92.E2.#2, *Implementation of the SIB System*

[5] Petersen T. and Barnett J. P.(editors), *Guide to Indexing and Cataloging Art and Architecture Thesaurus,* Oxford University Press, 1994, p.47-50.

Index