CASUEL: A Common Case Representation Language

Michel Manago, Ralph Bergmann, Noël Conruyt, Ralph Traphöner, James Pasley, Jacques Le Renard, Frank Maurer, Stefan Weß, Klaus-Dieter Althoff, Sylvie Dumont

ESPRIT project 6322
Task 1.1, Deliverable D1
Public
10/10/1994
Version 2.02

Collaborators:
- AcknoSoft (prime contractor)
- tecInno
- Irish Medical Systems
- University of Kaiserslautern

For further information, send email to: casuel@informatik.uni-kl.de or contact:

Ralph Bergmann
University of Kaiserslautern
PO-Box 3049
D-67653 Kaiserslautern
Tel: +49 631 205 3361
Fax: +49 631 205 3357

Michel Manago
AcknoSoft
58a rue du Dessous des Berges
75013 Paris
Tel: (33 1) 44 24 88 00
Fax: (33 1) 44 24 88 66

© 1994, INRECA consortium.
EXECUTIVE SUMMARY

CASUEL is the Common Case Representation language (CCRL) for the INRECA system. It is the interface language between all the INRECA component systems, but it is also intended to serve (1) as the interface language between the INRECA integrated system and the external world, and (2) as a standard for exchanging information between classification and diagnostic systems that use cases. Currently, CASUEL is NOT designed to handle design and planning tasks.

CASUEL is a flexible, object-oriented frame-like language for storing and exchanging descriptive models and case libraries in ASCII files. It is designed to model naturally the complexities of real cases. CASUEL represents domain objects in a class hierarchy using inheritance and slots to describe the attribute of these domain objects. Moreover, typing constraints on slot values, as well as different kinds of relationships between objects can be expressed in the language. In its current version, CASUEL additionally supports a rule formalism for exchanging and storing completion rules and adaption rules for cases, as well as a first mechanism for defining similarity measures.

CASUEL is more concise than flat feature-value vectors for the representation of objects with a large number of potentially relevant attributes of different types, only a few of which are applicable to any given case. Its use reduces the number of information gain calculations needed for induction systems or similarity computations required for case-based reasoning.

However, CASUEL provides a lot of features, it does not require applications to use all of them. Therefore, CASUEL is a keyword-driven language that allows different system component to easily ignore irrelevant definitions. On the other hand, CASUEL is also open in the sense that new features can be defined if necessary for a particular kind of application or component.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>About this Document</td>
</tr>
<tr>
<td>2.</td>
<td>Introduction</td>
</tr>
<tr>
<td>3.</td>
<td>Changes of the Language</td>
</tr>
<tr>
<td>3.1</td>
<td>Changes from Version 1.0 to Version 1.1</td>
</tr>
<tr>
<td>3.2</td>
<td>Changes from Version 1.1 to Version 2.0</td>
</tr>
<tr>
<td>4.</td>
<td>The CASUEL Syntax</td>
</tr>
<tr>
<td>4.1</td>
<td>Notional Conventions</td>
</tr>
<tr>
<td>4.2</td>
<td>Domain Description</td>
</tr>
<tr>
<td>4.3</td>
<td>Syntax of the Descriptive Model</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Class Definitions</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Slot Definitions</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Type Definitions</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Value Definitions</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Global Object Instances</td>
</tr>
<tr>
<td>4.4</td>
<td>Background Knowledge</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Definition of Completion Rules</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Definition of Adaptation Rules</td>
</tr>
<tr>
<td>4.5</td>
<td>Similarity Measures</td>
</tr>
<tr>
<td>4.6</td>
<td>Syntax of the Cases</td>
</tr>
<tr>
<td>4.7</td>
<td>Miscellaneous Definitions</td>
</tr>
<tr>
<td>4.8</td>
<td>The CASUEL Types and Features</td>
</tr>
<tr>
<td>5.</td>
<td>Principles Governing the CASUEL Language</td>
</tr>
<tr>
<td>5.1</td>
<td>Intended Use</td>
</tr>
<tr>
<td>5.2</td>
<td>An Object-Oriented Representation</td>
</tr>
<tr>
<td>5.3</td>
<td>Multiple Instances of the Same Class</td>
</tr>
<tr>
<td>5.4</td>
<td>A Structured Representation</td>
</tr>
<tr>
<td>5.5</td>
<td>Multivalued Slots</td>
</tr>
<tr>
<td>5.6</td>
<td>Using Rules</td>
</tr>
<tr>
<td>6.</td>
<td>An Example from the Marine Sponge Application</td>
</tr>
<tr>
<td>7.</td>
<td>Summary of the CASUEL Syntax</td>
</tr>
</tbody>
</table>
1. About this Document

We are happy to present the deliverable that describes the CASUEL Common Case Representation Language. Section 2, briefly introduces CASUEL. Section 3 describes the changes that were made to CASUEL since version 1.0. Section 4 describes the syntax of the language together with its pragmatics. Section 5, explains the principles underlying the syntax and gives some simple examples of how it is used.

Section 6 presents a small example extracted from the marine sponge application done at the Museum of Natural History in Paris. Finally, we summarise the pure syntax of the language as an easy to find compact reference.

2. Introduction

CASUEL is used to represent all the information relative to a particular application domain in a common format (i.e. class descriptions, slots and their values, cases etc.). The descriptive model (observable facts) defines the basic terms used to describe the cases. It is a part of the background knowledge to the CBR and inductive components of INRECA. The case library (observed facts) is then described according to this model. An application domain is made out of the descriptive model and the case library, which are both represented in CASUEL.

CASUEL is a syntax for representing the data structures, types and data needed for building a CBR application. The philosophy of the language is object-oriented, but CASUEL is not intended to be a computer implementation of an object-oriented language. It is merely a syntax for describing all the information that is relevant for an application. The files containing CASUEL information are stored in ASCII. Possible extensions of CASUEL might provide a more elaborated format such as ASN1, a rising ISO standard for storing data into files.

3. Changes of the Language

3.1 Changes from Version 1.0 to Version 1.1

In CASUEL 1.1, we are allowing any kinds of relations (previously, we only allowed the parts relation). A relation is defined by a defslot with a "class" keyword instead of a "type" keyword. This states that the values of the slot must be instances of the given class. Certain relations can carry the semantics of the parts relations by stating a "properties parts_semantic" in the defslot.

We have enforced the use of object identifiers which were optional in CASUEL 1.0. All objects are now referred by their identifiers instead of their class name and instance number. An object identifier is local to a case.
The cardinality description which was attached to the deftype for a slot and to the defclass for a part, has now been moved to the defslot. This is consistent with the idea that the cardinal describes the number of values allowed for a slot. In addition, we have clarified the semantics associated to having a multi-valued slot. When a slot has a cardinal greater than 1, this means that the slot holds several values simultaneously - that is, the value of the slot is the conjunction of the values. We can now also express that a disjunction of values is allowed for a slot (with an upper limit) using the disjunction keyword of defslot. An & (and) and | (or) sign in the case description reminds us of the interpretation of the list of values.

As a consequence of this last change, the CASUEL types have been simplified. We no longer have composed types like set and list (this notion has been transferred from the deftype to the description of the cardinal in the defslot). The notion of interval - that is, the value of a slot is an interval, has also been transferred to the defslot.

The defcase_structure now accepts a single class instead of a list of classes. There should be only one instance of that class as the topmost (root) object in the cases.

Finally, we are making a consistent use of the underscore instead of using both minus signs "-" and underscores ".". As a consequence, statements like defcase-structure are renamed defcase_structure and keywords like english-print-name are renamed english_print_name.

### 3.2 Changes from Version 1.1 to Version 2.0

CASUEL V 2.0 is a modified and extended version of CASUEL V1.1. Several things that have turned out to be incorrect, ambiguous or not necessary have been changed. Other features such as global objects, background knowledge, and similarity measures have been included. Please note that due to several necessary modifications, CASUEL V2.0 is not anymore compatible with the previous versions of CASUEL. However, the incompatible changes have been restricted to a minimum. The specific changes with respect to version 1.1 are explicitly noted at the end of each paragraph.
4. The CASUEL Syntax

4.1. Notional Conventions

We use the Extended Backus Naur Form (EBNF) as specified by Wirth. Characters such as {}, [], and | are meta characters that are not part of the language — that is, these tokens do not appear in the final text files. Lists are described as follows:

- `{x}` means exactly one occurrence of x
- `{x}`* means zero or more occurrences of x
- `{x}`+ means one or more occurrences of x
- `{x}`++ means two or more occurrences of x
- `[x]` means zero or one occurrence of x
- `x | y` means one occurrence of x or one occurrence of y

To avoid confusion, we have put in bold font character sequences, such as keywords which are part of the language. Single characters which appear in the language have additionally been put in quotes "". For example, "." means that a dot will appear in the final text file. This convention holds for tokens that are meta characters of the language: for instance, "[" and "]" designate square brackets that appear in the final text file instead of meaning that what is between the two brackets is optional.

Finally, in CASUEL files, line comments are preceded by a star and finish at the end of the line. Additionally, comments which require more than one line of text can be started with the "/*" token and are terminated with "/*". Nested comments are allowed.

4.2. Domain Description

The domain description specifies one domain of application for the INRECA system including the descriptive model and the respective cases data. The descriptive model specifies the structure of the domain including the structure of the cases. This descriptive model consists of a hierarchy of classes (including inheritance), a set of slots which represent the attributes of each class, a set of types which specify the range of possible values of the slots and additionally a set of rule and adaptation rule declarations for automatically modifying and completing case descriptions.

Domain description

```
<CASUEL sources> ::= <domain definition>
    { <descriptive model> |
      <case data> |
      { <descriptive model> <case data> } }
```

```
<domain definition> ::= defdomain <symbol>
    [ declaration_file {<string>}+ ";" ]
    [ case_file {<string>}+ ";" ]
    case_structure <class identifier> ";"
```

© 1994, INRECA consortium.
After the **declaration_file** keyword, a list of files containing parts of the descriptive model can be specified. These files are included in the sequence of their occurrence before the current descriptive model is read. The **case_file** keyword introduces a list of files which contain *case data*. These cases files are included in the sequence of their occurrence after all parts of the descriptive model have been parsed.

After the **case_structure** keyword, a name of a class must be stated which specifies the class whose instances represent the cases themselves. This is necessary, since the modelling of a complex domain usually consists of a large number of classes whose object instances stand in some relations. From all these classes, one class must be marked to indicated the top-most object of all cases. This class may for example be called "case".

Additionally, the **target** of classification must be specified as a slot of a specific class. This target must be a slot of an object which occurs once and only once in each case (for example, in the topmost object). This slot can be of any type or class.

The **case_reference** keyword is used to declare a unique slot in a specified class which is further used as a reference to the cases. This slot must not be a relational slot. The case reference is further used for displaying and editing facilities.

Finally, a set of different language identifiers (e.g. english, german, french) can be specified after the **languages** keyword. The first language in this list is assumed to be the *default language*. For each language specified, a print name, comment or question can be defined for the domain definition itself, as well as for values, slots, classes, rules and cases. The textual information provided with these statements is presented by the INRECA user-interfaces in various situations (e.g. presentation of values, help facilities).
Additionally, any kind of very specific user-defined statements can be declared with the `<additional statement>` declaration. These additional statements are also used in any kind of declarations in the descriptive model as well as in the case library. These statements may describe, for example, application specific behavior or parameters of the CBR-system which are currently not covered by the CASUEL language itself. An additional statement is always introduced by the "extra" keyword. An arbitrary sequence of strings symbols as well as all kinds of different values may follow. The definition and handling of the semantics of such an additional statement is up to the user.

Changes:

After the `language` keyword, a list of identifiers can be specified that declare the languages supported by the domain declaration. The first language in this list is assumed to be the default language. Rule definitions and global objects are now part of the descriptive model. The target slot can now be of any type or class and is no longer restricted to nominal values. Additionally, `print names`, `case_reference` and `comments` can be added to a domain description. Additional statements are now handled the same way since a specific keyword "extra" is introduced. Finally, files can now be parsed that contain the descriptive model or the case base only. Of course, a case base cannot be loaded if the descriptive model has not been filed into the system before. The case_structure definition has included into the domain definition. The statement `def_case_structure` has been removed.

4.3. Syntax of the Descriptive Model

4.3.1. Class Definitions

Classes are the central components of the descriptive models. Classes define the properties of certain objects, the cases consist of.

```
<class definition> ::= defclass <class identifier>
   a_kind_of { <class denoter> }
   [";" slots {<slot identifier>}+]
   [";" rules {<rule identifier>}+]
   [";" adaptation_rules {<adaptation rule identifier>}+]
   [";" discriminant no]
   "{";" [ <print name>| <comment> | <additional statement> ] }* "."
```

```
<class identifier> ::= <symbol>
<class denoter> ::= class | <class identifier>
[slot identifier] ::= <symbol>
<rule identifier> ::= <symbol>
<adaptation rule identifier> ::= <symbol>
```

The `defclass` keyword introduces the definition of a new class with a new `class identifier`. The class after the `a_kind_of` keyword is defined as the superclass of the new class. Note that the class-instance relation (often called is-a) is defined further down in the objects of the defcase and is treated differently than the class-superclass relation.
After the **slot** keyword, the names of the slots which are specific for the defined class are listed. The slot names assigned to a class must be different from the slot names of the respective superclasses, or must have been defined to be of a type which is a subtype of the slot's type in the superclass. Thereby, the type of inherited slots may only be restricted. All slot names mentioned in a class definition must be previously defined by a respective slot definition.

The **rules** keyword, if present, introduces a set of rules as background knowledge, referenced by a list of **rule names**. The rules themselves are described by the **defrule** declaration which must precede the definition of the class in which the rule set is used first. Furthermore, rules are inherited along the a_kind_of relation. Rules defined for a superclass are always assumed to be valid for all of its subclasses.

In a similar manner, adaptation_rules, if present, describe a set of rules for adapting an instance of **class** within a retrieved case towards a new instance of **class** suitable for the current query. This is required, if the case from the case base does not fit correctly the needs of the query.

If the **discriminant no** flag is present, the objects of the defined class are not significant for further reasoning from cases. They only represent additional information not relevant for the similarity assessment.

The <print-name> string gives an additional natural languages description of the defined class used in the INRECA-Questionnaire.

**Changes:**

Multiple inheritance has been removed for the classes, since it is not assumed to be of any value for the applications considered yet. A comment can now be included for the class definition. A set of rules and adaptation rules, describing the background knowledge, can also be added to a class now. Additionally, some non-terminals have been consistently renamed. The <additional statement> starts with a special keyword "extra" now.

### 4.2.2. Slot Definitions

Each class usually contains a set of slots (or attributes). These slots contain the actual values a concrete object consists of. A slot may contain a basic value (defined by a type) or an other object of some class. Furthermore, a slot (called multi-value slot) may also contain more than one value or object.

```
<slot definition> ::= defslot <slot identifier> [of <class identifier>] |
{[type <type denoter> [";" default <basic value>]] | |
{class <class denoter> [";" properties parts_semantic]| |
[";" {cardinal <cardinality restriction>} | |
{disjunction <upper limit>} | |
interval]} |
[";" discriminant no] |
[";" {<print name> | <comment> | <question> | <additional statement}> }]* "."
```

```
<upper limit> ::= <positive integer>
```

```
<cardinality restriction> ::= 'I' <positive integer> ".." <positive integer> ["*" "I"]
```

© 1994, INRECA consortium.
The slot definition describes a new global slot identifier, which is further available for all subsequent definitions. Specifying a class identifier after the of keyword indicates that the current slot definitions is only valid for the slot of the mentioned class. Thereby, different classes may have slots with the same name but different specifications.

The slot definition assigns each slot a unique type, which is either a basic type, a type previously defined with a deftype clause (see next section) or a class. Slots are only allowed to contain elements of this specified type or instances of the specified class. Slots that have instances of classes as their value are relational slots. For relations, we can specify whether the slot carries the parts_semantic or not. Slots which carry the parts semantic are not allowed to cause any cycles in the resulting instance network. So, it is not allowed to for an object a to be some part of an object b and at the same time to have object b to be a part of object a.

The cardinality restriction is used to state or restrict the cardinality of a slot. The first integer number in such a cardinality restriction declares the minimum number of elements allowed in the slots value, while the second integer specifies the maximum number of elements. A "*" instead of an integer indicates that no upper limit is specified. If no cardinality restriction is specified, [0..1] is assumed as default. The semantics of having several values is, in this case, a conjunction of values. The cardinality description can be employed to restrict the cardinality of an inherited slot. In this case, the cardinality restriction must define a new cardinality which is a subset of the cardinality of the slot for the super class.

We can also specify that we accept disjunction of values with the disjunction keyword. The <upper limit> states the highest number of disjuncts that is allowed. For the sake of simplicity, conjunctions and disjunctions of values are mutually exclusive (we do not allow disjunctions of conjunctions). Disjunctions are not allowed for relational slots.

The interval keyword states that the values of the slot are intervals. This is only compatible when the slot is of an ordered type: integer, real or ordered_symbol.

A slot which is inherited by a subclass may be defined to have a different type than the slot of the class from which it is inherited. The type of the slot in the subclass must be a type which is derived from the type of the class. The range or the cardinality of an inherited slot may be restricted but may never be enlarged.

The default keyword introduces a value for a slot that is assumed unless the slot is assigned differently. The discriminant no flag indicates that the defined slot is of no interest for the induction and CBR tools. This allows to attach information to an object that is useful for the user such as the serial number of a part or a comment about the case. The <print name>, <comment> and the <question> strings are used, for instance, by the INRECA-Questionnaire.

Note that the ability to enter additional keywords allows us to define annotations for the keywords (also called facets in frame languages)

Changes:

Comments have been included. Print names, questions, and comments can occur in any order. <value> has been corrected to <basic value>. In the cardinality restriction ":;" has been changed
to ".." to be consistent with other kinds of defining intervals in CASUEL. Additionally, non-terminal symbols of the language have been renamed. The <additional statement> starts with a special keyword "extra" now.

4.2.3. Type Definitions

In the following section, those parts of the CASUEL language are defined which cope with the type definitions. Types are used to define the set of values that may be used for a particular attributes (slots) in a case.

\[
\text{<type definition> ::= deftype <type identifier> } \\
\quad \text{a_kind_of <type denoter> } \\
\quad [";" range <range restriction>] \\
\quad \{";" <additional statement> \}* "."
\]

\[
\text{<type denoter> ::= <type identifier> } \mid \\
\quad \{ \text{integer | real | string | symbol | ordered_symbol | taxonomy } \mid \text{boolean | date | time } \}
\]

\[
\text{<type identifier> ::= <symbol>}
\]

The type definition defines a new type with the name Type identifier. This type is based on a supertype type denoter, which is either a type (with type identifier) which was defined by a previous deftype clause, or a basic type. Basic types are either ordinal types i.e. integer, real, time, date and strings or nominal types i.e. symbol, ordered_symbol (ex; small medium tall), taxonomy (hierarchy of values), and booleans. The types ordered_symbol and taxonomy need a range definition which explicitly defines the ordering of the elements.

The optional range restriction declares that the elements of the defined type must be out of a specific subrange of the supertype. The new type can impose additional range restrictions to its supertype, but can never relax restrictions which hold for the supertype.

Contrary to types in classical programming languages, each type implicitly contains a special value "unknown". So, unknown is a valid value for each type. Generally, this special value is treated differently than the other values of a type. Assigning a slot the value "unknown" means that its value is undetermined and cannot be known at all. See also section 4.2.6 for more information.

Changes:

Only the non-terminal symbols have been changed to assimilate the language definition to classical programming languages. The types date and time have been added. Additional statements are allowed now.

Range Restriction

\[
\text{<range restriction> ::= \{ <interval range definition> | <enumeration definition> \} }
\]

\[
\text{<interval range definition> ::= "[" <basic value> | "*" ] "}.." <basic value> | "*" ] }
\]
The range restriction within a new type definition is used to restrict the range of the defined type to a subrange of the underlying supertype.

The range definition can be either an interval range definition or an enumeration definition. The interval range definition simply restricts the range of a totally ordered supertype to a subrange between the specified values. The "*" which may replace the upper or the lower bound value, stands for the smallest or the largest value of the supertype, respectively. Additionally, the interval range definition can also be employed to abbreviate an enumeration of unordered symbols in case a large number of symbols has to be declared. In this case, the "*" is not allowed to substitute an upper or a lower bound. For example you can declare ten symbols such as 0 to 9 by simply declaring a range of [0 .. 9]. Note that the type is still unordered. This allows to easily define integers which will be treated like nominals by INRECA. For example, the error code of a machine displayed on a control panel is an integer that ranges between 0 and 99, but it does not carry the semantics of numbers and is defined as a symbol.

The enumeration definition defines a subrange of the supertype by enumerating all elements which the new type includes. For totally ordered types, the range definition specifies the concrete ordering of the elements. An order defined once cannot be changed by a later range definition of a sub-type. For taxonomies, the range definition specifies a hierarchy containing different levels of detail of a value, which reflects a partial ordering of the values.

We can distinguish three ways of specifying the ordering in the range restriction.

1. Range definitions for totally ordered supertypes: For a totally ordered supertype, the enumeration must be a list of elements which the enumerated type shall contain. The order in which these elements are written in this list represents the ascending order of the elements in the newly defined type.

2. Range definitions for unordered supertypes: For an unordered supertype, the order in which the elements are written in the list does NOT imply any order of the elements of the type. The new type is completely unordered.

3. Range definitions for taxonomies: A partially ordered supertype (a taxonomy) needs a hierarchical description of the items included in the type. Such a hierarchical description is a taxonomy which is a tree written as a value representing a node of the tree, followed by a sequences of subtrees enclosed in square brackets. All values at a node of the tree (not only the leave nodes) are valid elements of the new type. Note, that the root node of the hierarchy is not explicitly stated in the language. This is because, the root node of the hierarchy does not present any information about a value and has consequently the meaning of "unknown" which is already an implicit element of each type. The tree imposes a partial order on the elements of that type, similar to a Hasse-diagram. All values at a path
through that tree (starting at the root node) are comparable within the defined order, with the root node value representing the smallest value.

Changes:
The topnode of the hierarchy-tree has been removed, since it represents "unknown", which is already an element of each type.

4.2.4. Value Definitions

<value definition> ::= defvalue <basic value> of <type identifier>
   │   "" [ <print name> | <comment> ])*
   │   "" [ <image> | <sound> | <video> ] [ <language identifier> ] <media reference> )*""
   │   "" <additional statement> )* "".

<media reference> ::= <string>

Value definitions allow to attach different kinds of information to a value of a type such as a print name or an image. These information can be specified separately for each language. This is primarily used by the Hypermedia interface of the INRECA system. This display information is assigned to a specific basic value of a previously defined type called type identifier. The semantics of the different reference strings will be specifically determined by the INRECA-Questionnaire.

Changes:
The question has been removed. Image, sound and video references can now be specified in any sequence. <value> has been corrected to <basic value>. A <language identifier> has been introduced for the hypermedia information. The <additional statement> starts with a special keyword "extra" now.

4.2.5 Global Object Instances

CASUEL Version 2.0 allows it to define global object instances as part of the descriptive model. A global object instance is a collection of instances of one or more classes that have a unique global object identifier associated with them. Global objects are used to avoid the multiple definition and storage of objects that occur identically in many cases. These objects are not required to be defined repeatedly in each case. They can be defined globally in the descriptive model and can simply be referenced by a global object identifier associated with each global object. Generally, global objects are like regular values of a type (e.g. a symbol) but they have additionally an internal structure described by the slots, the global object consists of. This internal structure will usually be used for similarity assessment. With a global object, a set of local objects can be defined too. These objects can be referenced in the definition of the global object (relational value list) by their local object identifier. They cannot be seen outside the definition of the global object itself.

<global object definition>::= defglobals
   <global object> { "" <object> }* ""
The .slot instantiation describes the current values of the slots of the object. Each slot name which is assigned in an object must be defined in the class of the object or in one of its superclasses. The assigned slot values must be compatible with the types or class defined in the slot definitions. For relational slots, the respective objects must be referenced by global or local object identifiers. All global identifiers can be used, but only those local identifiers that are defined within the actual "defglobals" declaration.

The special value unknown can be assigned to a slot to indicate that its value is not known and can never be known. Additionally, the special value nothing can be assigned to a conjunctive multi-value slots if its the cardinality restriction states a lower bound of "0". The special value nothing means that the slot contains no value or object. Furthermore, the special value irrelevant is introduced to specify that the value for the slot may be known, but that this value is not relevant for the target of diagnosis.

For conjunctive multi-value slots, more than one value or object identifier can be specified for a slot. The number of objects or values assigned to a slot must be compatible with the slot’s cardinality restriction. A list of such values or objects may be terminated by the unknown or the nothing keyword. In this context, the unknown keyword means that it can never be known whether more objects or values are present for this slot. This is some kind of uncertainty. The nothing keyword specifies that it is absolutely known that no more object is present for this
multi-value slot. If none of these keywords are specified, a further object may still follow and may be asked for.

For disjunctive multi-value slots (note that disjunctive relational slots are not allowed), none of the special values unknown or nothing is allowed.

Changes:

This section is completely new. It contains the section about the slot instantiations which was previously in the section about the cases.

4.4. Background Knowledge

General knowledge - we call it background knowledge - can be represented by a set of rules associated to each class. These rules describe a generally known relation or a dependency between certain slots of a class. This background knowledge cannot be employed to represent uncertain rules. Rules always refer to the slots of the class which the rule is attached to. Slots that are inherited from a superclass can also be accessed. Within the condition of a rule, the value of a slot can be accessed and tested, while a rules consequence may fill an empty slot with some new value.

Two different kinds of rules are considered:

a) Completion rules refer to a single case and which describe knowledge that is valid for all cases and situations. These rules are applied to complete the descriptions of cases in the case base and to complete the current query.

b) Adaptation rules describe relations between three different situations, namely the situation that is described by the actual query, the situation that is described in the most similar retrieved case, and the situation that is the result of an adaptation of the retrieved case for the new query.

Please note that CASUEL is just a language for storing and exchanging case-bases and related information. So far it is not meant as a general knowledge representation language. Several issues of knowledge acquisition or representation are not covered by the language. However, respective extensions may appear in future version of CASUEL.

4.4.1 Definition of Completion Rules

The following statement describes the definition of a rule set. These rules always refer to a single case.

```
<rule definition> ::= defrule <rule identifier> [ of <class identifier> ]
   { ;" <rule> }+
   { ;" { <printname> | <comment> | <additional statement> } }# "."
```

Each rule set has a rule identifier associated with it and directly belongs to the class class identifier. The class identifier can be omitted if the rule identifier has the same definition for all classes in which it occurs.
A rule is always an implication, which is deductively true in the sense that if the precondition is fulfilled then the conclusion must be true. If the precondition is not fulfilled, then the rule does not fire that is the conclusion is not executed.

**Preconditions**

The precondition itself is a conjunction of possibly negated basic conditions ("~" indicates a negated condition). Variables are allowed in the precondition and conclusion of a rule. These variables must become instantiated by the first (left-most) basic condition in which they occur. No new variables can be introduced in the conclusion.

### Precondition Syntax

A basic condition can be either a 2-ary relation, a test on class membership of an object or the calculation of a new value or object by some function external to CASUEL. Arguments to those relations and functions can be either slots, variables, basic slot values or global object identifiers. The slot access describes the value, or a set of values for a slot. A variable is a new object that can hold any kind of value or object. Variables are always local to the current rule evaluation. Variables may be assigned by the evaluation of the = relation, by the a_kind_of relation or by the calculation of an external function. If variables occur in one of the other relations or as an argument of the calculation of an external function, they must be instantiated by one of the previous basic conditions.

The a_kind_of relation allows it to test or determine the class of an object. The a_kind_of relation tests the class of the argument if a class identifier or an already instantiated variable follows the a_kind_of keyword. If a free variable is given, this variable is bound to the class of the argument. This relation fails if the argument is not an object at all, or if the argument is of a different class than specified.

The calculate construct allows it to compute a new value for a variable by calling an external function of an underlying programming language. The external function identifier determines the name of the function to be called. The arguments must all be instantiated (instantiated variables or filled slots). The last variable in the calculate condition will be bound to the calculated value.
resulted by the function. If this last variable is already instantiated, but with a value different from the value the function returns, the condition fails. It also fails, if one of its arguments is not instantiated. Note that the definitions of the calculations themselves are not part of the language since it is not necessary to exchange them between different INRECA components yet. However, future versions of CASUEL may include extensions for declaring user-defined functions or predicates.

The built-in predicates are used to test different conditions and to bind free variables to (new) values. The predicates \(>, \leq, >, \geq\) can be used to test totally or partially ordered types. The \(\text{in}\) relation can be used to test whether a value is included in an interval or in a disjunction of values. The arguments of all these predicates always must have been instantiated before.

The \(=\) predicate can be used for all kinds of objects and types. Two arguments are equal if they represent the same value of the same type or super type (according to the type hierarchy). Two objects are equal if they are both of the same class and if all values contained in the corresponding slots are equal. Two multi-value arguments are equal if they represent the same set of objects or values. However, the order in which these elements occur is irrelevant.

Additionally, the \(=\) predicate can be employed to bind an uninstantiated variable. However, one of the arguments of the \(=\) predicate must be instantiated before.

The \(<>\) (not equal) predicate can be used to test any two arguments. The condition succeeds if both arguments are instantiated and if they are not equal according to the preceding definition of the \(=\) predicate.

**Slot Access**

Each rule has a limited and clearly defined scope of slots that can be accessed for the declaration of a rule's conditions and consequences. This scope includes all slots of the associated class and all inherited slots from the superclasses. Non relational slots can be directly accessed for testing conditions and for value assignment. Relational slots allow to access slots of related objects also.

\[
\text{slot access} ::= \text{<simple slot access>} | \text{<quantified slot access>}
\]

\[
\text{<simple slot access>} ::= \text{<slot identifier>} [ \{ "->" <slot access> \} | \{ \text{low} | \text{high} | \text{size} \} ]
\]

\[
\text{<quantified slot access>} ::= \{ \text{forall} | \text{exists} \} \text{<slot identifier>} [ \{ "->" <slot access>\}]
\]

The slot access describes the access to a slot value. If a simple slot access is just given by a slot name, it refers to the value or the object that is contained in this slot. This kind of access is valid for all kinds of slots. Objects which are related to an object of the current class (via a relational slot) can be accessed by using the "->"-operator. The right side of this operator must always specify a valid slot access for the related objects. This kind of slot access can obviously only be applied for relational slots that do not contain multiple values. Finally, slots which represent an interval of values can also be referenced with the "low","high" or "size" operator. In this case "low" represents the lower value of the interval, "high" represents the upper value of the interval and "size" stands for the size of the interval, which is always an integer or real number. Additionally, the "size" construct can also be used to test the size of a disjunctive slot. In this case, the size reflects the number of possible (disjunctive) values. For all other kinds of slots,
these operators are not allowed. Especially, the "size" access is not allowed for conjunctive multi-value slots.

Conjunctive multi-value slots can also be accessed via the quantified slot access. The **forall** quantifier states that all objects or values in the following multi-value slot identifier are referenced. The **exists** quantifier states that only one object or value in the following slot is referenced. Quantified relational multi-value slots can additionally be accessed by the "->" operator. The right side of this operator must always specify a valid slot access for the related objects.

**Conclusions**

\[
\text{<conclusion> ::= \{} \text{<action>} \text{\}+}
\]

\[
\text{<action> ::= \{} \text{<slot access>} \text{"="} \text{<argument>} \text{\} |}
\]

\[
\{ \text{<slot access>} \text{a}_\text{kind}_\text{of} \text{\{ <class identifier> | <variable> \} } \}
\]

The post condition of a rule consists of a list of actions. An action can be either the assignment to a slot value or the creation of a new object. Slots can be assigned any objects or values. If the **forall** quantifier occurs in the slot access, then all referenced objects are assigned. The **exists** quantifier is not allowed in an action.

The slots being assigned to must not already hold values different from the value being assigned. Otherwise, an inconsistency in a case is determined. However, conjunctive multi-value slots can be assigned a new object or value. In this case, the new object is simply added to the conjunction of objects that is already contained in the slot. Disjunctive multi-value slots can be assigned a new disjunction, if the new assignment is already contained in the values that are previously assigned to the slot. The same holds for intervals. Intervals can be assigned to a slot if they are a sub-intervals of the interval which the slot already holds.

### 4.4.2 Definition of Adaptation Rules

The following statement describes the definition of an adaptation rule set. Contrary to the previously defined rules, adaptation rules refer to three different situations or cases which must be consequently distinguished:

a) The **query case** describes the case of the current query. This case usually does not contain a solution, but a (partial) query only. All parts of the query case are either entered by the user or derived by completion rules.

b) The **retrieved case** refers to the case that is retrieved by the CBR-process or by the induction component. This case is assumed to be (most) similar to the query case.

c) The **target case** describes a case that is computed to be a solution to the current query case. If no adaptation is required, the target case is simply the retrieved case. But in general, a target case can be constructed from the current query case and the retrieved case. The adaptation rules describe how the target case is constructed out of the query case and the retrieved case.

\[
\text{<adaptation rule definition> ::=}
\]

\[
\text{defadaptationrule <adaptation rule identifier> [ of <class identifier> ]}
\]

© 1994, INRECA consortium.
Each rule set has an *adaptation rule identifier* associated with it and directly belongs to the class *class identifier*. The class identifier can be omitted if the *adaptation rule identifier* has the same definition for all classes in which it occurs.

A rule is always an implication, which is deductively true in the sense that if the precondition is fulfilled then the conclusion must be true.

**Preconditions**

The precondition itself is a conjunction of possibly negated (~) basic conditions. Variables are allowed in the precondition and conclusion of a rule. Variables must become instantiated by the first (left-most) basic condition in which they occur. No new variables can be introduced in the conclusion.

For the semantic of the preconditions refer to the previous section.

**Slot Access**

Each rule has a limited and clearly defined scope of slots that can be accessed for the declaration of a rule's conditions and consequences. This scope includes all slots of the associated class and all inherited slots from the superclasses. Non relational slots can be directly accessed for testing conditions and for value assignment. Relational slots allow to access slots of related objects also. For the slot access, the case that is referenced (query, retrieved, or target) must be explicitly stated.
The slot access describes the access to a slot value. If a simple slot access is just given by a slot name, it refers to the value or the object that is contained in this slot. This kind of access is valid for all kinds of slots. Objects which are related to an object of the current class (via a relational slot) can be accessed by using the "->"-operator. The right side of this operator must always specify a valid slot access for the related objects. This kind of slot access can obviously only be applied for relational slots that do not contain multiple values. Finally, slots which represent an interval of values can also be referenced with the "low", "high", or "size" operators. In this case "low" represents the lower value of the interval, "high" represents the upper value of the interval and "size" stands for the size of the interval, which is always an integer or real number. Additionally, the "size" construct can also be used to test the size of a disjunctive slot. In this case, the size reflects the number of possible (disjunctive) values. For all other kinds of slots, these operators are not allowed. Especially, the "size" access is not allowed for conjunctive multi-value slots.

Conclusions

<adaptation conclusion> ::= { <adaptation action> } +

<adaptation action> ::= { "target" <simple adaptation slot access> ":=" <argument> }

The post condition of a rule consists of a list of actions. An action is always the assignment of a slot value in the target case. Slots can be assigned any objects or values.

4.5 Similarity Measures

A similarity measure is defined as a reference to an external function in an arbitrary programming language. This source code of this function is assumed to be in the file <file name> and the function <function name> is expected to have exactly two arguments. These arguments are of type <type identifier> or contain an instance of class <class identifier>. Furthermore, the called function is assumed to compute a similarity assessment which is returned as a real number between 0 and 1. The concrete functionality and semantics of such measures is defined by the specific implementation. Note that by this definition, the similarity function itself is not part of the CASUEL language. Although INRECA consists many different ways for declaring similarity measures, the similarity definitions need not to be exchanged between different components of the INRECA system yet. However, to exchange complete case-based reasoning applications between different systems, the similarity definitions are of course required. Therefore, future versions of CASUEL may be extended accordingly.
4.6. Syntax of the Cases

<case> ::= defcase <positive integer>
     objects { <object> ";" }* <object> "."

The preceding grammar rules define the syntax of a case. A case consists of a unique identifier (a positive integer) and a sequence of objects (see section 4.2.5 for details). Note that each case must contain the slots declared as target and as case reference in the domain definition.

Changes:

There are lot's of changes in the definition of the slot instantiation which are moved to the definition of the global objects. The & and the | symbols have been removed, since they are redundant. Several ambiguities in the productions for the slot instantiations have been removed. The unit has been removed. In the definition of the interval values, ",.." has been introduced. The printname and comments have been removed.

4.7. Miscellaneous Definitions

In the following, some basic parts of the language are defined. It contains the definition of the values for the basic types such as integer, real, string, symbol and real. The whole language is not case-sensitive.

<integer> ::= ["-"]<positive integer>

<positive integer> ::= {"0".."9")+

<real> ::= ["-"{"0".."9")]+ [ "."{"0".."9")]+ ["E"["+"|"-"{"0".."9")]+

<time> ::= <hour> ":" <minute> ":" <second>

<hour> ::= { ["0" | "1"]{"0".."9") } | {"20" | "21" | "22" | "23" }

<minute> ::= ["0".."5"]{"0".."9")

<second> ::= ["0".."5"]{"0".."9")

<date> ::= <day> "\" <month>"\" <year>

<day> ::= ["0".."3"){"0".."9")

<month> ::= ["0"{"1".."9") | "10" | "11" | "12"

<year> ::= [{"0".."9"){"0".."9")} {"0".."9"){"0".."9")

<boolean> ::= "true" | "false"

<symbol> ::= {"a".."z" | "A".."Z"} {"0".."9") | "a".."z" | "A".."Z" | "_" | "-"}*
All symbols must be different from the keywords of the language. All keywords are printed in bold font in the previous sections.

<string> ::= "" { <string character> | "" }* ""

<string character> ::= any ASCII character except ""

Additionally, the "\" character at the end of a line indicates that the following line continues the actual line. This is important especially within long strings which don't fit into one line. Additionally, "\n" in a string is interpreted as a new line separator in the Questionnaire, and "\t" is interpreted as the tabulator symbol.

Changes:

The definition of print_names and questions has been changed and moved forward. Comments in different languages have been introduced. Exponents have been introduced for reals. Special characters are introduced for strings.

4.8. The CASUEL Types and Features

```
  Types
    Nominal
      Boolean  Symbol
    Ordered symbol
      Taxonomy
    Ordinal
      String
        Integer  Real  Time  Date
```

Figure 1. The CASUEL type hierarchy

```
  Ordinal
    Ordered enumeration  Interval
```

Figure 2. The CASUEL range definitions

```
  Type
    Range
      < User-type >
```

Figure 3. The CASUEL restriction / specialisation of a type
5. Principles Governing the CASUEL Language

5.1. Intended Use

It is important to understand that CASUEL is not a computer implementation of an object-oriented language: it is merely a syntax for storing objects' structure and definitions. It is the responsibility of each developer to parse those CASUEL features that are used by his own component system. For example, the INRECA case manager uses the information associated to the <question> to display the corresponding string on the screen when the user fills the questionnaire. This information is useless for the induction component, which simply skips it.

We have thus developed a keyword-driven language that makes it easy to ignore some information when loading a CASUEL file: the parser skips what follows an irrelevant keyword. This also makes the language easily extensible since the CASUEL interface of the individual component systems will skip new unrecognised keywords. Hence, it is not necessary to modify the CASUEL interface of each component system when a new feature is added to the language.

The goal of CASUEL is to provide the data types and structures required for representing all the information relevant for a particular application. It is therefore at least sufficiently powerful to cope with the application domains that are already covered by the standalone products KATE and PATDEX, such as the applications to marine sponge classification and CNC machine fault diagnosis.

![Figure 4: CASUEL Communications](image)

As a consequence, CASUEL subsumes the data types and structures of each of the two systems. However, it is up to the system's developer to decide whether he wants to implement internally
all the mechanisms for handling these data types and structures in a way that matches the philosophy of CASUEL. Classes of objects inherit some of their slots from classes higher up in the hierarchy. Nevertheless, a system's implementer might choose to duplicate all this information and saturate the class definition when the CASUEL file is loaded. Thus, although CASUEL makes use of inheritance when describing the classes, a component system that reads CASUEL files does not necessarily have to incur the computational cost of such an inheritance mechanism.

5.2. An Object-Oriented Representation

Using CASUEL, we declare and define the following:

- The basic objects of the application domain. Example: a patient in a medical application.
- Relations between the basic objects. Example: the component subparts of an object.
- Slots used to describe the basic objects. Example: the age, sex and medical history of a patient.
- Type of slots that specify:
  1. Whether a slot will accept a number, like “age”, a nominal value, like “sex”,
  2. The range of allowed values for the slot. “Age” must be between 0 and 120, “sex” is either “male” or “female”
  3. The number of allowed values. “Age” can only take a single value but “medical history” can have a conjunction of several values.

Once this general descriptive model has been designed and formalised in CASUEL, the cases are also represented in CASUEL according to this model. A particular patient named Smith is 36 years old, male and with a history of allergies to penicillin and an operation to his right kidney.

A case is a collection of objects. It is defined by the CASUEL statement defcase. The objects in a case are linked with one-another by relations.

The case_structure CASUEL statement in the domain definition is used to declare the topmost object of this component parts hierarchy. Each object in a case is an instance of a class that has been defined using a defclass statement. The slots appearing in an instance object must also appear in the slots list of the corresponding defclass.

In the following example, the topmost object in the component (parts) hierarchy is an instance of class family. An instance of class family is an object with two subparts (a man and a woman) and one slot (the number of children). A man and a woman are also instances of the corresponding classes that are linked to the family object through the parts relation.

```casuel
defslot man
class man.

defslot woman
class woman.

defslot children```
class child;
cardinal [0..*].
defslot number-of-children
type integer.

defclass family
    a_kind_of class;
    slots man woman number-of-children.
defclass man
    a_kind_of class;
    slots age weight hair-colour eyes-colour children.
defclass woman
    a_kind_of class;
    slots age weight hair-colour eyes-colour pregnant children.
defclass child
    a_kind_of class;
    slots sex age.
defcase 1
    objects
        family f1
            man : m1,
            woman : w1,
            number-of-children : 1;
        man m1
            children : c1,
            age : 32,
            weight : 78,
            hair-colour : blond,
            eyes-colour : blue;
        woman w1
            children : c1,
            age : 30,
            weight : 56,
            hair-colour : dark,
            eyes-colour : brown;
        child c1
            sex : boy,
            age : 3.

In addition, the slot value must be of a type that is compatible with the type declared in the defslot statement for the class. Note that the class specification in the defslot (example: "of man") is optional. When omitted, this means that the slot definition holds for all slots with that name independently of which class they are attached.
deftype child-age-type
    a_kind_of integer;
    range [0..18].
deftype man-age-type
    a_kind_of integer;
    range [15..120].
defslot age of man
type man-age-type.
defslot age of child
type child-age-type.
When a slot listed in the defclass does not appear in the slot-value pairs of an instance, this means that slot value is not known — that is, has not yet been entered by the user. Note that this has a different semantic than when the slot has the explicit value unknown. The latter means that the value of the slot can never be known.

As a consequence, if the slot does not appear in the description of the instance, we may ask for its value, but if it has value unknown (or this value may be inferred) we should not ask it. Likewise, an object describing a case may have three states:

- It may be present, in which case there is an instance with slots and subparts.
- It may be absent, in which case we explicitly write down that there is no instance of the class — that is, we know that there is no instance of the class
- It may be unknown — that is, it is impossible to know whether there is or there is not an instance of the class.

When the existence of an object is unknown, it means that the existence of its component subobjects and its slot values are all unknown. The fact that the components and slot values are all unknown is not explicitly written in the case description. This may be inferred at run time by an implementation of the CASUEL language. Alternatively, the CASUEL parser can saturate the case description when the CASUEL file is loaded.

defcase 2
  objects
    family f1
    woman : w1,
    man : nothing;
  woman w1
    children : unknown,
    age : unknown,
    eyes-colour : brown.

In the above example, we do not know the value of weight of the object woman. However, we know that her age cannot become known. We also know that there is not a man in this family. Thus, all the slots of man become irrelevant, which can be inferred immediately here but could not be with a flat attribute-based representation. Finally, we cannot know whether there is a child (parts : unknown child). As a consequence, all the slot values of the child are also unknown. An application developer is free to implement the language in a way that matches these conventions or to modify his own representation of this information when parsing the files.

### 5.3. Multiple Instances of the Same Class

When more than one instance of the same class is present in the same case, a local identifier, is used to distinguish the different instances.

defcase 3
  objects
    family f1
    man : m1,
woman : w1,
    number-of-children : c2;

man m1
    children : c2,
    age : 32;

woman w1
    children : & c1 c2,
    age : 30;

child c1
    sex : boy,
    age : 3;
    print_name english "john";
    print_name french "jean";

child c2
    sex : boy,
    age : 1;
    print_name english "peter";
    print_name french "pierre".

In this family, there are two children. The man is the father of the second child and the woman is the mother of both children. Note that we have chosen a number as an identifier but we can also associate a print name such as john or peter enclosed in square brackets.

The name of the identifier is local to the case: the object referenced by the identifier c1 in case 1 is not necessarily related to the object referenced by the identifier c1 in case 2. When matching the two cases 1 and 2, the case-based reasoning or induction system does not have to always unify two objects with the same identifier. The ability to have multiple instances of the same class in the same case implies the necessity to introduce the notion of a logical variable and hence the CBR and inductive systems must be powerful enough to handle first order logic.

5.4. A Structured Representation

Classes may be organised in a hierarchy of generality. For instance, man and woman can be defined as subclasses of human. We can then declare all the information that is common to these classes at the highest level of generality (as it is classically done in object-oriented programs). For example, man and woman share a common set of slots and relations: age, weight and parts. Let us redefine the classes by using this knowledge.

defclass human
    a_kind_of class;
    slots children age weight hair-colour eyes-colour.

defclass man
    a_kind_of human.

defclass woman
    a_kind_of human;
    slots pregnant.

defclass child
    a_kind_of human.
We notice that the classes lower down in the hierarchy inherit their slots from their ancestor class. We only declare at lower levels slots that are unique to the class. For example, the pregnant slot is declared at the woman level because it is not relevant for all humans (a man cannot be pregnant). Thus, unlike with a flat attribute language, the object language allows us to define slots that are always relevant in a given context and we do not need to introduce special attribute values such as "not-relevant". Note that the type declaration can be shadowed (restrained) lower down in the class hierarchy. For instance, we may state that a human has a weight that ranges between 3 and 250 kilos but that a woman has a weight that ranges only between 3 and 175 kilos. Again, this is only a syntactic issue and a particular system implementer should feel to implement inheritance or to saturate the class description when the CASUEL file is loaded.

Classes are not the only entities that can be structured in a hierarchy of generality. Values can also be structured by declaring the type to be a _kind_of taxonomy. For example, we can declare a colour-type to be a kind of taxonomy, and define value light-colour with subvalues white and yellow.

```
deftype colour-type
    a_kind_of taxonomy;
    range [light-colour [white yellow] dark-colour [blue [ultramarine navy-blue] black]].
```

![Figure 5. A taxonomy of colours](image)

5.5. Multivalued Slots

Slots can have more than a single value. This information is represented with the cardinal keyword. By default, if no cardinal is indicated for a slot, the slot has one and only one value - that is, a cardinal of 1. However, it may have no value or more than one value provided we set the cardinality of the slot in the `defslot` statement.

```
deftype traffic-light-colour
    a_kind_of symbol;
    range [red yellow green]
defslot colour of traffic-light
    type traffic-light-colour;
    cardinal [1..2].
defslot colour of french-traffic-light
```

© 1994, INRECA consortium.
5.6 Using Rules

Rules are used to derive additional slot values of a case which are not entered by the user. Furthermore, adaptation rules are used to compute new values when the given value in a case in the case base must be adapted to fit the current requirements specified in the query. In all those situations, strong dependencies between the different slots exist. Rules are one means to deal with those dependencies.

As an example, recall the "family domain" introduced in section 5.2 and 5.3. Now, let’s assume a new slot "surname" and a slot "married" in the definition of a human.

```plaintext
defslot surname of human
    string.

defslot married of human
    boolean.
```

If we now want to express that children always receive their surname from their father if the parents are married, we can describe this by the following rule¹.

```plaintext
defrule surname_rule of man
    rule married = true &
        surname = ?Name
    ==> forall children->surname = ?Name.
```

The precondition of this rule is fulfilled if the slot married of an object of kind man contains the value true and an arbitrary value is assigned to the slot surname. The variable ?Name is assigned

¹In this example, we do not want to consider the situation in which a man is married with another woman than the mother of his children.
the value of the slot surname. As a consequence of the rule, the slot surname is assigned the value of ?Name in all objects contained in the slot children (of the "man" object).

If we also want to express that the children of unmarried woman receive the surname from their mother, we have to write an additional rule as follows:

```
defrule surname_rule of woman
    rule married = false & surname = ?Name
    ==> forall children->surname = ?Name.
```

In this small example we can directly see the dependency between the slots surname in different objects contained in the case representation.

Such a dependency can also be the cause of a required adaptation. Imagine, for example, an application for a travel agency, in which several journeys are collected in a case-base. Each journey contains several slots to describe the different attributes of a journey (i.e. location, kind of accodmodation, etc) containing also two slots for the duration of the journey and the price.

```
defslot duration of journey
    type integer.

defslot price of journey
    type real.
```

Now, imagine a situation in which a client specifies a query journey with a duration of 2 weeks. An appropriate case can be found in the case-base with the only difference that a duration of 3 weeks is considered. To reuse this case for the current query, its price must be adapted (linear price reduction) according to the different durations according to the following formular:

```
TARGET PRICE = \frac{\text{RETRIEVED DURATION}}{\text{QUERY DURATION}} \times \text{RETRIEVED PRICE}
```

This can be represented by an adaptation rule as follows:

```
defadaptationrule priceadaptation of journey
    rule calculate (divide, retrieved duration, query duration, ?RelDuration) &
        calculate (multiply, ?RelDuration, retrieved price, ?AdaptedPrice ) ==>
        target price = ?AdaptedPrice.
```

In the precondition of this rule, the duration of the journey retrieved from the case base is divided by the duration which is entered by the user in the query. Thereby it is assumed that the external function "divide" is available. The result of the division is then assigned to the variable ?RelDuration. In the second calculate statement, the relative duration is then multiplied with the value of the slot price from the journey case retrieved from the case base. The result of this computation is then assigned to the variable ?AdaptedPrice. Finally, as a consequence of the rule, the slot price of the (new) constructed target case is assigned the value of the variable ?AdaptedPrice.
6. An Example from the Marine Sponge Application

The following section gives a more complex examples for using CASUEL for an application which deals with the classification of marine sponges.

********************** DOMAIN **********************

defdomain sponges
  declaration_file "users/conruyt/sponges/bk.text";
  case_file "users/conruyt/sponges/cases.text";
  case_structure case;
  target identification classe;
  case_reference identification reference;
  languages french.

*********************** TYPES **********************

deftype body_couleur_type
  a_kind_of symbol;
  range (blue red green).

deftype body_forme_type
  a_kind_of taxonomy;
  range [subcylindrique [en-bouteille cylindroique]  
       [renflee [en-flute en-corolle]].

deftype classe_type
  a_kind_of symbol;
  range (paradisconeman oonema ...).

deftype body_taille_type
  a_kind_of real;
  range [0..200].

************************* SLOTS ***********************

defslot classe
  type classe_type.

defslot reference
  type string.

defslot taille of body
  type body_taille_type.

defslot forme of body
  type body_forme_type;
  cardinality [1:2].

defslot couleur of body
type body_couleur_type.

defslot body of description
class body;
properties parts_semantic.

defslot pedoncule of description
class pedoncule;
properties parts_semantic.

defslot description
class description.

defslot identification
class identification.

defslot contexte
class contexte.

***************************** CLASSES *****************************

defclass case
   a_kind-of class;
   slots description identification contexte;
   print_name french "eponges marines".

defclass identification
   a_kind_of class;
   slots classe reference.

defclass description
   a_kind_of class;
   slots body pedoncule.

defclass body
   a_kind_of class;
   slots face-exhalante
      micro-elements
taille
forme
consistance
couleur;
   print_name french "corps";

***************************** CASES *****************************

defcase 1
   objects
      case case-1
         identification : i1,
         description : d1,
         contexte : c1;
identification i1
    classe: paradisconema,
    reference: "investigatoriis Schulze 1895";

description d1
    body : b1,
    pedoncule : nothing; *** no pedoncule ***

body b1
    face-exhalante : f1 nothing, *** exactly one face exhalante***
    micro-elements : m1,
    taille : 85,
    forme : en-bouteille,
    couleur : blanc;

face-exhalante f1
    cone-central : nothing.
7. Summary of the CASUEL Syntax

Domain description

<CASUEL sources> ::= <domain definition>
    { <descriptive model> | 
    <case data> | 
    { <descriptive model> <case data> } } 

<domain definition> ::= defdomain <symbol>
    [ declaration_file {<string>}+ ";" ]
    [ case_file {<string>}+ ";" ]
    case_structure <class identifier> ";"
    target <class identifier> <slot identifier> ";"
    case_reference <class identifier> <slot identifier>
    [";" languages { <language identifier>}+ ]
    {";" {<print name> | <comment> | <additional statement> } }* ";" 

<language identifier> ::= <symbol>

<print name> ::= print_name <language identifier> <string>

<comment> ::= comment <language identifier> <string>

<additional statement> ::= extra { <symbol> | <string> | <integer> | <real> | <time> | <date> }+

<descriptive model> ::= 
    { <type definition> }*
    { <value definition> }*
    { <slot definition> }+
    { <class definition> }+
    { <rule definition> }*
    { <adaptation rule definition> }*
    { <global object definition> }*

<case data> ::= { <case> }++ 

Class Definitions

<class definition> ::= defclass <class identifier> a_kind_of { <class denoter> } 
    [";" slots {<slot identifier>}+]
    [";" rules { <rule identifier> }+]
    [";" adaptation_rules {<adaptation rule identifier>}+]
    [";" discriminant no]
    {";" {<print name>| <comment> | <additional statement> } }* ";" 

<class identifier> ::= <symbol>
<class denoter> ::= class | <class identifier>
<rule identifier> ::= <symbol>
<adaptation rule identifier> ::= <symbol>
Slot Definitions

<slot definition> ::= defslot <slot identifier> [of <class identifier>] 
{["type" <type denoter> ["," default <basic value>]] | 
[class <class denoter> ["," properties parts semantic]]} 
[";" { [cardinal <cardinality restriction>] | 
[disjunction <upper limit>] | 
interval} 
[";" discriminant no] 
{";" [ print name ] [ comment ] [ question ] [ additional statement ]} ]* "."

<slot identifier> ::= <symbol>
<upper limit> ::= <positive integer>
<cardinality restriction> ::= 
{";" <positive integer> "." <positive integer> | 
"*" | 
".." [ <positive integer> | "*" ] ""]
<question> ::= question <language identifier> <string>

Type definitions

<type definition> ::= deftype <type identifier> 
 a_kind_of <type denoter> 
[";" range <range restriction>] 
{";" <additional statement>} ]* "."
<type denoter> ::= <type identifier> | 
{ integer | real | string | symbol | ordered_symbol | taxonomy | 
boolean | date | time }
<type identifier> ::= <symbol>

Range Restriction

<range restriction> ::= { <interval range definition> | <enumeration definition> } 
<interval range definition> ::= 
{";" [ <basic value> | "*" ] "." [ <basic value> | "*" ] ""]
<enumeration definition> ::= 
{ ("(" [ <basic value> ]+ ")") } | { <taxonomy definition> }
<taxonomy definition> ::= 
{";" [ <taxonomy tree> ]+ "}
<taxonomy tree> ::= 
<basic value> [ "(" [ <taxonomy tree> ]+ ")"]

Value Definitions

<value definition> ::= 
defvalue <basic value> of <type identifier> 
{";" [ print name ] [ comment ] }* 
{";" [ image | sound | video ] [ <language identifier> ] <media reference> }* 
{";" <additional statement> ]}* "."
<media reference> ::= <string>

**Global Object Instances**

<global object definition> ::= 
    defglobals <global object> { ";" <object> }* ".

global object ::= 
    <class identifier> <global object identifier>
    { <slot instantiation> ";" }* <slot instantiation>
    { "," { <printname> | <comment> | <additional statement> } }*

<object> ::= 
    <class identifier> <local object identifier>
    { <slot instantiation> ";" }* <slot instantiation>
    { "," { <printname> | <comment> | <additional statement> } }*

<slot instantiation> ::= 
    <slot identifier> ":" { <single or multiple value> | <interval value> | irrelevant | unknown | nothing }

<single or multiple value> ::= 
    <relational value list> | <basic value list>

<basic value list> ::= 
    { <basic value> }+ [unknown | nothing ]

<relational value list> ::= 
    { <object identifier> }+ [ unknown | nothing ]

<interval value> ::= 
    { "["<basic value> "." <basic value>"]

<basic value> ::= 
    <integer> | <real> | <string> | <symbol> | <boolean> | <time> | <date>

<object identifier> ::= 
    <local object identifier> | <global object identifier>

<global object identifier> ::= "@" <symbol>

<local object identifier> ::= "@" <symbol>

**Rules**

<rule definition> ::= 
    defrule <rule identifier> [ of <class identifier> ]
    { ";" <rule> }+
    { "," { <printname> | <comment> | <additional statement> } }* ".

<rule> ::= rule <pre cond> "==" <conclusion>

<pre cond> ::= ["~"] <basic cond> { "&" ["~"] <basic cond> }*

© 1994, INRECA consortium.
<basic cond> ::= 
    { <argument> <predicate> <argument> } |
    { <argument> a_kind_of { <class identifier> | <variable> } } |
    calculate "("<external function identifier> "," { <argument> "," }+ <variable>")"

<argument> ::= <slot access> | <variable> | <constant> | <global object identifier>

<variable> ::= "?" <symbol>

<constant> ::= "" <basic value>

<predicate> ::= "=" | "<" | "<" | "=" | ">" | ">=" | "in"

<external function identifier> ::= <symbol>

[slot access] ::= <simple slot access> | <quantified slot access>

<simple slot access> ::= 
    <slot identifier> [ { ">" <slot access> } | { low | high | size } ]

<quantified slot access> ::= 
    { forall | exists } <slot identifier> [ { ">" <slot access> }]}

<conclusion> ::= { <action> }+

?action> ::= 
    { <slot access> ":=" <argument> } |
    { <slot access> a_kind_of { <class identifier> | <variable> } } |

Adaptation Rules

<adaptation rule definition> ::= 
    defadaptationrule <adaptation rule identifier> [ of <class identifier > ] |
    { ";" <adaptation rule> }+ |
    { ";" { <printname> | <comment> | <additional statement> } }# "."

<adaptation rule> ::= rule <adaptation pre cond> ":==" <adaptation conclusion>.

<adaptation pre cond> ::= 
    ["~"] <adaptation basic cond> { "&" ["~"] <adaptation basic cond> }

<adaptation basic cond> ::= 
    { <adaptation argument> <predicate> <adaptation argument> } |
    { <adaptation argument> a_kind_of { <class identifier> | <variable> } } |
    calculate "(" <external function identifier> "," |
    { <adaptation argument> "," }+ |
    <variable>")"

<adaptation argument> ::= 
    <adaptation slot access> | <variable> | <constant> | <global object identifier>

<adaptation slot access> ::= 
    { "query" | "retrieved" | "target" } <simple adaptation slot access>
<simple adaptation slot access> ::= 
   <slot identifier> [ { "->" <simple adaptation slot access>} | { low | high | size } ]

<adaptation conclusion> ::= [ <adaptation action> ] +

<adaptation action> ::= { "target" <simple adaptation slot access> "=" <argument> }

Similarity Measures

<similarity definition> ::= 
   defsim <similarity identifier> of { <type identifier> | <class identifier> }  
   external <file name> <function name> 
   { ";" <additional statement> }* "."

Cases

<case> ::= defcase <positive integer> 
   objects { <object> ";" }* <object> "."

Miscellaneous Definitions

<integer> ::= ["-" ] <positive integer>

<positive integer> ::= [{"0".."9"}]+

<real> ::= ["-" ] {"0".."9"}+ [ "." {"0".."9"}+] ["E" ["+"|"-" ] {"0".."9"}+ ]

<time> ::= <hour> ":" <minute> ":" <second>

<hour> ::= { ["0" | "1"] {"0".."9"} } | { "20" | "21" | "22" | "23" }

<minute> ::= ["0".."5"] {"0".."9"}

<second> ::= ["0".."5"] {"0".."9"}

<date> ::= <day> "\" <month> "\" <year>

<day> ::= ["0".."3"] {"0".."9"}

<month> ::= [ [ ["0"] ["1".."9"] ] | "10" | "11" | "12"

<year> ::= [{"0".."9"} {"0".."9"}] {"0".."9"} {"0".."9"}

Only correct dates are allowed.

<boolean> ::= "true" | "false"

<symbol> ::= {"a".."z" | "A".."Z"} {"0".."9"} | "a".."z" | "A".."Z" | "." | "-" }

<string> ::= "" {<string character> | ""} * ""

<string character> ::= any ASCII character except ""

© 1994, INRECA consortium.
INDEX

?; 34
defclass; 5; 31
defdomain; 3; 31
defdomain; 10; 33
defrule; 12; 33
defsim; 17; 35
defslot; 6; 32
defvalue; 10; 32
discriminant no; 6; 32
disjunction; 6; 32
domain; 3
domain definition; 3; 31
domain description; 3
enumeration definition; 9; 32
evaluation; 14; 34
external; 35
external function identifier; 13; 34
extra; 4; 31
false; 35
file name; 17
forall; 14; 34
function name; 17
global object; 11; 33
global object definition; 10; 33
global object identifier; 10; 11; 33
Global Object Instances; 10
global object instance; 11; 33
high; 14; 34; 35
hour; 18; 35
image; 10; 32
integer; 8; 18; 32; 35
interval; 6; 32
interval range definition; 8; 32
interval value; 11; 33
irrelevant; 11; 33
language; 5
language identifier; 4; 31
languages; 4; 31
local object identifier; 10; 11; 33
low; 14; 34; 35
media reference; 33
minute; 18; 35
Miscellaneous Definitions; 18
month; 18; 35
multivalued slot; 6
Multivalued Slots; 25

© 1994, INRECA consortium.