APIIS Implementer Documentation

written by

Eildert Groeneveld
Hartmut Börner
Zhivko Duchev
Marek Imialek
Helmut Lichtenberg
Detlef Schulze

Mariensee, July 2004

Contents

Chapter 1

Introduction

This paper represents the documentation of APIIS. It tries to explain all steps required to setup and run an information system. This will be done using one reference database for breeding programs and is available as a module (ref_breedprg) in the CVS tree. It resides in $PDBL_HOME/ref_breedprg. Other modules come soon, for example for manage the datas from molecular laboratories or for cryo conservation systems.

The directory initial contains all data and programs required to load the historic data. The batch job runall should be sufficient to execute the complete procedure for creating the database and loading the historic data sets. There are two further runall jobs, runall_cleaning_and_LO.pl to demonstrate the handling of load objects and the error handling, and runall_output.pl to show some outputs from the database.

Chapter 2

Fixed APIIS Structure

This chapter describes that part of the database structure which is the same in all APIIS database.

2.1 TRANSFER

2.2 CODES

2.3 UNIT, NAMING, ADDRESS

These three tables cosntitute one block which administers all units, persons and addresses in the database. As such it should be quite generic and should be used, largely unchanged in every database.

2.3.1 ADDRESS

This table holds the individual physical addresses, one entry for each. The primary key is the sequence DB_ADDRESS

2.3.2 NAMING

Here we store all individual persons and organization which are part of the system. Many people can have the same address, that is why in a normalized way this has been split into ADDRESS and NAMING. The primary key is the DB_NAME which is a sequence.

2.3.3 UNIT

EXT_ID is the external identification used in the outside world. This may be the number or code "1" for the veterinarians. Then EXT_UNIT would be "VET". If we would merge veterinarians from Bavaria and Saxony in one system, and the vets were numbered within each state beginning with "1", the EXT_UNIT would be "VET-SAX" and "VET-BAV". In this way the old numbering systems can continue to be used (which is actually the motivation for

the EXT_UNIT/EXT_ID setup. Thus, EXT_UNIT has to be defined such that the EXT_IDs being used are made unique.

One other objective is the possibility to reuse external numbers. Assume that the head veterinarian with EXT_ID="1" retires and is replaced by his successor. This person should again get the EXT_ID="1". Then, the same as in TRANSFER, the old EXT_ID gets closed (CLOSING_DT is set) and a new record with the EXT_=ID="1" is created.

The unique index is EXT_UNIT——EXT_ID where CLOSING_DT is NULL. This means that we are allowing one "open" channel for the combination of external unit and external identification.

2.3.4 Navigation

Access to ADDRESS and NAMING is straight forward. Let us assume a new address is to be entered. The following steps are performed:

1. verify that the address to be entered does not already exist. This can be done by search the table ADDRESS on any fields with a LIKE expression.

2. enter a new address. The primary key is simply the sequence DB_ADDRESS. Note that EXT_ADDRESS is just a plain column which may be used for locating a record in an interactive manner. Notice that COUNTRY needs to be defined in CODES (Foreign Key).

It is the sole responsibility of the user to ensure that no duplicate addresses are entered. While this may sound dangerous it is actually the only logical way. If an address is entered twice there is really no harm done as this and the other may be used together as DB_ADRESS will be used to link the address to people.

NAMING is delt with in a similar manner. The insertion process is identical to ADDRESS:

1. verify that the person/organization to be entered does not already exist by searching on any of the fields interactively.

2. insert the new person/organization. The primary key DB_NAMING will automatically come from a the sequence seq_naming__db_nam. Notice that LANGUAGE needs to be defined in CODES (Foreign Key).

UNIT has a somewhat different position in this triplet. It defines the telefon-numbers, fax, e-mail belonging to a person or organization for a given role. To create a new external ID, let us say the second veterinarian, we would do the following:

1. type EXT_UNIT='VET' and EXT_ID='2' into unit

   (a) locate the address of this person in ADDRESS (remember DB_ADDRESS)

   (b) locate the person in NAMING (remember DB_NAMING)

2. insert record in UNIT with all data using DB_ADDRESS and DB_NAMING, DB_UNIT is derived from the sequence, OPENING_DT is set to the current date while CLOSING_DT is NULL.

Thus far, no statemens have been made as to the status of DB_UNIT. While in ADDRESS and NAMING the corresponding DB_ columns function as primary keys, this is not the case here. Thus, if the user choses to create another "input channel" for a different EXT_UNIT/EXT_ID with the same DB_UNIT nothing will prevent her from doing this. What needs to be considered is however that a select on DB_UNIT will then return two records instead of one.

Chapter 3

The business and modify rules

3.1 The business rules

The currently implemented business rules are shown in table 3.1 on page 14.

3.2 The Modify Rules

Sometimes it is necessary to modify the incoming value before it is fed to the business rules. These modify methods are: \par\vspace{4mm}

Method	Description	Example
UpperCase	converts all passed date into uppercase letters	"UpperCase"
LowerCase	converts all passed date into lowercase letters	"LowerCase"
ConvertBool	accepts YyJjNn and converts it to the appropriate boolean expression (true/false)	"ConvertBool"
CommaToDot	translates all commas , into dots . (mainly used for numerical date)	"CommaToDot"
DotToColon	translates all dots . into colons : (useful for fast typing of date/time values (16.34.00 =¿ 16:34:00	"DotToColon"
Encode	the current value is taken as external code within a class of table codes. It returns the encoded db_code.	"Encode BREED"
Encode_unit	the unit needs a special treatment for encoding	"Encode_unit"
Encode_animal	animal encoding is a special case, too	"Encode_animal"
SetNow	sets the value to the current time	"SetNow"
SetUser	sets the value to the user who is running this job	"SetUser"

3.3 Layering of Business Rules

As described above, all business rules are specified as properties of the the columns nd are thus part of their definition. This results in one set of rules applied to any database modifications. However, there may be circumstances

Table 3.1: Implemented Business Rules

Method	Description	Example
Range	value has to be within given range	"Range 10 33"
List	only values from this list are allowed	"List LR LW PI DU"
NotNull	has to have a value	"NotNull"
IsANumber	only a number is accepted (5, -.37, +5.3e-3)	"IsANumber"
NoNumber	value must not contain any number	"NoNumber"
IsAFloat	value must be a floating point number (+4.3, .27, -0.5)	"IsAFloat"
Unique	the value is a unique key in the current table	"Unique"
ForeignKey	value has to have an entry in the specified table and column, possibly with some additional conditions	"ForeignKey animal db_animal", "ForeignKey v_animal db_animal ext_sex=1 ext_breed=DL"
DateDiff	the difference between the content of farrowing date and current columns of the current record must be with the given range DateDiff takes the current value of the record and computes the difference to date given in the first parameter (compare_date). If the difference (in days) between the cure and compare_date is in the range given by min_diff and max_diff the rule is passed successfully. compare_date can either be a fixed format date like Mar-22-2001, a column of the current record or a date in a database table/column with the (hardcoded) db_animal value of the current record. The syntax for this format is 'tablename=¿columname'. If no compare date comes from the database the check is also successfully.	"DateDiff farrow_dt 20 56", "DateDiff Mar-22-2001 50 90", "DateDiff animal=¿birth_dt 1 65"
LastAction	LastAction is a conditional DateDiff depending on the value of the last action. For each element of this last action list an allowed range (in days) has to be specified. If last action was SEL[ECTION] the range is 20 100, if it was AI the range is 18 30	"LastAction SEL 30 100 AI 18 30 LITTER 10 34"
IsEqual	This example is placed in the CHECK attached to column db_sire e.g. in service and tests if the animal ID given is indeed a male. Thus, the second parameter (i.e. $data_column) takes its value from the current column Notice that the constant is specified as external code.	IsEqual (animal db_animal db_sex 'M');
ReservedStrings	ReservedStrings checks if the passed $date contains one of the reserved strings which are defined in pdblrc.	"ReservedStrings"

that one set of rules is not sufficient to describe all data coming into the database. For instance, the database may contain data from the nucleus level of a breeding program and also data from the production level. Clearly, business rules may be different for the two types of data. To accommodate this situation APIIS has implemented sets or layers of business rules. The philosophy behind this is, that data streams can be subdivided into distinct classes of data which have their own set of rules. Examples are (as mentioned above) nucleus versus production level. Others could be fat breeds versus less fat (like Meishan vs Landrace).

Operationally, business rules layers are defined as additional CHECK in the model file. They are written as CHECK1 for a layer 1, CHECK2 for a layer 2 etc. An example is given in table 3.2. The column db_sex requires is a foreign key in CODES and must not be NULL in the base (default) level as indicated by the the key CHECK. If no check_level (chk_lvl) is specified those given by CHECK apply. Its explicit chk_lvl is 0. On the other hand the chk_lvl=1 has only the foreign key requirement. Thus, at this level NULL values for sex will be allowed.

The procedure for specifying and using layered business rules is as follows:

1. determine the number of layers of business rules required in your set of data streams. This means that you should group together classes of records that have similar requirements regarding the business rules. Examples are: nucleus population, multiplier level, production level. Or small breeds version large breeds. Also, combinations are possible. These levels should get entries for documentation purposes in CODES under class CK_LVL.

2. write the basic set of rules in the model file using the key CHECK. The set of rules specified here will be the basic level that is used. Thus, it will be used if either CHK_LVL is set to 0 or not set at all. Then specify for each check level a corresponding CHECKn rule. Thus, if you decided to have three check levels you will have CHECK, CHECK1 and CHECK2 in your model file. While CHECK should be specified as the base set of rules, the other CHECKs are specified only if the base CHECK is not applicable. Thus, whenever a CHECK key exists for a given column corresponding the chk_lvl specified it will replace the base CHECK. Then this set of business rules will be executed. As can be seen in table 3.2 only for column db_sex are the business rules modified in level 1. In db_breed only the base is specified, thus for all other chk_lvl that may be specified only the base set of checks are performed.

3. as has been said above, prior to enforcing the business rules the programmer needs to specify which level she/he wants to fire. This is typically done in the load object by calling the routine: $pdbl-¿Model-¿set_checklevel (1); In this example it would be set to 1. If you specify a check level that does not exists in the model file at least once the calling program should stop.

4. With a number of possible check levels, the current level that was used when the database content was modified (insert or update) needs to be stored with the record in each table. This is done in the column CHK_LVL. This is read and used in the program check_integrity to fire the correct set of business rules.

Table 3.2: Specifying layers of business rules

AMS classes

```
col002 => { DATA => '',
            DB_COLUMN   => 'db_sex',
            DATATYPE    => 'BIGINT',
            LENGTH      => '1',
            DESCRIPTION => 'sex',
            DEFAULT     => '',
            CHECK       => ['ForeignKey codes db_code', 'NotNull'],
            CHECK1      => ['ForeignKey codes db_code'],
            MODIFY      => ['Encode SEX'],
            ERROR       => [],
},
col003 => {
            DATA        => '',
            DB_COLUMN   => 'db_breed',
            DATATYPE    => 'BIGINT',
            LENGTH      => '2',
            DESCRIPTION => 'breed',
            DEFAULT     => '',
            CHECK       => ['NotNull', 'ForeignKey codes db_code'],
            MODIFY      => ['Encode BREED'],
            ERROR       => [],

},
```

Chapter 4

Writing Load Objects

Load objects carry out the database manipulation originating from one record from a data stream. A load object groups together all database manipulations required for a separate selfcontained record from a data stream. As such all the database manipulations either fail (rollback) or succeed (commit). It is important to notice that a LO is completely self contained. Its only connection to the outside world is the model file (which is implicit) and the LO_keys. These are in fact the fields that constitute the data stream under consideration. Being selfcontained and parameterized from the outside the load object can either be called from a batch program or an interactive GUI program.

In case some of the business rules are violated, the transaction is aborted and rolled back (in case some database modifications were already done). The errors are returned in a data structure of its own and leave it processing to the calling program. In batch processing the errors are written into the appropriate database tables, GUIs usually have to shown the errors in the form immediately.

One function of the load objects is the relate error to the fields that held the data. These are usually the keys in the LO_keys. However, when 'encode' is used in the MODIFY part of the model file, this may become quite involved. Consider the database column db_animal. This is the internal animal number after translation from the outside external identifications to the internal. In particular, db_animal is a function of db_unit which in turn is derived from CLASS / EXT_code. This means that db_animal depends on three external identifications. In this case all three may get marked as being the culprit.

4.1 The pseudo SQL code, general syntax

The connection between the source data fields and the checking in the business rules layer (well hidden in the routine meta_db which is called in the load object) is derived from the pseudo SQL code used for database modifications in the LO. Up to now select, insert and update statements could be used with these pseudo SQL syntax. The example in table 4.1 implies an insert of a record in table MYTABLE with the columns animal_nr, weight, and weigh_dt. The values for the three columns are taken from the perl variables $animal_nr, $weight, and $measure_dt. It is a requirement that these source variables are part of the data input hash LO_keys to the LO. Typically, they will come from the outside

world (outside the LO) either from INSPOOL records for batch programs or the GUI fields for GUI programs. These keys are also used in the error hash (for description see 4.3)to collect any errors that may occur during execution of the business rules.

Sometimes data elements are also generated in the LO itself. This may be the case in the processing of birth records in pigs, where individual piglet identifications are generated on the basis of number of piglets born (see LO_DS02 for an example). Here, we would have additional source variables that are not part of the input hash LO_hash. If this variable is to be used in the pseudo SQL code and must be added to the hash LO_keys. Only then can the error processing code the 'culprit' i.e. the source variable responsible for the error (should there be an error).

Table 4.1: the pseudo SQL in the load objects

AMS classes

```
$pseudo_sql[0]= (
       'INSERT into MYTABLE (
                animal_nr,  weight,   weigh_dt
       )VALUES (
               $animal_nr, $weight, $measure_dt)'
);
```

In the following we will discuss the load object LO_DS01.pm with the matching DS01.pm as it is used in batch processing.

4.2 The pseudo SQL code with ENCODING

The APIIS design feature to disconnect the outside coding from it internal representation, while providing flexibility for future changes, does require additional action during database access. This action is the required translation of any external coding into internal codes used throughout the database. There are two ways to deal with this translation. Firstly, in the LO the programmer can retrieve the internal codes by using standard DBI accesses to the database and then use the retrieved internal codes in the pseudo SQL code. While it provides flexibility, it does require error handling which can be avoided by using the second mechanism.

Here, the model file contains in the MODIFY rule the expression ENCODING as described in table 3.2. If this is used, the BR layer must only contain external codes and not internal ones.

4.3 The error hash

The generell structure of the error hash is described in the library Pdbl::Errors. The predefined keywords and structure is shown in Table 4.2.

Table 4.2: Structure of error hash

AMS classes

```
my @type_values = qw/ DATA DB OS PARSE CODE PARAM CONFIG UNKNOWN /;
# Error types:
# DATA the passed data is not ok (usually in CheckRules)
# DB errors from the database (e.g. unique index violation)
# OS errors from the operation system (e.g. full hard disk)
# PARSE errors in ParsePseudoSQL with parsing pseudo SQL code
# CODE programming errors, e.g. from applications like load objects
# PARAM passed parameter is wrong or missing
# CONFIG one of the configuration files is wrong or has missing entries
# UNKNOWN is unknown :^)
my @severity_values = qw/ FATAL NON-FATAL /;
my @action_values = qw/ INSERT UPDATE DELETE SELECT DECODE ENCODE UNKNOWN /;
# structure of error messages:
my %struct; tie %struct, 'Tie::IxHash';
%struct = ( type => undef, # predefined values above
         severity => undef, # predefined values above
           action => undef, # predefined values above
             from => undef, # location where this error comes from (e.g. sub, rule)
        record_id => undef, # id of this record, e.g. record_seq from inspool
             unit => undef, # unit that provides this data
         db_table => undef, # database table concerned
        db_column => undef, # database column concerned
             data => undef, # just handled incorrect data
         ext_cols => undef, # involved external columns (array)
               ds => undef, # data stream name
         err_code => undef, # coded error message
        msg_short => undef, # main error message for end users
         msg_long => undef, # detailed error message
            misc1 => undef, # user defined scalar
            misc2 => undef, # user defined scalar
        misc_arr1 => undef, # user defined array
        misc_arr2 => undef, # user defined array
           );
```

4.4 An example: The Load Object LO_DS01

The simple load object LD_DS01 performs the database actions related with a new insemination record. In this example it needs to do:

1. Insert new record into SERVICE
2. Update last action (la_rep) in ANIMAL

lines 9 – 10: The input parameters are passed from DS01 (or from a GUI_DS01) (via subroutine `Process_LO_Batch`). Some data streams need the reporting unit for their processing, others (like LO_DS01) can simply ignore it.

lines 13 – 14: This defines the names of the external data fields which have to be used by any program that sends data to this load object. (Actually, LO_keys is a sufficient set of information to run the Load Object. It needs to be derived directly from the data streams that should have been described before.)

lines 17 – 18: CheckLO test the accordance of LO_keys with the keys of the %data_hash.

lines 20 – 39: A SQL-like description language is used to pass the parameters to the subsequent processing. This PseudoSQL is described in section .

line 41: A commit to the database is executed to start this transaction with a well defined status.

line 42: All needed parameters (PseudoSQL and the data hash) are passed to the subroutine meta_db() which prepares and executes the database actions.

line 43: Depending of the return status of meta_db() the transaction is committed or rolled back.

line 44: The load object returns either success ($err_status = 0) or an error status ¿ 0 and the accompanied error messages.

4.4.1 Passing parameters with PseudoSQL

To pass the parameters to the subroutine meta_db() for database transactions it was decided to choose a SQL-like syntax. This description language is parsed later to extract the important variables and data. These are the rules for parsing PseudoSQL:

- The Pseudo-SQL-Statements must be surrounded by **single** quotation marks. No substitution of the Perl variables should take place. PseudoSQL is only an artificial language to pass the parameters. Strings inside these statements must be framed by **double** quotation marks.
  Example: `$pseudo_sql[0] = 'INSERT into litter ( db_animal, delivery_dt, comment ) VALUES ( $dam_hb_nr, $delivery_dt, "this is a comment" );`

Table 4.3: LO_DS01.pm

```
 1 ########################################################################
 2 # $Id:  actual_docu.lyx,v 1.37 2003/12/15 13:54:40 eg Exp $
 3 # This is the Load Object for a new insemination record.
 4 # It is responsible to:
 5 #           1.  Insert new record into SERVICE
 6 #           2.  Update last action (la_rep) in ANIMAL
 7 ########################################################################
 8 sub LO_DS01 {
 9    my %data_hash = %{ shift () };
10
11
12    # These are the required LO_keys for DS01.pm:
13    my @LO_keys = qw( dam_hb_nr dam_society dam_breed sire_hb_nr
14      sire_society sire_breed service_dt );
15
16    # some basic checks:
17    my ( $err_status, $err_ref ) = CheckLO( \%data_hash, \@LO_keys );
18    return ( $err_status, $err_ref ) if $err_status;
19
20    my @pseudo_sql;
21    $pseudo_sql[0] = (
22       'INSERT into service (
23           db_animal,
24           db_sire,
25           service_dt,
26           db_service_typ
27       ) VALUES (
28           concat( "society|sex", $dam_society."|2", $dam_hb_nr ),
29           concat( "society|sex", $sire_society."|1", $sire_hb_nr ),
30           $service_dt,
31           "insem")'
32    );
33
34    $pseudo_sql[1] = (
35       'UPDATE animal SET
36         la_rep    = "SERVICE",
37         la_rep_dt = $service_dt
38         where db_animal = concat( "society|sex", $dam_society."|2",
$dam_hb_nr )'
39    );
40
41    $dbh->commit;    # clean start of transaction
42    $hash_ref = meta_db( \@pseudo_sql, \%data_hash );
43    $hash_ref->{err_status} ?  $dbh->rollback :  $dbh->commit;
44    return ( $hash_ref->{err_status}, $hash_ref->{err_ref} );
45 }
46
47 1;
```

- To concatenate values you can use the key word 'concat'. The single elements of this concatenation must be separated by commas. Fixed strings and variables can be mixed. The Perl concatenation operator . can also be used.
  Example: `concat( "society|sex", $dam_society . "|2", $dam_hb_nr )`

- Variables, whose names do not point to an external field of the incoming data stream can be assigned external names in brackets. The names in brackets **must** be surrounded by double quotation marks in total, not every single element.
  Examples: `$piglet["start_notch_no, born_alive_no"] concat("society|sex", $dam_society ."|2", $piglet["start_notch_no, born_alive_no"])`

- Do you use a variable that does not point to any external field you have to supply this variable with empty brackets.
  Example: `$today[]`

- All variables, that you use in the Pseudo-SQL-Statements have to appear as a key (without the dollar character $) in the %data_hash which you pass to meta_db(). Usually this will be the (already existing) field names of the incoming data. You have to add variables which newly appear in the load object like $piglet, $today, $now or also the separately passed external unit $ext_unit. Just have a look into LO_DS02.pm for some examples.

4.5 Calling a Load Object and some other issues

As stated above a load object needs to get its input values (i.e. the data elements defined for that particular data stream) from a calling program. There are two modes how this can be done: input comes from a batch program, i.e. data are read from a file or data are entered via a GUI program. A description of the former is to be found in the inspool section of this document. GUI programs on the other hand can be created automatically by the program mkLOform which may be called as: `mkLOform LO_* '$PDBL_LOCAL/model/apiis.model'` . As output it creates for each matching load object one GUI program. What happens is that mkLOform picks up the LO_key line from the load object which holds all elements defined for the data stream. Then it creates for each of these elements one GUI field. The GUI program can be run right away allowing the user to enter data into the fields and sending them to the load object by pressing the insert button. From there on the load object takes over doing all the database interactions defined under the control of the business rules.

The fields in the GUI program can be moved around the canvas using the FormDesigner to suit the artistic and ergonomic requirements of the user.

Sometimes data need to be derived from individual data. One example is the individual recording of piglet weight at birth from which we may want to derived the total number born, the sum of males and female piglets and the total litter weight at birth (while this implies redundancy we still may want to do it). To generate these sums in a Perl code is very easy, thus doing this in the load object is a piece of cake. However, there is the problem of not knowing in advance how many piglet a litter has. One way of approaching this problem would be to have one record for the sow and birthdate and the one for each piglet. This has a number of problems: firstly, the complete set of information,

i.e. all the records pertaining to one litter constitute a transaction. If we split this up into separate database activities, we cannot roll the already entered data back should a later database action require this. Secondly, one litter plus the piglet information are infact a unit that the person entering the data would like to see as one block. Thus, it would be desirable to have the GUI form start with the block on the sow and then have one line for each piglet. Thus, we would need to have enough space for a litter, 20 lines should be ok. Then after the last piglet has been entered the complete form content will be processed by the load object and do the necessary accumulations on those variables that are not undef. The program mkLOform will then again generate the GUI program. Because it places all fields below each other FormDesigner will have to be used to arrange the fields in a more meaningful way.

This little paragraph reflect the progress made since the last one was written. Because we are essentially lazy people Hartmut has meanwhile rewritten mkLOform into mkLOfForm (make f@PhdThesis, author = , title = , school = , year = , OPTkey = , OPTtype = , OPTaddress = , OPTmonth = , OPTnote = , OPTannote = ormatted forms from load objects) which produces GUI programs on the basis of the LO_Keys from the load objects that does multiple fields per line. This is simply done by arranging the entries in the LO_Keys as is desired in the GUI program. An example is given in table 4.4. The arrangement given will result in a GUI program with 5 lines. The first would be one field with ext_unit, the second line would hold the three fields dam_hb_nr, dam_society and dam_breed. The program is given in figure 4.1.

Table 4.4: Arrangement of keys for the GUI program

AMS classes

```
my @LO_keys = qw (
                  ext_unit
                  dam_hb_nr     dam_society     dam_breed
                  sire_hb_nr    sire_society    sire_breed
                  parity        start_notch_no  delivery_dt
                  born_alive_no weaned_no       weaning_dt
                 );
```

Figure 4.1: GUI program automatically created from LO_DS02.pm

Chapter 5

Initial Loading of Historic Data

Very rarely, a new information system can be set up without any previouly collected data. This implies the necessity to load all previously collected data into the new database before the routine data flow can start reaching the database. Historic data will typically be available from a number of different and often independant source or oven organization. As a consequence, many logical inconsistancies will be found in the data once they are loaded into a target structure that defines a possibly stringent set of business rules. As historic data may come in non predictable way (many files, any format), a procedure was developed which is as far as possible generic and thereby adaptable to any type of animal data recording system. It therefore allows any number of data files to be involved and furthermore, develops a strategy to deal with the potentially large number of errors that will become obvious once loaded into the business rule centered database.

5.1 The General Procedure

Once the structure of the database has been defined it has to be populated with historic data, i.e. information already available in some computer compatible form. As has been described above, all external codes or identifications are translated into internal database codes to allow for reuse of external codes should the meaning of them change at some time.

1. Operate on codes and external identification
   (a) Collect all temporary external codes
   (b) check external codes and determine their target code to go into the database
   (c) determine illegal values and handle them
   (d) identify and handle duplicate identifications
   (e) create input channels for corrected external codes in the database. The external codes can thus also be viewed as foreign keys which need to be available in the database prior to load the data.

2. Operate on data

   (a) load historic data using the channels created in the previous step

   (b) verify the loaded data against the business rules and flag records with errors in the database

3. Wrap up

   (a) rewrite the temporary external animal identification to the external IDs reaching the database in routine data flow

Now we shall describe the steps in a little more detail.

1. Operate on codes and external identification

   In this step only external codes and external identifications are considered, leaving other data on animals like their measurement aside. It is a step which basically collects, edits and loads Foreign Key information into the database. From the programming perspective, this process can be highly generic with little programming required by the adaptor (once the generic code has been written).

   (a) "Collect all external codes": In APIIS we have three groups of external codes:

   - animal identifications; this can be any combinations of data fields that identify an animal uniquely in the historic data set. This is not necessarily the external ID that may be used in later routine data reporting. Its sole purpose is to identifiy individual animals correctyl in the historic data sets. The definition of this temporary external ID can be different from the external IDs used and reported during routine data flow.
   - an address or partner identification
   - classical codes like the numeric representation of a breed or a code for a certain abnormality

   (b) "check external codes": like all data also identifications and codes can be erroneous or inconsistant. As an example the breed "Pietrain" may be coded in some files as "PI" in others as "pi". Not only should the data made consistant but also a procedure must exist to handle records that are identified as being incorrect. The basic principles that we have followed is to provide for each external code or identication the opportunity to translate it into a new target value or as illegal by manual intervention. Then in the following process (like loading of data) any occurence of the original value will be replaced by its translated value.

   (c) "determine illegal values": some identifications will be illegal. For example, unknown sires may be codes as " " or as "999999". These are not legal external identifications, and should thus not be used to create a data channel. Also, if these IDs come up later on in the loading process they should be skipped.

(d) "duplicates": The same ID must not be used for different animals. This can happen when animal IDs are being reused after a number of years. For some records we know that they should appear only once for an animal: a field performance test may be an example. Here we can test for multiple occurences. If duplicates are found then human interaction is required to resolve the problem.

(e) "create input channnels": Once all the target codes have been established by manual human intervention, likewise duplicates and undefined IDs have been dealt with, the external IDs get loaded into the database. This populate the tables TRANSFER , CODES and UNIT.

2. Operate on data

After the "foreign keys" have been collected, edited and loaded in the first block, now actual measurements have to be loaded into the database. The translation tables edited manually in the first step are also used here as filter.

(a) "load historic data": For each file containing historic data a parameter file has to be written. This requires knowledge of the structure of the data files and the database. The definition of the external codes are identical to those in the first step. They are then passed through the translation created by manual intervention. However, the business rule layer is bypassed as time sequence dependant checks cannot be made because data get processed in random order.

(b) "Verify the loaded data": Thus far all database modifications have bypassed the business rule layer of the model file. The business rule set can only be used if data come in correct time sequences which clearly cannot be done when loading historic data. Thus, conflicts with the business rules can be expected. Rows violating the business rules will be flagged "dirty". This allows a selection of correct data during later database operation. Furthermore, "dirty" records can be cleaned up successively later on.

3. Wrap up

In the previous step all historic data has been loaded, verified against the business rules and flagged accordingly. Now the routine data flow can start using the data stream model. But before this can be done, the data channels may have to be translated from the temporary external animal ID to the ID that comes in via the routine data streams. The temporariy external IDs were created for the purpose of uniquely identifying data records in the historic datasets. Thus, after loading, they have served their purpose and may be modified. In the historic dataset, initially there may have been duplicate IDs refering to different animals. By adding the birth year they would have been made unique: 4711 in 1990 and in 1999 would have been identified as 4711|90 and 4711|99. The former animal will not be active any more while data for the latter will still come into the system. However, it will be identified in the data stream only as 4711. Thus, its external number can be changed to 4711. Accordingly, incoming

data to animal 4711 will find an open data channel and be translated to the correct internal database number.

5.2 Description of the loading process for the reference database

As indicated above the reference database contains a script that should carry out all steps to create the actual reference database starting with the initial ASCII files containing the historic data. Prerequisites are:

- set APIIS_LOCAL to $APIIS_HOME/apiis
  (with bash: export APIIS_LOCAL=$APIIS_HOME/apiis)
- add bin/ and lib/ to the search path
  (with bash: export PATH=$PATH:$APIIS_HOME/bin:$APIIS_HOME/lib:...)
- the postmaster has to be up and running (use the -F option to speedup loading; possible -i to accept TCP/IP connections)
- the current user has to have the right to create and destroy a database
- run runall.pl (the total one may take a couple of hours or so depending on the speed of the computer, because the check routines need a lot of time; 7 hours on pIII 256 Mb RAM, until check_integrity nearly 3 hours)
- a log is written as runall.log for the succesfull run and as runall_long.log for statistics in data loading

In the following we shall describe the procedure which is generally applicable to any loading process of historic data (as we hope). However, we shall use the reference database for detail description. Table 5.1 gives a listing of the perlscript. To load the complete reference database no configuration is necessary. However, this script can also be used as the basis for user specific loading. Then the following lines need adaptation:

line 58 for testing purposes it is useful to load only a (consistant) subset of all historic data. The number of records loaded is set via $max_rec. To load all data use a number larger than the number of records in the data file accessed in program collect_ext_id.pl

line 61 etc the array @job holds all the programs that need to run.

Now the specific steps will be described on the basis of the reference database.

1. make a list of all files that hold historic data
2. start with identification of adult animals
   (a) determine the unique external id of the animal. In the reference dataset this is:

Table 5.1: creating the complete reference database (runall.pl)

```
"add_codes.pl",                                          # step 1
"collect_codes1.pl -m $max_rec",                         # step 2
                                                         # step 3 manual edit
"collect_codes2.pl -f codes.chg.cvs",                    # step 4
"collect_job1.pl -m $max_rec",                           # step 5
                                                         # step 6 manual edit
"collect_job2.pl -f job.chg.cvs",                        # step 7
"collect_ext_id.pl -i -m $max_rec",                      # step 8 find all ext
                                                         # and leave UNDEFINED in file
                                                         # step 9 manual edit
"collect_ext_id.pl -d -n hb_not_uniq.txt -m $max_rec", # step 10 find duplicates
                                                         # step 11 manual edit
"collect_ext_id.pl -f dup_animal.chg.cvs -m $max_rec -c ignore_animal.ok", #step 12
"transfer_to_animal.pl",                                 # step 13
"load_data.pl -p herdbook.par -m $max_rec",              # step 14 load the data herdbook
"load_data.pl -p station.par -m $max_rec -f codes.chg.cvs",    # station
"load_data.pl -p field.par -m $max_rec -f codes.chg.cvs",      # field
"load_data.pl -p litter.par -m $max_rec -f codes.chg.cvs",     # litter
"load_data.pl -p address.par -m $max_rec",                     # address
"clear_index.pl",                                        # step 15
"dates_to_transfer_hb.pl -m $max_rec",                   # step 16
"dates_to_transfer_station.pl -m $max_rec",              # step 17
"fill_last_action.pl",                                   # step 18 fill la_rep
"post_initial.pl",                                       # step 19 rewrite ext_animal
                                                         # step 20 manual edit 'codes_unit.chg'
"collect_codes2.pl -f codes_unit.chg.cvs",               # step 21 new codes from post_initial
# "clear_transfer.pl" # step 21a if duplicate animals exist after rewriting
"check_integrity -f ../model/breedprg.model -D -g check.erg", # step 22 (set dirty)
```

i. breed society (there are sires from other breed societys in station test records)

ii. herdbook number

iii. sex

(b) for each file determine the columns to create the EXT_ID on the above basis

3. load all ext_ids in a vector and note unique (u) and multiple (m) with each ext_id:

m: multiple occurences to be expected, these are occurences as parents, or sows with litter records

u: typically, in herdbook files we should have only one record for each ext_id. Thus, if an ext_id (as defined above) occurs more than once we know that we have a mistake.

When developing a new information system on the basis of APIIS normally historic data exists that has been collected by some other system or by a number of different systems. The problems and procedures to integrate all historic data in a new consistent APIIS based system is described in this chapter.

Because initial loading has to deal with data accumulated over time no time dependent business rules can be enforced. This means that during this phase the business rule enforcement system will be switched off. However, this means that after this phase the freshly loaded system has to be verified extensively to match the accuracy requirements stated in the model file.

5.3 Initial Datasets

Here it is assumed that the initial datasets are available in the form of flat ASCII files. These files constitute the complete historic data that are intended to be loaded into the central database. Furthermore, this implies that never more historic data is to be included. Thus, there will be only one initial step to load historic data. After this process has been carried out the newly established database will be the sole repository of the data. New data will afterwards be entered into the database via the time dependent stream entry using the model file.

The files used in the reference database reside in the directory $APIIS_HOME/ref_breedprg/initial and are:

1. herdbook data ..... herdbook.dat
2. field test data ..... field.dat
3. station test data ..... station.dat
4. reproduction data ..... litter.dat
5. address data ..... address.dat

Loading historic data requires the following steps:

1. list all data streams and their content (DS01 – DSnn)
2. normalize the database structure
3. list all files of historic data (in the reference database we have the 5 files given above)
4. create all other foreign keys such as codes and addresses
5. create external identifications for all animals referd to in the historic data files
6. load historic data
7. rewrite TRANSFER for currently used external identifications
8. check integrity against the defined business rules

5.4 Step 1: Adding code

The program add_codes.pl is used to fill the CODES table with the types that are being used in the later loading process (see 5.5). These are codes which are not came from data but from datastream and/or position in data, like type of the weight (station start test or fieldtest...) in table WEIGHT or role (breeder, owner) in table UNIT, where we have distinct columns in data. Entry and exit action are further examples. Here we can insert whole new classes or needed values which will come later in routine datachannels and are not present in historic data.

In the example (figure 5.1) a new code class 'SERVICE_TYPE' would defined with the two external codes for insemination and natural service.

Figure 5.1: Configuration section for add_codes.pl

```
## class        ext_code        [ shortname longname description ]
'SERVICE_TYPE' => {
                 'insem'      => ['insem', 'insemination (fresh semen)' ],
                 'nat'        => ['nat', 'natural (Natursprung)']
                   },
```

5.5 Step 2: Collecting and setting up CODES

The steps 2 until 7 are made before the external animal identifications would be collected, because also different codes from different files (ex. sex from different sources) could be part of the unique animal identification.

This block performed by the programs collect_codes1.pl and collect_codes2.pl which need to get adapted to the specific information system under consideration.

The configuration block in collect_codes1.pl is given in table 5.2. Each line

Figure 5.2: Configuration block for collect_codes1.pl

```
###############################################################################
# Begin configuration section:
###############################################################################
#               file           column           category
my @total = (
             [ 'herdbook.dat' , '0'  ,              'sex' ],
             [ 'herdbook.dat' , '9'  ,            'breed' ],
             [ 'herdbook.dat' , '20' ,              'mhs' ],
             [ 'station.dat'  , '0'  ,              'sex' ],
             [ 'station.dat'  , '2'  ,            'breed' ],
             [ 'station.dat'  , '27' ,  'slaughter_house' ],
             [ 'station.dat'  , '32' , 'slaugter_remarks' ],
             [ 'station.dat'  , '16' ,          'l_cause' ],
             [ 'field.dat'    , '0'  ,              'sex' ],
             [ 'field.dat'    , '4'  ,            'breed' ],
             [ 'field.dat'    , '10' ,              'sex' ],
             [ 'litter.dat'   , '4'  ,            'breed' ],
             [ 'litter.dat'   , '8'  ,            'breed' ],
           );
###############################################################################
# End configuration section:
###############################################################################
```

configures one column with codes. The files are again those that we had listed before. In "herdbook.dat" we have 3 columns with codes. They represent the sex of the animal, the breeds and the mhs (maligne hypthermia) status of the animal. Notice, that we need to specify **all occurences of codes.**

In historic data often many different codes are used for the same object. For instance for male animals we may find M, m and 1. These would have to be translated into one interal code number. A general procedure for loading CODES would follow these steps:

1. **count the number of raw tables (n_{tables})**
2. **for each table determine the columns that hold a code and give each a category name (column class in CODES)**
3. **for each table:**

(a) **column 1 do a select (code), count(code) group by code**

(b) **column 2 do a select (code), count(code) group by code**

4. **this will result in a table like this:**

					Names	
RAW table	**column**	**content**	**n**	**Target Code**	**short**	**long**
herdbook	**sex**	**1**	**231**			
herdbook	**sex**	**01**	**212**			
herdbook	**sex**	**m**	**2121**			
herdbook	**sex**	**f**	**3212**			
...	**...**	**...**	**...**			
litter	**anomaly**	**01**	**2123**			
litter	**anomaly**	**ok**	**216**			
...	**...**	**...**	**...**			
litter	**anomaly**	**99**	**3221**			

5. **write this table to a file**

The program collect_codes1.pl produces as out the ASCII file codes.chg. A few lines from this file are given in figure 5.3. This file needs to get edited by the developer.

Figure 5.3: Output file codes.chg from collect_codes1.pl

```
# seperate columns with blancs
# if you have more than one word in some columns you must insert these between >'<
# for example: breed raw_herdbook 2  02  8724 DL 'German Landrace' ' another kind of Landrace'
# for undefined TARGET-CODE use NULL
# CATEGORY   TABLE  COLUMN CONTENT    NUMBER  TARGET-CODE   SHORTNAME LONGNAME DESCRIPTION
#-------------------------------------------------------------------------
BREED      field.dat    '4'    '10'          1 '10'
BREED      field.dat    '4'    '11'          1 '11'
BREED      field.dat    '4'    '2'           1 '2'
BREED      field.dat    '4'    'DL'     126067 'DL'
BREED      field.dat    '4'    'LW'         27 'LW'
BREED      field.dat    '4'    'PI'       1514 'PI'
BREED      field.dat    '4'    'SH'       2369 'SH'
BREED      herdbook.dat '9'    'DL'      27473 'DL'
BREED      herdbook.dat '9'    'HA'         55 'HA'
BREED      herdbook.dat '9'    'LW'        149 'LW'
BREED      herdbook.dat '9'    'PI'       1384 'PI'
BREED      herdbook.dat '9'    'Pi'          7 'Pi'
```

5.6 Step 3: Manual edit of codes

1. **manually, determine the target external code to be used in the system (this will be done with your favorite editor). This table holds for each category all the external codes allowed, e.g. category SEX may be M, F, C for male, female and castrate. Table CODES would translate SEX/F into an internal code used throughout the database. For unknown codes it is possible to use 'NULL' in column target code. The table will then look like this:**

RAW table	column	content	n	Target Code
herdbook	sex	1	231	M
herdbook	sex	01	212	M
herdbook	sex	m	2121	M
herdbook	sex	f	3212	F
...	...	...	...	...
litter	anomaly	01	2123	1
litter	anomaly	ok	216	1
...	...	...	...	...
litter	anomaly	99	3221	9

Furthermore could be inserted the meaning of the additional columns in table CODES, like short name, long name and description. These meanings are filled only for one target code and could be also later filled from other source. The resultant file with the name codes.chg.cvs (which comes from cvs) is given in the figure 5.4.

Figure 5.4: Edited output file codes.chg

```
1  # seperate columns with blancs
2  # if you have more than one word in some columns you must insert these between >'<
3  # for example: breed raw_herdbook 2  02  8724 DL 'German Landrace' 'another kind of Landrace'
4  # for undefined TARGET-CODE use NULL
5  # CATEGORY   TABLE  COLUMN  CONTENT NUMBER TARGET-CODE SHORTNAME LONGNAME DESCRIPTION
6  #-------------------------------------------------------------------------------
7  BREED      field.dat    '4'    '10'      1 NULL
8  BREED      field.dat    '4'    '11'      1 NULL
9  BREED      field.dat    '4'    '2'       1 NULL
10 BREED      field.dat    '4'    'DE'   1769 'DE' 'DE' 'German Large White' 'another kind of ...'
11 BREED      field.dat    '4'    'DL'    140 'DL' 'DL' 'German Landrace' 'another kind of Landrace'
12 BREED      field.dat    '4'    'LW'     27 'LW' 'LW' 'Large White'
13 BREED      field.dat    '4'    'PI'   2551 'PI' 'PI' 'German Pietrain'
14 BREED      herdbook.dat '9'    'DE'   1403 'DE'
15 BREED      herdbook.dat '9'    'DL'    479 'DL'
16 BREED      herdbook.dat '9'    'HA'     55 'HA' 'HA' 'Hampshire'
17 BREED      herdbook.dat '9'    'SH'      4 'SH' 'SH' 'Schwäbisch Hällische'
18 BREED      herdbook.dat '9'    'LW'    149 'LW'
19 BREED      herdbook.dat '9'    'PI'   3438 'PI'
20 BREED      herdbook.dat '9'    'Pi'      1 'PI'
21 BREED      station.dat  '2'    '1'    3271 'DL'
22 BREED      station.dat  '2'    '2'     983 'PI'
23 BREED      station.dat  '2'    '3'      15 'DS'
```

In this example we have first some unknown codes inside which always only happens once in the ascii file field.dat and let it out from the database by changing the target to 'NULL'. The next lines leave the targets as they are and add here some short and longname and a little description which is needed only once for each target code. At the end here are some changes of target codes from 'Pi' to 'PI' and also for the data source station from numbers to describing characters.

5.7 Step 4: Load edited codes

Run the program collect_codes2.pl which reads the above edited file and loads it into table CODES.

5.8 Step 5: Collect jobs

The general procedure is the same as in collect codes, see above. The differences result from the specific table structure for UNIT, ADDRESS and NAMING.

5.9 Step 6: Manual edit of Jobs

Here you can specify if you want different db-sequences for the same person (unit) in different jobs. These are necessary when the same person can be either breeder or owner. Then you need two different entrys in UNIT but possible only one entry in ADDRESS and/or NAMING. As an example see file job.chg.cvs and figure 5.5.

Figure 5.5: job.chg

```
# seperate columns with blancs
# for undefined TARGET-CODE use NULL
# when  you need distinct sequence for db_name and db_address: see example
# below first 3 jobs get the same db_name but foster breeder get another db_address
# BREEDER   raw_herdbook   hb_zue_nr       208         21  208
# OWNER     raw_herdbook   owner_nr        208         12  208 BREEDER BREEDER
# F_BREEDER raw_herdbook   hb_auf_nr       208          2  208 BREEDER ''
# SOCIETY   raw_herdbook   society_nr      208        102  208
# the foster breeder below get the same db_name and db_address as breeder 208
# F_BREEDER raw_herdbook   hb_auf_nr       209          2  209 BREEDER(208) BREEDER(208)
# ext UNIT     TABLE    COLUMN  CONTENT  NUMBER  TARGET-CODE SAME(db_name) SAME(db_address)
#-------------------------------------------------------------------------------
BREEDER     herdbook.dat 18      57        1040  57
BREEDER     herdbook.dat 18      58         128  58
BREEDER     herdbook.dat 18      59           1  59
BREEDER     herdbook.dat 18      5           37  5
BREEDER     herdbook.dat 18      60           8  60
#-------------------------------------------------------------------------------
FBREEDER  herdbook.dat   19      570        171  570   BREEDER(57) BREEDER(57)
FBREEDER  herdbook.dat   19      57           1  570   BREEDER(57) BREEDER(57)
#-------------------------------------------------------------------------------
OWNER     herdbook.dat   11      0         1116  NULL
OWNER     herdbook.dat   11      10144        2  10144 BREEDER BREEDER
OWNER     herdbook.dat   11      10570        1  10570 BREEDER BREEDER
OWNER     herdbook.dat   11      105         73  105   BREEDER BREEDER
#-------------------------------------------------------------------------------
SOCIETY   herdbook.dat   2       54           2  54
SOCIETY   herdbook.dat   8       54           2  54
SOCIETY   herdbook.dat   2       56          77  56
SOCIETY   herdbook.dat   5       56          61  56
SOCIETY   herdbook.dat   8       56          54  56
```

In the example the breeder get all unique sequences in table ADDRESS and NAMING which will updated with the right informations later.

In the part foster breeder the two members get the same target as '570' and then get the same db_address and db_naming as the breeder with the external identification '57' (see the brackets at the end).

The next lines for the owner give then an empty (unknown) target for the external id '0' and then the other owners get the same sequencies as the identical numbered breeders.

At the end the class society get then unique db_address and db_unit for the identical target-code.

5.10 Step 7: Load jobs

Load all present units into the table UNIT and fill the right sequences to tables NAMING and ADDRESS.

5.11 Step 8: Collecting all external animal identifications

The step of creating external identifications (5) is performed by the program collect_ext_id.pl which resides in $APIIS_LOCAL/initial. Prior to being able to load historic data external identifications in TRANSFER need to be created together with their internal database numbers.

At this stage EXT_ID is defined as a concatenation of fields which is used in the historic dataset to make an identification unique. This may be society|hb_nr|sex in case of the reference. This same string will be used to load history data into the database in the following steps.

This step is done by the program collect_ext_id.pl. The program is parameterized in a configuration block right at its beginning. The example from the reference database is given in figure 5.6.

Figure 5.6: Configuration block of collect_ext_id.pl

```
1  ###############################################################################
2  # Begin configuration section:
3  ###############################################################################
4  #              file             new_nr        old_nr
5  my @link =  ('herdbook.dat', '2 1 0'        '8 7 c2 12',
6                'station.dat', 'cstation 11', '8 9 c2 10');
7
8  #              file              nr      unique / multiple
9  my @total = (
10              [ 'herdbook.dat' , '2 1 0' ,      'u' ],
11              [ 'herdbook.dat' , '8 7 c2 12' ,  'u' ],
12              [ 'herdbook.dat' , '5 4 c1' ,     'm' ],
13              [ 'herdbook.dat' , '8 7 c2' ,     'm' ],
14              [ 'field.dat'    , '2 1 c2 3' ,   'u' ],
15              [ 'field.dat'    , '2 1 c2' ,     'm' ],
16              [ 'station.dat'  , 'cstation 11', 'u' ],
17              [ 'station.dat'  , '8 9 c2 10' ,  'u' ],
18              [ 'litter.dat'   , '2 1 c2' ,     'm' ],
19              [ 'litter.dat'   , '6 7 c1' ,     'm' ],
20            );
21
22 my @regex = (
23              [ '\|0$', 'notch nr 0' ],
24              [ '\|999999\|', 'inside animalnr 999999' ]
25            );
26 ###############################################################################
```

Logically it breaks down into the following blocks:

1. **create LINK: in pigs we have different numbers or IDs in the historic datasets that refer to the same animal. This is usually the ID of the piglets that is kept through testing. Only after selection this animal gets a new number which is used for further reporting. Thus, the two ext_ids must refer to the same**

db_id, which is created as a counter. In this step two hashes are created from each link record, one for each ext_id holding the db_id.
In line 5 those two sets of columns are given that refer to the animal identification as young and as adult (selected) animals. The columns chosen are those that make the IDs unique in the complete data set. The first ID is made up of the columns 2, 1 and 0. This is herdbook society, herdbook number and sex. The second ID is made up of the dam herdbook society (column 8), the dam herdbook number (column 7), the sex of the dam (fix 2) and the running piglet number (column 12).
Line 6 describe the link between mother number + notch number with the internal station number which will refer to the information from this datasource.

2. for each historic dataset each ext_id (created as a concatenation as described above) is read one by one performing the following actions:

   (a) does ext_id exist in LINK(i)? if yes: use its DB_ID
   (b) does EXT_ID exist in TOTAL? if yes: increment COUNT (if UNIQUE print error)
   If it does not exist insert and add DB_ID as an incremented counter

It is configured also in figure 5.6 in lines 10 through 19. Notice the last column consisting of either 'u' or 'm'. This stands for unique or multiple and means if the ID given should be unique in the given file or if it can occur more than once. If the field test, only one record is allowed for each animal, thus, if it occurs more than once we know that there is a problem. Thus, the animal ID as given in line 10 must occur only once. On the other hand, the animals dam as given in line 12 will show up more than once. In this situation nothing can really be checked.

Note, that ALL datafiles that constitute the historic data set must be searched for external IDs and be configured in this section. This clearly includes parents of animals that are tested at the station and in the field (here, we sometimes have sires from different herdbook societies which are not part of the breeding program (i.e. herdbook) under consideration.

There are two problems that need attention at this point:

- unknown or undefined animals need to be identified. The run with the option '-i' create two files which should help you to identify such external animal identifications.
- then for some external IDs we know that they should occur only once. We have indicated this with a 'u' in the @total vector 5.6. If a this rule is violated we need to have means of resolving this problem. This will always require manual interference. In the second step collect_ext_id.pl (with option '-d') produces a file with duplicates with the name of 'dup_animal.chg'.

In the third configuration part (see figure 5.6 at @regexp) we have an aditional block which should help to find unknown or probably incorrect external animal identifications. Here you can describe simple regular expressions that match the wanted identifikations. At the end of this operation two files are generated.

5.12 Step 9: Manual edit

The file 'ignore_animal.chg' (see figure 5.7), craeted with option '-i' on 'collect_ext_id.pl', contain only little statistic which could help to identify external animals where usually used as unknown animals. The idea behind the creating of this file was that animal identifications which are only used as dummy number to create a full pedigree or can insert in the original software only animals which have parents but you don't know them, than the number of occurrence are normaly much higher as the normal use of an animal.

To really ignore further these animals you must write these identifications into the file 'ignore_animals.ok' or use option '-c filename' for this file and leave only animals there which should be ignored. (see file 'ignore_animal.ok' in directory $APIIS_LOCAL/initial for an example) If you need some more informations about the exact number of use the external animal identification in each input file you can use the option '-x' which create a file 'detailed_use' with this information.

Figure 5.7: ignore_animal.chg

```
1  # leave only animals which schould be ignored
2  # and rename to ignored_animals.ok or use the option c
3  # ext_animal count
4  32|400080|1  2563
5  0|9999999|0  2001
6  32|400061|1  1686
7  32|100048|1  1663
8  32|400167|1  1655
9  32|253020|1  1605
10     :
11 0|0|1  591
12 0|0|2  586
13 0|0|2|0  577
```

In this example seems to be that the first animal are an real one and if we see this was an boar and could be came as sire so often. But then we can identify such a number wich is probably not a valid identification of an living animal. unfortunateley in our case we found other invalid IDs later on the file (0|0|1 as unknown sire...).

The second file 'ignore_animal2.chg' cover the identifications from the defined regular expressions and also the animal identifications which has uninitialised values inside the concatenation. An example see in figure 5.8. Now you have to decide which animals are used as unknown or if something wrong with the id. Also these identifikations has to be added into file 'ignore_animal.ok'. mostly it would be better to copy these file to the ignore_animal.ok and add the other identifications found in ignore_animal.chg.

Figure 5.8: ignore_animal2.chg

```
# write the animals which schould be ignored
# to file ignored_animals.ok or use the option c with collect_ext_id.pl
# ext_animal  description of reason
32|999999|1  inside 999999
25|67682|2|0  notch_nr 0
32|104172|2|0  notch_nr 0
32|101338|2|0  snotch_nr 0
  :
||2|  NULL values
||2  NULL values
```

The example show all external identifications which match the given regular expressions in collect_ext_id.pl and additional all concatenated external IDs where not all elements are known.

5.13 Step 10: Find duplicates

Use of 'collect_ext_id.pl' whith option '-d' create a output file with the data of duplicated animals, if defined that these animals schould be unique in the raw datafile. (see file 'dup_animal.chg') To define ignored animals and find duplicates are two steps because animals to be ignored also could be duplicated...

5.14 Step 11: Manual edit of duplicates

In this step you can change the external animal identification in dependence of the whole data for these animal (if you know which external identification is wrong). If the lines are total identically one of the line will be accepted because there are no other information for the animal.

As example see file 'dup_animal.chg.cvs' (few lines are in figure 5.9), where only in the case of station data it is possible to change the identification. All other animals in this file will further be ignored in loading data, because nobody know if the external identification is ok.

Figure 5.9: example for changed duplicates

```
#  change only ext_animal
#  for example 32|400723|2|15 =>  32|400723|2|16
# file         ext_animal      key            line
     :
station.dat = 32|133575|2|57 ( 8 9 c2 10 )  =>
   ...|133575|57|202349|110|12.40|8.9.99 00:00:00||30.40|...
station.dat = 32|133575|2|58 ( 8 9 c2 10 )  =>
   ...|133575|57|202355|110|10.60|8.9.99 00:00:00|30.11.99 00:00:00|A9|28.20|...
```

The example in figure 5.9 show the changing of a notchnumber from 57 to 58 for the animal wich would be slaughtered (this information are on the end of the printed line). this means that the used boar have the correct notchnumber 57. both lines will be represented in the database.

5.15 Step 12: Load external identifications

This step really load the table TRANSFER with all external identifications and create the proper db_animal. Not inserted are the identifications from the file 'ignore_animal.ok' and the not edited animals from file 'dup_animal.chg.cvs'.

After this, the historic data can be loaded as all verified codes and units will be in the represent tables.

5.16 Step 13: write all animals to table animal

The script transfer_to_animal write all animal numbers into table ANIMAL. The reason is to have, after loading all external animals into table TRANSFER, you need to have all existing internal animal numbers also in table ANIMAL. After this step all needed access to this table could only be updates and no inserts anymore. With this step you guarantee also that all animals reside in this table and go down to unknown parents. Later all known parents overwrite this behavior.

5.17 Loading Data

The above steps ensure that all data channels are open and that each external identification is connected to the correct internal database number. The same holds true for all codes that are to be loaded. Also here, we have an external representation and an internal equivalent. In the following steps the tables get filled with information from the various data files. Clearly, these programs require a lot of adaptation, since the structure and content of the datafiles will be very different form one species to the other.

The order of loading the data are verry important which data have the better meaning if the information come in twice from different sources. For example use the sex from the herdbook data and only if there not filled here use other sources.

The configuration sections for each source file are located in a seperated ¡name¿.par file an will be run as *load_data.pl -p ¡name¿.par*.

5.17.1 Step 14: Loading Herdbook Records

As an example the parameter file for the loading program (herdbook) is shown in figures 5.10 until 5.12. The parametrisation require three blocks, first some informations about the included files, second the definition of animal identification (also sire and dam) and the definition of traits, and third the sql-section where defined the actions will taken with the data. All information from the source files are located by the position in it.

In the second section the names ext_animal, ext_sire and ext_dam are mandatory if exist and should have the same information as given in collect_ext_id.pl. Then follow the definition of the traits. The function get_value has in mind if the value is a code or a unit (defined

in collect_codes1.pl respective collect_job1.pl) and/or have to be redefined. The last parameter of the function could be an dateformat which have to splitted in the right order or an check for numeric values or upper / lower case any character (for details see pod from load_data.pl). If is no value on this position then the return value is 'NULL' (in database sense = absence), because also no values are allowed (ex: slaughter findings only some time appear).

The last section with the nativ sql-syntax has three possible parts:

1. **insert,**
2. **update and**
3. **any sql which should be executed only one time after all others.**

The reason for this is to make the inserts much more quickly, if the indices are droped and recreated later after this step. On the other side, for updates it is an advantage if indices exists. The use of the third part is in this example to drop all records from table GENES which have no mhs-status because mhs can also be 'NULL'.

Figure 5.10: first section from herdbook.par

```
1  #startconf
2  ###########################################################################
3  # Begin configuration section:
4  ################################## EDIT -> ##################################
5  $prg_name = 'herdbook';
6  $infile = 'herdbook.dat';
7  $sep = '\|';      # Delimiter/separator for input file (pipe | must be masked!)
8  $chg_file = 'dup_animal.chg.cvs'; # chg_file for duplicated animals
9  $file_cod = 'codes.chg.cvs';      # chg_file for codes or option -f
10 $file_job = 'job.chg.cvs';        # chg_file for jobs or option -j
11 #$file_id = 'keys.chg.cvs';       # chg_file for keys or option -i
12 ################################## EDIT <- ##################################
13 #endconf
```

The first section describe the names of the inputfiles wich should be used and probably the used delimiter for the raw datafile.

In this section we have three blocks:

- **animal / sire / dam identification**
- **value definition and**
- **some further data manipulation (further example see field.par)**

The animal identification have the same informations as described in collect_ext_id.pl. Birth date has in our case an content like '19981207' and wold be described by 'yyyymmdd'. The leading 'n' as option for get_value check the content for the teats count that they are real numbers. Else the content of the field are ignored.

The last line allow to get or manipulate some data. in this example we need the correct internal db_unit for the combination external unit and external id. The reason are that the society would only given if this is not the own one.

Figure 5.11: second section from herdbook.par

```
#startval
################################### EDIT -> ###################################
$ext_animal = $line[2] . '|' . $line[1] . '|' . $line[0];
$ext_sire   = $line[5] . '|' . $line[4] . '|' . '1';
$ext_dam    = $line[8] . '|' . $line[7] . '|' . '2';
$sex        = get_value( 0 , \@line );
$breed      = get_value( 9 , \@line );
$litter     = get_value( 13, \@line );
$teats_l_no = get_value( 16, \@line, 'n' );
$teats_r_no = get_value( 17, \@line, 'n' );
$mhs        = get_value( 20, \@line );
$exterior   = get_value( 23, \@line, 'n' );
$birth_dt   = get_value( 14, \@line, 'yyyymmdd' );
$name       = get_value( 10, \@line );
$breeder    = get_value( 18, \@line );
$f_breeder  = get_value( 19, \@line );
$society    = get_value( 2 , \@line );

$soc_fix = get_db_unit( '32', 'society' ); # where isnull
################################### EDIT <- ###################################
#endval
```

Figure 5.12: third section from herdbook.par

```
#startsql
@sql_insert=();
################################### EDIT -> ###################################
$sql_insert[0]="INSERT INTO genes
      ( db_animal, db_mhs, last_change_dt, last_change_user )
      VALUES ( $db_animal, '$mhs', '$now', '$user')";
$sql_insert[1]="INSERT INTO exterior
      ( db_animal, teats_l_no, teats_r_no, exterior, last_change_dt,
      last_change_user )
      VALUES ( $db_animal, '$teats_l_no', $teats_r_no, $exterior, '$now', '$user' )";
##################
 @sql_update=();
##################
$sql_update[0]="UPDATE animal SET db_sire = $db_sire, db_dam = $db_dam,
     db_sex = $sex, db_breed = $breed, parity = $litter,
     name = '$name', birth_dt = $birth_dt, db_society = $society,
     db_breeder = $breeder, db_foster_breeder = $f_breeder
  WHERE db_animal = $db_animal";
$sql_update[1]="UPDATE animal SET db_sex = $id_code{ 'SEX' . '#'
  . '1'}
  WHERE db_animal = $db_sire AND db_sex isnull";
$sql_update[2]="UPDATE animal SET db_sex = $id_code{ 'SEX' . '#'
  . '2'}
  WHERE db_animal = $db_dam AND db_sex isnull";
##################
@sql_only_one=();               #for use of sql statement which only one time executed
$sql_only_one[0]="DELETE FROM genes WHERE db_mhs isnull";
$sql_only_one[1]="UPDATE animal SET db_society = $soc_fix where db_society isnull";
## you must "'" the values wich are strings or dates ?? but you can do this for all...
################################### EDIT <- ###################################
#endsql
```

The last section describe the native SQL-statements to load the data in the tables. Inserts are made for tables GENES and EXTERIEUR. The variables *nowand*user could be used to fill the mandatory fields last_change_user and last_change_dt.

Remember that the filling of table ANIMAL are always have to be updates after loading table animal via transfer_to_animal.pl. Therefore here we update that table. The next two are examples for conditional updates only if the sex of the given animal not known. Here we have also an example for using codes which are not comming from the data. The string `$id_code{ 'SEX' . '#' . '1'}` give back the db_sex where the coding class are 'SEX' and the value are '1'. This information we have from the use of the animal either as sire or dam.

Last we delete all animals from table genes which not have any information about the mhs status and give to all animals which not have an society information the right one from the own society (which is the default)

The further data loading programs have the same structure. There will only describe some usefull structures which there are different from herdbook.par.

5.17.2 Step 15: Loading Station Records

Here we give an example (figure 5.13) for using 'if loops' to execute some SQL-statements only if the values are defined. Therefore we have also to use the push method to fill the array with the SQL-statements because we could not define the array element number as fix. Remember that the function get_value return 'NULL' (database empty) if the value are not defined. The delete statement in @sql_only_one (compare also herdbook.par) would then not be used here.

Figure 5.13: third section from herdbook.par

```
1 if ( $test_wt ne 'NULL' and $test_dt ne 'NULL' ) {
2   push @sql_insert, "INSERT INTO weight
3     ( db_animal, test_dt, test_wt, db_weight_type, last_change_dt, last_change_user )
4    VALUES
5     ( $db_animal, $buy_dt, $buy_wt,
6      '$id_code{ 'WEIGHT_TYPE' . '#' . 'buy_s'}', '$now', '$user')";
7 }
```

5.17.3 Step 16: Loading Fieldtest Records

A new not necessary parameter block are inserted here which will be run once after database access is created. This could be used if allways inserted informations are needed for the next steps. The example would be shown in figure 5.14. The use of the new informations are used in runing one insert only if some informations not always was inserted until here. The problem in the reference data are, that this information (number of teats) could either came from the herdbook and / or from the field data.

Figure 5.14: block perlrun from parameterfile field.par

```
#startperl
################################### EDIT -> ###################################
# block have to be run once after db access
###############################################################################
  :
  my $sqltext = "SELECT db_animal, teats_l_no FROM exterior";
  my $sql_ref = $apiis->DataBase->sys_sql($sqltext);
  $apiis->check_status;
  while ( my $ret = $sql_ref->handle->fetch ) {
  :
################################### EDIT <- ###################################
#endperl
```

The code in figure 5.14 have to be only valid perl so the cryptic example would not be explained.

Further here we found an example for calculating birth date from given testing date and age of animal (in section values).

5.17.4 Step 17: Loading Litter Records and Step 18: Loading Address Records

nothing special...

5.18 Step 19: Clear Index

This script clear such tables which couldn't create unique indices after loading historic data. This is required because indices are droped when inserting data there and recreated afterwards. The most possible reason are if there are the same information come from different sources.

However there are something wrong with the data. In the reference we get duplicated records with the same service date for the same animal. For example in the output you get the message `'Cannot create unique index. Table contains...'`, then you can specify the table, the unique index and the indexname like in figure 5.15.

Figure 5.15: EDIT section in clear_index.pl

```
################ E D I T ######################################
#                  table ;     unique index     ;   indexname ; where clause
$combination[0] = "service; db_animal, service_dt; uidx_service_2";
# $combination[2] = "weight; db_animal; uidx_weight_9; db_weight_type =
#   (select db_code from codes where ext_code = 'weaning' and class = 'WEIGHT_TYPE')";
################ E N D E  - E D I T ############################
```

The programm delete the further oids after the first one - if equal combinations exists. This means that you can use a hierarchical order which element should be saved if is one.

You can also use it for cleaning with not unique indices via using the where clause. The example means that we want to check if there in table weight more than one weaning weight is included (because only one weaning weight (see weight_type) are possible).

5.19 Houskeeping

So far merely data has been loaded into the correct structure. However, external data channels have been defined for convenience of the loading process but not as external ids as they will reach the database. Furthermore, dates in transfer will have to be set.

5.19.1 Step 20: Complete transfer with dates from Herdbook

Complete table TRANSFER with information from file 'herdbook.dat'. Create proper opening_dt, entry_action... for all animals in TRANSFER. This step is in the reference db a little difficult, because you have to regard a lot of dates in 'herdbook.dat' which can be filled in every combination or not. Concrete the following information have to go to TRANSFER:

1. **breeder**
2. **owner**
3. **foster breeder**
4. **birth date**
5. **registration date**
6. **leave date**
7. **buy date**

Therefor you must describe the sequence and the dependencies for each possible combination manually. For example opening date: first birth date if exists then buy date if come from other society, then registration ... Also this step require inserts in table TRANSFER because also buying, renaming (registration) and so on should be reflected here. :-(

5.19.2 Step 21: Complete transfer with dates from Station

Same procedure as in section 5.19.1 for informations comming from station.dat but rather simpler. :-)

5.19.3 Step 22: Fill last action

Fill the last reproduction actions in table ANIMAL if possible. Get simply the ordered actions from the involved tables (service, litter, animal) and fill the newest one to la_rep.

5.19.4 Step 23: Rewriting TRANSFER

In the previous step EXT_ID was defined to make animal ids consistent over the complete range of years in the historic dataset. They are often a concatenation of various kinds. Now that all historic data have been loaded we need to rewrite the EXT_ID and UNIT to a format that matches the incoming routine data streams. If there is not an living database required you have also to rewrite the db_unit but you can skip the steps for splitting the concatination.

The program post_initial.pl does the job under consideration. There are much specific code, because its verrry different which ids are needed for incomming datastreams. Also the file codes_unit.chg will be created for the new mandatory units to make the animal ids unique in each reporting unit. The program fill the related tables NAMING and ADDRESS with the correct sequences.

In general there are three parts to edit in the script:

1. **some global informations, the requirements of split (if use APIIS not for further incomming data) and the unit informations for the base parents.**
2. **create an sql-select which animals have to be rewritten. Here it is possible to let some IDs uncahanged and if required drop the unique index of table TRANSFER so far (then have a look at clear_transfer.pl).**
3. **a loop where desribe the roles under which circumstances what should happen with the external identification. Mainly here describe the new external animal ID and the parts of unit.**

For configuration block three in figure 5.16 we have the concret example from post_initial.pl. In this example we have the three different numbering systems: station numbers and herdbook numbers either with or without notch number.

The roles for spliting are:

1. **the word 'station are inside the concatenated number**
2. **else after splitting the concatenated number we have**
   (a) **three data elements which means a notch number is added or**
   (b) **all the rest. Then we have less than three elements an this mean it is an real herdbooknumber together with sex and society.**

The target numbers are than:

1. **station numbers**
   (a) **incoming animal number are the station number (position 1)**
   (b) **unit are 's57' as ext_id and ext_unit is 'station32'**

Figure 5.16: third EDIT section on post_initial.pl

```
1   ########### EDIT ##################################################
2   # split here:  EDIT  this loop
3   # 32|0815|2|16 => ext_animal 0815|16 (1 3|1) ext_unit 32|2 (0 2)
4   # $line represents the concatenated animal identification
5   # specific for each case!!
6   my @lline = split ( /\|/, $line[0] ); # split the elements
7   # station numbers
8   if ( $lline[0] =~ /station/ ) { # station|56789
9     #  ea == ext_animal, ei == ext_id, eu == ext_unit
10    my $ea = $lline[1];
11    my $ei = 's57';
12    my $eu = 'station32';
13    $new_unit{ $eu . $ext_sep . $ei } ++;
14    $new_id{ $line[0] } = [$ea, $ei, $eu];
15  # herdbooknumbers
16  # three elements means with notch number
17  } elsif ( $#lline == 3 ) { # 32|0815|2|16
18    my $ea = $lline[1] . '|' . $lline[3];
19    my $ei = $lline[0] . '|' . $lline[2];
20    my $eu = 'society|sex';
21    $new_unit{ $eu . $ext_sep . $ei } ++;
22    $new_id{ $line[0] } = [$ea, $ei, $eu];
23  # here without notch number
24  } else { # 32|0815|2
25    my $ea = $lline[1];
26    my $ei = $lline[0] . '|' . $lline[2];
27    my $eu = 'society|sex';
28    $new_unit{ $eu . $ext_sep . $ei } ++;
29    $new_id{ $line[0] } = [$ea, $ei, $eu];
30  }
31 }
32 ########### END EDIT ################################################
```

2. mothernumber plus notch

   (a) incoming animal are mothernumber plus (|) notch number

   (b) unit are society plus sex (position 0 and 2 from the splitted data) and ext_unit are 'society|sex'

3. real herdbooknumber

   (a) incoming animal are herdbook number

   (b) unit are society plus sex (position 0 and 2 from the splitted data) and ext_unit are 'society|sex'

The differences between the last two splits are than only the number of elements and an existing pipe sign inside the external identification. This information also could be later used to return the right external id to the right people (real herdbooknumber if exist else mothernumber plus notch or station numbers if the station have an request...).

5.19.5 Step 24: Manual edit

These step add the meaning (shortcode, long code, description) of the new units to table CODES like described in section 5.6.

5.19.6 Step 25: Insert new codes from post_initial

Fill in the new codes really.

5.19.7 Step 25a: Close some datachannels

Close some datachannels if there are now duplicated combinations for ext_animal and db_unit. For example if the split of an on farm used number has deleted the birth year and would be reused than, now we have the same external number more than one time and this require a close of the older chanel. For this purpose the script clear_transfer.pl could be used. The configuration section are described in figure 5.17.

Figure 5.17: configuration section from script clear_transfer.pl

```
1 ################ E D I T ########################################
2 # where is the used date_information
3 #                table , date
4 $used_date[0] = "animal, birth_dt";
5 ################ E N D E  - E D I T ############################
```

This script close all datachannels with more than one ext_animal db_unit combination in depense of one date. Here would be used the birth date from table animal.

5.20 Consistency checking against the business rules

So far, data has been loaded outside the business rules.

5.20.1 Checking the database content against the business rules

During the loading process the APIIS busines rule system has not been in effect. Thus, data will have been loaded that does not comply with the business rules specified in the model file. The program check_integrity does this job.

To this effect each table is accessed in turn then all rows are read one by one and validated against the model file business rules. Errors are logged in an ascii file with the name of the table used as file name and ".err" appended.

One time you must run this programm to fill the mandatory column 'dirty' in each table. Set 'true' if one busines rule is violated.

5.20.2 Treatment of errors

It should be the ultimate objective to have no data in the database that does not comply to the business rules. However, for practical reason, this status may be unachievable for a number of records. One reason will be that the original records are no longer available

for correction. On the other hand deleting a record with only some parts being incorrect may also not be an option – other information will be lost. Thus, if the corresponding flag has been set the status column "dirty" will be set to error for those records not complying with the business rules. These records can then be skipped on later processing. Furthermore, at a later stage after loading of historic data the database can be clean sucessively. The type of error can be regenerated by "verify_integrity" for individual records.

5.20.3 Fixing errors

Example:

- **check_integrity -f model -o 'table date from'**
  (check_integrity -f ../model/apiis.model -o 'animal birth_dt 1998-01-01')
- **edit file 'animal.birth_dt_since_1998-01-01.errors' (see '~.cvs' and add in the new column the correct values)**
- **update_from_error_file.pl -f animal.birth_dt_since_1998-01-01.errors**

Chapter 6

The INSPOOL System

This chapter describes the agreed setup of the INSPOOL system, i.e. that part of APIIS which handles the batch inflow of data. The system described will refer to the reference database where $APIIS_LOCAL must point to. This is usually (but not necessarily) located in the $APIIS_HOME/apiis directory.

The design goal of this batch interface to the database is:

- **fully automatic operation**
- **triggered from incoming data (ftp, e-mail, etc)**
- **scaling with the amount of data coming in**

6.1 The File System Structure of the INSPOOL System

Batch data will arrive at the central computer by some means that we shall discuss elsewhere. This may be via e-mail, ftp or manual copying.

6.1.1 Conventions for incoming data files

All data files will be copied to one location in the file system of the database machine which we shall call INSPOOL_DIR. The only requirement regarding file names is that no files are overwritten (preferably, the write flag should not be set on those files. Using the date and time at the arrival of the file may be a good idea. They would then look like:

AMS classes

```
--INSPOOL_DIR/
|             ds.1999.12.24-12:32:11
|             ds.1999.12.13-00:12:01
|             ds.1999.11.23-09:32:02
--INSPOOL_DIR/done/
|                   ds.1999.19.32_14:11:21
```

Data files which have been processed i.e. loaded into the database INSPOOL table are moved to the subdirectory done/.

The current naming convention for data files is: file names should start with "DS" followed by digits, e.g. "DS01", "DS12" , "DS102".

One special case is data file used for loading large binary data - like pictures, movies, scanned documents etc. This datastream consist of one file containing the file names of the pictures and the real picture files. In this case all files should be placed in a separate subfolder of the INSPOOL_DIR.

6.1.2 The INSPOOL Buffer of the database

All data coming into the database will be stored in the INSPOOL section of the database. This is the only place which will store data without consistency checks.Its purpose is to be the initial repository of all incoming data from which it will then be processed and loaded into the database properly under the constraints of the business rules in the model file. Records that pass this test and get loaded successfully will be flagged accordingly in the INSPOOL and skipped the next time it is processed.

6.2 The Database Structure of the INSPOOL System

The database structure connected to the INSPOOL system is given in table 6.1. It is manadatory and should not differ between systems, i.e. it should be identical across species.

6.3 Loading the files into INSPOOL

Depending on their type - binary or ASCII files are handled differently.

6.3.1 Loading ASCII files into INSPOOL

As stated above all incoming data files will arrive in the directory INSPOOL_DIR. The program `file2inspool.pl` will load each data file into the table INSPOOL. It is identical for ASCII input datafile. This can only work if the header file structure is identical across the data streams. The first two records need to give the datastream identifier (e.g. DS02) and the reporting (external) unit (e.g. 4711). The records that follow need to be selfcontained, i.e. each record needs to contain the complete set of information like date of testing, herd. Thus, the format of these files need to be accommodated at the time when they get written. This would be for instance when new records get extracted from a sow management package.

Table 6.1: INSPOOL Structure

```
CREATE TABLE inspool (
   ds                text,  -- datastream (dataset) name
   record_seq        int4,  -- unique ID of record(sequence)
   in_date           date,  -- Time stamp for initial entry
   ext_unit          int4,  -- Reporting Unit
   proc_dt           date,  -- time stamp for processing
   status            text,  -- Status column
   record            text,  -- the data record
   last_change_dt    date,  -- Date of last change, automatic timestamp
   last_change_user  text   -- User who did the last change
);
CREATE UNIQUE INDEX uidx_inspool_1 ON inspool ( record_seq );



CREATE SEQUENCE seq_inspool__record_seq;

CREATE TABLE inspool_err (
   record_seq        int4,       -- unique ID of record
   err_type          text,       -- Error type ( DB OS DATA...)
   action            text,       -- Error action
   dbtable           text,       -- Error point to table
   dbcol             text,       -- Error point to column (inside table)
   err_source        text,       -- Location where error occurred
   short_msg         text,       -- Error short message
   long_msg          text,       -- Error long message
   ext_col           text,       -- which external cols are involved
   ext_val           text,       -- external (incoming) value
   mod_val           text,       -- modified value
   comp_val          text,       -- compare values (2 in case of 1a)
   target_col        text,       -- Main/primary column of this record
   ds                text,       -- data stream
   ext_unit          text,       -- external unit
   status            text,       -- Active of historic?
   err_dt            timestamp,  -- timestamp for setting status
   last_change_dt    timestamp,  -- Timestamp of last change
   last_change_user  text,       -- Who did the last change
);
CREATE  INDEX idx_inspool_err_1 ON inspool_err ( record_seq );


CREATE TABLE load_stat (
   ds                text,       -- Program name
   job_start         timestamp,  -- timestamp start of job
   job_end           timestamp,  -- timestamp end of job
   status            int4,       -- completion code
   rec_tot_no        int4,       -- Number of Records processed
   rec_err_no        int4,       -- Number of erroneous records
   nrec_ok_no        int4,       -- Number of correct records - inserted
   last_change_dt    date,       -- Date of last change,automatic timestamp
   last_change_user  text        -- User who did the last change
);

CREATE TABLE blobs (
   guid              int4,      -- global identifier
   blob_id           int4,      -- number of blob
   blob              bytea,     -- binary large objects
   filename          text,      -- file name
   last_change_dt    date,      -- Date of last change,automatic timestamp
   last_change_user  text       -- User who did the last change
   owner             text,      -- record class
   version           int4       -- version
);
CREATE UNIQUE INDEX uidx_blobs_rowid ON blobs ( oid );


CREATE SEQUENCE seq_blobs__blob_id;
```

Table 6.2: Data file header

AMS classes

```
DS03
ini
blobs  1  3
cat|/home/zgr/duchev/pictures/IN00006A.JPG|123.56|/home/zgr/duchev/pictures/IN00009A.JPG|jpg

dog|/home/zgr/duchev/pictures/IN00004A.JPG|87.10|/home/zgr/duchev/pictures/IN00005A.JPG|jpg
```

6.3.2 Loading Binary files into INSPOOL

In this case there is one ASCII file containing the names of the binary files. The structure of this file is similar to the one of a normal datafile: the first two records contain datastream identifier (e.g. DS15) and the reporting (external) unit (e.g. farm32). The difference is in the third record - it has to start with the reserved word "blobs" followed by a set of numbers - the positions which have to be resolved as file names for the binary files. One example is shown in table 6.2. The file2inspool.pl should be executed with option -f ¡folder_name¿, where "folder_name" is the name of the subfolder containig the files. The program will read all binary files, place them in the BLOBS table and replace the file names with the returned blob_id pointers. Then the datafile is automatically loaded as a normal ASCII file in the INSPOOL table.

6.4 Batch Loading From INSPOOL

6.4.1 The driver program

The reference database contains a running example of the batch programs for loading data streams into the database. The main driver program is called "load_db_from_INSPOOL" and resides in $APIIS_HOME/bin.

"load_db_from_INSPOOL" is a program that can be run at any time. It only needs the name of the model file and the names of the data streams it should process. Just run "load_db_from_INSPOOL -h" to get the right syntax.

If new data are present in INSPOOL those records will be processed, if none exist, nothing much will happen. Thus, typically, this program will be started at certain time intervals (e.g. every 30 minutes) as a cron job.

6.4.2 The DS–routines

There is one DS subroutine for each data stream that finds its way into the INSPOOL table. Thus, if we have 12 data streams for which we get electronic data coming into the system, we need to have 12 subroutines in $APIIS_LOCAL/lib (i.e. DS01.pm, DS02.pm. . . DS12.pm (the names can be chosen differently, however it seems useful to stick

with this scheme)). The objective for each of these programs is as follows:

1. read the NEW records from INSPOOL pertaining to the DS under consideration. Thus, DS02 will read the NEW records for data stream 2.
2. split the record from INSPOOL according to its format and move them to variables.
3. create a hash to be passed to the corresponding load object. Thus, for each DSnn there will be a LO_nn. The latter executes the actual database modifications like inserts, updates, deletes.
4. the status of the INSPOOL records will be set to OK or ERR depending on the return status of LO_nn. This is done in the subroutine Process_LO_Batch().
5. in case of an error a record will be inserted into INSPOOL_ERR with all available information.

Now let us go through the code of DS01.pm (Table 6.3 on page 54):

line 6: The model file is the only parameter passed from load_db_from_INSPOOL

lines 8 – 10: A hash for storing several data is created and the name of this data stream is inserted.

line 12: This is only a label to leave the processing of this data stream prematurely in case of severe errors.

line 13: Some common tasks (setting up some counters, preparing database handles for data retrieving and statistic/error reporting) have been moved into this subroutine.

lines 15/16: Every new record record of this data stream is now handled separately.

lines 17 – 19: The important parts of the INSPOOL record are assigned to variables and the hash `%ds_conf` stores some more parameters.

lines 22 – 24: You have to know the structure of the data! In this case it is in fixed format, often you also find delimiter separated columns where you have to `split()` on the delimiter. The columns are assigned to the array `@data`.

line 27: Some information is added to `%ds_conf` to better associate errors to the responsible incoming data column, e.g. piglets to the litter sow data.

lines 35 – 37: Basic checks are exported to CheckDS(), currently only if the number of data columns coincides with the number of LO_keys.

Table 6.3: DS01.pm

```
1 ######################################################################
2 #  DS01.pm reads the Insemination records from INSPOOL;
3 #  $Id:  actual_docu.lyx,v 1.37 2003/12/15 13:54:40 eg Exp $
4 ######################################################################
5 sub DS01 {
6   my $model_file = shift;
7
8   my %ds_conf;
9   $ds_conf{ds}         = 'DS01';
10  $ds_conf{all_errors} = [];
11
12   DS_EXIT: {    # exit label for premature leaving in case of errors:
13      DS_PreHandling( \%ds_conf );
14
15      RECORD:
16      while ( my $data_ref = $ds_conf{sth_ds}->fetch ) {
17         my ( $record_seq, $ext_unit, $record ) = @$data_ref;
18         $ds_conf{ext_unit}   = $ext_unit;
19         $ds_conf{record_seq} = $record_seq;
20         my ( $err_status, $err_ref );
21
22         # we need to know the data structure of the record:
23         my $struct = "A20A20A20A20A20A20A20";
24         my @data   = unpack $struct, $record;
25
26         # this is text and data that shows up in the error report:
27         $ds_conf{target_col} = "sow:  $data[1] $data[0]";
28
29         # order of the incoming data, specified in LO_DS01.pm:
30         my @LO_keys = qw( dam_hb_nr dam_society dam_breed
31                           sire_hb_nr sire_society sire_breed
32                           service_dt );
33
34         # error checking:
35         ( $err_status, $err_ref ) = CheckDS( \@data, \@LO_keys );
36         push @{ $ds_conf{all_errors} }, @{$err_ref} if $err_status;
37         last DS_EXIT if $err_status;
38
39         ### some data manipulation:
40         # remove whitespace:
41         @data = map { s/^\s*//; s/\s*$//; $_ } @data;
42
43         # dam_society and sire_society have the value 32 if they are not
44         # defined or '00':
45         $data[1] = '32' if ( !$data[1] or $data[1] eq '00' );
46         $data[4] = '32' if ( !$data[4] or $data[4] eq '00' );
47
48         # reformat service date and use LocalToRawDate( 'EU', $date )
49         # getdate() does this job.  It resides in apiis_alib.pm.
50         ( $data[6], $err_status, $err_ref ) = getdate( $data[6] )
51           if defined $data[6];
52         if ($err_status) {
53            push @{ $ds_conf{all_errors} }, @{$err_ref};
54            next RECORD;
55         }
56
57         # now data elements are ready to be sent to the LO
58         $ds_conf{data}    = \@data;
59         $ds_conf{LO_keys} = \@LO_keys;
60
61         ######### this calls the LO and does the post processing of errors:
62         Process_LO_Batch( \%ds_conf );
63
64         print "Finishing data loop ...\n" if $debug > 5;
65         last RECORD if $debug > 5;
66      }    # record loop
67   }    # DS_EXIT label
68   DS_PostHandling( \%ds_conf );
69 }
70 ######################################################################
71 1;
```

lines 41 – 55: Parts of the data are prepared for further processing, e.g. leading and trailing blanks are removed, some columns values are changed according to the values of other columns and the date is converted to a predefined format. Errors are caught and processed.

lines 58 – 59: The prepared data and the LO_keys are pushed onto `%ds_conf`**.**

line 62: Process_LO_Batch get `%ds_conf` **passed. It creates the input hash for the LoadObject, calls the LO, and does the post processing of the errors (INSPOOL_ERR) . These tasks are hidden in a subroutine as no user interaction is needed here.**

line 68: DS_PostHandling does also some error handling and writes the counters into table LOAD_STAT.

6.4.3 The Load Objects

For a description of the load objects see the corresponding chapter.

cant seem to get the cross-refernce in.

6.5 Reporting

In this paragraph we will deal with reporting on LOAD_STAT and INSPOOL_ERR

6.5.1 Load_stat

6.5.2 Inspool_err

6.6 Error Correcting

Here we describe the correction facilities of INSPOOL.

6.6.1 GUI Interface to INSPOOL

Chapter 7

Access Rights

7.1 Implementation for the programs

7.1.1 Log-in to the APIIS system as a normal user

When user wants to log-in to the APIIS system (via apiis sheell or WWW) he has to use login name and password which are specified for his PostgreSQL account. If his login and password are proper then user object is created. This object is kept to close user session and contains all user data which are stored in the database (table users). After this when user object is successfully created then system user object is created. This object keeps meta_user password which is needed for the meta_user connection (we have to remember that all actions on the database are executed by the meta_user). Method which returns this password is internal and can not be call by the normal user. This password is taken to the object from the special passwd file where all passwords are kept as a coded string. There is a special method which encode this password for the object.

7.1.2 Log-in to the APIIS system as a root

7.2 Implementation for the database and the content of the database

7.2.1 Defining users

7.2.1.1 Adding new users

All information about users is stored in the database. The process of adding new user consists of the following steps:

- **creating new user in the APIIS system,**
- **creating new user in the PostgreSQL database,**
- **creating schema for the new user.**

There is a special Perl script which is used for adding new users - *access_control.pl* and it go through all these steps. This script has to be run with *-u* parameter and user name.

access_control.pl -p [project_name] -u [login name]

All access rights are revoked from the user. User has only access rights to views which are created in his schema (paragraph 7.2.3). Even if user logs-in directly via *psql* command, he can't create new tables or make any modifications on existing tables. User is able to make some modifications only if the administrator give him special roles (paragraph 7.2.2).

7.2.1.2 Removing users

Removing users from the system is also handled by *access_control.pl* script but with another paramater.

access_contol.pl -p [project_name] -d user -u [user name]

7.2.1.3 Presenting information about users

With *access_control.pl* script you can also see information about all users which are currently defined in the system. You have run this scripts with *-s* parameter.

access_control.pl -p [project_name] -s users

7.2.1.4 Log-in to the database

Direct connection to the database from shell via psql command is possible and has to be executed in the following way:

psql [database name] -U [user name]

You can only see there your system of views.

7.2.2 Defining roles and policies

7.2.2.1 Defining new roles

Information about roles and policies is stored in the database. Initially roles and policies are defined in the Roles.conf file and then are loaded into the database from this file. Following example show structure of this file and how roles should be defined:

```
[TEST]
SHORT_NAME = test role
LONG_NAME=  test role
DESCRIPTION = you can make insert and update on table transfer
ROLE_TYPE=DB (can be only DB or OS)
POLICIES=1,2

[OS_POLICIES]
```

```
1=runall.pl|program
2=enter data|www
3=new breeds in year 2004|report

[DB_POLICIES]
1=insert|transfer|oid,db_animal,ext_animal,db_unit,db_farm,synch|PL
2=update|transfer|oid,db_animal,ext_animal,db_unit,db_farm,synch|PL
```

Roles.conf consists of role sections (we can have more than one diferent definition of role in this file) and two policy sections. The policy sections are used through the roles. If role is OS (Operating System) type then the OS_POLICIES section is used else if role is DB (Database) type then the DB_POLICIES section is used.
OS policy definition consists of some action name and the category. This action can be a program name, report name, from name, subroutine name or some interface action. There are 5 categories defined now: program, form, report, subroutine, www, action. Categories are hardcoded now. If you want to add the new category, you have to add it to the AccessControl.pm module (lib/Apiis/Auth/ - line 484). DB policy is a definition of the one action on the one table in the one class (we can not have two definitions of the policies for the same action and the same table, and the same class, but with different column definitions). Each policy has to has unique number and following format:

```
|action|table|column1,column2,column3 ,...|class
```

The process of adding new role is the same for the OS roles and DB roles. It consists of the following steps:

- **defining role and policies in the Roles.conf file,**
- **adding new role into the database,**
- **adding policies into the database,**
- **assigning policies to role.**

First of these steps has to be done manually (as it was described above) and next are made automatically.

access_control.pl -p [project_name] -r [role name]

The role name has to be the same as the one in the square brackets, in Roles.conf file. In our example we have to execute the following command:

access_control.pl -p [efabis] -r test

7.2.2.2 Removing roles

Removing roles from the system is made by the following command:

access_control.pl -p [project_name] -d role -r [role name]

This command also removes all references to the role. If some user has ascribed role which is removed than this role is also revoked from him.

7.2.2.3 Granting roles to the users

Each role can be grant to the user.

access_control.pl -p [project_name] -r [role name] -u [user name]

If we execute this command with role name or user name which is currently not existing in the system then the script creates this role or user automatically.

7.2.2.4 Revoking roles from the users

Each role can be also revoked from the user.

access_control.pl -p [project_name] -d revoke -r [role name] -u [user name]

7.2.2.5 Presenting information about roles

With access_control.pl script you can also see information about all roles which are currently defined in the system. You have to run this scripts with -s parameter.

access_control.pl -p [project_name] -s roles

7.2.3 User views

All user views are created in his private schema on basis his access rights. Process creating user views consists of the following steps:

- **defining roles and policies (for select) in the Roles.conf file,**
- **granting roles to the user (paragraph 7.2.2.3),**
- **creating views in the user schema.**

All views are also created via access_control.pl script. This script reads access rights from the user access view and creates views for tables on which user can execute select statements.

access_control.pl -p [project_name] -v [login name]

Chapter 8

Synchronization of Database Content

8.1 Definition of Terms

Before presenting the basic principles of synchronization of animal biodiversity databases some terms need to be defined.

- **The smallest synchronization item will be referred as "data element".**
- **Each database that is part of the global information system will be called "node".**
- **Each node that provides new data element to the network is "owner" of this data element.**
- **"Representative" of the owner is a node that can distribute owners data. The representative has to have one to one relationship to owner or to other representative of this owner. Representative can be only node that needs this data, therefore there will be no nodes that serve as buffers or transporters for data.**
- **"Source" - any node that distributes data elements to other nodes is "source" for this data elements for that nodes**
- **"Target" is the set of nodes to which one source distribute a data element**
- **"Network manager" - the management authority that will route the traffic of information, preventing conflicts or inconsistencies**

8.2 Synchronization Requirements

We define the following list of features and constraints that we consider appropriate for the intended use in EFABIS or APIIS:

1. **Each data element has one and only one owner. Only the owner can modify the data element.**
   In animal breeding information systems, data is usually collected on different places - like AI stations, farms, research institutes. The common of all these sources of information is that they keep copy of data, there is somebody (human or organisation) that is officially responsible for the quality of data and all users of this data rely on its representative value. Therefore the definition of the term "owner" will be expanded as "member of the Information System who is presenting data to the information space and is responsible for its accuracy and up-to-date status". For example each national node in EFABIS that presents its country data to the European(EAAP) and global (FAO) node is owner of this data and is responsible for the data quality.

2. **Each data element has a "distribution target" or just "target". In general terms, the data collection process does not end in itself. Usually the collected data is intended to be used by someone and in most cases the data users are clearly defined. For example data collected on testing stations is sent to research institute for calculating the breeding values and the results are returned to the farmers.Or in EFABIS network each European country will send data to EAAP and EAAP will distribute this data to FAO. Therefore for each data element there is a well defined target group of consumers. This list of users is actually covered by the "distribution target". This list can consist of one or more addresses of nodes, these are the nodes to which this data item will be distributed. If the distribution target is an empty list then this element is only for private use and will not be distributed.**

3. **Distribution target may be freely changed if this will not produce inconsistencies.**
   This principle ensures that owner can freely choose target nodes unless this will disturb normal floating of data in the system. All actions have to be coordinated by the Network manager described in principle 7.

4. **Synchronizations cannot be refused.**
   When a session is started it automatically synchronizes all of the target content, thus do not allowing the user to refuse receiving of changes. This principle may look very restricting and leaving no choice to the users but it is not the case. The idea behind is that user who wants to have this data element is accepting by default all changes made by owner relying on the fact that all owner changes are representative. For example if owner deletes one data element then this element should be deleted everywhere. This is the situation in EFABIS where only the country can edit its data.

5. **All information to be transfered between nodes is considered element of the synchronization process.**

This principle states that we will not only synchronize data fields in the database containing quantitative values like size, milk, wool length, but also documents and multimedia data. This may look obvious, but it is important for the type and quantity of the data that will be transmitted.

6. **For one data element only one source is allowed.**
   This source could be any node that has (and needs) this data element.
   If the user node can establish connection to more than one representative then the user can choose in cooperation with the Network manager which one will be used as a source and also move from one source to another, but cannot use two sources simultaneously. This restriction will define clear status of the node - it is the status of the representative this node is using as a source for synchronization.

7. **The network is regulated by one authority -"Network manager".**
   To prevent conflicts and inconsistencies there should be one management authority in the whole network - the "Network manager". It will regulate data flow, prevent actions that are against the system consistence or resolve data exchange conflicts between the nodes. For example the Network manager will set the source/target information in cooperation with owner and the nodes that want to exchange some data elements. The need of such authority is clearly view by the following example: Let the node A target one of its data elements to node B and node B target this data element to node C. In this situation if node A wants to change the target of the same data element to node C then node B will loose its source. There are two possible solutions of this conflict - not allowing node A to change the target, or allowing the change, but also setting in node C the target to node B. Such a decision can be only made by authority that has a higher priority than the nodes, and this is the Network manager.

8. **A mechanism to be developed to ensure that the above rules are kept and doesn't produce inconsistencies**
   For the normal work of the Network manager it will need an automated mechanism for keeping track of the existing routes. It will look for possible route for each two nodes that need to exchange information and will also prevent destruction of existing route in case of changes made according to principles 3 or 6. Actually, this mechanism will serve as a basic information for the management authority mentioned in principle 3 for taking decisions about changes in source or target nodes.

As a result of this principles each data element(DE) route is described by its owner, by the source - the node that has supplied this element - and by the target, i.e. the list of addresses that this element will be delivered to. Using symbols this can be shortly written DE[O,S,T]

Figure 8.1: Data element information part example

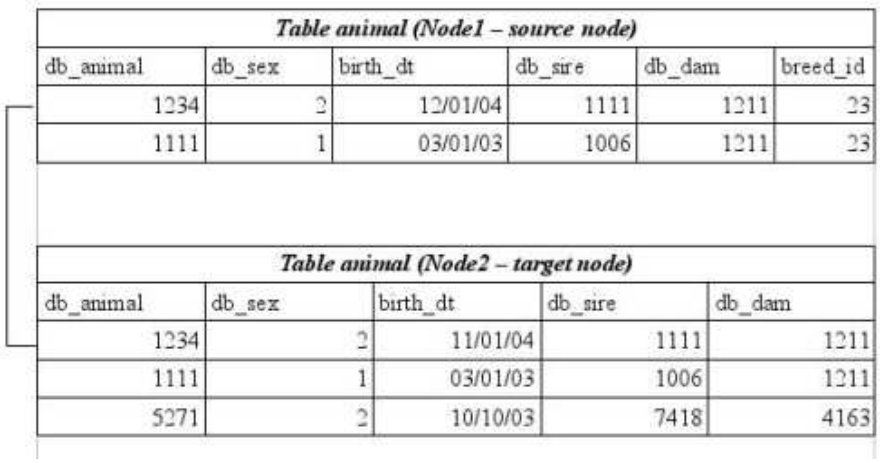


Table animal (Node1 – source node)

db_animal	db_sex	birth_dt	db_sire	db_dam	breed_id
1234	2	12/01/04	1111	1211	23
1111	1	03/01/03	1006	1211	23

Table animal (Node2 – target node)

db_animal	db_sex	birth_dt	db_sire	db_dam
1234	2	11/01/04	1111	1211
1111	1	03/01/03	1006	1211
5271	2	10/10/03	7418	4163

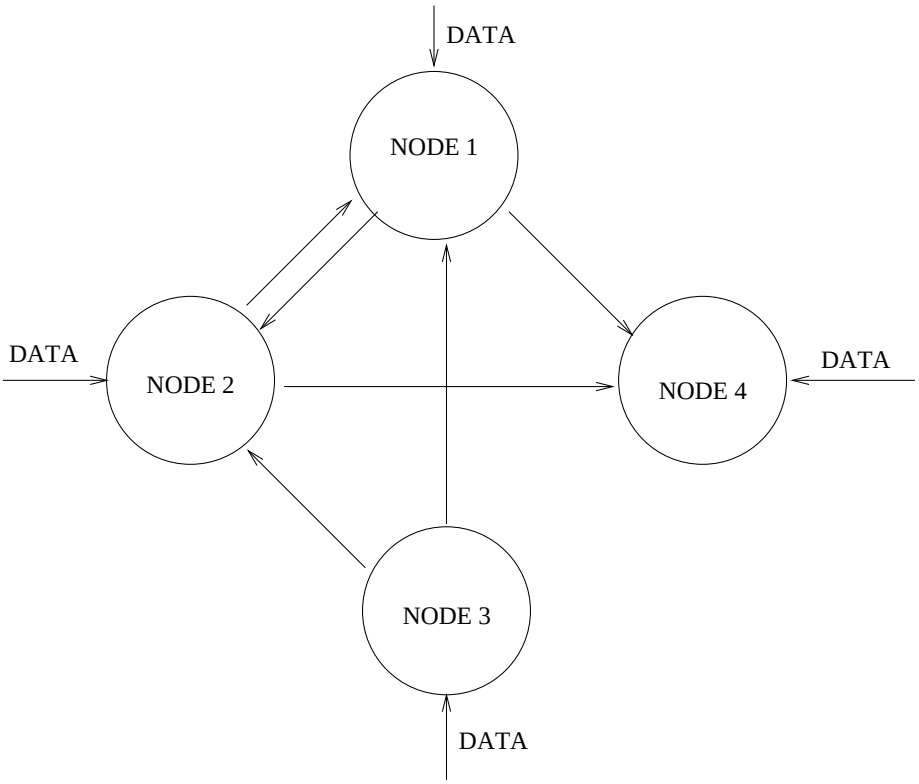
Information part: (1234,2,12/01/04,1111,1211)

where O is the owner, S - the source, T - the target. The first two fields should always have a value - even if it is the address of the same node (e.g. when DE occurs for the first time in the system, then the node where it has happened is the owner and also source for this data element). The third field is a set of addresses, and as it was mentioned before this set can be empty(in case of private data), or contain a list of some or all nodes in the network. The information part of one data element consists of one or more columns from one record. An example of the information part of data element is shown on Figure 8.1.
Two nodes can exchange information by setting source/target fields of the desired data element. For consistency reasons the following rule should apply: The source field of DE in the receiver node is the address of the sender node and the target field of the sender contains the receiver address .
This data flow, managed by setting source/target fields is regulated by Network manager in cooperation with all nodes, thus preventing inconsistencies in the system.
Also, as stated before, only owner can make representative changes of a data element and this changes are obligatory for all other nodes.

In a generic system there will be three type of nodes:

- **nodes that are only sources of information for the others**
- **nodes that are only targets**
- **nodes that are sources for some data elements and targets for other data elements**

Figure 8.2: General system topology



An example of such general animal biodiversity system topology is shown on Figure 8.2. NODE3 is an example of the first kind, NODE4 - of the second and NODE1 and NODE2 are examples of the third kind. In this example the content of NODE4 will look like on Figure 8.3. In the defined above terms "Data from NODE3" means data owned by NODE3 (initially loaded in NODE3). This data can be received via NODE1 or NODE2 and the route depends on the source-target pairs. It is possible that part of the data elements are distributed via NODE1 and part - via NODE2, but it is not possible that one data element has NODE1 and NODE2 as sources simultaneously.

8.3 Functional model

The functional model of the synchronization process of farm animal data is shown on Figure 8.4.

8.4 Implementation

The implementation has to include the following steps:

Figure 8.3: NODE4 data content

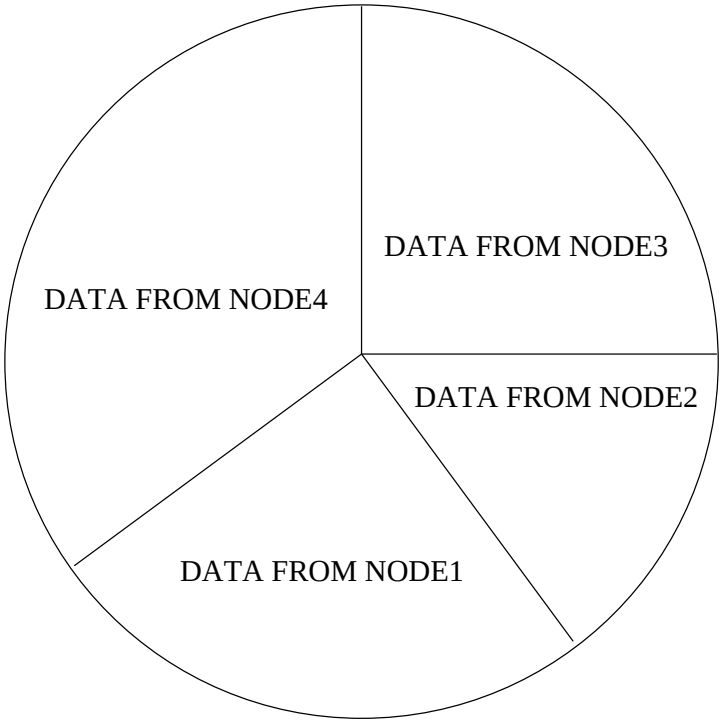
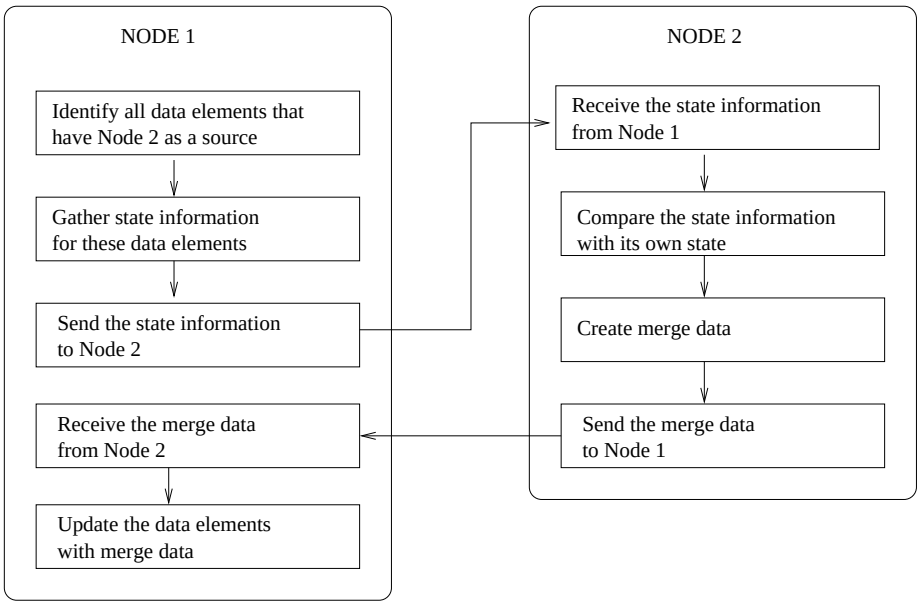


Figure 8.4: Functional model



- **Setting unique node name**
  Each node has to have unique name within the APIIS-network. THis name is given by the local administrator, but has to be co-ordinated with the Network Manager. The name is set in the local '/etc/apiisrc' file.

  ```
  [SYNCHRONISATION]
  node_name = EAAP
  ```

- **Setting unique node ip_address Each node must have static ip_address which is accessible from each node within the network. In addition port 5431 has to be open for incomming connections (this is the port where the server daemon listens). The ip_address is also set in local '/etc/apiisrc' file**

  ```
  [SYNCHRONISATION]
  node_ip = 10.1.1.126
  ```

- **Setting unique numbering interval Each node in the system has to have unique record numbers and also unique animal internal numbers. The numbering range has to be obtained from from Network Manager and set in local '/etc/apiisrc' file:**

  ```
  [SYNCHRONISATION]
  sequence_interval=1:100000000
  ```

  The initialization of all sequence generators is done via the 'CreateDatabase' subroutine. It encapsulates the subroutines: 'LoadNodeData' and 'SetSequences' which are responsible for loading the local nodename and address in the database and setting the numbering interval for all sequences.

- **Setting information about the other nodes in the network This is done via the program 'nodemanagement.pl' using the menu Settings>Nodes.**

- **Setting information about the sources This is done via the program 'nodemanagement.pl' using the menu Settings>Sources.**

- **Setting information about the targets This is done via the program 'nodemanagement.pl' using the menu Settings>Targets.**

Chapter 9

XML and APIIS model

Writing a model file as a simple text file always consumes a lot of time and is not protected of making mistakes that can have effect on further work with other APIIS tools. Working with a text editor usual does not give the completely picture of a document stricture that makes changes in the model file difficult and slow. In order to use faster and safety way for writing and editing a model file XML standard is implemented for describing data model structure of APIIS.

9.1 Why XML?

:)

9.2 Needed modules

The names of needed modules are written in file *needed_modules* in pdbl tree.

9.2.1 Parsing XML document

From many existing perl parsers of XML documents are chosen two that are standard for Perl 5.6.1 - XML::Parser and XML::Writer. Usual these modules are installed during regular perl installation in folder named *xml*. They use stream methods of parsing that are faster and take less memory for storage of data elements which was the reason for their usage.

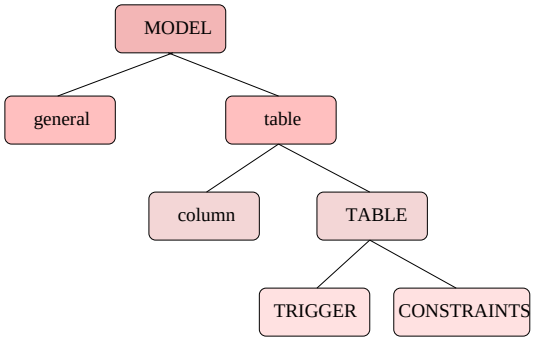
For better XML print a code, version of XML::Writer, *myWriter* is improved specially for APIIS.

9.2.2 Document Type Definition (DTD) of APIIS model in XML standard

APIIS model file is described in XML DTD format via the structure given in the figure 9.1 . The root element is the *model* with sub elements or children *general* and *table*. The sub elements of *table*

are *column* and *TABLE. TRIGGER* and *CONSTRAINTS* are sub elements of *TABLE.*

Figure 9.1: The XML structure of APIIS model



The DTD file on the figure 9.1 shows the elements definitions in APIIS model XML format as well as their attributes' lists. As attributes are better processed by some XML editors they are preferred here. Usage of attributes allows default values that facilitates the inserts of model elements. It can be seen that elements marked by '+' are these appearing more than ones in the model structure like *table* and *column*.

9.2.3 Converting the model file to XML format

The perl code *model2xml.pl* converts the model file to XML format. Location is in *pdbl/bin*. The syntax is the next:

***model2xml model_filename xml_filename*, where the first argument is the model file name and second is the xml file name that is better to have extension *xml* for further work with some XML editors.**

9.2.4 Extracting APIIS model file from XML document

Extracting the mode file from xml formatted file is done by perl code *xml2model.pl* located in *pdbl/bin*. The syntax is the next:

***xml2model xml_filename model_filename*, where the arguments are with the same meaning as in 8.2.3. Here the default values of the elements' arguments are taken automatically from DTD file. The size of xml format file is twice less than usual model format file because all default values are not saved.**

9.2.5 Editing of XML documents

XML editors process XML documents usual in GUI environment where all existing xml elements are accessible for editing, moving and copies. All operations are controlled via DTD file or XML scheme.

Table 9.1: DTD file of APIIS model

```
¡!DOCTYPE model [
¡!ELEMENT model (general,table+)¿
¡!ELEMENT general EMPTY¿
¡!ATTLIST general
dbdriver (Pg—Oracle—CSV—InterBase—Sybase) "Pg"
dbname CDATA #REQUIRED
dbhost CDATA "localhost"
dbport CDATA "5432"
dbuser CDATA "$user"
dbpassword CDATA ""¿
¡!ELEMENT table (column+,TABLE)¿
¡!ATTLIST table
name CDATA #REQUIRED¿
¡!ELEMENT column EMPTY¿
¡!ATTLIST column DATA CDATA ""
name CDATA #REQUIRED
DATATYPE
(CHAR—HUGEINT—BIGINT—SMALLINT—DATE—TIME—TIMESTAMP—SMALLFLOAT—BIGFLOAT—BOOL)
"CHAR"
LENGTH CDATA "20"
DESCRIPTION CDATA #REQUIRED
DEFAULT CDATA ""
CHECK CDATA ""
MODIFY CDATA ""
ERROR CDATA ""¿
¡!ELEMENT TABLE (TRIGGER,CONSTRAINTS)¿
¡!ELEMENT TRIGGER EMPTY¿
¡!ATTLIST TRIGGER PREINSERT CDATA ""
POSTINSERT CDATA ""
PREUPDATE CDATA ""
POSTUPDATE CDATA ""
PREDELETE CDATA ""
POSTDELETE CDATA ""¿
¡!ELEMENT CONSTRAINTS EMPTY¿
¡!ATTLIST CONSTRAINTS
PRIMARYKEY CDATA ""
SEQUENCE CDATA ""
INDEX CDATA ""¿ ]¿
```

The entire structure of the document is in tree view that can be easy processed.

So far editor of choice is *Xerlin* available in http://www.xerlin.org. It demands Java machine to be installed.

9.3 How to use?

The idea is using the document type definition in 'model.dtd' and working with the program 'xerlin' to write the model file in format of xml document. The process is facilitated via usage of libraries of elements: columns and tables in which the main DB structure of APIIS is implemented.

9.3.1 Creating a new model file

Steps:

1. **Open library apiis.xmllib from where the necessary elements could be copied or simply 'drop and drag'.**

2. **Start new file with choice of DTD file model.dtd from pdbl/lib that will be used for xml structure control and checks.**

3. **Insert the root element of the document - *model* and first its child - general.**

4. **Take from library the main table xml equivalents**

5. **Insert other elements**

For creating a complete model file is recommended to start with root element although the program offers you to choose from which element to start.

All attributes with mandatory values (#REQUIERED) are marked to be inserted before next element starts. The context menus of the right mouse button in Xerlin offer to choose among legal operations and elements according the DTD and current content of the document. All operations are restricted by context and logical structure in DTD that protect the user from making mistakes. In the context menu only legal elements are accessed and can be inserted after, before or into a current element.

When an element is chosen its attributes are shown in right panel, default values are there and we insert only attribute values which are different from default ones.

The result file contains all elements of table and column types and their attributes. The default attributes' values are not recorded in this file if the property 'merlot.write.default-attr' has a value 'false' in preference panel.

9.3.2 Editing the model file

To produce APIIS format model file in syntax we use in APIIS environment is used the module *xml2model.pl*. The result file is used by other APIIS applications. In case the model file needs to be changed in Xerlin it is reversed in xml format via module model2xml.pl.

9.3.3 Using alternative XML editors.

In case Xerlin is not accessible another xml editor could be used.

1. **xemacs**
2. **kate**
3. **kxmleditor**

If they are appropriate will be investigated.(so far they are weaker than Xerlin)

Chapter 10

Report Generator

This is Ulf's magnificent report generator. Her only needs to add the documentation here.

Index