Una técnica para una especificación inicial en RSL

María Virginia Mauco

Director: MSc. Daniel Riesco

Codirector: Mr. Chris George

Tesis presentada a la Facultad de Informática de la Universidad Nacional de La Plata como parte de los requisitos para la obtención del título de Magister en Ingeniería de Software.

La Plata, Marzo de 2004

Facultad de Informática
Universidad Nacional de La Plata - Argentina

A technique for an initial specification in RSL

Abstract

Formal methods have come into use for the construction of real systems, as they help to increase software quality and reliability. However, they are usually only accessible to specialists. This is particularly inconvenient during the first stages of software development, when the participation of stakeholders, unfamiliar with this kind of description, is crucial.

To address this problem, we present in this thesis a technique to derive an initial formal specification written in the RAISE Specification Language from requirements models based on natural language. In particular, we start from the Language Extended Lexicon (LEL) and the Scenario Model, two models of the Requirements Baseline, which are closer to stakeholders language. The derivation of the specification is structured in three steps: Derivation of Types, Derivation of Functions, and Definition of Modules. We provide a set of heuristics for each step which show how to derive types and functions, and how to structure them in modules by using LEL and scenarios information, thus contributing to fruitfully use the large amount of information usually available after problem analysis. We also propose to represent the hierarchy of modules obtained using a layered architecture which is the basis to start applying the steps of the RAISE Method. We show how the initial applicative and partially abstract specification derived could be developed into a concrete one to automatically obtain a quick prototype to validate the specification and get a feeling of what it really does.

Resumen

Los métodos formales se están usando actualmente para la construcción de sistemas reales, ya que contribuyen a aumentar la calidad y confiabilidad del software. Sin embargo, generalmente sólo son accesibles a especialistas. Esto resulta inconveniente sobre todo durante las primeras etapas del proceso de desarrollo de software cuando la participación de los *stakeholders*, no familiarizados con estos formalismos, es crucial.

Con el objetivo de aportar una solución a este problema, presentamos en esta tesis una técnica para derivar una especificación formal inicial escrita en el Lenguaje de Especificación RAISE a partir de modelos de requisitos basados en lenguaje natural. En particular, usamos el Léxico Extendido del Lenguaje (LEL) y el Modelo de Escenarios, dos modelos de la *Requirements Baseline* que están más cercanos al lenguaje de los *stakeholders*. La derivación de la especificación está estructurada en tres etapas: Derivación de Tipos, Derivación de Funciones, y Definición de Módulos. Para cada etapa, proponemos un conjunto de heurísticas que muestran cómo derivar tipos y funciones, y cómo estructurarlos en módulos usando la información del LEL y los escenarios, contribuyendo así a aprovechar la gran cantidad de información generalmente disponible después del análisis del problema. También proponemos representar la jerarquía de módulos obtenida usando una arquitectura por niveles, que es la base para comenzar a aplicar las etapas del Método RAISE. Mostramos cómo llegar a una especificación concreta, partiendo de la especificación aplicativa y parcialmente abstracta derivada, para luego obtener automáticamente un primer prototipo para validar la especificación.

Contents

Acknowledgements

I want to give my special thanks to my supervisors Daniel, Chris and Gustavo for their continuous guidance and assistance in spite of distance. To Daniel for his permanent availability to answer my requests, for his useful ideas and comments, and his constant support. To Chris for his valuable help and patience, specially at the beginning of this work. To Gustavo for all the support he gave me.

I am thankful to UNU/IIST's staff and fellows for making my fellowship in Macao a wonderful experience, and to MSc. Marcela Inza from Universidad Nacional del Centro de la Provincia de Buenos Aires, Argentina, and Ing. Agr. Atilio Magnasco for their work as domain experts.

My gratefulness also to my colleagues and friends Liliana, Carmen, and Laura for their permanent help and support.

Finally, I want to thank my family and friends for their love and support in my life, and to God for all the good things he has provided me.

Chapter 1

Introduction

1.1 Motivation

Formal methods have come into use for the construction of real systems, as they help to increase software quality and reliability, and even though their industrial use is still limited, it has been steadily growing [47]. When used early in the software development process, they can reveal ambiguities, incompleteness, inconsistencies, errors or misunderstandings that otherwise might be only discovered during costly testing and debugging phases. The RAISE Method [22], for example, is intended for use on real developments, not just toy examples. This method includes a large number of techniques and strategies for doing formal development and proofs, as well as a formal specification language, the RAISE Specification Language (RSL) [21], and a set of tools to help writing, checking, printing, storing, transforming, and reasoning about specifications [19].

One tangible product of applying a formal method is a formal specification [50]. A specification serves as a contract, a valuable piece of documentation, and a means of communication among stakeholders, specifiers, and implementers. Formal specifications may be used along the software lifecycle and they may be manipulated by automated tools for a wide variety of purposes such as model checking, deductive verification, animation, test data generation, formal reuse of components, and refinement from specification to implementation [47]. However, one of the problems with formal specifications is that they are hard to master and inappropriate as a communication medium, as they are not easily comprehensible to stakeholders, and even to non-formal specification specialists.

On the other hand, during first stages of system development the interaction with stakeholders is very important. System requirements must be described well enough so that an agreement can be reached between the stakeholders and the system developers on what the system should and should not do. A major challenge with this is that the stakeholders must be able to read and understand the results of requirements capture. To meet this challenge we must use the language of the stakeholders to describe these results [15, 24].

Domains are naturally informal because they reside in the real world. The problem domain is the home of real users and other stakeholders, people whose needs must be addressed in order to build the right system. Then, it becomes software engineers' problem to understand these people problems, in their culture and their language, and to build systems that meet their needs [29]. In addition, we end in the world, validating the specifications with the stakeholders. Therefore specifications are never formal at first. A good formal approach should use both informal and formal techniques [5]. To define properties precisely and formally, it is necessary to determine first what these properties are. And this must be done in a language all the people involved can speak and understand. For example, the Requirements Baseline [31, 32], one technique proposed to formalise requirements elicitation and modelling, includes two natural language models which ease stakeholder's actively participation and facilitates effective communication of the requirements among different stakeholders and software engineers.

In spite of the wide variety of formal specification languages and modelling languages, such as the Unified Modeling Language (UML) [24], natural language is still the method chosen for describing software system requirements [8, 24, 45, 47]. However the syntax and semantics of natural language, even with its flexibility and expresiveness power, is not formal enough to be used directly for prototyping, implementation or verification of a system. Thus, the requirements document written in natural language has to be reinterpreted by software engineers into a more formal design on the way to a complete implementation.

Considering what we have explained above, we think it would be useful to analyse and develop an integration between stakeholder-oriented requirements techniques and formal methods. In this way we could take advantage of both of them in the different steps of the software development process in order to improve the final product. Stakeholder-oriented requirements engineering techniques allow the development of a first specification of a system which can be validated with the stakeholders, and used as the basis to define a formal specification. But, as also some recent works point out [16, 28, 40, 41, 48], it would be necessary to look at ways for mapping the conceptually richer world of requirements engineering to the formal methods world.

In particular, our proposal aims at integrating the Requirements Baseline [31, 32] with the RAISE Method [20, 22]. We have developed a set of heuristics to help in the definition of an initial formal specification in RSL of a domain, starting from two natural language models belonging to the Requirements Baseline. In the next section we describe the goal of our proposal as well as some details of the steps we followed to achieve it.

1.2 Our proposal

1.2.1 The goal

The main goal of our work is to analyse and develop the integration of Requirements Engineering techniques with formal specifications written in RSL. We propose to develop a technique to derive an initial formal specification in RSL of a domain from two of the models of the Requirements Baseline, the Lexicon Model View and the Scenario View.

1.2.2 General description

When using the RAISE Method, writing the initial RSL specification is the most critical task because this specification must capture the requirements in a formal, precise way [20, 22]. But, as we have explained in the previous section, at the beginning of the software development process it would be better to use some kind of informal representations to allow stakeholders to participate actively in the requirements definition process. RSL specifications of many domains [6, 42, 43, 46] have been developed by starting from informal descriptions containing synopsis (introductory text which informs what is the domain about), narrative (systematic description of all the phenomena of the domain), and terminology (list of concise and informal definitions, alphabetically ordered). Others also include a list of events [11]. The gap between these kind of descriptions and the corresponding RSL formal specification is big, and thus, for example, it is difficult and not always possible to check whether the informal specification models what the informal description does and vice versa.

As we had some experience in using the Requirements Baseline [13], and we knew it had been used as the basis to an object conceptual model [33], we consider the possibility of using it as the first description of a domain from which a formal specification in RSL could be latter derived.

For this reason, our proposal aims at defining a technique to derive an initial formal specification in RSL from the Lexicon Model View and the Scenario View, two natural language models belonging to the Requirements Baseline. We organise the derivation of the specification in three steps, Derivation of Types, Derivation of Functions, and Definition of Modules, as RSL specifications are structured in modules, and each module may contain definitions of types, values (constants and functions) and axioms. We define for each step a set of heuristics which are guidelines about how to derive types and functions, and how to structure them in modules, taking into account the structured description of a domain provided by the Lexicon Model View and the Scenario Model. The Lexicon Model View contains structural features of the relevant terms in the domain, thus limiting the definition of types to those that correspond to significant terms in the domain, while using the behavioural description represented in the Scenario View, it is possible to identify the main functionality to model in the specification.

We also suggest to represent the hierarchy of RSL modules obtained using a layered architecture. Considering the Layers pattern implementation described in [24], the global architecture we propose is composed of three layers: specific layer, general layer and middleware layer. This layered architecture is then the basis to start applying the steps of the RAISE Method, encouraging separate development and step-wise development. For example, the initial applicative and partially abstract specification derived could be developed into a concrete one to make use of the SML translator [19] and, thus obtain a quick prototype to validate the specification and get a feeling of what it really does.

In order to validate our proposal, we applied it to a complete case study, the Milk Production System. We first developed the Lexicon Model View and the Scenario View for this domain, by working with two domain specialists (who are, besides, non-computer people). These models were then the basis to apply the three steps of the specification derivation process. Finally, we developed the initial specification obtained into a concrete one, in order to use the SML translator to obtain a prototype of the specification. We also defined a set of test cases to run the specification, and not only check it against the Lexicon and the Scenario Models but also help in finding poorly understood requirements, missing things, etc.

1.3 Publications

A Layered Architecture for a Formal Specification in RSL
María Virginia Mauco, Daniel Riesco, Chris George
International Conference on Computer Science, Software Engineering, Information Technology, e-Business and Applications (CSITeA'02). Brazil, June 2002.
pp 258-263. ISBN 0-9700776-3-7

Using a Scenario Model To Derive the Functions of a Formal Specification
María Virginia Mauco, Daniel Riesco, Chris George
8th Asia-Pacific Software Engineering Conference (APSEC 2001), IEEE Press, Macao, December 2001.
pp 329-332. ISBN 0-7695-1408-1.

Heuristics to Structure a Formal Specification in RSL from a Client-oriented Technique
María Virginia Mauco, Daniel Riesco, Chris George
1st Annual International Conference on Computer and Information Science (ICIS'01), USA, October 2001.
pp 323-330. ISBN 0-9700776-2-9

Deriving the Types of a Formal Specification from a Client-Oriented Technique
María Virginia Mauco, Daniel Riesco, Chris George
2nd International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'01), Japan, August 2001.
pp 1-8. ISBN 0-9700776-1-0

Una técnica para una especificacion inicial en RSL
María Virginia Mauco, Daniel Riesco
Poster presented in the Software Engineering Area of the Computer Science Researchers Workshop (WICC 2001). Universidad Nacional de San Luis, 2001.
Also, WICC 2001 Memories. pp 302-304.

Using Requirements Engineering to Derive a Formal Specification
María Virginia Mauco, Chris George
United Nations University/International Institute for Software Technology (UNU/IIST). Technical Report Number 223. December 2000. Macao.

Una Estrategia de Análisis Orientada a Objetos basada en Escenarios: Aplicación en un Caso Real
Laura Rivero, Jorge Doorn, Mariana del Fresno, María Virginia Mauco, Marcela Ridao, Carmen Leonardi
Workshop de Engenharia de Requisitos, XII Simposio Brasileiro de Engenharia de Software, Brasil, October 1998.
pp. 79-88

Derivación de Objetos utilizando LEL y Escenarios en un Caso Real
Mariana del Fresno, María Virginia Mauco, Marcela Ridao, Jorge Doorn, Laura Rivero
Workshop de Engenharia de Requisitos, XII Simposio Brasileiro de Engenharia de Software, Brasil, October 1998.
pp. 89-97.

1.4 Thesis organization

This work is organised as follows: in Chapter 2 we briefly describe the Requirements Baseline, making emphasis in the Lexicon Model View and the Scenario View. Chapter 3 presents the RAISE Method, and a brief description of the RAISE Specification Language. In Chapter 4 we present the three-step process to derive an RSL specification, describing and giving examples of the heuristics we propose for each step. Chapter 5 contains the application of our proposal to a complete case study, the Milk Production System, and Chapter 6 describes the approach we follow to validate the RSL specifications obtained. Then, Chapter 7 presents some conclusions and contributions of our work, as well as possible future works. Finally, Appendix A and B contain the Lexicon Model View and the Scenario View for the Milk Production System respectively, and Appendix C includes the complete RSL specification we obtained by developing the specification resulting from the application of our proposal.

Chapter 2

The Requirements Baseline

The Requirements Baseline [30, 32] is a mechanism proposed to formalise requirements elicitation and modelling. It is a structure which incorporates descriptions about a desired system in a given application domain. These descriptions are written in natural language following defined patterns. Thus, they provide an attractive way of communication and agreement between software engineers and stakeholders. As natural language can be read and understood by the stakeholders, they can participate actively in the requirements definition process.

The Requirements Baseline is developed during the requirements engineering process, but it continues to evolve during the software development process.

It is composed of five complementary views:

- the **Lexicon Model View**, a representation of the significant terms in the application domain language, defined as a Language Extended Lexicon (LEL) description
- the **Scenario View**, a description of behaviour in the application domain
- the **Basic Model View**, which uses the entity relationship framework as a representation language; its basic components are clients, actions, external events, inputs, outputs, restrictions, and diagnoses
- the **Hypertext View**, which allows one to link definitions between the Lexicon, Scenario and Basic Model Views
- the **Configuration View**, a versioning system to maintain the traceability of the different products and their revisions.

To the best of our knowledge, LEL and scenarios have been only used so far in case studies involving Information Systems.

In this work we will only use the Lexicon Model View and the Scenario View. They are explained in detail in Section 2.1 and in Section 2.2 respectively.

2.1 The Lexicon Model View

This view is implemented by the LEL, a meta-model designed to help the elicitation and representation of the language used in the application domain. It is a natural language

LEL: representation of the symbols in the application domain language.
Syntax: $\{\text{Symbol}\}_1^N$

Symbol: entry in the lexicon with a special meaning in the application domain.
Syntax: $\{\text{Name}\}_1^N + \{\text{Notion}\}_1^N + \{\text{Behavioural Response}\}_1^N$

Name: identification of the symbol; more than one represents synonyms.
Syntax: Word | Phrase

Notion: denotation of the symbol; it must be expressed using references to other symbols and using a minimal vocabulary.
Syntax: Sentence

Behavioural Response: connotation of the symbol; it must be expressed using references to other symbols and using a minimal vocabulary.
Syntax: Sentence

where Sentence is composed only by Symbols and Non-Symbols, and the last ones belong to the minimal vocabulary;
+ means composition, {x} means zero or more ocurrences of x, and | stands for **or**

Table 2.1: The Language Extended Lexicon Model

representation that aims at registering symbols (words or phrases) which are significant in the application domain language. The focus of the LEL is on the application domain language, rather than the details of the problem. It minimises the language, limiting it to the symbols that are related to the problem domain. In addition, it clearly defines these symbols and aims to eliminate possible inconsistencies, ambiguities and misinterpretations. Although the LEL is constructed before the scenarios, it evolves during scenarios definition.

Each entry in the LEL has a name (and possibly a set of synonyms), and two descriptions: Notion and Behavioural Response. The Notion, similar to a dictionary definition, describes the symbol denotation, i.e. what the symbol is. The Behavioural Response, describes the symbol connotation, that is how the symbol acts upon the system. Table 2.1, taken from [30], presents the LEL Model.

When describing symbols in the LEL, two principles must be followed: the principle of circularity, also called principle of closure, which aims at maximising the use of symbols defined in the LEL when describing other symbols, and the principle of minimal vocabulary, that intends to minimise the use of symbols external to the lexicon. As suggested in [30], the external symbols should belong to a small subset of words defined in a natural language dictionary, such as the Longman Defining Vocabulary [2]. The application of these two principles allows the definition of a self-contained set of highly connected symbols, which could be represented as a hypertext document.

LEL symbols may be classified according to its general use in the application domain.

Subject	*Notion:* who the subject is
	Behavioural response: register actions executed by the subject
Object	*Notion:* define the object and identify other objects with which the object has a relationship
	Behavioural response: describe the actions that may be applied to the object
Verb	*Notion:* describe who executes the action, when it happens and procedures involved in the action
	Behavioural response: describe the constraints on the happening of the action, and identify the actions triggered in the environment and new situations that appear as consequence
State	*Notion:* what it means and the actions which may be triggered by the state
	Behavioural response: describe other situations and actions related to it

Table 2.2: Heuristics to define LEL symbols

One possibility, as proposed in [34], is to classify each entry in the LEL as an object (passive entity), a subject (active entity, usually a person or organisation that represent relevant behaviour of the system), a verb phrase or a state. For this classification, some heuristics were proposed in [34] to suggest what to include in the notion and behavioural response of a symbol according to what the symbol defines. They are shown in Table 2.2.

Tables 2.3, 2.4, 2.5, and 2.6 show each an example of a LEL symbol, taken from the LEL of the Milk Production System. Underlined words or phrases correspond to other symbols defined in the LEL.

2.1.1 LEL Construction Process

The LEL construction process consists of six steps [17], which are dependent on each other and sometimes may overlap:

- Identification of the sources of information (a)
- Identification of the symbols (b)
- Classification of the symbols (c)
- Description of the symbols (d)
- Verification of the LEL (e)
- Validation of the LEL with the stakeholders (f)

As we have already said, the LEL evolves during the scenario construction process. This means the LEL produced after the six steps just mentioned may be modified because of discrepancies, errors or omissions founded while defining the scenarios.

The LEL construction elicits the vocabulary used in the application domain. As the sources of information are inside the application domain, the first step is the definition of the

DAIRY FARMER

Notion

- Person in charge of all the activities in a dairy farm.
- He has a name.
- He has a salary.
- He may have one or more employees

Behavioural Response

- He milks all the milking cow.
- He detects heat.
- He assigns to a group each cow of the dairy farm.
- He defines plot.
- He decides when to dry a cow for discard.
- He feeds groups of cows.
- He computes ration for each cow.
- He vaccinates each cow according to its needs.
- He weighs cow.
- He defines calf groups.
- He deparasites calves or heifers.
- He decides when to inseminate dairy cows or heifers.
- He saves birth.
- He registers heat.
- He sends calf to the calf rearing unit.
- He carries out calves artificial breeding.
- He takes calf out the calf rearing unit.
- He selects a calf group for each calf.
- He sells cow.
- He handles cow death.
- He computes individual production of a milking cow, a group or a dairy farm.
- He buys bull for the dairy farm.
- He discards bull.
- He computes birth date for each dairy cow or heifer.
- He inseminates artificially dairy cows or heifers.
- He sends to eat pasture each group in the dairy farm.
- He detects pregnant cow.
- He defines cow type.

Table 2.3: Subject LEL symbol

DAIRY COW

Notion

- It is a female cow which has had at least one calf.
- It is in a plot.
- It may be milking cow, dry cow, or discard cow.
- It weighs between 550 and 580 kilograms.
- Its useful life lasts more or less 4 years.
- It has an individual production.
- It belongs to a group of type 1, 2, pre-birth cow, dry cow or discard cow.
- It may be pregnant.
- It may be on heat every 21 days.

Behavioural Response

- When on heat, heat is registered.
- It is milked for approximately 10 months in each 12 months.
- It may be inseminated by artificial insemination or natural insemination.
- It generally gives birth to one calf per 12 months, and each birth is saved.
- When it is 4 years or more and approximately 580 kilograms weight, it may be dried

Table 2.4: Object LEL symbol

SAVE BIRTH

Notion

- A dairy farmer manages all the things related to a recent birth of a dairy cow or a heifer.

Behavioural Response

- The calf is assigned an identification number and it is added to the dairy farm set of cows.
- If a heifer is involved, define cow type as dairy cow.
- The new calf is added to the dairy cow's list of births.
- The date and the identification number of the calf and the dairy cow are registered in the Birth form.
- Define cow type as post-birth cow
- The post-birth cow is assigned to a group of type 1.

Table 2.5: Verb LEL symbol

LACTATION PERIOD/LACTATION

Notion

- Period after the birth of a calf in which a dairy cow produces milk.
- Dairy cow should be a milking cow.

Behavioural Response

- It lasts approximately seven months.
- Dairy cows can be milked.

Table 2.6: State LEL symbol

context where the requirements engineering process will take place. Documents and people involved in the application domain are the most important sources of information. However, other sources should be also considered such as books about related topics, and other systems available (a).

Once established the sources of information, the next step is the selection of the strategies to extract the symbols from these sources. Although the strategies depend heavily on the sources of information, structured and unstructured interviews are the most common way to recognise the vocabulary the stakeholders use in their domain. Interviews are usually combined with reading of documents, such as forms and manuals. The result is a list of symbols organised by some criteria, as for example alphabetical order (b).

The symbols are then classified according to the general classification (subject / object / verb / state), a refined or an alternative one (c).

The description of the symbols (d) consists in the definition of their notions and behavioural responses considering the type assigned to each symbol during the classification step.

The goal of the verification step (e) is to do an internal test to control if the produced LEL is consistent and homogeneous.

At last, by validating the LEL with the stakeholders (f) notions and behavioural responses of symbols already defined are corrected, the definitions of the symbols are confirmed, and new symbols and synonyms may be identified. The validation process generally consists of structured interviews with the stakeholders. As LEL is written in natural language the stakeholders do not have difficulties in understanding it, and thus they can participate actively in this process.

2.2 The Scenario View

A scenario describes a situation in the application domain, with an emphasis on the behaviour description [32]. Although each scenario describes a particular situation, none of them is entirely independent of the rest [30]. Scenarios use also a natural language description as their basic representation and they are naturally linked to the LEL. This link is reflected by underlying words or phrases defined in the LEL every time they appear in a scenario description.

Table 2.7 shows the structure proposed in [30, 32] to describe scenarios. A scenario must satisfy a goal which is fulfilled by performing the episodes. Episodes represent the main course of action and each of them corresponds to an action performed by an actor, with the participation of other actors, and the use of resources. Each episode may be a simple, an optional or a conditional one. Simple episodes are those necessary to complete the scenario. Conditional episodes depend on a specified internal or external condition, and optional episodes are those that may or may not take place according to conditions that cannot be explicitly detailed. Though main and alternative courses of action are treated within one scenario, many times understanding a scenario turns easier if well-bounded situations are detected and treated as sub-scenarios. A sub-scenario is then used when common behaviour is detected in several scenarios, complex conditional or alternative course of action appears in a scenario, or the need to enhance a situation with a concrete and precise goal is detected inside a scenario.

The context describes the initial state of the scenario, by using preconditions, and geographical and temporal locations.

Actors are entities actively involved in the scenario, generally persons or organisations, and resources identify passive entities with which actors work.

Constraints and Exceptions may be added to some of the components of a scenario. A constraint refers to non-functional requirements, and it may be applied to context, resources or episodes. An exception, only applied to episodes, causes serious disruptions in the scenario, asking for a different set of actions. These actions may be described separately as an exception scenario. The treatment of the exception may or may not satisfy the original goal. Table 2.8 contains an example of one scenario taken from the Milk Production System Scenario Model. Underlined words or phrases are symbols defined in the LEL, and phrases written in the episodes using capital letters correspond to the title of other scenarios. Episodes may be enclosed between # and #, to represent a parallel or arbitrary sequential order.

Scenarios can be derived from the LEL by applying a set of heuristics [30] or they can be constructed directly from the application domain. However, these two alternatives can be combined. First, the scenarios are derived directly from the LEL by applying the heuristics which produce a set of candidate scenarios. These scenarios are then improved and extended, returning to the application domain when necessary. Also, new scenarios may be added.

Scenario: description of a situation in the application domain.
Syntax: Title + Goal + Context + $\{\text{Resources}\}_1^N$ + $\{\text{Actors}\}_1^N$ + $\{\text{Episodes}\}_2^N$ + Exceptions

Title: identification of the scenario; in case of a sub-scenario the title is the same as the corresponding episode sentence, without the constraints.
Syntax: Phrase | ([Actor|Resource] + Verb + Predicate)

Goal: aim to be reached in the application domain; the scenario describes the achievement of the goal.
Syntax: ([Actor |Resource] + Verb + Predicate)

Context: composed by at least one of the following sub-components: geographical location (physical set of the scenario), temporal location (time specification for the scenario development), precondition (initial state of the scenario).
Syntax: Geographical location + Temporal location + Precondition
where Geographical location is Phrase + Constraint
Temporal location is Phrase + Constraint
Precondition is [Subject|Actor|Resource] + Verb + Predicate + Constraint

Resources: relevant physical elements or information that must be available in the scenario.
Syntax: Name + Constraint

Actors: persons, devices or organisation structures that have a role in the scenario.
Syntax: Name

Episodes: set of actions that details the scenario and provides its behaviour. An episode can also be described as a scenario.
Syntax: see [30]

Exceptions: usually reflect the lack or malfunction of a necessary resource. An exception hinders the achievement of the scenario goal. The treatment of the exception may be expressed through another scenario.
Constraint: a scope or quality requirement referring to a given entity. It is an attribute of Resources, basic Episodes or sub-components of Context.

+ means composition, {x} means zero or more ocurrences of x, () is used for grouping, [x] denotes that x is optional and | stands for **or**

Table 2.7: The Scenario Model

TITLE: Manage birth
GOAL: Manage all things related to a recent birth.
CONTEXT: Pre: The cow is a dairy cow or a heifer which has just given birth to a calf.
RESOURCES: Cow Calf Date of birth Birth form Dairy farm set of cows
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer assigns an identification number to the calf.
- # The dairy farmer adds the new calf to the dairy farm set of cows.
- The dairy farmer adds the new calf to the dairy cow's or heifer's list of given birth to calves.
- The dairy farmer records in the Birth form the date and the identification number of the calf and the cow.#
- ASSIGN A GROUP TO A COW, a group of type 1.
- DEFINE COW TYPE as post-birth cow.

Table 2.8: Example of a scenario

Chapter 3

RAISE

RAISE (Rigorous Approach to Industrial Software Engineering), which was originally the name of a CEC funded ESPRIT project, gives now its name to a wide spectrum specification and design language, the RAISE Specification Language (RSL), an associated method, and an available set of tools. In this chapter we provide a brief introduction to RSL and to the RAISE Method, as well as a short description of the RAISE tools. Complete descriptions of the RAISE Method and RSL can be found in the corresponding books [22] and [21], while the tools are described in [19], and they can be downloaded from UNU/IIST's web site (www.iist.unu.edu).

3.1 The Language

The RAISE Specification Language (RSL) is a powerful formal specification and design language used in the RAISE Method. The language provides a range of specification styles (axiomatic and model-based; applicative and imperative; sequential and concurrent), and supports specifications ranging from abstract (close to requirements) to concrete (close to implementations). RSL allows specifications and designs of large systems to be modularised and permits separate subsystems to be separately developed. It also allows low-level operational designs to be expressed, to a level of detail from which final code extraction is straightforward. This means most of the construction of a system, from specification to design, may be done using one and the same formalism, thus providing precise, mathematical arguments for correctness of development steps and of other critical properties.

In the following sections, we briefly describe some basic concepts of the language emphasizing those we will need to use in the Three-step Process we present in Chapter 4. Detailed definitions can be found in [20] and [21]. The examples used to shown RSL constructions were taken from the specification of an University Library [42].

3.1.1 Basic Class Expressions

A specification in RSL is a collection of modules. A module is basically a named collection of declarations and it can be a scheme or an object. However, the kernel module concept is that of a class expression. A basic class expression is a collection of declarations enclosed by the keywords **class** and **end** and it represents a class of models. Each declaration is a keyword followed by one or more definitions of the appropriate kind (Table 3.1).

Declaration	**Kind of definition**
object	Embedded modules
type	Types
value	Values: constants and functions
variable	Variables that may store values
channel	Channels for input and output
axiom	Axioms: logical properties that must always hold
test_case	Test cases: expressions to be evaluated by a translator or interpreter

Table 3.1: Declarations and their definitions

No declarations are compulsory, and thus, many classes only contain type and value declarations. Though the declarations may come in any order, the order shown in Table 3.1 is a common one to use.

3.1.2 Types

RSL is a typed language. This means each ocurrence of an identifier representing a value, variable or channel must be associated with a unique type. Besides, it must be possible to check each ocurrence of an identifier is consistent with a collection of typing rules.

A type is a collection of logically related values, and it may be specified by an abstract or a concrete definition. An abstract type, also referred to as a sort, has only a name. It is a type we need but whose definition we have not decided on yet. A concrete type can be defined as being equal to some other type, or using a type expression formed from other types. For example

type
  Book_id,
  Copy_id,
  Book_key = Book_id × Copy_id

defines Book_id and Copy_id as abstract types, and Book_key as a concrete one.

In order to provide concrete definitions for types, we need a collection of types to use. RSL has seven built-in types (Bool, Int, Nat, Real, Char, Text, and Unit) with their corresponding operators, and a number of ways of constructing types from other types (type constructors, record types, variant types, union types, and subtypes).

Type constructors allow the definition of composite types: products (×), functions (→ for total functions, $\overset{\sim}{\rightarrow}$ for partial ones), sets (**-set** for finite sets, **-infset** for infinite ones), lists (* for finite lists, $^\omega$ for infinite ones), and maps ($\underset{m}{\rightarrow}$ for finite maps, $\underset{m}{\overset{\sim}{\rightarrow}}$ for infinite ones). Sets, lists and maps define collections of values of the same type. A set is an unordered collection of distinct values, while a list is a sequence of values, possibly including duplicates. A map is a table-like structure that maps values of one type into values of another type.

For example, the following definition models the collection of all books of the library by using a map type.

type

```
    Book_id,
    Book,
    Books = Book_id $\underset{m}{\rightarrow}$ Book
```

Records are very much like those common in programming languages. This example defines the type Borrower as a record with three components:

```
type
 Borrower_detail,
 Borrower_level == academic | non_academic | student,
 Br_copies,
 Borrower::
       borr_detail: Borrower_detail ↔ chg_borr_detail
       level: Borrower_level ↔ chg_level
       copies: Br_copies ↔ chg_copies
```

Each component has an identifier, called a destructor, and a type expression. Optionally a record component can have a reconstructor. A record type definition also provides an implicit constructor function for creating a record value from its component values. In the previous example, we have the constructor mk_Borrower, which is formed by putting mk_ on the front of the corresponding identifier of the type.

Destructors are total functions from the record type to their components type expression. So, we can apply, for example, level to a value of type Borrower to get its level, and then for a borrower value br, we write level(br).

Reconstructors are total functions that take their components type expression and a record to generate a new record. If we write chg_level(student, br) we get a new borrower value with the same borr_detail and copies, but with the level component set to student.

Variant types allow the definition of types with a choice of values, perhaps with different structures. The type Borrower_level shown above is an example of a variant type definition.

Union type definitions allow us to make new types out of existing ones. If B and C are types defined somewhere, then we can define the type A as their union:

```
type
    A = B | C
```

Subtypes are types that contain only some of the values of another type, the ones that satisfy a predicate. For instance, we can define the type Student as the one containing values that satisfy the predicate is_student.

```
type
    Student = {| br : Borrower • is_student(br) |}
```

3.1.3 Values

Values are constants and functions, and they may be implicit or explicitly defined. In both cases, the definition must include at least the signature, that is a name, and types for the result, and for the arguments, in case of a function. A value declaration consists of the keyword **value** followed by one or more value definitions separated by commas.

For example, to specify the maximum number of copies a borrower can be reading in the reading room, we may provide the following implicit value definition:

```
value
   reading_limit: Nat • ≤ 3
```

However, if we knew the exact value of the constant, we can use an explicit value definition:

```
value
   reading_limit: Nat = 3
```

A function is a mapping from values of one type to values of another type, and it can be total or partial. It is total when it is defined for every value of the arguments, and it is considered partial when it is not known to be total. Functions can be also classified as either generators, when the type of interest appears directly or indirectly in the result type, or observers, when it does not. In the following definitions:

```
value
   can_remove_item: Book_id × Books → Bool
   can_remove_item(bi, bs) ≡ exist_id(bi, bs) ∧ B.has_copy(bs(bi)),

   remove_item: Book_id × Books ~→ Books
   remove_item(bi, bs) ≡ bs \ {bi}
   pre can_remove_item(bi, bs)
```

remove_item is a partial generator function, while can_remove_item is a total observer one. Both definitions are explicit ones.

3.1.4 Axioms

Axiom declarations are introduced by the keyword **axiom** and consist of axiom definitions separated by commas. An axiom definition is a predicate, optionally preceded by an identifier in square brackets. For example, instead of defining:

```
value
   reading_limit: Nat • ≤ 3
```

we could write:

```
value
   reading_limit: Nat
axiom
   [reading_ copies_limit] reading_limit ≤ 3
```

In fact, all value definitions, can be written in this style, a typing plus an axiom. Though this "axiomatic" or "algebraic" style can be used within RAISE, RAISE allows the use of the pre-defined sets, lists, maps, and products that are characteristic of model-based specification languages.

3.1.5 Test Cases

Test cases were added to RSL after the publication of the two books on RAISE [21] and [22]. They have no semantic meaning: they are like comments directed at an interpreter or translator meaning "please provide code to evaluate these expressions and report the results".

Their syntax is similar to the one of axioms, except that the test case expressions can be of any type. For example, the following test case tests a function to sum a list of integers

```
test_case
   [sum0] sum(⟨⟩)
   [sum1] sum(<1,2,2>)
```

giving the results

```
[sum0] 0
[sum1] 5
```

However, including the expected result in the test case may be a more useful style of test case. This means to write

```
test_case
   [sum0] sum(⟨⟩) = 0
   [sum1] sum(<1,2,2>) = 5
```

so that the output for every test case should be true.

Test cases are always evaluated in order of definition, and this is useful for imperative specifications [20, 21] when there are variables storing information. As the information stored as a result of one test case is available for the next one, it would be possible for example, to test use-cases step-by-step by using a sequence of test cases, outputting intermediate observations as the result of each.

3.2 The Method

Though using the syntax and type rules of RSL you can describe and develop software in any way that you choose, there are a number of ideas for using RSL that have been found useful and that are collectivelly described as the RAISE Method.

The RAISE Method is based on a number of principles:

- *Separate development*

  To develop systems of any size, we must be able to decompose their description into components and compose the system from the developed components. But, we will need a contract between the developers and the users. For the developer a contract will say what he must provide; for the users, what they may assume.

A specification of a module, or group of modules, can act as this contract, as the specification says precisely what the essential properties of the thing being specified are.

- *Step-wise development*

  It is important to be able to develop software in a sequence of steps. Then we can start with a suitable abstraction, decide what are the main design decisions we need to make, and plan the order in which to do them. Among typical design decisions we can mention giving concrete definitions for abstract types, providing explicit definitions for values previously given only signatures or implicit definitions or axioms, adding new definitions or axioms, and so on. Dealing with one or more such decisions means we make a development step. It is important to be able to make only one, or at least a few, design decisions in each development step in order to deal with one problem at a time.

- *Invent and verify*

  It is a style that forces the developer to invent a new design to later verify its correctness.

- *Rigour*

  It is impractical to prove everything, given the current state of theorem provers. Then, it can be necessary to select the properties worthy of closer investigation, and to formally prove only those we suspect. But while investigating some property in part informally we should also note down the argument, explaining why we think is true, as part of the documentation. An argument that may be wholly or partly informal is called a justification. Arguments that contain informal steps are termed rigorous. A justification that is completely formal is a proof.

A method consists essentially of procedures to be followed and techniques that facilitates the procedures. Most of the RAISE Method consists of techniques for four major procedures, which are detailed described in [22]:

- **specification** starts with identified requirements, written mostly in natural language, and produces a description in RSL generally structured in modules. The output of this procedure is often referred to as the initial specification, not because it is the first one written but because it is the basis for more detailed specifications produced during development. This initial specification should define what the system is to do rather than how it is to do it.

- **development** starts with the initial specification and produces a new, more detailed RSL specification, the final specification, that conforms to the original and that is ready for translation.

- **justification** is an argument showing the truth of some condition. Such an argument can be totally formal or it can be constructed more informally indicating how formal proofs could have been constructed. In this last case, arguments are called rigorous. The typical scenario for doing a justification is that the developer starts from a given condition whose truth should be justified.

- **translation** begins with the final specification in RSL and produces a program or collection of programs in some executable language.

3.2.1 Choice of specification style

There are four main alternatives in the styles of writing specifications:

- **applicative sequential:** a "functional programming" style with no variables or concurrency
- **imperative sequential:** with variables, assignment, sequencing, loops, etc., but with no concurrency
- **applicative concurrent:** functional programming but with concurrency
- **imperative concurrent:** with variables, assignment, sequencing, loops, etc. and concurrency.

Applicative concurrent specifications are often inappropriate as the basis for programming language implementations, as the main processes are recursive in structure. From the remaining three the applicative style is the easiest both to formulate and to reason about in justifications. Then, it is easy to start with applicative specifications and develop them later into imperative or concurrent ones.

We can also distinguish between abstract and concrete styles. In abstract specifications we leave as many alternative development routes open as possible. The following are the options, which by no means are absolute ones as, for example, a module may be abstract in some ways and concrete in others:

- **abstract applicative** modules use abstract types and signatures and axioms rather than explicit definitions for some or even all functions.
- **concrete applicative** modules use concrete types and contain more explicit function definitions.
- **abstract imperative** modules do not define variables but use **any** in their accesses and use axioms.
- **concrete imperative** modules define variables and contain more explicit function definitions.
- **abstract concurrent** modules do not define variables or channels but use **any** in their accesses and use axioms.
- **concrete concurrent** modules define variables and channels and contain more explicit function definitions.

Usually the first specification is an abstract, applicative and sequential one, which is later developed into a concrete specification, initially still applicative and then, imperative and sometimes concurrent.

3.2.2 Writing the initial specification

It is the most critical task in software development, because if it fails to meet the requirements, the following work will be largely wasted.

The main problem at the start is understanding the requirements. Generally, requirements are set in some domain in which we are usually not experts, while the people who wrote them tend to forget to explain what to them is obvious. Besides, as requirements are written in natural language they are likely to be ambiguous. They are also developed by several people over a period of time, and thus they are often contradictory.

The aim of the initial specification is to capture the requirements in a formal, precise manner, to obtain a model of what the system will do. In order to check this model we create accurately models what the writer of the requirements has in mind, we should take into account the following suggestions:

- Be abstract: the specification should leave out as much detail as possible.
- Use users' concepts: as the specification should describe the problem, and not its solution, the specification should not refer to concepts like databases, tables, and records. The concepts in the specification should be the same as the users' concepts.
- Make it readable: as specifications are intended to be read by others, we want to make them as readable as possible. The guidelines are very much like those for programming languages: meaningful identifiers, comments, simple functions, modules that are coherent and loosely coupled, etc.
- Look for problems: we should concentrate on the things that appear difficult, strange, or novel, and defer things that are straightforward, so that we can avoid mistakes, or find them quickly.
- Minimise the state: by state of a system (module) we mean the information that is stored, that persists between interactions with it. In order to make state information minimal, we should try hard not to include in the state dependent information, i.e. information that can be calculated from other information in the state. For example, if C can be calculated from A and B, then we should not model C as part of the state. If we store C, together with A and B, we will need a consistency condition that what is stored for C is the same as would be calculated from stored A and B. There is a general notion that the simpler the set of consistency conditions needed, the better the state is designed. However, in a later stage of development we may decide to store C to achieve sufficient speed.
- Identify consistency conditions: though we try to minimise state information, it is still usually the case that we need consistency conditions and policy conditions. Consistency conditions are needed if some possible state values cannot correspond to reality, for example, two users of a library borrowing the same copy of a book simultaneously. Policy conditions are the ones that might perhaps arise in reality, but we intend that they should not happen, as for example a user borrowing too many books at one time. Preserving consistency conditions is more critical for the healthiness of our system than keeping within policy.

Consistency requirements should be identified first because sometimes it is possible to design a state that will reduce the need for consistency conditions. For example, sometimes consistency may be dealt with by a subtype (we can record the number of books someone can borrow as a **Nat** to prevent it from being negative), while others it would be better to define a function expressing it, as when consistency requirements involve more than one module.

Policy conditions are generally separated from consistency. States that violate policy requirements are possible in the real world, and then if our system is to faithfully model the real world, it must also allow them.

3.2.3 Modules

As was stated in Section 3.2 separate development is one of the principles the RAISE method is based on. When developing systems of any size, we must be able to decompose their description into components and compose the system from the components. Moreover, for most systems it may be necessary to have different people working on different components at the same time.

Modules are the means to decompose specifications into comprehensible and reusable units. As we defined in Section 3.1.1, a module can be a scheme or an object. A scheme is a named class expression and an object is a named model chosen from a class of models represented by some class expression. Objects may be embedded, i.e. defined inside a class expression, or global, i.e. not defined as a scheme parameter or within a class expression. Embedded objects are used wherever possible because they make the objects visible only in the class expression within which they are defined and, if not hidden, to other users of the scheme or object defined using that class expression. Schemes may be parameterised with objects.

A separately developed component module can be used in other modules in esentially three ways. If the module is a scheme, it can be used in a formal parameter or to make an embedded object. If the module is a global object, its name can be mentioned in qualifications. As a result, all mentions of the entities defined in the module will be qualified, by the name of the formal parameter, by the name of the embedded object, or by the name of the global object respectively. Schemes and global objects form a space of names that may potentially be used in modules. To provide some control over visibility and hence dependency, a context clause indicates those that may actually be used. More precisely, any name in the transitive closure of the context and the context's contexts may be used.

Global objects are declared at the top level, in a separate file. Though in general they are not advised because they have a too wide scope, they are defined to contain a collection of types that we need to use in many places. Types such as dates and periods are candidates to be defined in global objects as well as types that should be visible to users, i.e. types that occur as parameters to user functions or in the results of user functions.

Most modules will contain a type modelling (a part of) the state, together with functions to observe and generate values of the state. The type is often called the type of interest of the module. Such modules are usually defined as schemes, and typically instantiated as embedded objects within others.

For example, to model the collection of books in a library we define one module (a scheme) with type of interest Books and another one with type of interest Book, and we use the scheme BOOK to make the object B in the scheme BOOKS. The context clause of the module BOOKS contains the scheme BOOK, used to define the embedded object B.

```
scheme BOOK =
  class
    type
      Book
  end
```

```
context: BOOK
scheme BOOKS =
  class
    object
      B: BOOK
    type
      Books = Book_id $\overset{\sim}{\to}$ B.Book
  end
```

The RAISE Method encourages the use of embedded objects or global objects for expressing the dependency of a module on others. A dependent module is called a client and the modules it instantiates within it or mentions are called suppliers.

Modules are hierarchically structured in order to make possible to understand a particular component by reference only to it and its suppliers, to limit the effects of changes to a module to it and its clients, and to limit the properties of a module to it and its suppliers. To achieve these aims each module should have only one type of interest, clients should only extend their suppliers conservatively, and global objects should only be used with care. Besides, a module A should only mention the entities of a module B if A is a client of B, and clients should only refer to the entities of their immediate suppliers.

3.3 The Tools

UNU/IIST has produced a portable type checker, rsltc, for the RAISE Specification Language. The type checker is portable across Unix and PC platforms and is available free from UNU/IIST's web site (www.iist.unu.edu).

There is also a collection of related tools all based on the type checker. We briefly describe them below. A complete description of all the tools as well as how to install them on Unix, Linux and Windows platforms can be found in [19].

- **Type checker:** type checking is performed on context files first, followed by the input module mentioned in the command. The tool outputs the names of the modules it is checking, and if it finds errors it also outputs the corresponding messages.
- **Pretty printer:** provided there are no syntax errors, a pretty-printed version of the input module is output on standard output.
- **Confidence condition generator:** confidence conditions are conditions that should generally be true if the module is not to be inconsistent, but that cannot in general be

determined as true by a tool. The complete list of the conditions that can be generated by the tool can be found in [19].

- **Showing module dependencies:** they are shown in a simple ASCII representation.
- **Drawing a module dependency graph:** if run on a file X.rsl this generates input for the Visualisation of Computer Graphs (VCG) tool in a file X.vcg. Schemes are drawn as red ellipses, objects as blue rectangles, theories as yellow diamonds, and development relations as cyan triangles. The graph can be exported as a graphic file in a various formats for printing or use in documents.
- **SML translator:** it maps a specification in RSL to the functional programming language Standard ML [1], giving as result a first prototype of the specification. Only a subset of RSL is accepted by the translator.
- **C++ translator:** it produces an automatic translation of a RSL specification into C++. A similar subset of RSL to the SML translator is accepted.

Chapter 4

The three-step process

As we defined in Section 3, a specification in RSL is a collection of modules, where a module is basically a named collection of declarations. Usually the first specification is an abstract, applicative and sequential one, which is later developed into a concrete specification, initially still applicative and then, imperative and sometimes concurrent. A typical applicative module contains type and value (constants and functions) definitions, and probably some axiom definitions too.

When using the RAISE Method, writing the initial RSL specification is the most critical task in software development, because this specification must capture the requirements in a formal, precise way [20]. But, domains are naturally informal as they reside in the real world. Then, at the beginning of the software development process it would be better to use some kind of informal representations, such as natural language, to allow stakeholders to participate actively in the requirements definition process [44].

To bridge the gap between these two worlds, we propose a technique to derive an initial formal specification in RSL from requirements models, such as LEL and scenarios which are closer to stakeholders language. The derivation of the specification is structured in three steps which show how to derive RSL types and functions, and how to structure them in modules using the information provided by the LEL and the Scenario Model. We call the steps Derivation of Types, Derivation of Functions, and Definition of Modules. They are not strictly sequential; they can overlap or be carried out in cycles. For example, function definitions can indicate which type structures are preferable.

The Derivation of Types step produces a set of abstract as well as concrete types, which model the relevant terms in the domain. During this step, the LEL is the main source of information. We perform the derivation of the types in two steps. First we identify the types, and then we decide how to model them. Most of the types derived in the Identification step will be abstract types, and many of them will be replaced by more concrete ones in the Elaboration step. This way of defining types follows one of the key notions of the RAISE Method: the step-wise development (Section 3.2).

The Derivation of Functions step gives as result a set of functions that model the functionality in the application domain. The heuristics we propose help to identify and to model the functions, by showing how to derive arguments and result types of functions, how to classify functions as partial or total, and how to define function bodies by analysing descriptions of scenarios. As scenarios are natural language descriptions of the functionality in the domain, the Scenario Model plays a significant role in this step.

The Definition of Modules step helps to organise types and functions in RSL modules, as modules are the means to decompose specifications into comprehensible and reusable units. As we described in Section 3.2.3, the RAISE Method proposes to structure modules hierarchically in order to make possible to understand a particular component by reference only to it and its suppliers, to limit the effects of changes to a module to it and its clients, and to limit the properties of a module to it and its suppliers. The decomposition into modules is particularly useful when designing complex systems, because it facilitates and encourages separate development, one of the principles the RAISE Method is based on.

In the following sections of this chapter, we describe in detail each of the three steps mentioned above. As we have applied this three-step process to a complete case study, the Milk Production Systems domain, most of the examples we will use to show the application of the heuristics in each step will come from this domain. However, we will include examples from other case studies [13, 42] when the domain we selected does not provide appropriate examples.

4.1 Derivation of Types

A type is a collection of logically related values, and it may be specified by an abstract or a concrete definition, as we defined in Section 3.1.2. An abstract type, also referred to as a sort, has only a name while a concrete one can be defined as being equal to some other type, or using a type expression formed from other types.

There is a standard piece of advice in specification that you do not choose a design until you have to [20]. Abstract types are the mechanism to define a type we need but whose definition we have not decided on yet. As we explained in Section 3.1.2, they are typically used in two situations: when defining simple types, such as identifiers for people, books in a library, and cows in a farm that we expect to implement easily in the final program, and when working with complicated types whose designs are not known yet. In the last situation, using an abstract type provides a way to delay the design until it is clear enough.

Following this piece of advice, we define a set of heuristics to derive the types of an initial RSL specification of a given domain, starting from the LEL and the Scenario Model. We perform the derivation of the types in two steps. First we identify the types, and then we decide how to model them. Most of the types derived in the Identification step will be abstract types, and many of them will be replaced by more concrete ones in the Elaboration step. This way of defining types follows one of the key notions of the RAISE Method: the stepwise development. The replacement of an abstract type by a more concrete one follows the implementation relation. Implementation is very important because if an initial specification meets the requirements and all its developments follow the implementation relation, then they all meet the requirements.

Sections 4.1.1 and 4.1.2 present the heuristics to identify the types of the RSL specification and to model them respectively.

4.1.1 Identification of Types

The main goal of this step is to determine an initial set of types that are necessary to model the different entities present in the analysed domain. This initial set will be completed, or even modified, during the remaining steps of the specification derivation. For example, during the Definition of Modules step may be necessary to define a type to reflect the domain state.

Also, when defining functions may be useful to define some new types to be used as result types of functions.

The LEL is the source of information during this step as LEL subjects and some objects represent the main components or entities of the analysed domain. In general, LEL subjects and objects will correspond to types in the RSL specification. In some cases, LEL verbs may also give rise to the definition of more types, as when they represent an activity which has its own data to save. In order to define just the relevant types, we suggest some heuristics which are summarised in Table 4.1, and explained later in detail. The prefix HIT, used to distinguish each heuristic, means Heuristics for the Identification of Types.

HIT1: Types coming from subjects/objects whose name is a singular noun

A subject/object whose name is a singular noun may correspond to one of two different kinds of domain components: those with only one instance or those which are elements of a collection, i.e they have more than one instance. In any case, we model the subject/object as an abstract type and then, we must only provide its name.

```
type
   Symbol_name
```

where Symbol_name comes from the name assigned to the subject/object in the corresponding LEL entry.

It is frequent that subjects representing an organisation are candidates to have only one instance. For example, the subject Administrator, which stands for an enterprise in the Saving Plan for Automobile Acquisition System [14], has one instance. Objects describing places, such as the object Library in a Library System [12] or the object Dairy farm in the Milk Production System, are also candidates to have only one instance. However, when subjects/objects may have more than one instance they represent each element in the corresponding collection. This happens, for example, with objects such as Cow and Field.

Then, for the Milk Production System we may define

```
type
   Dairy_farm,
   Cow,
   Field
```

as abstract types which may be developed later.

This first definition of types could be even refined a bit more in somes cases, as we show in the following heuristics.

HIT1.1: Types coming from objects defining computable properties

LEL symbols classified as objects may also represent some property of another object or subject in the LEL, computable from some other properties of the object or subject. In some cases, the LEL contains a verb symbol in which the way to compute this property is defined. Moreover, this verb symbol may have a corresponding scenario where more details are given.

HITid	**LEL symbol**	**RSL type**	**RSL specification**
HIT1	Subject/object name is a singular noun	Abstract type	Symbol_name
HIT1.1	Object representing a computable property	Abstract type	Property_name
HIT1.2	Subject/object name is a noun, also a symbol in the LEL, modified by a phrase:		
HIT1.2.1	If it represents a category, state or situation	Subtype expression (though not always)	Main_type, /∗ already defined ∗/ Subtype = {| s: Main_type • is_subtype(s) |}
HIT1.2.2	If it represents a different subject/object	Abstract type	Symbol_name
HIT2	State		
HIT2.1	If name references a symbol in the LEL	Subtype expression (though not always)	Main_type, /∗ already defined ∗/ Subtype = {| s: Main_type • is_subtype(s) |}
HIT2.2	If name does not reference a symbol in the LEL	Abstract type	State_name
HIT3	Verb represents an action with data to save	Abstract type	Verb_name
HIT4	Symbol name is a plural noun or symbol is an element of a collection:		
HIT4.1	If instances have an attribute or set of attributes for identification	Map type expression	Sym_id, Sym_name, /∗ already defined ∗/ Map = Sym_id $\overrightarrow{m}$ Sym_name
HIT4.2	If instances need an ordering	List type expression	Sym_name, /∗ already defined ∗/ List = Sym_name*
HIT4.3	Otherwise	Set type expression	Sym_name, /∗ already defined ∗/ Set = Sym_name-**set**

Table 4.1: Heuristics to identify RSL types

If the verb symbol does not exist, the behavioural response or the notion of the LEL symbol defining the property indicates how to calculate it.

The RAISE Method recommends to minimise state information, as we have already explained in Section 3.2.2. This means that we should try to avoid including in the state dependent information, that is information that can be calculated from other information in the state. Then, following this recommendation, we decide to model such deducible properties with a function. However, we suggest to define a type to be used as the function result type. We continue this discussion in Section 4.1.2.

```
type
   Property_name
```

For example, the object Individual Production is a property of the objects Milking cow, Group and Dairy farm which can be computed as established in the verb symbol Compute Individual production and in the scenarios Compute milking cow individual production, Compute group individual production, and Compute Dairy farm individual production. Another example is the object Hectare_loading, a property of a field which can be computed dividing the number of cows in a field by the size of the field in hectares. Although we will define functions to model each deducible property, as suggested by the heuristic, we define one type for each of them to be used as the corresponding function result type.

```
type
   Indiv_prod,
   Hectare_loading
```

HIT1.2: Types coming from subjects/objects whose name is a noun, also a symbol in the LEL, modified by a phrase

When the name of the subject/object is composed of a noun, which is a subject/object in the LEL, modified by some phrase, for example an adjective, it may correspond to a category of the symbol referred to by the noun, a state or situation in which the subject/object could be, or even a different subject/object. In the first two cases, we propose to consider the definition of a subtype (HIT1.2.1). However, in the last case it is necessary to define a new type to reflect that the subject/object is a different one (HIT1.2.2).

Subtypes may be useful to capture a particular concept, and also to define as total functions that would be partial on any larger subtype. Then, if the decision is to define a subtype the specification will look like:

```
type
   Main_type,
   Subtype = {| s: Main_type • is_subtype(s) |}
```

where is_subtype(s) is a predicate (boolean function) defined to constrain the main type. For example, the object term Pregnant Cow represents a possible state for a dairy cow or a heifer, and then it could be modelled as a subtype if necessary.

type
  Dairy_cow,
  Pregnant_dairy_cow = {| dairy_cow: Dairy_cow • is_pregnant(dairy_cow) |}

Another example, taken from the Library System [12], is the object symbol Book with red label which is composed of the noun Book, an object in the LEL, modified by a phrase. Book with red label is a category of Book. Then, following the heuristics we have just proposed, we could model the object Book with a type, and the object Book with red label as a subtype of Book. Overdue book is a possible state for a book, and thus, it could also be modelled as a subtype if necessary.

type
  Book,
  Book_red_label = {| b: Book • has_red_label(b) |},
  Overdue_book = {| b: Book • is_overdue(b) |}

But a different case is, for example, the one appearing in the Meeting Scheduler System [34] with the symbol Possible Meeting. Even though Meeting is a symbol in the LEL modified by an adjective, Possible Meeting has a different semantics, making necessary the definition of a different type, independent of the type defined for Meeting.

type
  Meeting,
  Possible_Meeting

HIT2: Types coming from state symbols

This heuristic is closely related with the previous one, as generally a symbol classified as a state may define a situation or state in which a subject/object in the LEL could be. Moreover, the symbol name may be composed of a LEL subject/object name modified by a phrase. If this is the case, we consider the definition of a subtype and we proceed as explained in heuristic HIT1.2.1. But, if the symbol name has no reference to any other symbol in the LEL, we simply define an abstract type (HIT2.2) which might be developed later during the Elaboration of Types Step.

type
  State_name

For example, for the LEL symbol Pregnant we define the following abstract type

type
  Pregnant

HIT3: Types coming from verb symbols

Symbols classified as verbs should also be analysed. It is frequent that a verb represents an action or activity which has its own data to save. We specify verb symbols of this kind using types, where the type models the data to be saved.

```
type
   Verb_name
```

where Verb_name comes from the name assigned to the verb symbol in the corresponding LEL entry. This abstract type will be later replaced by a concrete one which models the characteristic data of each activity or action.

For example, the verb term Vaccinate cow/Vaccination has its own attributes like the date and the vaccine given to the cow. The verb Milk a cow/Milking implies saving the date and the quantity of litres of milk extracted from a milking cow. Both verb symbols represent actions performed on cows. Then, we could define

```
type
   Milking,
   Vaccination
```

HIT4: Types coming from symbols defining each element of a collection or whose name is a plural noun

Symbols whose name is a plural noun generally define a collection of some component of the analysed domain. However, as we have pointed out, a LEL symbol identified with a name in singular may represent each element of a collection. It is common practice not to include in the LEL symbols defining a collection of another LEL symbol when the actions that could be applied to the collection are the classical ones such as adding, removing or recovering elements. This means that when we consider a subject, object or verb modelling an activity with its own data to save we should find out if it may have more than one instance in order to model the corresponding collection.

Collections can be defined in RSL using map, list or set type expressions (Section 3.1.2). In many cases it is possible to find or create one attribute or set of attributes that identify unambiguously each instance of a subject/object. Then, one good alternative is to specify the collection as a map, involving the definition of three types: one for the map domain, one for the values in the range of the map, and another one for the map itself (HIT4.1). In addition, many common operations applied to collections like adding, removing or recovering elements, correspond closely to map operators thus, reducing the number of functions to be defined to manipulate the collection.

```
type
   Symbol_id,     /* map domain */
   Symbol_name,   /* values in the range of the map */
   Map_name = Symbol_id →m Symbol_name    /* the map */
```

The first two types, defined as abstract ones, may be replaced by a concrete definition later in the Elaboration of Types step (Section 4.1.2). The type Symbol_name might be already defined if the heuristic HIT1 had been applied before.

When there is some order to be maintained among the elements of the collection, we choose a list expression, involving the definition of two types, one to represent each element in the list and another one for the list itself (HIT4.2).

type
  Symbol_name,   /$*$ element of the list $*$/
  List_name = Symbol_name* /$*$ the list $*$/

Finally, if the symbol instances do not have an attribute or a set of attributes to identify them, and there is no need of an ordering among the instances, we select a set expression. Working with set expressions implies the definition of two types, one to represent each member of the set and another one for the set itself (HIT4.3).

type
  Symbol_name,   /$*$ member of the set $*$/
  Set_name = Symbol_name-**set**   /$*$ the set $*$/

When specifying the collection as a list or a set, the type Symbol_name is an abstract definition for each element in the list or each member of the set respectively that might be already defined if the heuristic HIT1 had been applied before. These abstract types may be developed later in the Elaboration of Types step (Section 4.1.2).

For example, the subject Dairy farmer may have more than one instance, so we need to model the collection of dairy farmers. Besides, it is possible to determine an attribute that identifies each instance of a dairy farmer. Thus, we specify the collection of dairy farmers using a map expression. Three types are defined, one for the map domain, one to contain the information relevant to each dairy farmer, and another one for the collection of dairy farmers.

type
  Dairy_farmer_id,
  Dairy_farmer,
  Dairy_farmers = Dairy_farmer_id $\underset{m}{\rightarrow}$ Dairy_farmer

In the same way, the objects Field and Cow may have more than one instance in the domain and it is possible to define an attribute that distinguishes each of their instances. So, we model each of them with a map involving three types.

type
  Field_id,
  Field,
  Fields = Field_id $\underset{m}{\rightarrow}$ Field,
  Cow_id,
  Cow,
  Cows = Cow_id $\underset{m}{\rightarrow}$ Cow

The types Dairy_farmer_id, Dairy_farmer, Field_id, Field, Cow_id, and Cow, as yet defined as abstract types by applying the heuristic HIT1, may be replaced later by more concrete ones as we show in Section 4.1.2.

We made a similar analysis to discover that the objects Bull, Group and Plot may also have more than one instance in the domain and thus, they can also be modelled using maps, involving the definition of three types each.

For vaccinations and milkings, which come from verb symbols, ordering by date is natural. Then, for each of these verb symbols we define two types:

type
  Vaccination,
  Vaccinations = Vaccination*,
  Milking,
  Milkings = Milking*

The types Vaccination and Milking, as yet defined as abstract types, will contain the characteristic data of each activity.

4.1.2 Elaboration of Types

The result of the previous step is a preliminary list of types, many of them abstract, which specify subjects, objects and activities taken from the application domain. In order to remove under-specification [22], we propose to return to the information contained in the LEL and the Scenario Model. In particular, the analysis of the notion, and sometimes the behavioural response, of each symbol that motivated the definition of an abstract type, can help to decide if the type could be developed into a more concrete type. As we have already mentioned, all the developments we present satisfy the implementation relation. Table 4.2 summarises the heuristics we propose, which are then explained in detail. The prefix HDT, used to distinguish each heuristic, means Heuristics for the Development of Types.

We want to remark that during this step, it might be necessary to introduce some type definitions that do not correspond to any entry in the LEL. They appear, in general, when modelling components of some other type. Symbols without an entry in the LEL may represent an omission or a symbol considered outside the application domain language. When an omission is detected, it is necessary to return to the LEL to add the new definition, and update the Scenario Model to maintain the consistency between its vocabulary and the LEL itself. We return to this issue in Section 5.6.

HDT1: Development of types coming from subject/object symbols

To give more concrete definitions for the abstract types identified applying heuristics HIT1 and HIT2, we propose to analyse subject/object notions as well as objects behavioural responses.

- **HDT1.1: The notion contains one or more properties of the symbol**

  Notions written as "It/he/she has ..." suggest a property of the symbol that may be modelled as an attribute. Then, we define a record type containing as many components as properties identified.

HDTid	**Type comes from**	**RSL type**	**RSL specification**
HDT1	Subject/object symbol:		

HDT1.1	Notion contains one or more properties of the symbol	Short record definition	Sym_name::
			  prop_1: Prop_type_1
			  ...
			  prop_n: Prop_type_n
			  /∗ n >0 ∗/

HDT1.2	Notion contains a deducible property of the symbol	Simple type	Prop_name = Type_expression /∗ Remove the property from the record ∗/
HDT1.3	Notion represents symbol state or category	Variant type	Categ == cat_1 | ... |cat_n /∗ (n >0) ∗/

HDT1.4	Categories or states share some attributes	Variant type
		Short record definition
		Subtypes
		(if necessary)
			Categ == cat_1 | ... |cat_n,
			Main_type::
			  common_attr_1: Attr_type_1
			  ...
			  common_attr_m: Attr_type_m
			  distinguishing_attr: Categ,
			St_Cat_1 = {| mt : Main_type •
			    has_cat_1(mt) |},
			...
			St_Cat_n = {| mt: Main_type •
			    has_cat_n(mt) |}
			/∗ (n, m > 0) ∗/

HDT1.5	Object behavioural response suggests a property of the symbol		/∗ In general, model the property as part of the object∗/
HDT2	Verb describing an action applied to an object		/∗ In general, model the action as part of the object∗/

Table 4.2: Heuristics to develop identified types

```
type
    Property_type_1,
    ...
    Property_type_n,
    Symbol_name ::
     property_1 : Property_type_1
     ...
     property_n : Property_type_n
```

for n $>$ 0.

We also want to remark that Property_1, ..., Property_n might have been already defined if any of them came from a symbol in the LEL, and if so they had to be considered when applying the heuristics for the Identification of Types.

In case only one property could be identified from the notion, a record with only one field may have no sense. As the formal specification derived with our technique is an initial one it will be refined and modified later. Then, we would choose to define the record anyway, leaving to the software engineer the decision of removing it later. But, if we are quite sure the notion defines the subject/object with only one property, we can either set the specification of the abstract type coming from the subject/object equals to the type which corresponds to the property, or even leave it as an abstract type. For example, from the object Identification number we can identify only one property, more precisely the one that defines an identification number. So we can give a concrete definition for an identification number, or we can defer it for a future refinement.

If the collection for the symbol also exists and it has been specified with a map type expression, the properties that correspond to the attributes used to define the map domain are not included as componentes of the record. However, if the collection was modelled with a list or set expression all the properties identified are included as components of the record.

For example, a first development for the abstract type Field coming from the LEL object Field and identified following the heuristic HIT1 from Section 4.1.1, could be

```
type
    Location,
    Size,
    Pasture,
    Hectare_loading,
    Plots,
    Field::
        location: Location
        size: Size
        pasture: Pasture
        hectare_loading: Hectare_loading
        plots: Plots
        past_plots: Plots
```

It is worth remarking that as the collection of fields was defined using a map expression (Section 4.1.1), the property Field_id used to define the map domain, is not included as a record component.

- **HDT1.2: The notion contains a deducible property of the symbol**

  As we have already explained in Section 4.1.1 a property that can be deduced is defined as a function, following the general principle of excluding from the specification what can be computed. Storage of what can be computed implies redundancy, and hence consistency conditions. Then, we remove any component of a record representing a computable property. However, any of them may be added later as a refinement if necessary for efficiency.

  In the previous example, it is not necessary to include hectare_loading as a component of the type Field, because it can be computed from the number of cows in the field and the size of the field, as explained in the notion of the LEL symbol Hectare loading, and so we will model it with a function. Then, we remove the component hectare_loading from the record Field because it will be computed any time it is needed. The new definition for the type Field is as follows

```
type
   Location,
   Size = Real,
   Pasture,
   Hectare_loading = Real,
   Plots,
   Field::
      location: Location
      size: Size
      pasture: Pasture ↔ chg_pasture
      plots: Plots ↔ chg_plots
      past_plots: Plots ↔ chg_past_plots
```

  where we have replaced abstract definitions for the types Size and Hectare_loading by concrete ones, and we have added the reconstructors chg_pasture, chg_plots, and chg_past_plots, to indicate these components of the record Field may be modified.

  Another example is the property individual production which can also be computed, as we have already explained when defining the heuristic HIT1.1. We have suggested the definition of three functions, one to compute a dairy cow individual production, another for a group individual production, and another one for the dairy farm. So, this property is not included as a component of the types Cow, Cow_group and Dairy_farm, at least in this step of the specification. But, the previous abstract definition is replaced by the following concrete one:

```
type
   Indiv_prod = Real
```

- **HDT1.3 and HDT1.4: The notion represents a state or category of the symbol**

  An entry in the notion containing the verb “may be” suggests the possibility of different states or categories for the subject/object. In such cases, we define a variant type to model the property (HDT1.3).

```
type
    Category == Categ_1 | ... | Categ_n
```

  for n > 0.
  To include all the alternatives in this variant type definition, sometimes may be necessary to analyse the rest of the entries in the notion because they may be expressed separately.

  In addition, it is frequent that states and, more commonly categories, share some attributes while differ in others. In this kind of situation, we use a variant type to describe the distinguishing attribute (the state or category), and a record type to include the common attributes plus the distinguishing one. When it is useful, subtypes can also be defined to represent each alternative appearing in the variant type definition (HDT1.4).

```
type
    Attr_type_1,
    ...
    Attr_type_m,
    Category == Categ_1 | ... | Categ_n,
    Main_type ::
       common_attr_1: Attr_type_1
       ...
       common_attrib_m: Attr_type_m
       distinguishing_attr: Category,
    St_Categ_1 = {| mt : Main_type • has_categ_1(mt) |},
    ...
    St_Categ_n = {| mt: Main_type • has_categ_n(mt) |}
```

  for n, m > 0.
  In case any of the different alternatives specified for the variant type has particular attributes, these attributes could be modelled as components of the variant type.

  For example, the notion of the object Calf says a calf may be either male or female. Neither male nor female are defined as entries in the LEL. We define a new variant type Calf_gender to reflect this two possibilities. We add the component photo to register that a female calf has a picture, as indicated in the notion of the object Calf.

```
type
    Photo,
    Calf_gender == male | female(photo: Photo)
```

Similarly, the notion of the LEL symbol Cow says that a cow may be a calf, a heifer or a dairy cow. But this example is a bit different from the previous one because calf, heifer and dairy cow have an entry in the LEL. Moreover, these definitions show that, even though calves, heifers and dairy cows share some attributes because they are defined as cows, each of them has some special features. The category of the cow is the distinguishing attribute. We model it with a variant type, and we include the particular attributes of each alternative as components of the variant type. We define the type Cow as a record whose fields are the common attributes, like date of birth, and the distinguishing attribute. Then, we introduce the following definitions to model a cow and its three different categories, each of them with its particular information. The definition of the type Cow is still not finished.

```
type
   Calf_info,
   Heifer_info,
   Dairy_info,
   Cow_classif ==
      calf(info: Calf_info) |
      heifer(info: Heifer_info) |
      dairy(info: Dairy_info),
   Date,
   Cow::
      birthday: Date
      cow_classif: Cow_classif
```

Sometimes only the fact that a subject/object can be in a state is indicated, but the complementary one is implicit. This case is also specified with a variant type containing the explicit alternative as well as the implicit one.

For example, the notion of the object Dry cow says that a dry cow may be a pre-birth cow, leaving implicit that if not, it will be a non pre-birth cow. Then, we model this property of dry cows as follows

```
type
   Dry_classif == pre_birth | non_pre_birth
```

- **HDT1.5: An object behavioural response suggests a property of the symbol**

  In the case of LEL objects, the behavioural response should also be analysed since attributes can appear as a result of operations applied to the object. The behavioural response of an object contains the actions that may be applied to the object. Sometimes, each of these actions is described with a verb phrase which is an entry in the LEL. If not, the behavioural response itself might contain a description of the action. Frequently, it is necessary to record some results of applying the actions to objects, and so we should define the information we want to save and where to store it. For example, the behavioural response of the object Cow establishes that a cow is vaccinated. The verb symbol Vaccinated is an entry in the LEL, where it is written that some information about each vaccination is registered, as the date and the vaccine injected. One possibility would be to add an attribute to the type defining cows to contain all the vaccinations. In the same way, each of the different operations applied to cows could be modelled.

In general, when modelling actions applied to objects, there are two possibilities for saving the corresponding information: one is to save it as part of the corresponding object, and the other one is to put together the results corresponding to the application of the action to all the objects. In the next section, we analyse advantages and disadvantages of each alternative, and we propose some heuristics to follow.

HDT2: Development of types coming from verb symbols

To determine how to model types coming from actions, we analyse the corresponding verb symbol in order to find the data to be recorded, and to decide where to store the information. These verb symbols describe actions applied to LEL objects, and then one possibility is to model the result of the action as part of the type defining the object in the specification. The other possibility is to store together the results of an action applied to all the occurences of the LEL object under consideration. As we show in the following example, the types used to model the object and the actions applied to the object should be carefully analysed in order to find a solution that minimises the number of consistency conditions to be defined.

To continue with the example introduced in Section 4.1.1, we define the following types to represent all the vaccinations belonging to a cow as part of the definition of the cow.

```
type
    Date,
    Vaccine,
    Vaccination::
        date: Date
        vaccine: Vaccine,
    Vaccinations = Vaccination*,
    Cow::
        ...
        vaccinations: Vaccinations
        ...
```

Another possibility would be to store together the vaccinations belonging to all the cows in the dairy farm. If this were the case, each vaccination should also contain a reference to the corresponding cow, and so the Cow_id component should be added to the Vaccination component. Then, the types would be defined as follows

```
type
    Cow_id,
    Date,
    Vaccine,
    Vaccination::
        cow_id: Cow_id
        date: Date
        vaccine: Vaccine,
    Vaccinations = Vaccination*
```

However, it is important to remark that this last alternative is not as good as the previous one for two main reasons. As we defined in Section 4.1.1, the type Cow_id represents the domain of the map Cows, and thus it is a key to access each cow in the map. A key inside a list suggests that a map having the key as domain and the rest of the attributes as range would be better, i.e.

```
type
    Cow_id,
    Date,
    Vaccine,
    Vaccination::
        date: Date
        vaccine: Vaccine,
    Vaccinations = Cow_id $\underset{m}{\rightarrow}$ Vaccination$^*$
```

The second reason is that if there are two maps with the same key type, there is typically a consistency condition that they have the same domain.

In this case we have the map type Vaccinations and we also have a map describing cows from Cow_id to Cow. If we merge these two maps the consistency condition will be guaranteed "by construction".

Therefore, we chose the first alternative, and we model all the actions involving cows as part of the Cow type. The new definition for the type Cow is then

```
type
    Calf_info,
    Heifer_info,
    Dairy_info,
    Cow_classif ==
        calf(info: Calf_info) | heifer(info: Heifer_info) | dairy(info: Dairy_info),
    Date,
    Vaccine,
    Vaccination::
        date: Date
        vaccine: Vaccine,
    Cow::
        birthday: Date
        cow_classif: Cow_classif
        vaccinations: Vaccination$^*$
        ...
```

The ellipsis ... represent the remaining actions applied to cows such as milkings, deparasitations, and inseminations.

So in general, when developing types coming from verb symbols we suggest modelling the information necessary to be saved as part of the corresponding object.

More heuristics for the development of types

Another thing to consider when defining attributes to give a more concrete definition of a type, is the existence of one to n relationships. Usually, these kind of relationships are cross-referenced between symbols in the LEL. For example, the notion of the symbol Group defines a group as a set of calves, heifers or dairy cows. On the other hand, the notion of the symbols Calf, Heifer and Dairy cow say that each of them belongs to only one group at any moment. A first, and still incomplete, definition of the type Group could be

```
type
   Cow_group::
      cows: Cow_id-set
```

Thus, if this definition is adopted we will need a consistency predicate to ensure that a cow can be in only one group at any moment. In addition, it is necessary to check that each cow identification appearing in the set belongs to a cow of the dairy farm. However, if instead of storing the cows in a group, the group is defined as a component of the type Cow, the first consistency predicate can be avoided. But in this case, we need to ensure that the group identification corresponding to each cow is a valid one in the dairy farm. A similar analysis should be made with one to one relationships, which also contain cross-references. In general, it is possible to store the relation on either of its sides defining the appropriate consistency predicates. But sometimes one side would be better than the other. We will show some examples in Section 5.4.1.

4.2 Definition of modules

Modules are the means to decompose specifications into comprehensible and reusable units. This decomposition into modules is particularly useful when designing complex systems, as it eases and encourages separate development, one of the principles the RAISE Method is based on.

As we have already explained in Section 3.2.3, there are some principles to follow when defining a collection of modules to model a system:

- Each module should have only one type of interest, defining the appropriate functions to create, modify, and observe values of the type.
- The collection of modules should be, as far as possible, hierarchically structured. This means that each module below the top should be instantiated in only one another, its parent, as an embedded object, and its functions should only be called from its parent.

In this section we present a set of heuristics which help to organise in modules all the types produced by the Derivation of Types step in order to obtain a more legible and maintainable specification. These modules would be latter completed with the definition of functions in the next step, and probably they will be completed with more type definitions. In defining these heuristics, we followed closely the two principles mentioned above as well as the other features RSL modules should have according to Section 3.2.3. So, we first identify class expressions to define schemes, and then we assemble these schemes defining objects to express dependencies between them. In Table 4.3 we present a summary of the heuristics we propose. The prefix HDM, used to distinguish each heuristic, means Heuristics for the Definition of Modules.

The modules obtained by applying the heuristics we propose can be hierarchically organised to show the system module structure. In addition, this hierarchy of modules can be represented using a layered architecture, as we will show in Section 4.2.3.

4.2.1 Modules coming from class expressions with no type of interest

Class expressions with no type of interest are commonly used to define types that we need in many places. They provide a simple mechanism for sharing them and for changing them if necessary. Class expressions of this category are usually schemes, and they are always instantiated as global objects. So, any time we refer to a type or a function defined in any of these schemes we prefix the type or function name with the name of the corresponding object. Global objects can always be avoided, by the use of parameterisation. However, we use them for commonly ocurring types because it is tedious to have to parameterise all the other modules with them.

In general, all the types used across a specification that must be visible to users are defined in this category of class expression. These are the types used as parameter or result types of top level functions, and they are the basis to define the domain components. Besides, it is common to include types used in at least two modules, as for example the types that define map domains. So, to gather all these definitions of types we define a scheme

```
scheme GLOBAL_TYPES =
  class
    type
      Global_type_1,
      ...
      Global_type_n
  end
```

for n > 0, and then, we instantiate it as a global object so that all the types could be accesible from any module in the specification.

```
context: GLOBAL_TYPES
object GT:GLOBAL_TYPES
```

Then, for the Milk Production System we define the scheme GENERAL_TYPES, which is instantiated as the global object GT. In this scheme, we put the definition of types used as argument or result types of the top level functions, such as the types Cow_classif, and Vaccine. We also include the types that define map domains such as Cow_id, Group_id, Field_id and Plot_id.

```
scheme GENERAL_TYPES =
  class
    type
      Cow_id,
```

HDMid	**Type**	**RSL module**	**RSL definition**

HDM1	For all the types that must be visible to users or used in at least two modules	Two modules:
		a scheme and a global object which is an instantiation of the scheme
			scheme GLOBAL_TYPES =
			class
			  type
			    Global_type_1,
			    ...
			    Global_type_n
			end /∗ n > 0 ∗/
			
			context: GLOBAL_TYPES
			object GT:GLOBAL_TYPES

HDM2	Models an element of a collection	Scheme	**scheme** COLL_ELEM =
			class
			  type
			    Coll_elem
			end

HDM3	Models a collection	Scheme, where the scheme modelling each element in the collection is defined as an embedded object	context: COLL_ELEM
			scheme THE_COLLECTION =
			class
			  object CE: COLL_ELEM
			  type
			/∗ if collection specified with a map ∗/
			    The_Collection =
			      GT.Coll_id $\underset{m}{\rightarrow}$ CE.Coll_elem
			/∗ if collection specified with a list ∗/
			    The_Collection = CE.Coll_elem*
			/∗ if collection specified with a set ∗/
			    The_Collection = CE.Coll_elem-**set**
			end

HDM4	For all the types modelling domain components	One top level module defined as a scheme, with each scheme defining a domain component instantiated as an embedded object	context: DOM_COMP_1, ...,
			DOM_COMP_n
			scheme DOM_STATE =
			class
			  object
			    DC_1: DOM_COMP_1,
			    ...,
			    DC_n: DOM_COMP_n
			  type
			    Dom_state::
			      dom_comp_1: DC_1.Dom_Comp_1
			      ...
			      dom_comp_n: DC_n.Dom_Comp_n
			/∗ n > 0 ∗/
			end

Table 4.3: Heuristics to define modules

```
        Group_id,
        Field_id,
        Plot_id,
        ...
  end
```

```
context: GENERAL_TYPES
object GT: GENERAL_TYPES
```

When the specification is large and if there is a natural division into smaller objects, more than one of this kind of module can be defined.

4.2.2 Modules coming from class expressions with a type of interest

Class expressions with a type of interest are used to specify the main hierarchy of modules. Usually each module is defined as a scheme and instantiated as an object in some module above it. We use this kind of class expression to specify application domain components.

During the derivation of types, we defined a number of maps which represent collections of application domain components, such as cows, fields and dairy farmers. Each of these maps came from a LEL subject or object with more than one instance in the application domain. For example, as the LEL object Cow can have more than one instance, the corresponding collection should also be modelled.

Then, for each collection specified during the derivation of types, we define two scheme modules, one having the type modelling the collection as its type of interest, and the other having the type of the elements in the collection as its type of interest. We use the last one to make an object in the scheme containing the collection.

Schemes and also global objects form a space of names that may potentially be used in modules. To provide some control over visibility and hence dependency, a context clause indicates those that may actually be used. More precisely, any name in the transitive closure of the context and the context's contexts may be used.

```
scheme COLL_ELEM =
  class
    type
      Coll_elem
  end
```

```
context: COLL_ELEM
scheme THE_COLLECTION =
  class
    object CE: COLL_ELEM
    type
    /* if collection specified with a map */
      The_Collection = GT.Coll_id $\overrightarrow{m}$ CE.Coll_elem
    /* if collection specified with a list */
```

```
            The_Collection = CE.Coll_elem*
        /* if collection specified with a set */
            The_Collection = CE.Coll_elem-set
    end
```

In the specification of the map, the prefix GT refers to the global object where the type of the map domain is defined, according to the heuristic HDM1 explained in the previous section. The scheme COLL_ELEM must be always defined in the context clause of the scheme THE_COLLECTION. However, it is possible that later, depending on the final hierarchy of modules, this context clause may have to be modified or completed with the name of other schemes. Moreover, the scheme COLL_ELEM may also need to include a context clause.

For example, we define one module with type of interest Cows and another one with type of interest Cow, and we use the scheme COW to make the object C in the scheme COWS. The context clause of the module COWS contains the scheme COW, used to define the embedded object C. Then, a first, and still incomplete, specification of the modules COWS and COW is as follows

```
scheme COW =
    class
        type
            Cow
    end

context: COW
scheme COWS =
    class
        object C: COW
        type
            Cows = GT.Cow_id $\overrightarrow{m}$ C.Cow
    end
```

Likewise, each of the remaining maps such as Fields, Plots, Bulls, Cow_groups, and Dairy_farmers motivates the definition of two schemes, one of them instantiated as an object in the scheme containing the map type.

Sometimes a scheme needs to be shared between two or more schemes above it, and in this case it is made a parameter of those schemes. For example, COWS is shared by COW_GROUPS and DAIRY_FARM; then as we will show later in Section 5, we made it a parameter of the scheme COW_GROUPS.

Finally, it is also necessary to define a top level module having as type of interest the type which is the state of the system or application domain specified. In some cases, the LEL contains the definition of a symbol that concentrates information about the application domain being modelled, listing the main components of the domain. When it exists, this symbol can help in the definition of the type of interest of the top level module. For example, in the LEL for the Milk Production System, the symbol Dairy farm, which is shown in Appendix A, contains in its notion an enumeration of the main components of the domain.

Anyway, to identify the application domain components we propose to consider LEL subjects and objects. Usually, subjects are relevant components of the application domain, so they will be part of the type of interest. However, an object may represent a main domain component or it may define a component of other objects or subjects. The notion of the corresponding symbol in the LEL can help to decide whether an object should be included in the system module type of interest or not. Another thing that may help is considering maps, sets or lists defining LEL objects and not used in the definition of any other type. In our case study, for example, the maps Cows, Fields, Bulls, and Cow_groups are types which are not used in the definition of any other type. So, each of them represents potentially one of the main components in the domain. On the other hand, the map Plots is not considered because it is used to define one of the components of the type Field.

Once the main application domain components are identified, we define an embedded object for each component. Each object is an instantiation of the scheme defining the corresponding component. Then, we gather all these objects into a record type definition which will represent the domain state. Each field in this record corresponds to one of the components of the domain, and it is modelled as an instance of the scheme defining the corresponding component.

```
context: DOM_COMP_1, ..., DOM_COMP_n
scheme DOM_STATE =
   class
      object
         DC_1: DOM_COMP_1,
         ...,
         DC_n: DOM_COMP_n
      type
         Dom_state::
            dom_comp_1: DC_1.Dom_Comp_1
            ...
            dom_comp_n: DC_n.Dom_Comp_n
   end
```

for n > 0, where the schemes DOM_COMP_1, ... DOM_COMP_n listed in the context clause are the schemes defining the different application domain components.

For example, in the Milk Production System maps were defined to represent cows, bulls, cow groups, dairy farmers and fields in a dairy farm. Each map is defined in a scheme module and corresponds to one component of the dairy farm. So, to reflect the system state, we define the record type Dairy_farm in the top level module, which is called DAIRY_FARM. All the modules used to define embedded objects in the module DAIRY_FARM are listed in the module context.

```
context: FIELDS, COW_GROUPS, BULLS,DAIRY_FARMERS
scheme DAIRY_FARM =
   class
      object
         CS: COWS,
         BS: BULLS,
         FS: FIELDS,
         CGS: COW_GROUPS(CS),
        DFS: DAIRY_FARMERS
    type
        Dairy_farm::
            cows: CS.Cows
            bulls: BS.Bulls
            fields: FS.Fields
            groups: CGS.Cow_groups
            dairy_farmers: DFS.Dairy_farmers
            past_cows: CS.Cows
end
```

4.2.3 The architecture of the specification

The modules defined by applying the heuristics we proposed in Sections 4.2.1 and 4.2.2 can be hierarchically organised to show the system module structure. The root of this hierarchy is the system module, the second level contains the modules that define each one of the domain components and the remaining levels correspond to the modules that help to define the upper ones, as for example the general types module, a module defining the date type, etc. This hierarchy can be shown graphically in a diagram. Such diagrams can be generated automatically by the RAISE tools (Section 3.3), as we will show in Section 5.

In Section 3.2.3, we explained many guidelines the RAISE Method provides to hierarchically structure a specification, aiming at encouraging separate development and step-wise development. These guidelines allow one to obtain a hierarchy of modules that could be specified using the Layers pattern [10]. The heuristics applied during the three-step process we proposed were defined following closely all these guidelines. As a consequence the RSL specification derived can be structured in layers. Considering the Layers Pattern implementation defined in [24], the global architecture we propose is composed of three layers: specific layer, general layer and middleware layer.

A layer is a set of RSL modules that share the same degree of generality. Lower layers are general to several domain specifications, while higher ones are more specific to a concrete domain. The specific layer contains application-specific modules not shared by other parts. The general layer includes modules that are not specific to a single application and then they can be reused for many different applications within the same domain or business. The middleware layer has modules that are so general that can be used in any domain. Examples of middleware layer modules are standard specifications such as bags, stacks, queues, etc. in different levels of abstraction.

A specific module, which is located in the specific layer, can use modules of the general layer or the middleware layer. Modules located in the general layer can use modules in the middleware layer. This way of defining use relationships between layers is similar to the one proposed in [24] but more flexible than the one described in [10].

A module in RSL can be a scheme or an object. Schemes and global objects form a space of names that may potentially be used in modules. To provide some control over visibility and hence dependency, every RSL module must include a context clause indicating those that are actually used. The use of context clauses allows a layer to be partially opaque, this means that some of its modules are only visible to the next higher layer, while others are visible to all higher layers. This is particularly helpful when having global objects because though global they must be included in the context clause of any module that needs them.

In general, the development of a specification into another one has no impact on the layered architecture of the specification. Development in RAISE typically involves replacing more abstract with more concrete modules, and sometimes it also involves introducing new child modules. A child appears when the development of a module introduces a new component or concept worthy of its own module. A developed module will be in the same layer as its more abstract counterpart, while the new child may be in the same layer or in a lower one.

Our proposal of using the Layers Pattern to structure the hierarchy of modules of a specification in RSL assumes all the modules have the same specification style, as for example applicative sequential as in our case study. When developing the modules into a different style, such as imperative sequential, the architecture could be respected as long as the implementation relation holds between the modules of the different layers of both specifications.

In Section 5 we will explain how we could define the layered architecture for the Milk Production System RSL specification.

4.3 Derivation of functions

A function is esentially a mapping from values of one type to values of another type. As we showed in Section 3.1.3, functions can be total or partial, and they may be defined in a variety of styles, ranging from abstract property-oriented styles on one side to concrete algorithm-oriented styles at the other. Functions are essential to the specification of a system, as activities within a system may be modelled as functions [21].

In this section we present a set of heuristics that help to identify and to model the functions of the RSL specification. We explain how to derive arguments and result types of functions, how to classify functions as partial or total, and, when possible, how to define function bodies by analysing descriptions of scenarios. The Scenario Model describes domain situations, with an emphasis on the behaviour description. So, scenarios are the main source of information when defining functions. In addition to functions that are specific to the considered application domain, we show how to define the appropriate functions to create, modify and observe the type of interest of each module defined in the Definition of Modules step (Section 4.2).

We perform the derivation of functions in two steps: Definition of top level functions (Section 4.3.2) and Definition of lower level functions (Section 4.3.3). Before proposing the corresponding heuristics, we present a brief discussion to support our decision of modelling functions in a hierarchical way.

4.3.1 Hierarchical definition of functions

As we have already mentioned, scenarios play a significant role when deriving functions. A scenario can produce a change in the domain by modifying any of the components of the domain. In addition, scenarios that produce a change in the domain usually contain an episode saying that some information is stored, recorded, registered or saved. From now on, we will call this kind of scenario a modifying scenario. In the same way we will use the term observing scenario to refer to a scenario that only accesses information in the domain without performing any change. The scenario goal can help in classifying each scenario as modifying or observing, and in most cases this should be enough as, by definition, the goal contains the aim to be reached in the domain by performing the episodes in the scenario. For example, the scenarios Assign a group to a cow and Define cow type are modifying scenarios while the scenarios Check ration distribution and Control weight of a cow are observing scenarios.

A first conclusion might be that modifying scenarios will always correspond to generator functions while observing scenarios will correspond to observer ones. However, this is not always true. For example, a scenario like Compute group individual production can be first classified as modifying because it contains an episode stating that the individual production computed is stored. But, as we will show later, it is modelled with an observer function. The reason for this decision is that the individual production is a component of a group that can be calculated from some other components of the group and, as we have already explained in Section 4.1.2, we do not store what can be computed. So, any time the individual production of a group is required we compute it. The same reasoning can be applied to scenarios such as Compute next birth date, Compute dairy cow individual production, and Compute pasture eaten to find that they are modelled with observer functions, although they apparently store information.

The hierarchy of modules produced by the Definition of Modules step (Section 4.2), has a great influence in the way functions should be specified. To respect this hierarchy, any function in a module should only be called from its parent. Then, functions at one level in the hierarchy of modules frequently have counterparts at lower levels, but with different parameters.

For example, the function milk_cow that comes from the scenario Record milking is modelled by defining three functions in different levels:

- In the DAIRY_FARM module

  value
  can_milk_cow: GT.Cow_id × D.Date × Dairy_farm → **Bool**
  can_milk_cow(ci, d, df) ≡ CS.can_milk_cow(ci, d, cows(df)),

  milk_cow: GT.Cow_id × D.Date × GT.Litres × Dairy_farm $\overset{\sim}{\to}$ Dairy_farm
  milk_cow(ci, d, lts, df) ≡ chg_cows(CS.milk_cow(ci, d, lts, cows(df)), df)
  pre can_milk_cow(ci, d, df)

- In the COWS module (instantiated as the object CS in DAIRY_FARM)

  value
  can_milk_cow : GT.Cow_id × D.Date × Cows → **Bool**
  can_milk_cow(ci, d, cs) ≡ ci ∈ cs ∧ C.can_milk_cow(d, cs(ci)),

  milk_cow: GT.Cow_id × D.Date × GT.Litres × Cows $\overset{\sim}{\to}$ Cows
  milk_cow(ci, d, lts, cs) ≡ cs † [ci ↦ C.milk_cow(d, lts, cs(ci))]
  pre can_milk_cow(ci, d, cs)

- In the COW module (instantiated as the object C in COWS)

```
    value
       can_milk_cow : D.Date × Cow → Bool
       can_milk_cow(d, c) ≡ is_milking_cow(c) ∧ ∼milked(d, c),

       milk_cow: D.Date × GT.Litres × Cow ∼→ Cow
       milk_cow(d, lts, c) ≡ chg_history(CH.add_event(d, CE.milkings(lts), history(c)), c)
       pre can_milk_cow(d, c)
```

The definitions of the functions is_milking_cow and milked in COW can be found in Appendix C, page 172.

There are some things worth explaining about these definitions: there is an appropriate type of interest at each level, each function is only called from its parent module, and the identifying parameter changes while the other parameters are typically the same.

This may appear to require many unnecessary functions, but if we try to use only one function at the top level module we could get something like

```
value
   milk_cow: GT.Cow_id × D.Date × GT.Litres × Dairy_farm ∼→ Dairy_farm
   milk_cow(ci, d, lts, df) ≡
      let c = cows(df)(ci),
      new_c = CS.C.chg_history(CH.add_event(d, CE.milking(lts), CS.C.history(c)), c)
      in
      chg_cows(cows(df) † [ci ↦ new_c], df)
      end
   pre ci ∈ cows(df) ∧ CS.C.is_milking_cow(cows(df)(ci))
         ∧ ∼CS.C.milked(d, cows(df)(ci))
```

This last definition is hard to write and also to read. It is not much shorter than the others, and, in addition, it is harder to change if, for example, a type at some level needs to be modified.

Therefore, we suggest following the first approach we explained above and thus, model for each function in the top level module the necessary functions in lower level modules, in order to simplify the legibility and maintainability of the specification.

4.3.2 Definition of top level functions

Functions are usually identified at the top level because scenarios help to generate them there. Top level functions represent the main functionality in the system and they are defined in the system module.

Behavioural responses of LEL subjects include the main functionality in the domain, and each of them is usually described with more details in a scenario. So, in general, each scenario will motivate the definition of a top level function. However, a scenario may be a sub-scenario listed in the episodes of another scenario. A sub-scenario does not necessarily represent a domain situation. We explained in Section 2.2, sub-scenarios are mainly introduced to group common behaviour detected in several scenarios, when complex conditional or alternative courses appear in a scenario, or when the need to enhance a situation with a concrete and

HTFid	**Scenario Model**	**RSL**
HTF1	Scenario describes a LEL subject behavioural response	Top level function
HTF2	Scenario type	Function type
HTF2.1	Modifying	Generator (though not always)
HTF2.2	Observing	Observer
HTF3	Scenario components	Function signature
HTF3.1	Modifying scenario	
HTF3.1.1	Resources to modify and resources with data to modify	Arguments
HTF3.1.2	Actors	Probably arguments
HTF3.1.3	Resources and/or actors modified	Result
HTF3.2	Observing scenario	
HTF3.2.1	Resources to access information	Arguments
HTF3.2.2	Actors	Probably arguments
HTF3.2.3	Information returned	Result
HTF4	Context	Total or partial function

Table 4.4: Heuristics to model top level functions

precise goal is detected inside a scenario. It is not possible to determine from the LEL and the Scenario Model if a scenario should only be a sub-scenario or if it is also a scenario that defines a relevant functionality in the domain. For example, the scenario Compute pasture eaten is used as a sub-scenario when it appears as an episode of the scenario Feed a group. But with the information available, we cannot ensure that it does not represent some independent functionality. In summary, subjects' behavioural responses and their scenarios are only a source of candidate top level functions.

Table 4.4 summarises the heuristics we propose to specify top level functions (HTF stands for Heuristics for Top Level Functions). After determining which functions to define in the top level module (HTF1), the next steps are the formulation of their signatures (HTF2, HTF3, HTF4) and the definition of their bodies. The definition of the signature of a function involves determining its arguments and result type as well as classifying it as partial or total.

- **Definition of the signature**

  To determine the function arguments and result, we analyse actors and resources in the scenarios. Resources in general will be arguments because they represent information that should be available in the scenario, and thus in the function. However, this is not always the case for actors as sometimes they only represent the ones who execute the action in the application domain. So, they will be arguments only if the goal of the scenario is either to access or modify the information they contain.

  To find out the function result we analyse if the function is a generator or an observer. In case of a generator function, the result is determined by the subject(s)/object(s) that are modified in the scenario. In case of an observer function, the subjects/objects containing the information returned by the scenario represent the function result.

  For example, the modifying scenario Feed a group (Appendix B) motivates the definition of a generator function feed_group. The resources of the scenario suggest that the group, the date, the quantities of concentrated food, hay and corn silage, and the feeding form should be the arguments of the function. The actor dairy farmer should not be an argument because its only responsibility is the execution of the action described by the scenario. The resource Feeding form is the place where the change performed by the scenario is stored, so it will represent the function result. An informal definition for this function would be

feed_group: group × date × quantity of corn silage × quantity of hay
  × quantity of concentrated food × feeding form → feeding form

We use this kind of informal definition to help in the identification of function arguments and results. When deriving a generator function, the arguments may be divided into two groups: the ones to identify the component to be modified, and the ones that contain the information with which to modify the component. The function result is the component to be modified. In case of an observer function, the arguments are only those necessary to access the information to be returned by the function, while the result is precisely this information to be returned.

These informal arguments and result are replaced by the types previously defined to represent each actor and resource that was proposed as argument or result. We showed in Section 4.1 that, in general, each actor and resource has its corresponding type definition. However, as we will explain later in this section, there may be some exceptions that need a slightly different treatment.

For a generator function, the result type is always the record type representing the system state. This record type is also included as an argument type because it contains the definition of all the domain components. The rest of the argument types correspond to the types used to define the data required to identify the components to modify as well as the information with which to modify them. Then, when the type comes from a subject or an object whose collection was modelled with a map, the type of the corresponding identification argument will defined by the types used to model the set of attributes defined as the map domain.

The signature of a top level generator function may be specified as follows:

```
value
   gen_function_name: Identifying_attr × Modifying_attr × Sys_state
                      → Sys_state
```

where Identifying_attr and Modifying_attr may be composed of more than one type, and Sys_state is the record type specifying the system state.

For the function feed_group, informally defined above, the signature is the following:

```
value
   feed_group: GT.Group_id × D.Date × GT.Corn_sil × GT.Hay
      × GT.Conc × Dairy_farm → Dairy_farm
```

We use Group_id as argument because the LEL object Group is an argument whose collection was represented with the map Groups. Feeding form is apparently not included as argument. But, when deriving the types we decided to model all the events applied to groups of cows as part of the type Group. This signature is still not definitive, because it is necessary to classify each function as partial or total, as we show below.

For an observer function, the result type is obtained from the types corresponding to the information returned by the scenario. The record type representing the system state is always an argument for the same reasons we explained above for a generator function. The rest of the arguments are the types corresponding to subjects or objects that are necessary to access the information that should be returned by the function, plus some additional arguments like the date, not always present as a resource in the scenario. As we pointed out above, when arguments are objects or subjects whose collection was modelled with a map, the type of the corresponding arguments will be defined by the types used to model the set of attributes defined as the map domain.

Then, the signature of an observer top level function may be specified as follows:

value
  obs_function_name: Identifying_attr $\times$ Sys_state $\rightarrow$ Info_returned_type

where Identifying_attr as well as Info_returned_type may be composed by more than one type, and Sys_state is the record type defining the system state.

For example, the scenario Compute next birth date is modelled with the top level function next_birth_date, which can be first informally defined as follows

  next_birth_date: cow $\times$ date $\times$ insemination form $\rightarrow$ date

The signature for this function is:

value
  next_birth_date: GT.Cow_id $\times$ D.Date $\times$ Dairy_farm $\rightarrow$ D.Date

Cow_id is the argument type for the cow as the collection of cows was modelled with the map Cows. The argument Insemination form is included in the definition of the type Dairy_farm because inseminations were modelled as part of the type Cow, and the collection of cows is a component of the dairy farm.

The signatures just presented are still not definitive, because it is necessary to classify each function as partial or total, as we show below.

To classify each function as partial or total, we analyse the context component of the corresponding scenario. The context describes the scenario's geographical and temporal location as well as the scenario's initial state. According to some general heuristics to describe scenarios, at least one of the components of the context should be filled in, i.e. the context should not be empty. So if a function cannot be total because it needs some preconditions to be satisfied, most of these preconditions will appear in the scenario context. In general, we found that a context description which contains a temporal location or an initial state motivates the definition of a partial function. Preconditions are better expressed as calls of functions [22], so additional functions will have to be defined. Preconditions of top level functions will commonly call functions defined in the top level module itself, which in turn will typically call functions defined in the second level modules.

For example, as we have stated the function next_birth_date comes from the scenario Compute next birth date. The context of this scenario contains a precondition, which establishes that next birth date can be calculated only for pregnant dairy cows or heifers. The function must be then defined as partial with the precondition formulated as a call of a function defined in the top level module.

```
value
    can_give_birth: GT.Cow_id × D.Date × Dairy_farm → Bool
    can_give_birth(ci, d, df) ≡ CS.can_give_birth(ci, d, cows(df)),

    next_birth_date: GT.Cow_id × D.Date × Dairy_farm ~→ D.Date
    next_birth_date(ci, d, df) ≡ ...
    pre can_give_birth(ci, d, df)
```

In the same way, the already introduced function feed_group is finally classified as partial, as the context of the scenario Feed a group establishes that only non empty groups should be fed once a day:

```
value
    feed_group: GT.Group_id × D.Date × GT.Corn_sil × GT.Hay
        × GT.Conc × Dairy_farm ~→ Dairy_farm
```

The precondition is formulated as a call to a function defined in the top level module, as we will show later.

A different situation occurs with the function define_cow_classif which comes from the scenario Define cow type. Although the context of the scenario appears to be empty, we define the function as partial because it is necessary to check that the cow belongs to the Cows map.

```
value
    define_cow_classif: GT.Cow_id × D.Date × Dairy_farm ~→ Dairy_farm
    define_cow_classif(ci, d, df) ≡ ...
    pre ci ∈ cows(df)
```

A function accessing a map with a key will almost always be partial, independently of what is described in the context of its corresponding scenario. Checking whether the key belongs to the domain is likely to be in the precondition of all such functions.

Another situation worth mentioning here, is the one that appears when any of the components in the informal definition of the function has not a corresponding type definition because it was modelled with a variant type. When we use a variant type to model a component, the functions involving this component will typically have an additional precondition to check the function is only applied to the appropriate component. We will model this kind of additional preconditions as separate functions, in order to simplify future changes in case the definition of the component type is modified.

To show an example, in the informal definition of the function milk_cow, which comes from the scenario Record milking

milk_cow: milking cow × date × litres × milking form → milking form

appears the component milking cow, which actually has no associated type definition, because we modelled the different categories of cows with a variant type. To manage this, we add a precondition to the function to ensure that it is only applied to milking cows. So, a first formal definition could be

value
  is_milking_cow: GT.Cow_id × Dairy_farm → **Bool**
  is_milking_cow(ci, df) ≡ CS.is_milking_cow(ci, cows(df)),

  can_milk_cow: GT.Cow_id × D.Date × Dairy_farm → **Bool**
  can_milk_cow(ci, d, df) ≡ CS.can_milk_cow(ci, d, cows(df)),

  milk_cow: GT.Cow_id × D.Date × GT.Litres × Dairy_farm $\overset{\sim}{\rightarrow}$ Dairy_farm
  milk_cow(ci, d, lts, df) ≡
  ...
  pre is_milking_cow(ci, df) ∧ can_milk_cow(ci, d, df)

The function can_milk_cow contains all the necessary conditions established in the scenario context, while is_milking_cow verifies the function milk_cow is only applied to the appropriate category of cows, i.e milking_cows.

- **Definition of the body of the function**

  The hierarchy of modules defined in Section 4.2, has a great influence on the specification of functions. Most of the top level functions will call functions in the second level, which in turn will call functions in the levels below, thus motivating the definition of more functions in lower level modules.

  In general, the body of each top level function will contain a call to one or more functions defined in modules in the second level. In the case of a generator function, the body will contain at least one call to a chg_component function which in turn will call to the function or functions that perform the modification of the corresponding component(s). Thus, each chg_component function will have as its first argument a call to a second level function in charge of doing the change, and as its second argument the system state. The RSL specification below provides a general guide to follow when defining these functions:

```
value
   precond_name: Identifying_attr × Modifying_attr × Sys_state → Bool
   precond_name(ia, ma, ss) ≡ SL.precond_name(ia, ma, comp_i(ss)),

   gen_function_name: Identifying_attr × Modifying_attr × Sys_state
                      ~→ Sys_state
   gen_function_name(ia, ma, ss) ≡
         chg_comp_i(SL.gen_function_name(ia, ma, comp_i(ss)), ss)
   pre precond_name(ia, ma, ss)
```

As before, Identifying_attr and Modifying_attr may be composed of more than one type, and Sys_state stands for the record type modelling the system state. The function chg_comp_i is the reconstructor that corresponds to the ith component of the system state. The first argument of chg_comp_i, the call to a second level function, is prefixed with SL, the object which is an instance of the scheme that defines the ith component. Finally, precond_name is a function defined in the same top level module which contains the necessary preconditions for the function. It is worth noting that, depending on the preconditions to be defined, it may not be necessary to include all the arguments shown in the pattern above.

For the generator function feed_group, its informal definition

```
feed_group: group × date × quantity of corn silage × quantity of hay
    × quantity of concentrated food × feeding form  → feeding form
```

shows that the corresponding function will need to access the data of a group and will also return a group as result, as we decided to model all the events applied to groups of cows as part of the type Group. We modelled the collection of groups of cows with the map Cow_Groups, and we defined this map in a separate module called COW_GROUPS. Besides, as groups of cows are one of the components of the domain, the record type Dairy_farm contains the component groups defined as CGS.Cow_Groups. For this reason, we write in the body of this function a call to the function chg_groups, with first parameter CGS.feed_group.

The complete definition for the function feed_group is

```
value
 can_feed_group: GT.Group_id × D.Date × Dairy_farm → Bool
 can_feed_group(gt, d, df) ≡
    CGS.can_feed_group(gt, d, groups(df), cows(df)),

 feed_group : GT.Group_id × D.Date × GT.Quantity × GT.Quantity
    × GT.Concentrate × Dairy_farm ~→ Dairy_farm
 feed_group(gt, d, corn, hay, conc, df) ≡
    chg_groups(CGS.feed_group(gt, d, corn, hay, conc, groups(df), cows(df)), df)
 pre can_feed_group(gt, d, df)
```

In the case of an observer function, the body will contain one or more calls to the appropriate second level functions. These functions will be defined in the module which contains the component that has the information to be retrieved. The following RSL specification provides a pattern to help in the definition of these functions:

```
value
   precond_name: Identifying_attr × Sys_state → Bool
   precond_name(ia, ss) ≡ SL.precond_name(ia, comp_i(ss)),

   obs_function_name: Identifying_attr × Sys_state ~→ Info_returned_type
   obs_function_name(ia, ss) ≡ SL.obs_function_name(ia, comp_i(ss))
   pre precond_name(ia, ss)
```

As before, Identifying_attr and Info_returned_type may be composed of more than one type, and Sys_state stands for the record type modelling the system state. The function SL.obs_function_name is a second level function placed in the scheme where the component which has the information to be retrieved is defined, and SL is the object which is an instance of the scheme that defines this component. Finally, precond_name is a function defined in the same top level module which contains the necessary preconditions for the function. As we pointed out before, the inclusion of all the arguments shown in the pattern above may not be necessary.

Following these guidelines, we define the function next_birth_date as follows

```
value
   can_give_birth: GT.Cow_id × D.Date × Dairy_farm → Bool
   can_give_birth(ci, d, df) ≡ CS.can_give_birth(ci, d, cows(df)),

   next_birth_date: GT.Cow_id × D.Date × Dairy_farm ~→ D.Date
   next_birth_date(ci, d, df) ≡ CS.next_birth_date(ci, d, cows(df))
   pre can_give_birth(ci, d, df)
```

In any case, to determine the appropriate second level function to call, which we will have to define later, we analyse the informal definition we have previously proposed for top level functions. This informal definition always shows which components are modified or which is the information the function will need to access. From these components, the types defined applying the heuristics from Section 4.1, and the module structure obtained with the heuristics from Section 4.2, we can identify the arguments and result type of the second level function, and the second level module in which it should be defined.

4.3.3 Definition of lower level functions

As we showed in the previous section, top level functions and their preconditions are modelled in terms of functions in the second level modules. For each function that is called in the body or the precondition of a top level function, and whose name has as prefix an object name, we analyse the object name to determine in which second level module we should define the function. From the definition of the object in the top level module we can find out in which module the function has to be defined. For example, the function CGS.feed_group(gt, d, corn, hay, conc, groups(df), cows(df)) should be defined in the module COW_GROUPS because CGS is an instance of the scheme COW_GROUPS. From the call in the top level module and the informal definiton obtained from the corresponding scenario, we can also find out the signature of the second level function, i.e. function arguments and result type and its classification as partial or total function.

In what follows we provide some guidelines to define the signature as well as the body of second level functions.

- **Definition of the signature**

  The function informal definition we used to determine arguments and result type for top level functions, could be of help to define the signature of second level functions. As we explained in Section 4.3.1, we decided to model functions in a top-down style, following the hierarchy of modules. We also showed that the definitions of the functions across the different levels only change the identifying parameters while the others are typically the same. Then, the only thing we will have to do to define the signature of these functions, is to determine the identifying parameters taking into account the component to be modified/accessed and the second level module which contains the type to model the component.

  When the type of interest of the second level module is a map, and the function is a generator one, a general definition for the signature is

  value
  gen_function_name: Map_id × Attrib × Map $\overset{\sim}{\to}$ Map

  where Map_id represents the type expression for the map domain, Attrib the information necessary to modify the map (perhaps composed of more than one type), and Map stands for the map type. The function is always partial because it is necessary to ensure that the map is only applied to values that belong to the domain of the map. According to this, the signature for the function feed_group is as follows

  value
  feed_group : GT.Group_id × D.Date × GT.Quantity × GT.Quantity
  × GT.Concentrate × Cow_Groups × CS.Cows $\overset{\sim}{\to}$ Cow_Groups

  When the function is an observer, a general definition for the signature is

  value
  obs_function_name: Map_id × Attrib × Map $\overset{\sim}{\to}$ GO.Result

  Map_id and Map are the same as above. Attrib may be empty or not, depending on some additional information the function may need to compute the result, as for example a period of time. Result represents the type of the value returned by the function, and it is prefixed with the name of the object that instantiates the module of global types. It is only necessary when the type of Result is not a built-in type. For the observer function next_birth_date, we have already introduced, the signature is

```
value
   next_birth_date: GT.Cow_id × D.Date × Cows $\xrightarrow{\sim}$ D.Date
```

- **Definition of the body**

  Concerning the bodies of this second level functions, they may contain in general a call to one or more functions defined in a lower level module.

  When the type of interest of the second level module is a map, the call will be to a function that manipulates each individual value in the map range. If the function is a generator, this call appears as an argument of the map override operator. In general

```
value
   precond_name: Map_id × Attrib × Map → Bool
   precond name (id, attrib, map) ≡ id ∈ map ∧ ...,

   gen_function_name: Map_id × Attrib × Map $\xrightarrow{\sim}$ Map
   gen_function_name(id, attrib, map) ≡
      map † [ id ↦ O.lower_function(attrib, map(id)) ]
   pre precond_name(id, attrib, map)
```

  where lower_function stands for a function located in a module one level below that manipulates individual values in the map range, and O is the name of the object which is an instance of the scheme that has as type of interest the type of the values in the map range. The function precond_name specifies the conditions that have to be satisfied to apply the function gen_function_name. It will always contain the test to ensure that id belongs to the map domain, and it may also contain other necessary tests or even calls to functions in lower level modules in charge of checking conditions concerning the values in the map range. It is worth noting that the definition of a function to specify the precondition can be avoided by writing directly after the keyword **pre** all the conditions connected by the operator and. However, we modelled most preconditions in the top level module as a conjunction of calls to functions in second level modules. So as each of these functions must be written in a second level module, we can also use them to express the preconditions of the second level functions. The general form for a function modelling a precondition in a second level module is

```
pre_f: Map_id × Attrib × Map → Bool
pre_f(id, attrib, map(id)) ≡ id ∈ map ∧ Condition_1 ∧ ...
     ∧ Condition_n ∧O.f′(attrib, map(id))
```

The call of the function f′ is present whenever is also necessary to check the value represented by map(id) satisfies certain conditions. Condition_1, ..., Condition_n stands for any other additional checks necessary to do.

But, if the only condition to be checked is that the map key that appears as argument belongs to the map domain, we can avoid the definition of a function for the precondition.

For example, the complete formal definition for the function feed_group is

```
value
   can_feed_group: GT.Group_id × D.Date × Cow_Groups × CS.Cows
                           → Bool
   can_feed_group(gt, d, cgs, cs) ≡ gt ∈ cgs ∧
      ∼empty(gt, cgs, cs) ∧ CG.can_feed_group(d, cgs(gt)),

   feed_group : GT.Group_id × D.Date × GT.Quantity × GT.Quantity
      × GT.Concentrate × Cow_groups × CS.Cows ∼→ Cow_groups
   feed_group(gt, d, corn, hay, conc, cgs, cs) ≡
    let
       ration = GT.mk_Ration(0.0, corn, hay, conc),
       new_r = GT.chg_pasture(
          compute_pasture_eaten(gt, ration, cgs, cs), ration)
    in
       cgs † [ gt ↦ CG.feed_group(new_r, d, cgs(gt)) ]
    end
   pre can_feed_group(gt, d, cgs, cs)
```

However, when the function is an observer one the body will only contain a call to the appropriate next lower level function.

```
value
   precond_name: Map_id × Attrib × Map → Bool
   precond name (id, attrib, map) ≡ id ∈ map ∧ ...,

   obs_function_name: Map_id × Attrib × Map ∼→ GO.Result
   obs_function_name(id, attrib, map) ≡
       ...  O.lower_function(attrib, map(id))  ...
   pre precond_name(id, attrib, map)
```

where lower_function and O have the same meaning defined above for generator functions. Result represents the type of the information returned by the function, and the prefix GO is the name of an object that instantiates the module of global types. The same comments we made with respect to preconditions for generator functions also hold for observer ones.

For the observer function next_birth_date the final definition is

```
value
    can_give_birth: GT.Cow_id × D.Date × Cows → Bool
    can_give_birth(ci, d, cs) ≡ ci ∈ cs(ci) ∧ C.can_give_birth(d, cs(ci)),

    next_birth_date: GT.Cow_id × D.Date × Cows →̃ D.Date
    next_birth_date(ci, d, cs) ≡ C.next_birth_date(d, cs(ci))
    pre can_give_birth(ci, d, cs)
```

We define the function can_give_birth to contain the conjunction of the conditions that have to be satisfied to apply the function, including the test to ensure that the map is only applied to values in its domain. This function has to check also, if the corresponding cow is pregnant and, as this information is stored in the Cow type, this check must be done calling a function in the module COW.

We showed that, in general, when the type of interest of a second level module is a collection, functions in this level will probably contain at least one call to a function placed in a module one level below, the one containing the definition of the type of interest of each element in the collection.

If the collection is modelled in a second level module with a map, then a third level module defines the values in the map range. As we have already explained, the top-down style we chose to model functions makes that identifying parameters change across different levels in the hierarchy while the rest of the parameters are basically the same. So, to define the signature of these functions, we only have to find out the identifying parameters taking into account the element of the collection to be modified/accessed and the third level module which contains the type to model the element. The following general forms may be used to define a generator function or an observer one:

```
value
    pre_f: Attrib × Map_range_value → Bool
    pre_f(attrib, map_range_value) ≡ ... ,

    f: Attrib × Map_range_value →̃ Map_range_value
    f(attrib, map_range_value) ≡ ...
    pre pre_f(attrib, map_range_value)


    pre_f: Attrib × Map_range_value → Bool
    pre_f(attrib, map_range_value) ≡ ... ,

    f: Attrib × Map_range_value →̃ GO.Result
    f(attrib, map_range_value) ≡ ...
    pre pre_f(attrib, map_range_value)
```

In both cases, Attrib represents any kind of information necessary to access or modify the adequate map value, Map_range_value is one of the types defined in Section 4.1 to represent individual domain components, and GO.Result stands for the type of the value returned by the function. The prefix GO is the name of the object which is an instance of the module of global types where the type Result is defined. It is only necessary when the type of Result is not a built-in type. Functions can be total or partial. They will be partial when coming from a scenario whose context has a temporal location or an initial state for the scenario.

For example, the following are the signatures for functions called from the module COWS, defined in the second level of the hierarchy.

```
value
    can_milk_cow: D.Date × Cow → Bool
    can_milk_cow(d, c) ≡ ... ,

    milk_cow: D.Date × GT.Litres × Cow $\overset{\sim}{\to}$ Cow
    milk_cow(d, lts, c) ≡ ...
    pre can_milk_cow(d, c)


    can_give_birth: D.Date × Cow → Bool
    can_give_birth(d, c) ≡ ... ,

    next_birth_date: D.Date × Cow $\overset{\sim}{\to}$ D.Date
    next_birth_date(d, c) ≡ ...
    pre can_give_birth(d, c)
```

Most of these lower level functions do the real work, because they access or modify the information stored in each individual component of the domain. The episodes in a scenario contain a set of actions describing the scenario behaviour and thus they are a good source of information when trying to define the body of such functions. However, it is not easy to provide guidelines for this definition process because the decisions taken to determine how to represent the components in the domain using types, functions, and a module structure, have a great influence on the way function bodies will have to be defined.

Chapter 5

The Case Study: A Milk Production System

In order to validate our proposal, we present in this chapter a complete case study. We briefly describe the Milk Production System, the domain selected, and we explain the steps we followed to define its LEL and Scenario Model. The construction of the LEL and the Scenario Model for the Milk Production System helped us to have a better understanding of this domain. Moreover, we defined both models with a considerable level of detail because the domain was not a conventional one. We also show the application of the technique we proposed in Chapter 4 to derive a RSL formal specification of the domain selected, and we present some experiences gained with the development of this real and quite complete case study.

5.1 Brief Description

A dairy farm breeds cows with the goal of producing good quality milk and obtaining a good income. All the necessary activities to achieve this goal are performed by one or more dairy farmers, sometimes with the help of one or more employees.

Cows are divided into groups according to their features. Each group receives a daily ration which may be composed of corn silage, hay, and concentrated food. Besides, each group is sent out to pasture in a field. Fields are divided into plots to ensure a good use of the pasture.

Cows are deparasited and vaccinated against different diseases, such as brucellosis and diarrhoea. The date as well as some information about the deparasitation or vaccination are registered.

After birth, calves are with their mother until they are more or less 5 days old, and then they are sent to the calf rearing unit. In the calf rearing unit, they receive milk or milk replacement and balanced food. When they can eat at least one kilogram of balanced food, they are sent out to pasture and they do not receive any more milk or milk replacement. In general, male calves are sold upon birth.

Female calves of twelve months old are considered heifers. Heifers can be inseminated when they reach 15 months age and their weight is nearly 350 kilograms. After giving birth to the first calf, a heifer is considered a dairy cow.

Dairy cows and heifers are on heat every 21 days approximately. When this is detected, they can be inseminated in the next twelve hours. Insemination can be natural or artificial. In any case, the date is recorded, plus some additional information relative to the procedure followed, such as the identification of the bull in case of natural insemination. Two months after the insemination, it is possible to detect if the dairy cow or the heifer became pregnant or not. After birth, dairy cows are milked for approximately seven months. In this period dairy cows are called milking cows, and they are milked twice a day, once in the morning and once in the evening. The quantity of litres and the date of each milking are registered. In the second menstrual cycle after the birth, dairy cows are again inseminated in order to make each dairy cow give birth to one calf per 12 months. A pregnant milking cow is dried, i.e. it is no longer milked, in the seventh month of pregnancy, and kept in a separate group where it receives a special ration.

A dairy cow can be discarded for many reasons, as for example an illness or when it cannot become pregnant for a long time. A discarded cow is kept eating only pasture until it reaches the appropriate weight to be sold.

The history of a cow, that is all the relevant events that happened since its birth, is very important as a basis for taking decisions about what to do with each cow.

5.2 The LEL Definition

To construct the LEL for the Milk Production System domain, we followed the LEL Construction Process described in Section 2.1.1. Two different domain experts were our main sources of information, as well as some documents and books about Milk Production Systems. Then, to identify the symbols used in the selected domain, we initially carried out two unstructured interviews, each one with a different domain expert, and we also read some books and documents related to Milk Production Systems. With all the information gathered, we wrote a preliminary list of the symbols characteristic of the domain. Each symbol of the list was then classified as subject, object, verb or situation/state, and its notions and behavioural responses were described, following the guidelines described in Table 2.2. Although these guidelines establish what to write in the notion and behavioural response of each symbol, usually the same meaning may be expressed with many different natural language sentences. As in some cases it is possible to define a standard form, without restricting the power of expression of natural language, when writing this LEL, we tried to use the same natural language structure to describe similar semantics in different symbols. For example, when defining the notion of a symbol x classified as a subject or an object, we use a consistent natural language structure to express a component of the symbol: "An x has a y" or "It/he/she has a y". Table 2.3 shows, for instance, how this form was used to express a dairy farmer has a name, a salary and one or more employees.

Once we had defined the notion and behavioural response of each symbol in the list, we carried out a verification process to check, for instance, the syntax and classification of each symbol in the list. By using this list as a guide, we developed some structured interviews in order to validate the definition of the symbols with the domain experts. Though they were not software engineers and they had no previous knowledge about LEL and scenarios, they found no problems reading and understanding them. They could make some corrections and suggestions, and following them we deleted some symbols, we detected some synonyms, and we found necessary to add some new symbols. Besides, we corrected and/or completed notions and behavioural responses of some symbols.

The final LEL has 68 symbols, from which one was classified as subject, 32 as objects, 32 as verbs, and three as states. As the domain we selected is not a conventional one, this LEL contains all the information available in a quite detailed way. Tables 5.1 and 5.2 show the complete list and the classification for each symbol. The complete definition for each symbol can be found in Appendix A.

5.3 The Scenario Model Construction

To define the Scenario Model we chose the combined strategy explained in Section 2.2. We first derived from the LEL a list of candidate scenarios by applying the heuristics. Then, we completed and improved this list to obtain the final one which actually contains 32 scenarios (Table 5.3) and which was validated with the stakeholders. Each scenario was defined following the structure shown in Table 2.7. Table 5.3 shows how we classify each scenario as observing or modifying, considering if the scenario produces a change in the domain or not.

The complete description of each scenario can be found in Appendix B. The structure used to define scenarios allows the description of any situation in the application domain with the required level of detail. For example, it would be possible to write a scenario called Milk a milking cow, enumerating in the episodes all the activities performed by a dairy farmer, such as taking the milk from the milking cow, putting the milk in a container, measuring the litres extracted, and recording this information. But the system to be developed will not actually milk cows, move them to pasture or give vaccinations, because we decided to model an information system instead of a control one.

Then, although scenarios were derived from the LEL, we filtered some information coming from the LEL in order to include only those situations that could be modelled in the specification of an information system. We return to this issue in Section 5.6.

5.4 The derivation of the RSL specification using our technique

In Chapter 4 we presented the three steps of the technique we proposed to derive an initial RSL specification from natural language models, such as LEL and scenarios. We mentioned there that those steps were not strictly sequential; they could overlap or carried out in cycles. However, as modules are defined to isolate a type of interest, and functions are defined to generate, modify, and observe values of a type, we should start identifying at least a preliminary set of types. For this reason, we begin the derivation of the initial RSL specification with the Derivation of types step.

5.4.1 Deriving the types

The LEL of the Milk Production System (Appendix A) and the corresponding classification of the symbols (Tables 5.1 and 5.2) contain the necessary information to derive, applying the heuristics we proposed in Section 4.1, an initial set of types which model the main components of our case study.

As set in our proposal, the first step consists in identifying a preliminary list of types by applying the heuristics HIT1, HIT2, HIT3, and HIT4 described in Section 4.1.1. Tables 5.4

LEL SYMBOL	CLASSIFICATION
ARTIFICIAL BREEDING/BREED ARTIFICIALLY	VERB
ARTIFICIAL INSEMINATION/INSEMINATES ARTIFICIALLY	VERB
ASSIGNS TO A GROUP/ASSIGNED TO A GROUP	VERB
BALANCED FOOD/BALANCED	OBJECT
BE ON HEAT/ON HEAT/HEAT	STATE
BIRTH/CALVING/GIVE BIRTH	VERB
BRAND	OBJECT
BULL	OBJECT
BUYS BULLS	VERB
CALF	OBJECT
CALF REARING UNIT	OBJECT
COMPUTES BIRTH DATE	VERB
COMPUTES INDIVIDUAL PRODUCTION	VERB
COMPUTES PASTURE EATEN	VERB
COMPUTES RATION	VERB
CONCENTRATED FOOD/CONCENTRATED	OBJECT
CONTROLS WEIGHT	VERB
CORN SILAGE	OBJECT
COW	OBJECT
DAIRY COW	OBJECT
DAIRY FARM	OBJECT
DAIRY FARMER	SUBJECT
DEFINES CALF GROUP	VERB
DEFINES COW TYPE/DEFINE COW TYPE	VERB
DEFINES PLOT	VERB
DEPARASITES/DEPARASITATION	VERB
DETECT PREGNANT COW	VERB
DISCARD COW	OBJECT
DISCARDS BULL	VERB
DRIED FEEDSTUFFS/DRY MATERIAL	OBJECT
DRY A COW FOR DISCARD/DRIED	VERB
DRY COW	OBJECT
EARLY PREGNANT COW	OBJECT
EMPTY COW	OBJECT
FEEDS GROUP/FEED A GROUP/FED/FEEDING	VERB
FIELD	OBJECT
GROUP/COW GROUP	OBJECT
HANDLES COWS DEATH/HANDLE COW DEATH	VERB
HAY	OBJECT
HEAT DETECTION	VERB
HEAT IS REGISTERED/REGISTERS HEAT	VERB
HECTARE LOADING	OBJECT
HEIFER	OBJECT
IDENTIFICATION NUMBER	OBJECT
INDIVIDUAL PRODUCTION/ MILK INDIVIDUAL PRODUCTION	OBJECT
INSEMINATION/INSEMINATE/INSEMINATES	VERB
LACTATION/LACTATION PERIOD	STATE
MAXIMUM LACTATION/PEAK LACTATION	OBJECT

Table 5.1: Classification of the symbols in the LEL

LEL SYMBOL	CLASSIFICATION
MILK	OBJECT
MILKING/TO MILK/MILKED/MILKS	VERB
MILKING COW	OBJECT
MILK REPLACEMENT/MILK SUBSTITUTE	OBJECT
NATURAL INSEMINATION/INSEMINATE NATURALLY	VERB
PASTURE	OBJECT
PLOT/PLOT AREA	OBJECT
POST-BIRTH COW	OBJECT
PRE-BIRTH COW	OBJECT
PREGNANT/PREGNANCY	STATE
RATION	OBJECT
SAVE BIRTH/SAVES BIRTHS/BIRTH IS SAVED	VERB
SELECTS A CALF GROUP	VERB
SELLS COW/SOLD	VERB
SENDS CALF TO THE CALF REARING UNIT	VERB
SENDS TO EAT PASTURE/SENT TO EAT PASTURE	VERB
TAKES CALF OUT THE CALF REARING UNIT	VERB
VACCINATES COW/VACCINATES/VACCINATION	VERB
VACCINE	OBJECT
WEIGHS COW/WEIGH COW/WEIGHED	VERB

Table 5.2: Classification of the symbols in the LEL

and 5.5 summarise the types identified from the LEL symbols and the heuristic/s applied to define each one.

Heuristic HIT1 makes us define a first set of 33 abstract types, as the LEL contains one subject and 32 objects whose names are singular nouns. The remaining abstract types come from heuristic HIT3 and correspond to verb symbols representing actions with data to save.

From this preliminary list of types, we consider the abstract types to apply the heuristics we proposed in Section 4.1.2 in order to develop them into more concrete ones when possible. Table 5.6 contains some of the types developed, the heuristic/s applied, and the RSL constructions used in the definition. The complete specification for each type can be found in Appendix C. However, we include below some complete examples in order to clarify the way we applied the heuristics.

For example, the abstract type Dairy_farmer, which comes from a subject LEL symbol according to heuristic HIT1, may be developed into a more concrete one by specifying it with a short record definition with two components, as two properties (the salary and the set of employees) can be identified from its notion (HDT1.1). We consider the dairy farmer's name as the attribute to identify each dairy farmer, and as it will be used to define the map domain, it is not included in the short record definition. Then, the RSL definition for the type is

```
type
   Salary = Real,
   Employee,
   Dairy_farmer ::
      salary : Salary ↔ chg_salary
      employees : Employee-set ↔ chg_employee
```

where we include the reconstructors chg_salary and chg_employee to indicate the components salary and employees may be modified.

SCENARIO TITLE	CLASSIFICATION
Assign a group to a cow	Modifying
Breed artificially	Modifying
Buy a bull	Modifying
Check ration distribution	Observing
Compute next birth date	Modifying
Compute dairy farm individual production	Modifying
Compute milking cow individual production	Modifying
Compute group individual production	Modifying
Compute pasture eaten	Modifying
Compute ration	Observing
Define calf group	Modifying
Define cow type	Modifying
Define plot	Modifying
Discard a bull	Modifying
Dry dairy cow	Modifying
Feed a group	Modifying
Handle cow death	Modifying
Inseminate artificially	Modifying
Inseminate naturally	Modifying
Manage birth	Modifying
Record cow deparasitation	Modifying
Record milking	Modifying
Register cow weight	Modifying
Register cows on heat detection	Modifying
Register heat	Modifying
Register pregnancy test	Modifying
Select a calf group	Modifying
Sell cow	Modifying
Send calf to the calf rearing unit	Modifying
Send out to pasture	Modifying
Take calf out the calf rearing unit	Modifying
Vaccinate cow	Modifying

Table 5.3: List of scenarios and their classification

LEL Symbol	HITid	RSL type	Type id
ARTIFICIAL BREEDING/ BREED ARTIFICIALLY	HIT3 HIT4.2	Abstract type List type expression	Artif_breeding Artif_breedings
ARTIFICIAL INSEMINATION/ INSEMINATES ARTIFICIALLY	HIT3 HIT4.2	Abstract type List type expression	Artif_insem Artif_insems
ASSIGNS TO A GROUP/ ASSIGNED TO A GROUP	HIT3 HIT4.2	Abstract type List type expression	Cow_to_group Cows_to_group
BALANCED FOOD/BALANCED	HIT1	Abstract type	Balanced
BE ON HEAT/ON HEAT/HEAT	HIT2.2	Abstract type	On_heat
BIRTH/CALVING/ GIVE BIRTH	HIT3 HIT4.2	Abstract type List type expression	Calving Calvings
BRAND	HIT1	Abstract type	Brand
BULL	HIT1 HIT4.1	Abstract type Map type expression	Bull Bulls
BUYS BULLS	HIT3 HIT4.2	Abstract type List type expression	Bought_bull Bought_bulls
CALF	HIT1 HIT4.1	Abstract type Map type expression	Calf Calves
CALF REARING UNIT	HIT1	Abstract type	Cru
CONCENTRATED FOOD/CONCENTRATED	HIT1	Abstract type	Conc
CONTROLS WEIGHT	HIT3 HIT4.2	Abstract type List type expression	Weigh Weighs
CORN SILAGE	HIT1	Abstract type	Corn_sil
COW	HIT1 HIT4.1	Abstract type Map type expression	Cow Cows
DAIRY COW	HIT1.3 HIT4.1	Subtype expression Map type expression	Dairy_cow Dairy_cows
DAIRY FARM	HIT1	Abstract type	Dairy_farm
DAIRY FARMER	HIT1 HIT4.1	Abstract type Map type expression	Dairy_farmer Dairy_farmers
DEPARASITES/ DEPARASITATION	HIT3 HIT4.2	Abstract type List type expression	Deparasitation Deparasitations
DISCARD COW	HIT1.3 HIT4.1	Subtype expression Map type expression	Discard_cow Discard_cows
DRIED FEEDSTUFFS/ DRY MATERIAL	HIT1	Abstract type	Dry_mat
DRY A COW FOR DISCARD/DRIED	HIT3 HIT4.2	Abstract type List type expression	Cow_dried Cows_dried
DRY COW	HIT1.3 HIT4.1	Subtype expression Map type expression	Dry_cow Dry_cows
EARLY PREGNANT COW	HIT1.3 HIT4.1	Subtype expression Map type expression	Early_preg_cow Early_preg_cows
EMPTY COW	HIT1.3 HIT4.1	Subtype expression Map type expression	Empty_cow Empty_cows
FEEDS GROUP/FEED A GROUP/FED/FEEDING	HIT3 HIT4.2	Abstract type List type expression	Feeding Feedings
FIELD	HIT1 HIT4.1	Abstract type Map type expression	Field Fields
GROUP/COW GROUP	HIT1 HIT4.1	Abstract type Map type expression	Cow_group Cow_groups
HANDLES COWS DEATH/ HANDLE COW DEATH	HIT3 HIT4.2	Abstract type List type expression	Death Deaths
HAY	HIT1	Abstract type	Hay

Table 5.4: Identification of types

LEL Symbol	HITid	RSL type	Type id
HEAT DETECTION	HIT3 HIT4.2	Abstract type List type expression	Heat_detection Heat_detections
HEAT IS REGISTERED/ REGISTERS HEAT	HIT3 HIT4.2	Abstract type List type expression	Heat Heats
HECTARE LOADING	HIT1.1	Abstract type	Hect_loading
HEIFER	HIT1 HIT4.1	Abstract type Map type expression	Heifer Heifers
IDENTIFICATION NUMBER	HIT1	Abstract type	Cow_id
INDIVIDUAL PRODUCTION/MILK INDIVIDUAL PRODUCTION	HIT1.1	Abstract type	Indiv_prod
INSEMINATION/ INSEMINATE/INSEMINATES	HIT3 HIT4.2	Abstract type List type expression	Insemination Inseminations
LACTATION/ LACTATION PERIOD	HIT2.2	Abstract type	Lact_period
MAXIMUM LACTATION/ PEAK LACTATION	HIT1	Abstract type	Max_lactation
MILK	HIT1	Abstract type	Milk
MILKING/TO MILK/ MILKED/MILKS	HIT3 HIT4.2	Abstract type List type expression	Milking Milkings
MILKING COW	HIT1.3 HIT4.1	Subtype expression Map type expression	Milking_cow Milking_cows
MILK REPLACEMENT/ MILK SUBSTITUTE	HIT1	Abstract type	Milk_repl
NATURAL INSEMINATION/ INSEMINATE NATURALLY	HIT3 HIT4.2	Abstract type List type expression	Nat_insem Nat_insems
PASTURE	HIT1	Abstract type	Pasture
PLOT/PLOT AREA	HIT1 HIT4.1	Abstract type Map type expression	Plot Plots
POST-BIRTH COW	HIT1.3 HIT4.1	Subtype expression Map type expression	Post_birth_cow Post_birth_cows
PRE-BIRTH COW	HIT1.3 HIT4.1	Subtype expression Map type expression	Pre_birth_cow Pre_birth_cows
PREGNANT/PREGNANCY	HIT2.2	Abstract type	Pregnant
RATION	HIT1	Abstract type	Ration
SAVE BIRTH/SAVES BIRTHS/ BIRTH IS SAVED	HIT3 HIT4.2	Abstract type List type expression	Birth Births
SELLS COW/SOLD	HIT3 HIT4.2	Abstract type List type expression	Cow_sale Cow_sales
SENDS CALF TO THE CALF REARING UNIT	HIT3 HIT4.2	Abstract type List type expression	Calf_to_cru Calves_to_cru
SENDS TO EAT PASTURE/ SENT TO EAT PASTURE	HIT3 HIT4.2	Abstract type List type expression	Group_to_plot Groups_to_plot
TAKES CALF OUT THE CALF REARING UNIT	HIT3 HIT4.2	Abstract type List type expression	Calf_out_cru Calves_out_cru
VACCINATES COW/ VACCINATES/VACCINATION	HIT3 HIT4.2	Abstract type List type expression	Vaccination Vaccinations
VACCINE	HIT1	Abstract type	Vaccine
WEIGHS COW/ WEIGH COW/WEIGHED	HIT3 HIT4.2	Abstract type List type expression	Cow_weigh Cows_weigh

Table 5.5: Identification of types

Type id	**HDTid**	**RSL type**
Artif_breeding	HDT2	Short record definition
Artif_insem	HDT2	Short record definition
Cow_to_group	HDT2	Short record definition
Balanced	HDT1.1	Simple type
Brand	HDT1.1	Simple type
Bull	HDT1.1	Short record definition
Conc	HDT1.1	Short record definition
Corn_sil	HDT1.1	Simple type
Cow	HDT1.1 HDT1.4 HDT1.5	Short record definition Variant type List expression
Dairy_farm	HDT1.1	Short record definition
Dairy_farmer	HDT1.1	Short record definition
Deparasitation	HDT2	Short record definition
Dry_mat	HDT1.1	Simple type
Cow_dried	HDT2	Short record definition
Feeding	HDT2	Short record definition
Field	HDT1.1 HDT1.2	Short record definition
Cow_group	HDT1.1 HDT1.5	Short record definition List expression
Death	HDT2	Short record definition
Hay	HDT1.1	Simple type
Heat_detection	HDT2	Short record definition
Heat	HDT2	Short record definition
Hect_loading	HDT1.1	Simple type
Cow_id	HDT1.1	Simple type
Indiv_prod	HDT1.1	Simple type
Insemination	HDT2	Short record definition
Lact_period	HDT1.1	Simple type
Milking	HDT2	Short record definition
Nat_insem	HDT2	Short record definition
Pasture	HDT1.1	Simple type
Plot	HDT1.1	Short record definition
Ration	HDT1.1	Short record definition
Birth	HDT2	Short record definition
Cow_sale	HDT2	Short record definition
Calf_to_cru	HDT2	Short record definition
Group_to_plot	HDT2	Short record definition
Calf_out_cru	HDT2	Short record definition
Vaccination	HDT2	Short record definition
Vaccine	HDT1.1	Short record definition
Cow_weigh	HDT2	Short record definition

Table 5.6: Elaboration of types

The abstract type Plot, coming from an object LEL symbol as set by heuristic HIT1, may be specified with a short record definition with four components (HDT1.1), representing the properties identified from the notion of the symbol (location, size, starting date and duration of the plot). The RSL definition for the type Plot is

```
type
    Location,
    Size = Real,
    Date,
    Plot ::
        plot_location : Location
        size : Size
        starting : Date
        days : Nat ↔ chg_days
```

The components plot_location, size, and starting do not have reconstructors as once set their values they cannot be modified. Besides, we have not included the plot identification as a component, because it will be used as the map domain when defining the collection of plots.

We want to remark that we could have included one more component in the record to save the group eating in a plot. But, analysing the LEL we discovered a one to one relationship between a plot and a group of cows, meaning that a plot can contain only one group at any time and a group can be eating pasture in only one plot at any time. Plots are only defined when it is necessary to send a group to a field, and each plot can contain at most one group at any time. If the plot is saved with the group, besides checking the existence of the plot, it would be necessary to define a consistency predicate to ensure that the group is unique in the plot. If the group is stored with the plot it is guaranteed that the group is the only one in the plot, but it should be checked that each plot has always one group assigned. We chose the first alternative, maintaining the plot with the group, and thus we deleted from the record Plot the component to store the group.

As another example, we show below the development of the abstract type Cow, coming from the object symbol Cow (HIT1). From the LEL symbol, we can identify some properties which, as suggested by heuristics HDT1.1, could be modelled with a short record definition. But, before determining the number of components the record will have, it is necessary to analyse some things more.

One of the notions of the symbol Cow says that a cow may be a calf, a heifer or a dairy cow. The words calf, heifer and dairy cow are symbols in the LEL classified as objects. LEL subjects and objects may appear in the notions of other subjects/objects in ways that indicate different states or alternatives. As we have just mentioned, the symbols Dairy cow, Calf and Heifer, which are objects whose name is a noun, appear in the notion of the object Cow as different categories of cow animals. In addition, in the notion of each one of these symbols it is explicitly written that they are cows with some special or additional features. This means that dairy cows, calves, and heifers share some attributes, but also have some special attributes depending on their category. Following heuristic HDT1.4, we model this notion with a variant type (Cow_classif), leaving the particular attributes of each category as components of the variant type. However, we decide not to model subtypes for each category of cows because actually, for each cow, categories are temporary as the cow will be continuously changing from one category to another depending on specific events in its life.

Another thing to consider is that, as we have shown in Section 4.1.2, we suggested to model actions applied to cows as part of the type Cow (HDT1.5 and HDT2), and we proposed modelling each of the actions applied to a cow as a list: Milkings list, Vaccinations list, Inseminations list, Births list, Deparasitations list, etc. (Tables 5.4 and 5.5). By analysing these lists, we observe that they are all ordered by date and basically the same operations are applied to each of them, like adding a new element (checking it had not been previously added), returning the elements whose date was in a given period, and so on. Besides, the definitions for the elements in each list are quite similar; all of them include the date plus some special information concerning the specific action, and thus each of them can be modelled with a short record definition (Table 5.6). Then, to give a more general solution we propose to define the type History as a list of events ordered by date. Each event in the list will represent one of the actions that could be applied to cows. We return to this issue in Section 5.4.2.

As a consequence of this decision, we delete the types representing list expressions modelling each collection of actions that could be applied to a cow. Then, though not finished, the new definition for the type Cow is as follows:

```
type
   Date,
   Event_info,
   Event ::
      date : Date
      event_inf : Event_info,
   History = {| h : Event* • is_ordered(h) |},
   Calf_info,
   Heifer_info,
   Dairy_info,
   Cow_classif ==
      calf(info : Calf_info) | heifer(info : Heifer_info) | dairy(info : Dairy_info),
   Cow ::
      birthday: Date
      cow_classif: Cow_classif ↔ chg_classif
      history: History ↔ chg_history
```

The type History is a subtype of the type of finite lists of events: only lists in date order are allowed. The complete specifications for the types Date, Calf_info, Heifer_info, and Dairy_info can be found in Appendix C, page 208 for the first one and page 204 for the remaining ones.

Following a similar analysis, we discovered that lists of actions applied to a group of cows, such as feedings list and heat detections list, are also ordered by date. Besides, the same operations are applied to them, and each element in each list can be modelled with a short record definition whose components are the date and specific information concerning the corresponding action. So, as we will explain in more detail in Section 5.4.2, we will use the type History we have just defined above to model actions applied to a group of cows.

Finally, during this elaboration of types, it was necessary to include some type definitions that do not correspond to any entry in the LEL. They appear when modelling components of some other type. For some of them we could give a concrete definition, while for others we provided a first abstract one to be developed later. For example, in the definition of the type Dairy_farmer we included the abstract types Salary and Employee which were not LEL entries. Similarly, when specifying the type Plot, we defined the abstract types Location and Date, and the concrete type Size, though they have no entry in the LEL. We return to this issue in Section 5.6.

5.4.2 Defining the modules

The set of types obtained in the previous section are the main source to define a first hierarchy of modules of the RSL specification. This hierarchy may be modified later when deriving the functions.

As suggested by heuristic HDM1, we define a scheme called GENERAL_TYPES to contain those types we are sure will be used in at least two modules, as the types defining map domains. This module includes then the types Bull_id, Cow_id, Dairy_farmer_id, Field_id, Group_id, and Plot_id. Besides, it is instantiated as the global object GT. The following is a first, and still incomplete definition for these two modules, as the Derivation of Functions step may show the need to include in the GENERAL_TYPES module many other types used in at least two modules.

```
scheme GENERAL_TYPES =
   class
      type
         Bull_id,
         Cow_id,
         Dairy_farmer,
         Group_id,
         Field_id,
         Plot_id,
         ...
   end

context: GENERAL_TYPES
object GT: GENERAL_TYPES
```

The type Date is another one that will be used in more than one module, and so it should be included in the scheme GENERAL_TYPES. However, as the date is an important element in our domain, and there are many functions to manipulate dates, we propose the definition of a scheme DATE in a separate module in order to isolate the types Date and Period, with their associated functions. We instantiate this module as the global object D and include it in the context of the scheme GENERAL_TYPES. The RSL specification of these two new modules is

```
scheme DATE =
   class
      type
         Date,
         Period
   end
```

```
context: DATE
object D: DATE
```

As another example of types used in at least two modules, we can mention constant values. When analysing LEL symbols to define types, we found many constat values that will be used throughout the specification of the Milk Production System, such as lactation period, pregnancy period, and discard age. We suggest putting them in a separate module, named CONSTANTS and instantiated as the global object K. Both specifications are in Appendix C, page 202.

According to heuristic HDM2, each type that models an element of a collection motivates the definition of a scheme. For this reason we define six schemes to contain the types Bull, Cow, Dairy_farmer, Field, Cow_group, and Plot respectively. In addition, as the types Bulls, Cows, Dairy_farmers, Fields, Cow_groups, and Plots model each a collection we define one scheme for each. Besides, each of these schemes includes the scheme modelling the corresponding element of the collection as an embedded object (HDM3).

To continue with one of the examples introduced in the previous section, we show the still incomplete specifications for the two scheme modules PLOT and PLOTS respectively. We define one module with type of interest Plots and another one with type of interest Plot, and we use the scheme PLOT to make the embedded object P in the scheme PLOTS. Complete specifications for both modules can be found in Appendix C, page 197. Prefixes GT and D refers to the global objects where the corresponding types are defined.

```
context: GT
scheme PLOT =
  class
    type
      Plot ::
        plot_location : GT.Location
        size : GT.Size
        starting : D.Date
        days : Nat ↔ chg_days

  end


context: PLOT
scheme PLOTS =
  class
    object P : PLOT
    type
      Plots = GT.Plot_id →m P.Plot
  end
```

In the previous section, we proposed a solution to model actions applied to a cow and to a group of cows, and we introduced the definition of the type History as an ordered list of actions of type Event. On the way to completing the specification, we have to decide in which module we should place the types History and Event. In order to model any kind of events, we suggest the definition of a parameterised module HISTORY, with History as its type of interest and parameterised over the types Event_info and Event_kind defined in the scheme EVENT_INFO. The context clause of the module HISTORY contains the scheme EVENT_INFO and the global object GT, which is an instantiation of the scheme GENERAL_TYPES. Although not listed in the context clause, the global object D may be mentioned in qualifications because we have included it in the context clause of the module GENERAL_TYPES.

```
scheme EVENT_INFO =
   class
      type
         Event_kind,
         Event_info
      value
         kind_of: Event_info → Event_kind
   end

context: GT, EVENT_INFO
scheme HISTORY(E: EVENT_INFO) =
   class
      type
         Event::
            date: D.Date
            event_inf: E.Event_info,
         History = {| h : Event* • is_ordered(h) |}
   end
```

There is still one thing to do and it is to define the two modules with which EVENT_INFO will be later instantiated: one to model actions applied to a cow and the other one to model actions applied to a group of cows. Thus, we define the scheme COW_EVENT, instantiated as the global object CE, to contain appropriate type definitions for each action that could be applied to an individual cow, and the scheme GROUP_EVENT, instantiated as the global object GE, to include appropriate type definitions for each action that could be applied to a group of cows. Complete specifications for these new modules are in Appendix C, pages 192 and 163.

With the introduction of these new modules, the final type definitions for the schemes COW and COWS are as follows, where CH is a global object defined as the instantiation HISTORY(CE), and K, GT, and D are the global objects previously defined. GT and D may be mentioned in the scheme COW because we included them in the context clause of the module CONSTANTS.

```
context: K, CH
scheme COW =
   class
      type
         Cow::
            birthday: D.Date
            cow_classif: GT.Cow_classif
```

```
          history: CH.History
    end

 context: COW
 scheme COWS =
    class
       object C : COW
       type
          Cows = GT.Cow_id →m C.Cow
    end
```

Finally, to define the top level module that will model the system state, we have to consider LEL subjects and objects to see which of them model a main domain component. The LEL object Dairy_farm may be of help here as it enumerates the main components of our domain (dairy farmers, cows, bulls, groups of cows, and fields). Besides, dairy farmer is a subject and usually subjects are relevant components of the domain. Concerning the objects cow, bull, field, and cow group, as their collections were specified using map expressions and these expressions are not used in the definition of any other type, they are potentially main domain components. However, we do not consider the LEL object Plot because the map modelling its collection is used to define a component of the type Field. Then, following the heuristic HDM4 we define the top level module, which is called DAIRY_FARM, with the record type Dairy_farm as its type of interest. The schemes Cows, Cow_groups, Bulls, Fields and Dairy_farmers are instantiated as embedded objects, and the corresponding modules are listed in the module context. As the scheme COWS needs to be shared between the schemes COW_GROUPS and DAIRY_FARM, we make it a parameter of the scheme COW_GROUPS.

```
 context: FIELDS, COW_GROUPS, BULLS,DAIRY_FARMERS
 scheme DAIRY_FARM =
    class
       object
          CS: COWS,
          BS: BULLS,
          FS: FIELDS,
          CGS: COW_GROUPS(CS),
          DFS: DAIRY_FARMERS
       type
          Dairy_farm::
             cows: CS.Cows
             bulls: BS.Bulls
             fields: FS.Fields
             groups: CGS.Cow_groups
             dairy_farmers: DFS.Dairy_farmers
             past_cows: CS.Cows
    end
```

All the modules we have just defined can be hierarchically organised to show the specification module structure. Figure 5.1 displays the diagram automatically generated using the

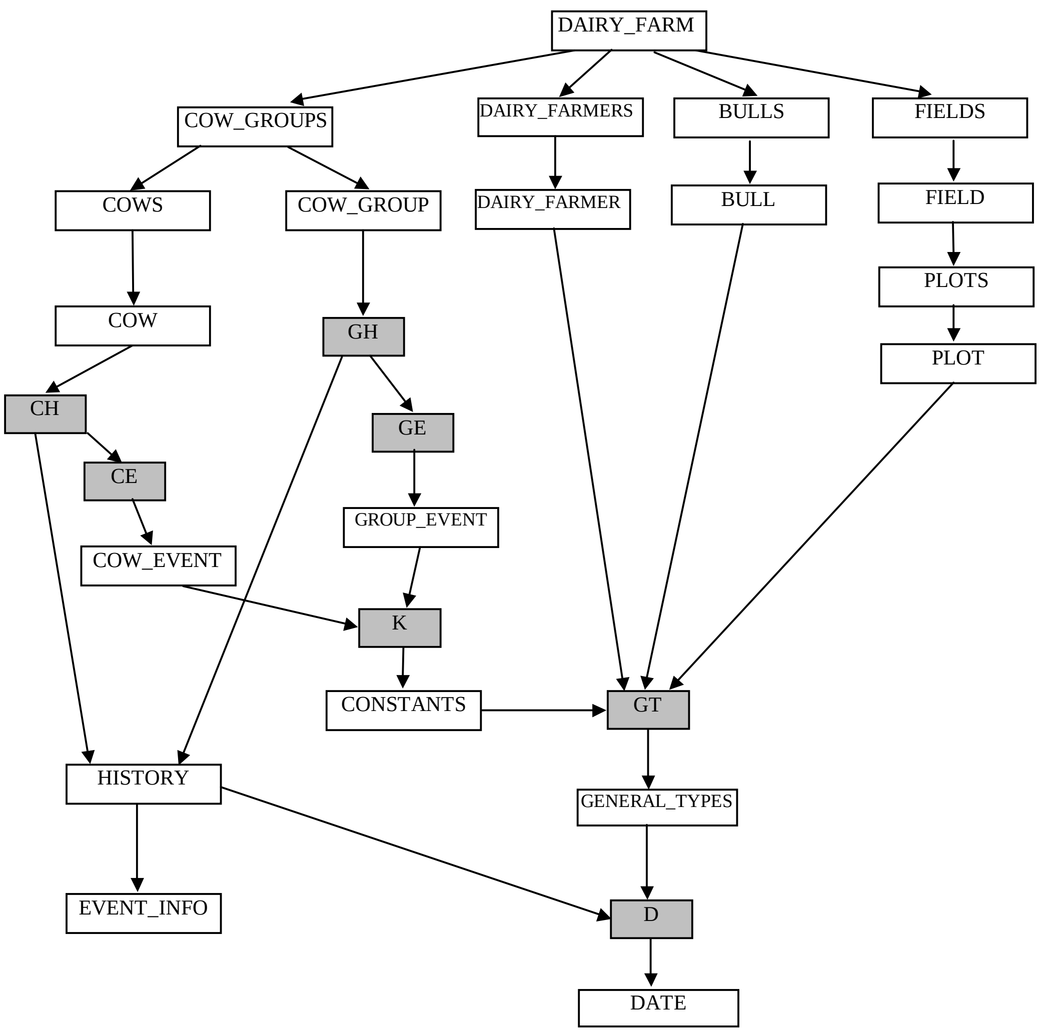

Figure 5.1: Module Structure of the Milk Production System RSL Specification

Type id	HDMid	Module
Bull_id	HDM1	GENERAL_TYPES
Bull	HDM2	BULL
Bulls	HDM3	BULLS
Cow_id	HDM1	GENERAL_TYPES
Cow	HDM2	COW
Cows	HDM3	COWS
Group_id	HDM1	GENERAL_TYPES
Cow_group	HDM2	COW_GROUP
Cow_groups	HDM3	COW_GROUPS
Dairy_farmer_id	HDM1	GENERAL_TYPES
Dairy_farmer	HDM2	DAIRY_FARMER
Dairy_farmers	HDM3	DAIRY_FARMERS
Dairy_farm	HDM4	DAIRY_FARM
Field_id	HDM1	GENERAL_TYPES
Field	HDM2	FIELD
Fields	HDM3	FIELDS
Plot_id	HDM1	GENERAL_TYPES
Plot	HDM2	PLOT
Plots	HDM3	PLOTS
Date	HDM1	GENERAL_TYPES

Table 5.7: Some types and the modules that contain them

RAISE tools (Section 3.3). Shadowed boxes correspond to objects, while the others represent schemes. The root of the Milk Production System hierarchy of modules is the DAIRY_FARM module, the top level one, the second level contains the modules that define each one of the domain components, as for example COW_GROUPS, FIELDS, BULLS, etc., and the remaining levels include the modules used to define the upper ones, such as HISTORY, CONSTANTS, and DATE.

The definition of this hierarchy allowed us to place many of the types originated in the previous section, in the different modules of the hierarchy. Moreover, many of the modules exist to isolate a type of interest. Table 5.7 shows some of the types we are quite sure will remain in the modules indicated. However, there are many types we are still not sure in which module to store. As they will be used as arguments or results of functions, it would be necessary to model first the functions in order to decide where to place them.

5.4.3 Deriving the functions

From the 32 scenarios listed in Table 5.3 we can identify 29 top level functions, as three of the scenarios (Check ration distribution, Compute pasture eaten and Register heat) are only sub-scenarios listed in the episodes of other scenarios. This is not the case for scenarios such as Assign a cow to a group and Define cow type. Though they appear as an episode in some scenarios, and thus they are sub-scenarios, they also model independent domain situations. Each of these 29 scenarios comes from a behavioural response of the unique subject in the domain: the dairy farmer (HTF1).

Observing scenarios will always motivate the definition of an observer function. However, as we have mentioned in Section 4.3.1, some modifying scenarios may be modelled with observer functions. So, before starting the definition of the signatures and bodies of these top level functions and in order to know which of the heuristics to apply, we analysed the list of modifying scenarios to determine that the scenarios Compute next birth date, Compute dairy farm individual production, Compute milking cow individual production, and Compute group individual production will be modelled with observer functions because the information they store can always be computed. Then, in the beginning, the module DAIRY_FARM will include the definition of 24 generator functions and five observer ones, one coming from the scenario Compute ration, classified at first as observing, and the remaining four coming from the scenarios we have just mentioned (HTF2.1, HTF2.2).

Tables 5.8 and 5.9 contains the informal definition for these first 29 top level functions as well as the signature of each function specified in RSL. We also include the title of the scenario motivating the definition of each function, and the heuristics applied in each case are listed. We want to explain that for the four modifying scenarios we have just decided to model with observer functions, we will apply the heuristics corresponding to observing scenarios.

All these top level functions are defined in the system module, the DAIRY_FARM module. This module contains the complete definition (signature and body) for each of them (Appendix C, page 138). As we suggested in Section 4.3.2, the body of these functions contain a call to one or more functions in second level modules. Each of these calls motivates the definition of new functions in the corresponding modules. Besides, for almost all partial functions we modelled preconditions as a call to another top level function, which in turn calls the corresponding lower level ones. We will use the top level function vaccinate_cow to exemplify the way we proceed to define the rest of the functions. The complete RSL specification for this function is

```
value
   can_receive_vacc: GT.Cow_id × D.Date × GT.Vaccine × Dairy_farm → Bool
   can_receive_vacc(ci, d, vacc, df) ≡
      CS.can_receive_vacc(ci, d, vacc, cows(df)),

   vaccinate_cow: GT.Cow_id × D.Date × GT.Vaccine × Dairy_farm ~→ Dairy_farm
   vaccinate_cow(ci, d, vacc, df) ≡
      chg_cows(CS.vaccinate_cow(ci, d, vacc, cows(df)), df)
   pre can_receive_vacc(ci, d, vacc, df)
```

The signature of the function was obtained by analysing the function informal definition taken from the corresponding scenario. We use Cow_id as argument because the LEL object Cow is an argument whose collection was modelled with the map Cows. The date and the vaccine are replaced by the corresponding types, but the Vaccination form is apparently not used. The reason is that when deriving the types we decided to model all the actions applied to cows as part of the type Cow (Section 5.4.1). The set of cows was modelled with a map, and this map is one of the components of the dairy farm state. This is why we use the type Dairy_farm as an argument. As we had already mentioned, the preconditions of this partial function are modelled with a call to the function can_receive_vacc also defined in the top level module. The bodies of these two functions contain a call to lower level functions. In both cases, the call is to a function defined in the module COWS, because the aim of the function is to register the vaccination of an individual cow and we modelled each cow as an element of the map Cows. The informal definition of the top level function could be of help to decide where to make the call, depending on the domain component the function needs to modify/access.

TOP LEVEL FUNCTIONS SIGNATURE	**HTFid**
Assign a group to a cow	HTF2.1
assign_group_to_cow: cow × date × current groups × group form → group form	HTF3.1
assign_group_to_cow: GT.Cow_id × D.Date × Dairy_farm $\overset{\sim}{\to}$Dairy_farm	HTF4
Breed artificially	HTF2.1
breed_artif: calf × date × quantity of milk replacement × quantity of balanced food × artificial breeding form → artificial breeding form	HTF3.1
breed_artif: GT.Cow_id × D.Date × GT.Litres × GT.Balanced × Dairy_farm $\overset{\sim}{\to}$Dairy_farm	HTF4
Buy a bull	HTF2.1
buy_bull: bull × date of purchase × bull birth date × field × bull features × list of bulls → list of bulls	HTF3.1
buy_bull: GT.Bull_id × D.Date × D.Date × GT.Field_id × GT.Features × Dairy_farm $\overset{\sim}{\to}$Dairy_farm	HTF4
Compute next birth date	HTF2.2
next_birth_date: cow × Insemination form → date	HTF3.2
next_birth_date: GT.Cow_id × D.Date × Dairy_farm $\overset{\sim}{\to}$ D.Date	HTF4
Compute dairy farm individual production	HTF2.2
d_farm_indiv_prod: period × Milking form → Individual production	HTF3.2
d_farm_indiv_prod: D.Period × Dairy_farm $\overset{\sim}{\to}$ GT.Indiv_prod	HTF4
Compute milking cow individual production	HTF2.2
cow_indiv_prod: dairy cow × period × Milking form → Individual production	HTF3.2
cow_indiv_prod: GT.Cow_id × D.Period × Dairy_farm $\overset{\sim}{\to}$ GT.Indiv_prod	HTF4
Compute group individual production	HTF2.2
group_indiv_prod: group × period × Milking form × Group form → Individual production	HTF3.2
group_indiv_prod: GT.Group_id × D.Period × Dairy_farm $\overset{\sim}{\to}$ GT.Indiv_prod	HTF4
Compute ration	HTF2.2
compute_ration: group × weight of average cow in group → ration	HTF3.2
compute_ration: GT.Group_id × Dairy_farm $\overset{\sim}{\to}$ GT.Quantity	HTF4
Define calf group	HTF2.1
define_calf_group: calves minimum age × calves maximum age × current groups → current groups	HTF3.1
define_calf_group: Nat × Nat × Dairy_farm $\overset{\sim}{\to}$Dairy_farm	HTF4
Define cow type	HTF2.1
define_cow_classif: cow → cow	HTF3.1
define_cow_classif: GT.Cow_id × D.Date × Dairy_farm $\overset{\sim}{\to}$Dairy_farm	HTF4
Define plot	HTF2.1
define_plot: group × field → field	HTF3.1
define_plot: GT.Group_id × GT.Field_id × Dairy_farm $\overset{\sim}{\to}$Dairy_farm	HTF4
Discard a bull	HTF2.1
discard_bull: bull × date × discard causes × list of bulls → list of bulls	HTF3.1
discard_bull: GT.Bull_id × D.Date × GT.Discard_cause × Dairy_farm $\overset{\sim}{\to}$ Dairy_farm	HTF4
Dry dairy cow	HTF2.1
dry_cow: cow × date × drying causes × discard form → discard form	HTF3.1
dry_cow: GT.Cow_id × D.Date × GT.Dried_cause × Dairy_farm $\overset{\sim}{\to}$Dairy_farm	HTF4
Feed a group	HTF2.1
feed_group: group × date × quantity of corn silage × quantity of hay × quantity of concentrated food × feeding form → feeding form	HTF3.2
feed_group: GT.Group_id × D.Date × GT.Corn_sil × GT.Hay × GT.Conc × Dairy_farm $\overset{\sim}{\to}$ Dairy_farm	HTF4
Handle cow death	HTF2.1
save_cow_death: cow × date × causes of death × List of cows × List of calves in calf rearing unit × Dead cows form → list of cows × dead cows form	HTF3.1
save_cow_death: GT.Cow_id × D.Date × GT.Death_cause × Dairy_farm $\overset{\sim}{\to}$Dairy_farm	HTF4

Table 5.8: Definition of top level functions

TOP LEVEL FUNCTIONS SIGNATURE	**HTFid**
Inseminate artificially	HTF2.1
insem_cow_artif: cow × date × method × insemination form → insemination form	HTF3.1
insem_cow_artif: GT.Cow_id × D.Date × GT.Artif_info × Dairy_farm $\overset{\sim}{\to}$Dairy_farm	HTF4
Inseminate naturally	HTF2.1
insem_cow_natural: cow × date × bull × insemination form → insemination form	HTF3.1
insem_cow_natural: GT.Cow_id × D.Date × GT.Bull_id × Dairy_farm $\overset{\sim}{\to}$Dairy_farm	HTF4
Manage birth	HTF2.1
give_birth: cow × calf × date of birth × birth form × set of cows → birth form × set of cows	HTF3.1
give_birth: GT.Cow_id × GT.Calf_sex × D.Date × Dairy_farm $\overset{\sim}{\to}$Dairy_farm	HTF4
Record cow deparasitation	HTF2.1
deparasite_cow: cow × date × substance × dose × deparasitation form → deparasitation form	HTF3.1
deparasite_cow: GT.Cow_id × D.Date × GT.Dep_inf × Dairy_farm $\overset{\sim}{\to}$Dairy_farm	HTF4
Record milking	HTF2.1
milk_cow: cow × date × litres of milk × milking form → milking form	HTF3.1
milk_cow: GT.Cow_id × D.Date × GT.Litres × Dairy_farm $\overset{\sim}{\to}$Dairy_farm	HTF4
Record cow weight	HTF2.1
weigh_cow: cow × date × weight × weight form → weight form	HTF3.1
weigh_cow: GT.Cow_id × D.Date × GT.Weight × Dairy_farm $\overset{\sim}{\to}$Dairy_farm	HTF4
Record cows on heat detection	HTF2.1
detect_heat: group × date × times × list of cows on heat × group form → group form	HTF3.1
detect_heat: GT.Group_id × D.Date × (GT.Cow_id × Bool)* × Dairy_farm $\overset{\sim}{\to}$ Dairy_farm	HTF4
Register pregnancy test	HTF2.1
detect_pregnant_cow: cow × date × test × insemination form → insemination form	HTF3.1
detect_pregnant_group: GT.Cow_id × D.Date × Bool × Dairy_farm $\overset{\sim}{\to}$ Dairy_farm	HTF4
Select a calf group	HTF2.1
select_calf_group: calf × list of groups → group	HTF3.1
select_calf_group: GT.Cow_id × D.Date × Dairy_farm $\overset{\sim}{\to}$ GT.Group_id × Dairy_farm	HTF4
Sell cow	HTF2.1
sell_cow: cow × date × set of cows × sale form → set of cows × sale form	HTF3.1
sell_cow: GT.Cow_id × D.Date × Dairy_farm $\overset{\sim}{\to}$ Dairy_farm	HTF4
Send calf to the calf rearing unit	HTF2.1
send_calf_to_cru: calf × date × calf rearing unit set of calves → calf rearing unit set of calves	HTF3.1
send_calf_to_cru: GT.Cow_id × D.Date × Dairy_farm $\overset{\sim}{\to}$ Dairy_farm	HTF4
Send out to pasture	HTF2.1
send_to_pasture: group × date × period × plot ×group form → group form	HTF3.1
send_to_pasture: GT.Group_id × D.Date × × Nat × GT.Plot_id × GT.Field_id × Dairy_farm $\overset{\sim}{\to}$ Dairy_farm	HTF4
Take calf out the calf rearing unit	HTF2.1
take_calf_out_cru: calf × date × calf rearing unit set of calves → calf rearing unit set of calves	HTF3.1
take_calf_out_cru: GT.Cow_id × D.Date × Dairy_farm $\overset{\sim}{\to}$ Dairy_farm	HTF4
Vaccinate cow	HTF2.1
vaccinate_cow: cow × date × vaccine × vaccination form → vaccination form	HTF3.1
vaccinate_cow: GT.Cow_id × D.Date × GT.Vaccine × Dairy_farm $\overset{\sim}{\to}$ Dairy_farm	HTF4

Table 5.9: Definition of top level functions

To define these lower level functions we proceed as we explained in Section 4.3.3. The signature of the functions across the different levels only change in the identifying parameters. Functions vaccinate_cow and can_receive_vacc need to access and modify an individual cow, so considering the set of cows in a dairy farm were modelled with a map, the identifying parameters would be in both cases Cow_id and Cows, the map domain and the map respectively. The rest of the arguments remain the same. Concerning the result type of the function vaccinate_cow (it is a generator function), we use the map Cows as it contains the domain component to be modified. The bodies of these second level functions will contain a call to a function defined in a lower level module, in this case the module COW, the one that manipulates each individual cow. We include below the RSL specification for all these lower level functions.

In the module COWS (instantiated as the object CS in DAIRY_FARM)

```
value
   can_receive_vacc : GT.Cow_id × D.Date × GT.Vaccine × Cows → Bool
   can_receive_vacc(ci, d, vacc, cs) ≡
      ci ∈ cs ∧ C.can_receive_vacc(d, vacc, cs(ci)),

   vaccinate_cow: GT.Cow_id × D.Date × GT.Vaccine × Cows ~→ Cows
   vaccinate_cow(ci, d, vacc, cs) ≡ cs † [ ci ↦ C.vaccinate_cow(d, vacc, cs(ci)) ]
   pre can_receive_vacc(ci, d, vacc, cs)
```

In the module COW (instantiated as the object C in COWS)

```
value
   can_receive_vacc : D.Date × GT.Vaccine × Cow → Bool
   can_receive_vacc(d, vacc, c) ≡ ...,

   vaccinate_cow: D.Date × GT.Vaccine × Cow ~→ Cow
   vaccinate_cow(d, vacc, c) ≡
      chg_history(CH.add_event(d, CE.vaccination(vacc), history(c)), c)
   pre can_receive_vacc(d, vacc, c)
```

When defining top level functions, it may be possible to determine in which module to store some of the types derived during the Derivation of Types step and which could not be precisely located when defining the modules, as we mentioned in Section 5.4.2. For example, from the definition above we can see the type Vaccine must be accesible from at least two modules, the module DAIRY_FARM and the module COWS. For this reason, we store it in the module GENERAL_TYPES. Making a similar analysis, we could determine that some many other types such as Corn_sil, Hay, Conc, and Death_cause should be also be defined in the GENERAL_TYPES modules.

There is another point we would like to comment. In addition to the 29 top level functions coming from heuristic HTF1, and the functions modelling their corresponding preconditions the module DAIRY_FARM contains some other definitions of functions. For example, according to heuristic HDT1.2 we do not store what can be computed. Then, we should include the function field_hectare_loading to calculate the hectare loading of a field, as this property was not modelled as a component of the type Field (Table 5.6). Besides, there are some many other functions not derived from the heuristics but included later to check consistency conditions.

5.5 The architecture of the Milk Production System RSL specification

The RSL specification derived in the previous section represents the activities that occur in a dairy farm. This specification is in an applicative sequential style. Figure 5.2 displays how this specification can be structured in layers using the Layers Pattern (Section 4.2.3). Shadowed boxes represent global objects while the others represent schemes.

Although global objects could be avoided, we decided to use them in some cases in order to avoid having modules with many parameters. For example, as the operations applied to cows and groups of cows were basically the same, we decided to provide a general solution and then we had to define the global objects CE, CH, GE and GH as we showed in Figure 5.1. These objects were placed in the specific layer.

The specific layer includes schemes and global objects that are specific to a dairy farm application. For example, the schemes COWS, COW, and COW_EVENT contain the specific operations applied to cows that belong to a dairy farm while the schemes COW_GROUPS, COW_GROUP, and GROUP_EVENT specify the particular way in which cows are grouped in a dairy farm. Schemes such as BULLS and FIELDS are defined in the general layer because they are common to different applications within the agricultural systems infrastructure. Although the scheme GENERAL_TYPES and its corresponding global object GT may seem to be application-specific, as they contain the definition of most of the types used in the specification, we placed them in the general layer because they must be accessible not only from modules in the specific layer but also from modules in the general layer itself. The schemes that appear in the middleware layer can be used in any domain. HISTORY specifies a list of events ordered by date, EVENT_INFO defines the individual elements to be included in the list, and DATE and its corresponding global object D contain definitions of types and functions related to dates and periods of time.

5.6 Conclusions from the case study developed

We have presented the derivation of a specification in RSL for a Milk Production System, by applying the technique we proposed in Chapter 4. There are several issues that arose while deriving the specification from the LEL and the scenarios.

5.6.1 Ambiguity

Natural language is suited to validation with stakeholders. But its expressiveness and flexibility mean that natural language descriptions are open to misinterpretation, i.e. can be ambiguous. Their syntactic flexibility also make them hard to process by automatic tools.

However, some of the problems found in the derivation, and associated with natural language flexibility, could be overcome if stronger standards or guidelines were imposed on the way of describing LEL terms and scenarios. Even though they have a precise structure and it is established what to write in their components, the same semantics may be usually expressed with many different natural language sentences. We think in some cases would be possible to define a standard form, without restricting the power of expression of natural language. When writing the LEL for the Milk Production System domain, we followed some informal rules to describe similar features in different terms. For example, we use a consistent natural

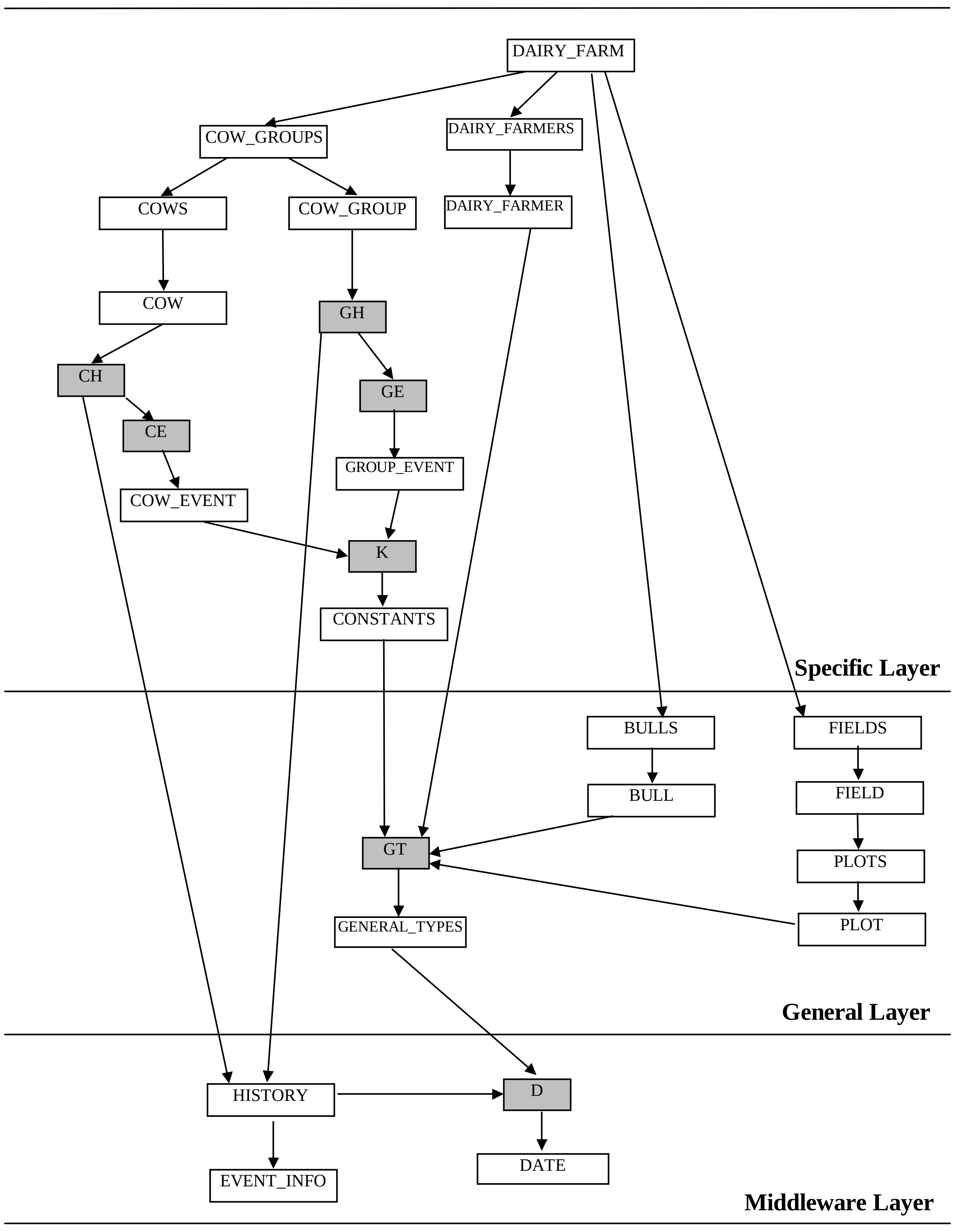

Figure 5.2: Architecture of the Milk Production System RSL Specification

language structure to express a component of a term: “An x has a y”. Other LELs have used other structures, like just mentioning “Published by a publisher” without stating “A book has a publisher”. But, a deeper analysis should be carried out.

The decision about which restrictions or structure to impose to natural language should be very carefully taken. The use of a restricted or controlled natural language, defined as a subset of natural language, may simplify natural language computational processing, and as it is still natural language, stakeholders can understand it correctly. Some of the disadvantages of this solution may be the reduction of the expressive power of natural language, and the training people involved will need in order to use this controlled language correctly. Some people have gone much further in this direction, e.g. [9, 40]. The risk is that when introducing constraints to natural language one finishes defining something similar to a new formal language. This new language will have a new meaning and thus stakeholders will not be able to interpret it correctly. In addition, if there is a need to use a formal language it would be possible to choose one from the formal languages available nowadays, or directly use mathematics.

Sub-scenarios may be used when common behaviour is detected in several scenarios. We showed in Table 2.8, Section 2.2, that two sub-scenarios appeared as episodes of the scenario Manage birth. We could not find proper rules for relating subjects or objects mentioned in the episodes of a scenario, to the scenario resources and actors. Moreover, the general heuristics to write scenarios presented in [30] suggest not including the resources and actors of a sub-scenario when describing the main scenario. In the scenario Manage birth, for example, the inclusion of the sub-scenario ASSIGN A GROUP TO A COW hides the fact that the object Cow is the one involved. Without the definition of the sub-scenario this would not have happened, because from the episodes it would have been clear enough that the group should be assigned to the Cow and not to the Calf.

5.6.2 Completeness

A LEL's main goal is to describe the application domain language, and the principles of circularity and of minimal vocabulary are suggested as ways of achieving this goal. However, an important point is determining where domain language ends, i.e. which terms must and must not be included in the LEL. We think there is no absolute definition because although some agreement can be reached about specific terms, such as dairy cow, milking, and vaccination from our case study, the inclusion of others may depend on what the software engineer who is writing the LEL considers in or out the domain language. We show an example in Section 4.1.2 involving the words Size and Location, which were not included in the LEL. This might be because they were omitted, in which case it would be necessary to return to the LEL to define them. But this omission might be also justified saying that they are “basic” words not worth defining. It is true that their meaning may be more or less clear to almost everybody, and this kind of situation gives rise to another problem which are unspoken assumptions. Domain experts do not explain some things assuming software engineers know them, and on the other hand when software engineers construct models they take some decisions assuming their understanding of the terms coincides with the domain experts understanding. The word Size, for example, appears in the Longman Defining Vocabulary [2]. The entry of the word in the dictionary includes several definitions with quite different meanings. So, if its meaning in the application domain is not precisely stated, a software engineer may assume it represents the measure of a field in hectares, while for the domain expert it is any of a set of measures, such as small, medium, or large.

The solutions to the problems explained above should be in some way complementary. Unspoken assumptions should be avoided in order to obtain a complete LEL, in which an agreement on each definition could be reached by domain experts and software engineers. In addition, if the LEL is expected to be the input to a next software development step, such as the specification phase in our proposal, it should be as complete as possible. However, a detailed LEL implies a big effort not only in the construction process but also in its maintenance. This is even worse when the domain is well-known, because people will find it tedious to write and maintain definitions of concepts that are supposed to be well understood.

How to balance details with unspoken assumptions is an issue difficult to solve.

Concerning the LEL for our case study, we should say that it may appear more detailed than others, and thus the effort to develop it was quite considerable. The most important reason is that, as it is not a conventional domain, it was necessary to include all the information available. In addition, as we knew it would be an important resource for the derivation step, we tried to make it as complete as possible.

5.6.3 Maintenance

Documentation maintenance is a problem in any software development project.

The specification should be consistent with the description the LEL and scenarios provide about the domain. It is common to detect errors or omissions when deriving and refining the specification, or even when writing the scenarios. This implies returning to the LEL or the scenarios to change what was wrong or to add what was missing.

Without the appropriate tools support, this process is very tedious and time consuming, and thus, the size of the LEL and the number of the scenarios are again a critical point.

5.6.4 Domain analysis/Requirements analysis

According to the Requirements Baseline [30, 32] approach, the LEL aims at describing the application domain language, and the scenarios describe specific application domain situations, according to the main actions performed outside the software system. Thus, scenarios can be used to describe, with the required level of detail, any situation in the application domain. For example, in our case study it would be possible to write a scenario called Milk a milking cow, enumerating in the episodes all the activities performed by a dairy farmer, such as taking the milk from the milking cow, putting the milk in a container, measuring the litres extracted and recording this information.

The initial specification represents a mapping from real-world concepts onto RSL constructions. Before writing this specification it should be decided the kind of system that will be finally implemented. In our case study, we decided to model an information system instead of a control one. This means that, for example, the system we modelled does not milk cows, it only records the information related to this activity.

As we have previously explained, our goal is the definition of a set of heuristics to derive an initial specification from the LEL and the scenarios. When constructing the LEL we take into consideration all the information that could be recovered from the application domain. However, as scenarios would be the main source to define the functions in the specification, we filtered the information coming from the LEL to include in the scenarios only those situations that could be modelled in the specification of an information system. So, most of the scenarios which describe the recording of information are quite similar and trivial, and they may look like use-cases [24]. The way in which each scenario was defined, following the structure proposed in [30] that distinguishes actors, resources, initial states, etc., turned out to be useful in the specification of the functions.

The LEL and scenarios for our case study are closer to domain analysis than to requirements analysis. They represent how things actually occur in a Milk Production System. Requirements analysis will show the need for new functions, for example to carry out statistical analysis of the data, or many other specific system requirements not covered in our analysis such as the ones related to hardware, users, access rights, and backups.

Chapter 6

Validating the RSL specification

There are two aspects to showing correctness that are commonly distinguished [22]:

- **Validation**, which is the check that we are creating what is required, and it can be expressed as the check that we are "solving the right problem". It is necessarily informal, as the check is against requirements written in natural language.
- **Verification**, that consists in checking, with varying degrees of formality, the development process is correct. It can be expressed as the check that we are "solving the problem right".

The aim of validating the RSL specification is to check that we have written the right specification, i.e. that we have met the requirements. As we try to make the initial specification a contract between software engineers and stakeholders, the validation of the specification turns to be a very important step. Discovering and fixing requirements problems can help to reduce the amount of rework to do because of mistakes in the initial specification. To validate a specification we must look outside it, at the requirements. But as usually requirements are written in natural language, validation cannot be formalised and so there is no way to demonstrate that a requirements specification is correct. The validation process can only increase stakeholders'confidence that the initial specification represents a clear description of the system for design and implementation [45].

Verification is concerned with checking the final implementation conforms to the initial specification. As it assumes the correctness of the initial specification, it must be done after validation.

Together, validation and verification help assure we are "writing the right specification right". Validation needs requirements traceability, in order to relate them to where they are met either in the initial specification or in a later development. Once we are sure a requirement has been captured, we use verification to control it remains captured.

In this chapter we briefly describe some techniques proposed in [20] to validate an initial specification in RSL, and then we present the approach we adopt to validate the RSL specification obtained by applying the technique we proposed in Chapter 4.

6.1 Techniques to validate a RSL specification

There are a number of techniques to validate a specification in RSL [20]. The main technique in validation is to check that each requirement is met. After writing the initial specification, we go back to the requirements to determine, for each issue that we can find, one of the following:

- It is met.
- It is not met, and so we will have to change the specification.
- It is not met because, for some reason, we think it is not a good idea and so we need to discuss with the customers.
- It is deferred to later in development, and so we add it to a list of the requirements against which later development steps will be validated. This applies to non-functional requirements and to things that we have not yet designed, as for example aspects of user interface or particular algorithms to be used.

The rest of the validation techniques mentioned are the following:

- Read the specification to look for properties it will have and which are not mentioned in the requirements.
  It is typical that customers omit to mention issues that seem too obvious to them, as for example if a data structure should be initialised, and if so to what. As a consequence scenarios, or use cases, often lack essential but, to the customer, obvious things. Then, it would be necessary to set up a formal procedure of queries to customers and their answers being documented.
- Develop system tests (test cases and expected results)
  This may help to clarify the requirements, as it is a way of revealing problems such as incompleteness and ambiguity. Besides, the tests can be shown to the stakeholders who will usually find them easier to read than the formal specification.
- Rewrite the requirements from the specification
  Although it is an expensive task, it generally helps one obtain requirements documents thar are clearer, better structured, more concise, and more complete than the originals.
- Prototype all or parts of the system.
  By trying out the system prototype stakeholders will see if it meets their real needs, and then, they can make suggestions for improvements. One possibility is to do a quick and simplified refinement of the abstract types in the specification, and then use the translators to SML [49] or C++ [3]. These tranlators are part of the RAISE tools [19], and they allow one to run test cases in order to get a feeling of what the specification really does.

The application of any of the validation techniques mentioned above, provides early feedback to the stakeholders. This has the added advantage of committing them to what has be done so far. It also helps stakeholders understand the added cost and danger of later requirements changes. The aim is to make the initial specification a contract between the software engineers and the stakeholders.

6.2 Our approach to validate the specification

Our goal in validating the RSL specification obtained applying the technique we proposed is to check this specification meets the requirements modelled with LEL and scenarios.

Considering the validation techniques presented in the previous section, we think a combination of prototyping with system tests would help us to achieve our goal. The main reason for selecting these techniques is we could take advantage of the translators already implemented as part of the RAISE tools [19], such as the SML translator, thus minimising the development costs for the prototype. The prototype obtained using the SML translator will not only help us in checking the specification against LEL and scenarios, but it may also assist in clarifying the real requirements for the system, as the stakeholders may participate in this validation task. Running the prototype with appropriate test cases stakeholders may find easier to discover problems with poorly understood requirements, and then suggest how the requirements may be improved.

6.2.1 The SML Translator: a brief description

RSL is a wide-spectrum specification and design language. This means it can be used to formulate initial, very abstract specifications as well as to express low level designs suitable for translation to programming languages. Though RSL is more suitable for the abstract specification of general problems, it provides a complete set of syntactical primitives to describe concrete implementations. It would be of help to have an RSL interpreter to run concrete specifications in order to get a feeling of what each specification really does.

As an answer to this issue it was proposed in [49] to use an existing runtime system, the Standard ML of New Jersey [1], to be the back end. Standard ML of New Jersey is a compiler for the Standard ML '97 (SML) programming language with associated libraries, tools, and documentation. SML [1] is a safe, modular, strict, functional, polymorphic programming language with compile-time type checking and type inference, garbage collection, exception handling, immutable data types and updatable references, abstract data types, and parametric modules. It has efficient implementations and a formal definition with a proof of soundness.

The advantage of using this existing runtime system, instead of interpreting RSL directly, is that it saves the time and eliminates the difficulties in implementing a runtime system, such as instruction generation, and garbage collection, which are not trivial. Although translating the functional part of RSL into SML manually is not difficult, the RSL to SML translator avoids the users to have to learn another programming language. Translation to SML is mainly intended for prototyping and testing.

RSL has a very rich set of features but not all of them can be translated into a functional programming language, like SML. RSL elements such as abstract types, axioms, post expressions, and implicit values and implicit functions cannot be currently translated into SML. So, sometimes would be necessary to make some refinements in order to get a concrete RSL specification which could be translated into SML. A complete description of this translator can be found in [49], and as the rest of the RAISE tools it can be downloaded from UNU/IIST's web site.

6.2.2 Validating our specification

The initial specification derived applying the three step process we described in Chapter 4 is an applicative one, i.e it is written in terms of definitions and applications of functions. Besides, it may have some abstract type and function definitions. So, to make use of the SML translator, we did a quick and simplified refinement of the abstract types we found to obtain a concrete applicative version of the derived specification. For some of the types, as we did not have enough information, we gave a temporary definition which could be replaced later by a more appropriate one. For example, for types such as Photo, Brand and Features we did not have information, then we defined them of type Text, leaving for a later development their definitive specification.

In addition, we defined an appropriate set of test cases in order to run the specification with them and check if the specification does what was required. Scenarios may be of great help when designing appropriate test cases. The goal of a scenario contains the aim to be reached in the domain after performing the scenario. Then, to validate each function in the specification we suggest going to the scenario that motivated its definition, and analysing the goal to define one or more test cases. In addition, the scenario context may help to define appropriate test cases to check partial functions. In our case study, Tables 5.8 and 5.9 contain for each top level function derived applying the heuristics, the scenario that caused its definition.

Test cases are always evaluated in order of definition, and this is particularly useful for imperative specifications with variables to store information [20]. As the information stored as a result of one test case is available for the next one, it would be possible to test scenarios step-by-step by using a sequence of test cases. To achieve this, we formulated a concrete imperative specification from the concrete applicative one.

Chapter 7

Conclusions

The advantage of formal methods such as RAISE is they help to avoid requirements ambiguities and misinterpretations, and they provide a correct software development process based on mathematical proofs. But, formal specifications are usually only accessible to formal methods specialists. This is particularly inconvenient during the first stages of the software development process, when the participation of stakeholders, unfamiliar with this kind of description, is crucial as the definition of complete and precise requirements cannot be done without an involvment of the stakeholders, who are responsible for supplying information and validating the final requirements. Stakeholders' participation can be guaranteed if natural language is used.

Then, to contribute to bridge the gap between stakeholders and the formal methods world, we have presented a three-step process to derive an initial formal specification in RSL from LEL and scenarios, two natural language models belonging to the Requirements Baseline. Once this initial specification is derived, the process may continue with the steps proposed in the RAISE Method. For example, the initial applicative and partially abstract specification derived could be developed into a concrete one to make use of the SML translator and, thus, obtain a quick prototype to validate the specification and get a feeling of what it really does.

For each step of the process we proposed, we defined a number of heuristics which are guidelines about how to start with the definition of an initial specification, taking into account the structured description of a domain provided by LEL and scenarios. The LEL provides structural features of the relevant terms in the domain, thus limiting the definition of types to those that correspond to significant terms. Using the behavioural description represented in the scenarios, it is possible to identify the main functionality to model in the specification. In addition, the structure proposed in [30] to describe each scenario makes simpler the derivation of function signatures. However, even though LEL and scenarios have a precise structure and it is established what to write in their components, the same semantics may be usually expressed with many different natural language sentences. But, as we have explained in Section 5.6 we think some of the problems found in the derivation, and associated with natural language expressiveness and flexibility, could be overcome if stronger standards or guidelines were imposed to the way of describing LEL symbols and scenarios.

In order to validate our proposal, we applied the three-step process designed to a complete case study, the Milk Production System (Chapter 5). The experiences gained during this development helped us to complete, improve, and refine the heuristics proposed.

7.1 Main contributions

The following are the main contributions of our work:

- *A technique to be used in the first stages of development using the RAISE Method*

  [20] and [22] illustrates how to specify and develop systems using RAISE. When analysing developments of RSL specifications of different domains, we found they start from informal descriptions containing synopsis, narrative, and terminology (Chapter 1). Once obtained these informal descriptions, in general, each case followed its own approach to obtain the RSL specification, though of course they all considered the principles proposed in the RAISE Method.

  We proposed and defined a concrete and detailed three-step process that could be applied in any domain, allowing to take profit of informal descriptions and reducing the gap between them and the final RSL specification.

- *The possibility of using a layered architecture for the specification*

  The three-step process we developed gives as result a set of modules hierarchically structured, aiming at increasing the maintainability and legibility of the specification. The hierarchy of RSL modules obtained can be mapped onto a layered architecture by describing the structure of modules using the Layers pattern. This architecture is the basis to start applying the steps of the RAISE Method and provides the specific properties all its developments should have. This means that, for example, any implementation or extension development step should preserve the layers and the relationships among them. The use of a layered architecture is particularly useful when designing complex systems, because it facilitates and encourages not only reuse but also separate and step-wise development.

- *Fruitful use of the large amount of information usually available after problem analysis*

  LEL and scenarios provide a detailed description of an application domain, and as we have already mentioned, they are valuable for supporting communication among software engineers and stakeholders. But, an important point with them is how to fruitfully use all the information they contain along the software development process. By using the three-step process we proposed, the effort to define complete requirements models is worth doing because, though partially, they could be later mapped onto a formal specification.

- *Achievement of an executable specification for a rapid prototyping of requirements*

  The heuristics we defined followed closely the principles the RAISE Method proposes, so the initial specification derived could be later developed into a concrete one according to the steps provided by the RAISE Method. With a concrete specification the SML translator could be used in order to have a quick prototype and get a feeling of what the specification really does. In Chapter 6, we exemplified how to achieve this by using the case study we selected.

- *Tracking of traceability relationships* A significant factor in quality software implementation is the ability to trace the implementation through the stages of specification, architecture, design, implementation, and testing [29]. The traceability relationship may be defined in terms of a simple “traced-to” and “traced-from” model.

The tables we presented in Chapter 5, may be considered a first attempt to track these relationships. Though they contain the information in a “traced-to” way (for example, they show how a LEL symbol is modelled with a type or a function), the “traced-from” relationships could be added by including appropriate comments in the RSL specification derived. However, a more detailed and deeper analysis should be made because tracing relationships are not always one-to-one.

7.2 Future work

We plan to improve the three-step process we proposed by refining and completing the heuristics presented in this work, though obviously a complete automatic derivation is by no means possible, as LEL and scenarios contain all the necessary and unavoidable ambiguity of the real world, while the specification contains decisions about how to model this real world. The analysis of other case studies may help in this point.

It would also be interesting to have a tool to assist in the derivation process. At present, there are two groups of students working in the development of two different tools. One of the groups, is developing a web-based application that not only implements the three-step process we have proposed but also assists in the construction of LEL and scenarios. Extreme Programming (XP) [4] is the software development methodology selected to guide the construction of the tool, and Java Server Pages (JSP) is used to separate the dynamic part of the web pages from the static HTML. Both tools could be later integrated with the RAISE tools in order to have assistance in the RSL specification complete development process.

As we have mentioned in Section 7.1, tracking of traceability relationships is an important issue that needs further analysis. The tool we have already mentioned may also include assistance to follow relationships that may exist between elements in the requirements models and the RSL specification, and vice versa.

Bibliography

[1] Standard ML of New Jersey. http://www.smlnj.org/.

[2] *Longman Dictionary of Contemporary English.* Longman, 3rd edition, 1995.

[3] U. Ahn and C. George. C++ Translator for RAISE Specification Language. Technical Report 220, United Nations University/International Institute for Software Technology, Macau, November 2000.

[4] K. Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 2000.

[5] D. Bjorner. Software Engineering: A New Approach. Lecture notes, Technical University of Denmark, 2000.

[6] D. Bjorner, C. George, and S. Prehn. *Industrial-Strength Formal Methods in Practice*, chapter "Scheduling and rescheduling of trains". Springer-Verlag, 1999.

[7] B. Bryant and B. Lee. Two-Level Grammar as an Object-Oriented Requirements Specification Language. In *Proceedings of the 35th Hawaii International Conference on System Sciences*, pages 1–10. IEEE Press, 2002.

[8] B. R. Bryant. Object-Oriented Natural Language Requirements Specification. In *Proceedings of ACSC 2000, 23rd Australasian Computer Science Conference*, pages 24–30, 2000.

[9] J.F.M Burg. *Linguistic Instruments in Requirements Engineering.* IOS Press, Netherlands, 1997.

[10] F. Buschman, Meunier R., H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture.* John Wiley and Sons, 1996.

[11] A. Dasso. A Course on Formal Methods using RAISE. Technical Report 114, United Nations University/International Institute for Software Technology, Macau, June 1997.

[12] L. De Bortoli. Estudo de Caso: Biblioteca do Instituto de Informatica da Universidade Federal do Rio Grande do Sul. Porto Alegre, Brasil, 1999.

[13] M. Del Fresno, J. Doorn, C. Leonardi, V. Mauco, M. Ridao, and L. Rivero. Modelo de Escenarios y LEL para el Caso de Estudio del Círculo Cerrado de Compra de un Automóvil. Work presented for the Course Requirements Engineering, Universidad Nacional del Centro de la Provincia de Buenos Aires, Argentina, 1997.

[14] M. del Fresno, V. Mauco, M. Ridao, J. Doorn, and L. Rivero. Derivación de Objetos Utilizando LEL y Escenarios en un Caso Real. In *Proceedings of WER'98 - Workshop en Engenharia do Requisitos*, pages 79–90, 1998. Maringa. Brazil.

[15] W. Dzida and R. Freitag. Making Use of Scenarios for Validating Analysis and Design. *IEEE Transactions on Software Engineering*, 24(12):1182–1196, December 1998.

[16] N. Fuchs, U. Schwertel, and S. Torge. Controlled Natural Language can Replace First-order Logic. In *Proceedings of IEEE Int.Conf. on Automated Software Engineering*, 1999.

[17] O. Garcia and C. Gentile. Escenarios de la Construcción de Escenarios: Autoaplicación de la Metodología. Tesis de grado, Universidad Nacional del Centro de la Provincia de Buenos Aires, Argentina, 2000.

[18] D. Garlan. Software Architecture: a Roadmap. In *The Future of Software Engineering.* ACM Press, 2000.

[19] C. George. RAISE Tools User Guide. Research Report 227, UNU/IIST, Macau, February 2001.

[20] C. George. Introduction to RAISE. Technical Report 249, United Nations University/International Institute for Software Technology, Macau, March 2002.

[21] The RAISE Language Group. *The RAISE Specification Language.* BCS Practitioner Series. Prentice Hall, 1992.

[22] The RAISE Method Group. *The RAISE Development Method.* BCS Practitioner Series. Prentice Hall, 1995.

[23] G. Hadad, G. Kaplan, and J. Leite. Léxico Extendido del Lenguaje y Escenarios del Meeting Scheduler. Technical report, Universidad de Belgrano, Buenos Aires, Argentina, 1998.

[24] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.

[25] N. Juristo, A. Moreno, and M. Lopez. How to Use Linguistic Instruments for Object-Oriented Analysis. *IEEE Software*, pages 80–89, May-June 2000.

[26] B. Lee. Automated Conversion from a Requirements Documentation to an Executable Formal Specification. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE 2001)*, 2001.

[27] B. Lee and B. Bryant. Automated Conversion from Requirements Documentation to an Object-Oriented Formal Specification Language. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, pages 932–936, 2002.

[28] B. Lee and B. Bryant. Prototyping of Requirements Documents Written in Natural Language. In *Proceedings of SESEC 2002, the 2002 Southeastern Software Engineering Conference*, 2002.

[29] D. Leffingwell and D. Widrig. *Managing Software Requirements: A Unified Approach.* Addison-Wesley, 2000.

[30] J. Leite, G. Hadad, J. Doorn, and G. Kaplan. A Scenario Construction Process. *Requirements Engineering Journal*, 5(1):38–61, 2000. Springer-Verlag.

[31] J. Leite and A. Oliveira. A Client Oriented Requirements Baseline. In *Proceedings of the Second IEEE International Symposium On Requirements Engineering*, pages 108–115, 1995.

[32] J. Leite, G. Rossi, V. Maiorana, F. Balaguer, G. Kaplan, G. Hadad, and A. Oliveros. Enhancing a Requirements Baseline with Scenarios. *Requirements Engineering Journal*, 2(4):184–198, 1997. Springer-Verlag.

[33] C. Leonardi. Una Estrategia de Modelado Conceptual de Objetos basada en Modelos de Requisitos en Lenguaje Natural. Master's thesis, Universidad Nacional de La Plata, Argentina, November 2001.

[34] C. Leonardi, V. Maiorana, and F. Balaguer. Una Estrategia de Análisis Orientada a Objetos basada en Escenarios. In *Actas II Jornadas de Ingeniería de Software JIS '97*, pages 87–100, Donostia, San Sebastián, España, 1997.

[35] V. Mauco and C. George. Using Requirements Engineering to Derive a Formal Specification. Technical Report 223, United Nations University/International Institute for Software Technology, Macau, December 2000.

[36] V. Mauco, D. Riesco, and C. George. Deriving the Types of a Formal Specification from a Client-Oriented Technique. In *Proceedings of the 2nd International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pages 1–8, Japan, 2001.

[37] V. Mauco, D. Riesco, and C. George. Heuristics to Structure a Formal Specification in RSL from a Client-oriented Technique. In *Proceedings of the 1st Annual International Conference on Computer and Information Science (ICIS 01)*, pages 323–330, U.S.A., 2001.

[38] V. Mauco, D. Riesco, and C. George. Using a Scenario Model To Derive the Functions of a Formal Specification. In *Proceedings of the 8th Asia-Pacific Software Engineering Conference (APSEC 2001), IEEE Press*, pages 329–332, Macao, 2001.

[39] V. Mauco, D. Riesco, and C. George. A Layered Architecture for a Formal Specification in RSL. In *Proceedings of the International Conference on Computer Science, Software Engineering, Information Technology, e-Business and Applications (CSITeA02)*, pages 258–263, Brazil, 2002.

[40] A. Moreno Capuchino, Juristo N., and Van de Riet R.P. Formal justification in Object-oriented Modelling: A Linguistic Approach. *Data and Knowledge Engineering*, 33(1):25–47, April 2000. Elsevier.

[41] B. Nuseibeh and S. Easterbrook. Requirements Engineering: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 35–46. ACM, 2000.

[42] Pak Jong Ok, Ri Hyon Sul, and C. George. A Management System for a University Library. Technical Report 186, United Nations University/International Institute for Software Technology, Macau, February 2000.

[43] M. Patras and R. Moore. A Formal Model of an Agent-mediated Electronic Market. Technical Report 211, United Nations University/International Institute for Software Technology, Macau, August 2000.

[44] I. Sommerville. *Software Engineering*. Addison-Wesley, 2001.

[45] I. Sommerville and P. Sawyer. *Requirements Engineering: A Good Practice Guide*. John Wiley and Sons, 1998.

[46] J. Tapamo. Domain Analysis of a System of Assessment of Natural Resource Usage. Technical Report 179, United Nations University/International Institute for Software Technology, Macau, November 1999.

[47] A. van Lamsweerde. Formal Specification: a Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 147–159. ACM, 2000.

[48] A. van Lamsweerde. Requirements Engineering in the Year 00: A Research Perspective. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 5–19. ACM, 2000.

[49] K. Wei and C. George. RSL to SML Translator. Technical Report 208, United Nations University/International Institute for Software Technology, Macau, August 2000.

[50] J. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, pages 8–24, September 1990.

Appendix A

The Lexicon View

ARTIFICIAL BREEDING/BREED ARTIFICIALLY
Notion

- It is the breeding of a calf away from its mother, in a place called calf rearing unit.
- It is carried out by a dairy farmer.

Behavioural Response

- It starts when a dairy farmer sends calf to the calf rearing unit.
- Calf is fed with 4-5 litres of milk or milk replacement and at most 1 kg of balanced food per day.
- The quantity of milk replacement and balanced food given per day to each calf in the calf rearing unit is saved in the Artificial breeding form.
- It finishes when the dairy farmer takes calf out the calf rearing unit.

ARTIFICIAL INSEMINATION/INSEMINATES ARTIFICIALLY
Notion

- Insemination without using a bull.
- A dairy farmer puts sperm taken, from a bull, into the dairy cow or heifer body.

Behavioural Response

- It is performed 12 hours after the dairy cow or the heifer is detected to be on heat.
- Specific information relative to the procedure is saved in the Insemination form.

ASSIGNS TO A GROUP/ASSIGNED TO A GROUP
Notion

- A dairy farmer adds a cow to a group.

Behavioural Response

- It is done daily.
- A dairy farmer has a list of current defined groups.
- If the cow is a post-birth cow which has a recent birth, or a milking cow whose individual production is at least 10 per cent greater than the individual production of the dairy farm, group of type 1 is selected.
- If the cow is a dairy cow in group of type 1 and its last birth date is greater or equal than 3 months, group of type 2 is selected.
- If the cow is a dry cow or a pregnant heifer whose next birth date is within 15-20 days, pre-birth cow type group is selected.
- If the cow is an early pregnant cow in its seventh month of pregnancy, dry cow type group is selected.
- If the cow is a discard cow, discard cow type group is selected.
- If the cow is a calf, a dairy farmer selects a calf group.
- If the cow is a heifer, heifer type group is selected.
- The arrival date for the new group and the identification number of the cow are registered.

BALANCED FOOD/BALANCED
Notion

- It is a mixture of cereals given to cows.

Behavioural Response

- It is measured in kilograms of dried feedstuffs.
- The quantity given to each cow is registered.

BE ON HEAT/ON HEAT/HEAT
Notion

- A dairy cow or a heifer is in a sexual condition ready for being inseminated.
- It is detected by a dairy farmer.

Behavioural Response

- It happens every 21 days.
- The dairy cow or heifer should not be pregnant.

- Heat is registered.
- The dairy cow or heifer can be inseminated.

BIRTH/CALVING/GIVE BIRTH
Notion

- It is the coming of a calf out of a dairy cow's or heifer's body.

Behavioural Response

- If a dairy cow is involved, it should be a pre-birth cow.
- If a heifer is involved, it should be in the ninth month of pregnancy.
- The dairy farmer saves birth.

BRAND
Notion

- Mark which identifies the dairy farm.

Behavioural Response

- It is stamped on calves bodies with a hot piece of metal when they are in the calf rearing unit.

BULL
Notion

- Male in the dairy cow family.
- It has a name.
- It has a date of birth.
- It has a set of characteristics.
- It has a date of purchase.
- It is in a field.

Behavioural Response

- It is used to inseminate naturally dairy cow or heifer.
- It may be discarded.

BUYS BULLS
Notion

- The dairy farmer decides to add a new bull to the dairy farm.

Behavioural Response

- The bull is added to the set of bulls of the dairy farm and it is sent to a field.
- The date of the transaction, the features of the bull, and its date of birth are registered.
- The bull is ready to inseminate dairy cows or heifers in the dairy farm.

CALF
Notion

- It is a cow of less than 12 months age.
- Its mother is a dairy cow.
- It may be with its mother, in the calf rearing unit or in a group of type calf.
- It may have a current group.
- It may be in a plot.
- It weighs 40 kg at birth.
- When it is 60 days, it weighs approximately 60 kg.
- It may be male or female.

Behavioural Response

- If it is a male calf, it can be sold after birth.
- After birth, it is kept with its dairy cow mother from 1 to 5 days.
- At most 5 days after birth, it is sent to the calf rearing unit for artificial breeding.
- In the calf rearing unit, it is tied to a stake for 45-60 days.
- During the 45-60 days in the calf rearing unit, it receives 4-5 litres of milk replacement or milk and up to 1 kg of balanced food per day.
- When it is able to eat 1 kg of balanced food, the milk or replacement milk is suspended and it is sent to a small plot to eat grass.
- After leaving the calf rearing unit, it is assigned to a group of type calf.
- It is given balanced food until it is 6 months age.
- After leaving the calf rearing unit, it is deparasited every 2 or 3 months.
- When it is 3, 6, 9 and 12 months it receives vaccination with the triple vaccine.
- If it is female, it has a photograph.

- If it is female, it receives vaccination against brucellosis.

CALF REARING UNIT
Notion

- It is a place where calves of less than 60 days age are kept for artificial breeding.
- It has a set of calves.

Behavioural Response

- Each arrival or departure of a calf is registered.

COMPUTES BIRTH DATE
Notion

- A dairy farmer computes the approximate next birth date for a dairy cow or heifer.

Behavioural Response

- The dairy cow or heifer should be pregnant.
- 9 months are added to the last insemination date.
- Computed date is saved.

COMPUTES INDIVIDUAL PRODUCTION
Notion

- The dairy farmer calculates individual production of a dairy farm, a milking cow or a group.

Behavioural Response

- For a milking cow, it is computed dividing the addition of the litres produced in a period into the number of milking in the period.
- For a group, it is computed dividing the addition of the litres produced by the whole group in a period into the number of milking of the group in that period.
- For a dairy farm, it is computed dividing the addition of the litres produced by all the milking cow in a period into the number of milking in that period.
- The individual production calculated is registered.

COMPUTES PASTURE EATEN
Notion

- A dairy farmer determines the quantity of pasture eaten by each cow in a group.

Behavioural Response

- It is computed as the difference between the ration a cow should eat and the addition of corn silage, hay, concentrated food the cow is given.
- The quantity computed, the date and the identification number of each cow in the group are registered.

COMPUTES RATION
Notion

- A dairy farmer defines the total kilograms for the ration each cow in a group should be given.

Behavioural Response

- If the type of the group is 1 or pre-birth cow, the total kilograms are the 3,5 percent of the weight of an average dairy cow in the group.
- If the type of the group is 2, the total kilograms are the 3 percent of the weight of an average dairy cow in the group.
- If the type of the group is dry cow or discard cow, the total kilograms are the 2 percent of the weight of an average dairy cow in the group.
- If the type of the group is heifer, the total kilograms are the 3 percent of the weight of an average heifer in the group.
- If the type of the group is calf and group range of ages are between 2 and 4 months, the total kilograms are the 2.2 percent of the weight of an average calf in the group.
- If the type of the group is calf and group range of ages starts from 4 months or more, the total kilograms are the 2.5 percent of the weight of an average calf in the group.
- The total kilograms of the ration and the date for each cow in the group are registered.

CONCENTRATED FOOD/CONCENTRATED
Notion

- It is a mixture of grains (corn, barley, wheat) or balanced food given to cows as food.

Behavioural Response

- It is measured in kilograms of dried feedstuffs.
- The quantity given to each cow is registered.

CONTROLS WEIGHT
Notion

- A dairy farmer compares the current weight of a cow to the expected weight, to determine if it is according to standards.

Behavioural Response

- If the cow is a just calf, the weight should be nearly 40 kgs.
- If the cow is a 60 days old calf, the weight should be nearly 60 kgs.
- If the cow is a 15 months old heifer, the weight should be nearly 350 kgs.
- If the cow is a dry cow or a milking cow, the weight should be between 550 and 580 kgs.
- If the cow is a discard cow, the weight should be 580 kgs or more.

CORN SILAGE
Notion

- One of the foods given to cows, which is prepared using the whole corn plant.

Behavioural Response

- It is measured in kilograms of dried feedstuffs.
- The quantity given to each cow is registered.

COW
Notion

- It is a large animal kept in a farm to produce milk or meat.
- It has an identification number in its ear.
- It also has an earring with the identification number.
- It may have a brand.
- It has a date of birth.
- It may be male or female.
- It has a current weight.
- It may be a calf, a heifer, or a dairy cow.

Behavioural Response

- It is weighed every month or every three months.
- It receives vaccination against different diseases.
- It is assigned to a group.

- It is fed every day.
- It receives deparasitation.
- It is placed in a plot.

DAIRY COW
Notion

- It is a female cow which has had at least one calf.
- It is in a plot.
- It may be milking cow, dry cow, or discard cow.
- It weighs between 550 and 580 kilograms.
- Its useful life lasts more or less 4 years.
- It has an individual production.
- It belongs to a group of type 1, 2, pre-birth cow, dry cow or discard cow.
- It may be pregnant.
- It may be on heat every 21 days.

Behavioural Response

- When on heat, heat is registered.
- It is milked for approximately 10 months in each 12 months.
- It may receive insemination by artificial insemination or natural insemination.
- It generally gives birth to one calf per 12 months, and each birth is saved.
- When it is 4 years or more and approximately 580 kg weight, it may be dried

DAIRY FARM
Notion

- Farm where cows are bred with the goal of producing good quality milk and obtaining a good income.
- It has at least one dairy farmer.
- It has a set of cows.
- It may have a set of bulls.
- It is divided into a set of fields.

- It has a set of groups of cows.
- It has a calf rearing unit.
- It has an average individual production.
- It has a brand which identifies it.

Behavioural Response

- Arrivals and departures of cows are registered.
- Arrivals and departures from calf rearing unit are registered.
- The individual production is computed.
- Any change in the fields location or size is recorded.
- Any change in the set of groups is saved.

DAIRY FARMER
Notion

- Person in charge of all the activities in a dairy farm.
- He has a name.
- He has a salary.
- He may have one or more employees.

Behavioural Response

- He milks all the milking cow.
- He detects heat.
- He assigns to a group each cow of the dairy farm.
- He defines plot.
- He decides when to dry a cow for discard.
- He feeds groups of cows.
- He computes ration for each cow.
- He vaccinates each cow according to its needs.
- He weighs cow.
- He defines calf groups.
- He deparasites calves or heifers.

- He decides when to inseminate dairy cows or heifers.
- He saves birth.
- He registers heat.
- He sends calf to the calf rearing unit.
- He carries out calves artificial breeding.
- He takes calf out the calf rearing unit.
- He selects a calf group for each calf.
- He sells cow.
- He handles cow death.
- He computes individual production of a milking cow, a group or a dairy farm.
- He buys bull for the dairy farm.
- He discards bull.
- He computes birth date for each dairy cow or heifer.
- He inseminates artificially dairy cows or heifers.
- He sends to eat pasture each group in the dairy farm.
- He detects pregnant cow.
- He defines cow type.

DEFINES CALF GROUP
Notion

- A dairy farmer defines one group of type calf for a set of calves which have 60 or less days difference in their birth date.

Behavioural Response

- The group is assigned an identification.
- The group is set minimum and maximum ages.

DEFINES COW TYPE/DEFINE COW TYPE
Notion

- A dairy farmer sets the type of a cow according to its characteristics.

Behavioural Response

- If the cow is a 12 months female calf, its type is set to heifer.
- If the cow is a heifer which has a recent birth, its type is set to dairy cow.
- If the cow is a dairy cow which has a recent birth , its type is set to post-birth cow.
- If the cow is a pregnant post-birth cow whose last birth was 3 months ago, its type is set to early pregnant cow.
- If the cow is an early pregnant cow in its seventh month of pregnancy, its type is set to dry cow.
- If the cow is a dry cow whose next birth is within 15-20 days, its type is set to pre-birth cow.
- If the cow is a non pregnant dairy cow and it is a post-birth cow which could not become pregnant after 4 inseminations, its type is set to empty cow.
- If the cow is a dairy cow recently dried, its type is set to discard cow.

DEFINES PLOT

Notion

- It is the definition of a plot in a field to send a group to eat pasture.
- It is performed by a dairy farmer.

Behavioural Response

- Considering the pasture, the group type and the number of cows in the group, a dairy farmer determines a division of the field.
- The identification, size and location of the plot is registered in the corresponding field.

DEPARASITES/DEPARASITATION

Notion

- A dairy farmer gives a substance to a calf or a heifer to protect it against parasites.

Behavioural Response

- It is first applied to a calf when it leaves the calf rearing unit.
- It is applied to each calf every 2 or 3 months.
- It is applied to a heifer every 2 or 3 months until it is pregnant.
- The dose given, the date and the identification number of the calf or heifer are recorded.

DETECT PREGNANT COW

Notion

- A dairy farmer examines a dairy cow or heifer to detect if it is pregnant or not.

Behavioural Response

- The dairy cow or heifer was mated in its last on heat period.
- If it is pregnant, the date and the dairy cow or heifer identification number are saved in the Insemination form.
- If it is a pregnant dairy cow, a dairy farmer defines cow type as early pregnant cow and dairy cow is assigned to a group.
- If it is a non pregnant dairy cow and it is a post-birth cow which could not become pregnant after 4 inseminations, define cow type as empty cow.

DISCARD COW
Notion

- It is a dairy cow recently dried.
- It belongs to group of type discard cow.

Behavioural Response

- It is kept in group of discard cow for some months until its weight is 600 kg or more.
- When its weight is 600 kg or more, it is sold.

DISCARDS BULL
Notion

- The dairy farmer deletes a bull from the set of bulls because it is not in conditions to mate cows or because it died.

Behavioural Response

- The bull is deleted from the set of bulls.
- The causes, the bull name and the date are registered.

DRIED FEEDSTUFFS/DRY MATERIAL
Notion

- It is a unit of weight used to measure the quantity of bulky foods (pasture, hay or corn silage), concentrated food and balanced food that compose a ration.

Behavioural Response

- It is expressed in kilograms.

DRY A COW FOR DISCARD/DRIED

Notion

- The dairy farmer stops milking a milking cow.

Behavioural Response

- The milking cow is an empty cow at the end of its lactation period or it may be a dairy cow which has a disease or reproductive problems.
- The date, the identification number of the dairy cow and the drying causes are registered.
- Define cow type as discard cow.
- The dairy cow is assigned to a group of type discard cow.

DRY COW

Notion

- It is a pregnant dairy cow whose next birth date is within 2 months.
- It has a next birth date.
- It may be a pre-birth cow.
- It belongs to group of type dry cow.

Behavioural Response

- It is not milked.
- Between 15 and 20 days before the birth, it is assigned to a group of type pre-birth cow.

EARLY PREGNANT COW

Notion

- It is a pregnant milking cow whose last birth was at least 3 months ago.
- It may belong to a group of type 1 or type 2.
- It has an approximated next birth date.

Behavioural Response

- After the seventh month of pregnancy, it is assigned to group of type dry cow.

EMPTY COW

Notion

- It is a milking cow which could not become pregnant after 4 inseminations.
- It belongs to a group of type 1 or 2.

Behavioural Response

- It is in lactation period.
- When the lactation period finishes, it is dried.

FEEDS GROUP/FEED A GROUP/FED/FEEDING
Notion

- A dairy farmer gives a group the corresponding ration.

Behavioural Response

- It is done once a day.
- A dairy farmer computes ration.
- The ration is distributed as follows:

  if the type of the group is 1 or heifer, 30-35 per cent is concentrated food, 25-30 per cent is corn silage, 10 per cent is hay;

  if the type of the group is 2 or dry cow or pre-birth cow , 25-30 per cent is corn silage, 15-20 per cent is concentrated food, 10 per cent is hay;

  if the type of the group is discard cow, only pasture are given;

  if the type of the group is calf and calves age is less than 6 months or calves are female of more than 6 months old, 40 per cent of balanced food is given.

  if the type of the group is calf and calves age is greater than 6 months and calves are male, only pasture are given.
- The quantities of concentrated food, corn silage and hay given to each cow of the group and the date are registered in the Feeding form.
- A dairy farmer computes pasture eaten.

FIELD
Notion

- Land where cows eat pasture.
- It has an identification.
- It has a precise location in the dairy farm.
- It has a size.
- It has a pasture.

- It has an hectare loading.
- It is divided into a set of plots.
- It has a list of previous plots

Behavioural Response

- A dairy farmer divides it into a set of plots, separated by electric wires.
- Many different groups can be eating in it simultaneously.

GROUP/COW GROUP
Notion

- It is a set of only calves, only heifers or only dairy cows.
- It has an identification.
- It may be of one of the following types: 1, 2, pre-birth cow, discard cow, dry cow, heifer or calf.
- If it is of type calf, it has a range of ages of its members.
- Except for the ones of type calf, all the others are unique.
- If the type is 1 or 2 it has an individual production.

Behavioural Response

- It is sent out to pasture in a plot.
- It is daily fed.
- Groups of type 1, 2 and heifer are examined to detect cows on heat.

HANDLES COWS DEATH/HANDLE COW DEATH
Notion

- The dairy farmer records the death of a cow and its body is taken away.

Behavioural Response

- The date, the causes of the death and the history of the cow are saved.
- The cow is deleted from the dairy farm set of cows.
- If the cow is a calf in the calf rearing unit, it is deleted from the calf rearing unit set of calves.
- If not, it is deleted from the group to which it belongs.
- The cow's body is taken away from the dairy farm.

HAY
Notion

- Grass cut and dried to be used as cows food.

Behavioural Response

- It is measured in kilograms of dried feedstuffs.
- The quantity given to each cow is registered.

HEAT DETECTION
Notion

- To observe a group of milking cow or heifer to detect which of them are on heat.
- It is done by a dairy farmer while the cows are in a plot.

Behavioural Response

- It is done twice a day.
- It is applied only to a group of type 1, 2 and heifer.
- A dairy farmer observes carefully each cow.
- For each milking cow or heifer on heat, heat is registered.
- Milking cow and heifer detected on heat can be inseminated.

HEAT IS REGISTERED/REGISTERS HEAT
Notion

- The dairy farm records that a heifer or a dairy cow has been detected on heat.

Behavioural Response

- A dairy farmer has done heat detection.
- The date, time and heifer or dairy cow identification number are saved in the Insemination form.

HECTARE LOADING
Notion

- It is the number of cows per hectare.

Behavioural Response

- It should be maintained near 1,4.

HEIFER
Notion

- It is a female cow of 12 months age or more which has not yet had a calf.
- It may be pregnant.
- It may be on heat every 21 days.
- It belongs to group of type heifer.
- It is in a plot.
- It weighs approximately 350 kilograms when it is 15 months age.

Behavioural Response

- It may receive the first insemination by artificial insemination or natural insemination when it reaches 64 per cent of the weight of an adult dairy cow.
- When on heat, heat is registered.
- After the first birth, it is considered a dairy cow.
- It receives deparasitation every 2 or 3 months until it becomes pregnant.

IDENTIFICATION NUMBER
Notion

- It is a number that uniquely identifies a cow.

Behavioural Response

- It is assigned upon birth.
- It is tattooed in the ear when the calves are in the calf rearing unit.
- It is the number that appears in the earring.
- It is required to make any reference to a cow.

INDIVIDUAL PRODUCTION/MILK INDIVIDUAL PRODUCTION
Notion

- It is the average of the litres of milk produced in a period of time by a milking cow, a group or a dairy farm.

Behavioural Response

- It is measured in litres.
- The dairy farmer computes individual production.

INSEMINATION/INSEMINATE/INSEMINATES

Notion

- To put sperm into a dairy cow's or heifer's body to make it pregnant.
- It may be artificial insemination or natural insemination.

Behavioural Response

- Each dairy cow or heifer is given at most 4 possibilities.
- A dairy farmer decides if it will be artificial insemination or natural insemination.
- It can be performed only on an on heat heifer or on heat post-birth cow or empty cow which has been detected on heat in the last 12 hours.
- Date, type and identification number of the dairy cow or heifer are registered in the Insemination form.

LACTATION/LACTATION PERIOD

Notion

- Period after the birth of a calf in which a dairy cow produces milk.
- Dairy cow should be a milking cow.

Behavioural Response

- It lasts approximately seven months.
- Dairy cows can be milked.

MAXIMUM LACTATION/PEAK LACTATION

Notion

- It is the maximum value of a milking cow's individual production.

Behavioural Response

- It is reached between 60 and 70 days after the birth.
- It coincides with the milking cow's maximum ration.

MILK

Notion

- White liquid produced by dairy cows as food for their calves or to be drunk by humans.

Behavioural Response

- It is measured in litres.
- It can only be taken from a milking cow.
- Each calf in the calf rearing unit may receive 4-5 litres per day.

MILKING/TO MILK/MILKED/MILKS/MILKING
Notion

- Take the milk from a milking cow.
- It is done by a dairy farmer twice a day, in the morning and in the evening.

Behavioural Response

- It is applied only to a milking cow.
- A dairy farmer extracts the milk and puts it in a bucket to be measured.
- The litres of milk produced by the milking cow, the date and the time are registered in the Milking form.

MILKING COW
Notion

- It is a dairy cow currently producing milk.
- It may be an early pregnant cow, a post-birth cow or an empty cow.
- It may belong to group of type 1 or type 2.
- It may be pregnant.
- It may be on heat every 21 days.

Behavioural Response

- It is in lactation period.
- It is milked twice a day, in the morning and in the evening.
- It may receive insemination by artificial insemination or natural insemination.
- When on heat, heat is registered.

MILK REPLACEMENT/MILK SUBSTITUTE
Notion

- It is a kind of liquid food given to calves when they are in the calf rearing unit.
- It has a trade mark.

Behavioural Response

- It is measured in litres.
- 4-5 litres are given to each calf in the calf rearing unit per day.

NATURAL INSEMINATION/INSEMINATE NATURALLY
Notion

- Insemination done by a bull.

Behavioural Response

- The bull and the dairy cow or heifer are brought together in a field by the dairy farmer 12 hours after the last one is detected to be on heat.
- The bull name is registered in the Insemination form.

PASTURE
Notion

- Growing grass.
- It may be of different species.
- It has a level of quality.

Behavioural Response

- It is directly harvested by cows.
- It is measured in kilograms of dried feedstuffs.
- For each cow in a group, a dairy farmer computes pasture eaten.

PLOT/PLOT AREA
Notion

- Each one of the parts in which a field is divided into.
- It has an identification.
- It has a location inside the field.
- It has a size.
- It has a starting date.
- It has an approximated period of duration in days.
- In any time it is occupied by one group.

Behavioural Response

- Its size is defined by a dairy farmer.
- A group is sent out to pasture in it.

POST-BIRTH COW
Notion

- It is a milking cow whose last birth was in the last 3 months.
- It belongs to group of type 1.
- It has a last birth date.

Behavioural Response

- It can be inseminated up to 4 times to become pregnant.
- The first on heat period after birth is not considered for a new insemination.
- It receives insemination between 45 and 60 days after the birth.
- Three months after the birth, it is assigned to a group of type 2.

PRE-BIRTH COW
Notion

- It is a dry cow whose next birth is within 15 to 20 days.
- It belongs to group of type pre-birth cow.

Behavioural Response

- In the ninth month of pregnancy, it gives birth to a calf and the birth is saved.
- After birth, it is assigned to a group of type 1.

PREGNANT/PREGNANCY
Notion

- A heifer or a dairy cow has a calf developing in the uterus.

Behavioural Response

- The heifer or dairy cow has been inseminated.
- It lasts 9 months.
- The heifer or dairy cow cannot be on heat.

RATION
Notion

- It is the quantity of kilograms of dried feedstuffs to satisfy the daily requirements of a cow.
- It is composed by kilograms of corn silage, hay, concentrated food, balanced food and pasture.

Behavioural Response

- The dairy farmer computes ration for each cow in each group.

SAVE BIRTH/SAVES BIRTH/BIRTH IS SAVED
Notion

- A dairy farmer manages all the things related to a recent birth of a dairy cow or a heifer.

Behavioural Response

- The calf is assigned an identification number and it is added to the dairy farm set of cows.
- If a heifer is involved, define cow type as dairy cow.
- The new calf is added to the dairy cow's list of birth.
- The date and the identification number of the calf and the dairy cow are registered in the Birth form.
- Define cow type as post-birth cow
- The post-birth cow is assigned to a group of type 1.

SELECTS A CALF GROUP
Notion

- A dairy farmer chooses a group of type calf according to calf's age.

Behavioural Response

- A dairy farmer analyses current groups looking for a group of type calf which fits the calf's age.
- If it does not exist, dairy farmer defines calf group.

SELLS COW/SOLD
Notion

- A dairy farmer sends a cow to the market to be sold.

Behavioural Response

- The cow should be a discard cow or a male calf.
- The cow was recently weighed.
- The date, weight and the history of the cow are saved.
- The cow is deleted from the dairy farm set of cows.
- The cow is taken to the market.

SENDS CALF TO THE CALF REARING UNIT
Notion

- A calf is sent to the calf rearing unit for artificial breeding.
- It is carried out by a dairy farmer.

Behavioural Response

- It happens at most 5 days after calf's birth.
- The calf is added to the set of calves of the calf rearing unit.
- The entry date and the calf identification number are registered.
- Calf is tattooed the identification number.
- Calf may be stamped the brand.
- Calf is put on the earring.
- If the calf is a female calf, it is taken a photograph.

SENDS TO EAT PASTURE/SENT TO EAT PASTURE
Notion

- A dairy farmer sends a group to eat pasture in a plot of a field.

Behavioural Response

- The leaving date for the previous plot is saved.
- The entry date and the period the group is expected to be in the plot are recorded.
- The plot identification is registered.
- The group identification is recorded.

TAKES CALF OUT THE CALF REARING UNIT
Notion

- A dairy farmer decides a calf should finish artificial breeding.

Behavioural Response

- It happens between 45-60 days after birth when the calf is able to eat at least 1 kilogram of balanced food.
- Leaving date and identification number of the calf are registered.
- The calf is removed from the set of calves of the calf rearing unit.
- Calf is assigned to a group of type calf.

VACCINATES COW/VACCINATES/VACCINATION
Notion

- To inject a vaccine to a cow to protect it against a disease.
- It may be against brucellosis, diarrhoea or triple disease.
- It is performed by a dairy farmer.

Behavioural Response

- The vaccine has not expired.
- If the cow is a female calf of 3-10 months age and it has not received it yet, brucellosis vaccine is given.
- If the cow is a pregnant heifer or a dry cow in its seventh or ninth month of pregnancy, diarrhoea vaccine is given.
- If the cow is a calf of 3, 6, 9 or 12 months age, triple vaccine is given.
- The identification number of the cow, the date and the vaccine serial number given are registered.

VACCINE
Notion

- It is a substance to protect against a disease.
- It has a serial number.
- It may be against brucellosis, diarrhoea or triple disease.
- It has an expiration date.

Behavioural Response

- It is given to cows by a dairy farmer.

WEIGHS COW/WEIGH COW/WEIGHED
Notion

- A dairy farmer takes a cow to the scale to determine its weight.

Behavioural Response

- It can be done monthly or every 3 months.
- A dairy farmer takes the cow to the place where the scale is located.
- The cow comes up the scale.
- The weight showed by the scale, the date and the identification number of the cow are recorded.

Appendix B

The Scenario View

TITLE: Assign a group to a cow
GOAL: Add a cow to a group.
CONTEXT: Pre: Cow is in the calf rearing unit or cow is a member of a group.
RESOURCES: Cow Date List of current groups Group form
ACTORS: Dairy farmer
EPISODES:

- **If** the cow is a post-birth cow which has just given birth to a calf, or a milking cow whose individual production is at least 10 per cent greater than the dairy farm individual production **then** the dairy farmer selects group of type 1.
- **If** the cow is a dairy cow in group 1 and its last birth date is greater than 3 months or the cow is a pregnant dairy cow in less than seventh month of pregnancy **then** the dairy farmer selects group of type 2.
- **If** the cow is a dry cow or a pregnant heifer whose next birth date is within 15-20 days **then** the dairy farmer selects pre-birth cow type group.
- **If** the cow is an early pregnant cow in its seventh month of pregnancy **then** the dairy farmer selects dry cow type group.
- **If** the cow is a discard cow **then** the dairy farmer selects discard cow type group.
- **If** the cow is a calf **then** SELECT A CALF GROUP.
- **If** the cow is a heifer **then** the dairy farmer selects heifer type group.
- The dairy farmer registers in the Group form the arrival date for the new group and the identification number of the cow.

TITLE: Breed artificially
GOAL: Register the artificial breeding of a calf.
CONTEXT: It is done daily while the calf is in the calf rearing unit. Pre: The quantity of milk or milk replacement is approximately 4-5 litres and the quantity of balanced food is at most 1 kilogram.
RESOURCES: Calf Date Quantity of milk replacement Quantity of balanced food

Artificial breeding form
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer records in the Artificial breeding form the quantity of balanced food and milk replacement given to each calf per day.

TITLE: Buy a bull
GOAL: Add a new bull to the dairy farm.
CONTEXT: Pre: Bull is in conditions to inseminate cows
RESOURCES: Bull Date of purchase Bull date of birth Field Bull features List of bulls
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer adds the bull to the set of bulls of the dairy farm.
- The dairy farmer sends the bull to a field.
- The dairy farmer registers the date of the transaction, the bull features and date of birth.

TITLE: Check ration distribution
GOAL: Check if ration distribution is according to group type.
CONTEXT: It may be done once a day, before feeding a group. Pre: Group is not empty.
RESOURCES: Group Total ration Quantity of concentrated food Quantity of hay Quantity of corn silage
ACTORS: Dairy farmer
EPISODES:

- **If** the type of the group is 1 or heifer **then** the dairy farmer defines the ration as 30-35 per cent of concentrated food, 25-30 per cent of corn silage, 10-15 per cent of hay.
- **If** the type of the group is 2 or dry cow or pre-birth cow **then** the dairy farmer defines the ration as 15-20 per cent of concentrated food, 25-30 per cent of corn silage, 10-15 per cent of hay.
- **If** the type of the group is discard cow **then** the dairy farmer sets all the quantities to zero.
- **If** the type of the group is calf and calves age is less than 6 months or calves are female of more than 6 months age **then** the dairy farmer defines the ration as 40 per cent of balanced food and the remaining quantities are zero.
- **If** the type of the group is calf and calves age is greater than 6 months and calves are male **then** the dairy farmer sets all the quantities to zero.

TITLE: Compute next birth date
GOAL: Determine the approximate next birth date for a dairy cow or heifer.
CONTEXT: Pre: The cow is a dairy cow or a heifer and it is pregnant.
RESOURCES: Cow Insemination form
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer adds 9 months to the last insemination date.
- The dairy farmer saves the computed date.

TITLE: Compute dairy farm individual production
GOAL: Determine the individual production of a dairy farm in a period
CONTEXT: Pre: The dairy farm has at least one milking cow with at least one milking in the period.
RESOURCES: Period Milking form
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer divides the addition of the litres produced by all the milking cow in the period into the number of milking corresponding to the milking cows in that period.
- The dairy farmer registers the individual production calculated.

TITLE: Compute milking cow individual production
GOAL: Determine the individual production of a dairy cow in a period
CONTEXT: Pre: The dairy cow has at least one milking in the period
RESOURCES: Dairy cow Period Milking form
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer divides the addition of the litres produced by the dairy cow in the period into the number of milking in the period.
- The dairy farmer registers the individual production calculated.

TITLE: Compute group individual production
GOAL: Determine the individual production of a group in a period
CONTEXT: Pre: The group is type 1 or 2 and it has at least one milking cow which has at least one milking.
RESOURCES: Group Period Milking form Group form
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer divides the addition of the litres produced by the whole group in the period into the total number of milking corresponding to all the milking cow in the group in that period.

- The dairy farmer registers the individual production calculated.

TITLE: Compute pasture eaten
GOAL: Determine the quantity of pasture eaten by each cow in a group.
CONTEXT: Pre: The total kilograms of the ration have been calculated and the quantity of corn silage, hay and concentrated has been decided
RESOURCES: Group Feeding form
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer computes the difference between the ration a cow should eat and the addition of corn silage, hay, concentrated food the cow was given.
- The dairy farmer registers the quantity of pasture computed, the date and the identification number of each cow in the group.

TITLE: Compute ration
GOAL: Determine the total kilograms of the ration each cow in a group should be given in a day.
CONTEXT: Pre: Group is not empty
RESOURCES: Group Weight of an average cow in the group
ACTORS: Dairy farmer
EPISODES:

- **If** group type is 1 or pre-birth cow **then** the dairy farmer sets the total kilograms to 3,5 percent of the weight of an average dairy cow in the group.
- **If** group type is 2 **then** the dairy farmer sets the total kilograms to 3 percent of the weight of an average dairy cow in the group.
- **If** group type is dry cow or discard cow **then** the dairy farmer sets the total kilograms to 2 percent of the weight of an average dairy cow in the group.
- **If** group type is heifer **then** the dairy farmer sets the total kilograms to 3 percent of the weight of an average heifer in the group.
- **If** group type is calf and range of ages are between 2 and 4 months **then** the dairy farmer sets the total kilograms to the 2.2 percent of the weight of an average calf in the group.
- **If** group type is calf and range of ages starts from 4 months or more **then** the dairy farmer sets the total kilograms to 2.5 percent of the weight of an average calf in the group.

TITLE: Define calf group
GOAL: Define a new group of type calf.
CONTEXT: Pre: Range of ages has at most 60 days difference.
RESOURCES: Calves minimum age Calves maximum age List of current groups
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer assigns the group an identification.
- The dairy farmer sets the minimum and maximum ages for the new group.
- The dairy farmer adds the new group to the list of groups.

TITLE: Define cow type
GOAL: Set the type of a cow according to its characteristics
CONTEXT: Pre:
RESOURCES: Cow
ACTORS: Dairy farmer
EPISODES:

- **If** the cow is a 12 months female calf **then** the dairy farmer sets the type to heifer.
- **If** the cow is a heifer which has just given birth to a calf **then** the dairy farmer sets the type to dairy cow.
- **If** the cow is a dairy cow which has just given birth to a calf **then** the dairy farmer sets the type to post-birth cow.
- **If** the cow is a pregnant post-birth cow whose last birth was 3 months ago **then** the dairy farmer sets the type to pregnant cow.
- **If** the cow is pregnant cow in its seventh month of pregnancy **then** the dairy farmer sets the type to dry cow.
- **If** the cow is a dry cow whose next birth is within 15-20 days **then** the dairy farmer sets the type to pre-birth cow.
- **If** the cow is a non pregnant dairy cow and it is a post-birth cow which could not become pregnant after 4 inseminations **then** the dairy farmer sets the type to empty cow.
- **If** the cow is a dairy cow recently dried **then** the dairy farmer sets the type to discard cow.

TITLE: Define plot
GOAL: Delimit a plot in a field to send out a group to pasture.
CONTEXT: Pre: The group is not empty
RESOURCES: Group Field
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer determines a division of the field considering the pasture, the group type and the number of cows in the group.
- The dairy farmer registers the identification, size and location of the plot, and duration period in the corresponding field.

TITLE: Discard a bull
GOAL: Delete a bull from the dairy farm set of bulls
CONTEXT: Pre: Bull has just died or is not in conditions to inseminate cows.
RESOURCES: Bull Date Discard causes List of bulls
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer deletes the bull from the dairy farm set of bulls.
- The dairy farmer register the causes, the bull name and the date.

TITLE: Dry dairy cow
GOAL: Stop milking a milking cow
CONTEXT: Pre: Milking cow may be at the end of its lactation period or it may have a disease or it may have reproductive problems
RESOURCES: Milking cow Date Drying causes Discard form
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer registers the date, drying causes and the identification number of the milking cow in the Discard form.
- DEFINE COW TYPE as discard cow.
- ASSIGN A GROUP TO A COW.

TITLE: Feed a group
GOAL: Register the corresponding daily ration given to a group.
CONTEXT: It is done once a day. Pre: Group is not empty.
RESOURCES: Group Date Quantity of corn silage Quantity of Hay Quantity of concentrated food Feeding form
ACTORS: Dairy farmer
EPISODES:

- COMPUTE RATION.
- The dairy farmer records, in the Feeding form, the date and the quantities of corn silage, hay and concentrated food given to each cow in the group.
- COMPUTE PASTURE EATEN.

TITLE: Handle cow death
GOAL: Register the death of a cow
CONTEXT: Pre: The cow is in a group or if it is a calf of at most 60 days age, it may be in the calf rearing unit.
RESOURCES: Cow Date of death Causes of death Dairy farm set of cows

Calf rearing unit set of calves   Dead cows form
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer saves the date, the causes of the death and the history of the cow in the Dead cows form.
- # The dairy farmer deletes the cow from the dairy farm set of cows.
- **If** the cow is a calf in the calf rearing unit **then** the dairy farmer deletes it from the calf rearing unit set of calves.
- **If** not **then** the dairy farmer deletes it from the group to which it belongs.#

TITLE: Inseminate artificially
GOAL: Register the artificial insemination of a dairy cow or heifer.
CONTEXT: Pre: The cow is a post-birth cow which has had more than one on heat period after the last birth, or is a dairy cow or a heifer, and has been detected on heat in the last 12 hours and it has been inseminated at most 3 times without becoming pregnant
RESOURCES: Cow   Date   Method   Insemination form
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer registers, in the Insemination form, date, identification number of the post-birth cow or heifer and information relative to the procedure followed.

TITLE: Inseminate naturally
GOAL: Register the natural insemination of a dairy cow or heifer.
CONTEXT: Pre: Cow is a post-birth cow which has had more than one on heat period after the last birth, or is a dairy cow or a heifer, and has been detected on heat in the last 12 hours and it has been inseminated at most 3 times without becoming pregnant.
RESOURCES: Cow   Date   Bull   Insemination form
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer saves, in the Insemination form, date, identification number of the post-birth cow or heifer and the name of the bull.

TITLE: Manage birth
GOAL: Manage all things related to a recent birth.
CONTEXT: Pre: The cow is a dairy cow or a heifer which has just given birth to a calf.
RESOURCES: Cow   Calf   Date of the birth Birth form Dairy farm set of cows
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer assigns an identification number to the calf.
- The dairy farmer adds the new calf to the dairy farm set of cows.

- # The dairy farmer adds the new calf to the dairy cow's or heifer's list of given birth to calves.
- The dairy farmer records in the Birth form the date and the identification number of the calf and the cow.#
- ASSIGN A GROUP TO A COW, a group of type 1.
- DEFINE COW TYPE as post-birth cow.

TITLE: Record cow deparasitation
GOAL: Record the deparasitation of a calf or a heifer.
CONTEXT: Pre: Cow is a calf which has just left the calf rearing unit or a calf that has not been deparasited in the last 2 or 3 months or a heifer that is not pregnant and that has not been deparasited in the last 2 or 3 months.
RESOURCES: Cow Date Substance Dose Deparasitation form
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer records the dose given, the date and the identification number of the cow.

TITLE: Record milking
GOAL: Record the milking of a milking cow.
CONTEXT: It is done in the morning or in the evening. Pre: Milking cow has not been milked yet.
RESOURCES: Milking cow Date Litres of milk Milking form
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer records, in the milking form, the litres of milk measured, the milking cow identification number, date and time of extraction.

TITLE: Register cow weight
GOAL: Register the weight of a cow.
CONTEXT: It occurs in the place where the scale is located. Pre: The cow has not been weighed in the last 3 months or in the last month or the cow is going to be sold.
RESOURCES: Cow Date Weight form Weight
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer saves, in the Weight form, the weight, the date and the identification number of the cow.

TITLE: Register cows on heat detection
GOAL: Register which milking cow or heifer in a group is on heat.

CONTEXT: It occurs in a plot.Pre: Group is of type 1, 2 or heifer and it has been examined at most once that day.
RESOURCES: Group Date Time List of cows on heat Group form
ACTORS: Dairy farmer
EPISODES:

- REGISTER HEAT for each cow detected on heat.
- The dairy farmer registers date, time and group examined in the Group form.

TITLE: Register heat
GOAL: Record that a dairy cow or heifer is on heat
CONTEXT: Pre: A dairy farmer has done heat detection and the cow is a dairy cow or heifer detected on heat
RESOURCES: Cow Date of detection Time of detection Insemination form
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer records in the Insemination form, the date, time and cow identification number.

TITLE: Register pregnancy test
GOAL: Register the result of the pregnancy test for a dairy cow or heifer.
CONTEXT: Pre: Cow is a dairy cow or heifer inseminated in its last on heat period.
RESOURCES: Cow Date Test Insemination form
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer saves the date, the dairy cow or heifer identification number and the test in the Insemination form.
- **If** it is a pregnant dairy cow **then** DEFINE COW TYPE

  ASSIGN A GROUP TO A COW.

TITLE: Select a calf group
GOAL: Choose a group of type calf for a calf according to its age.
CONTEXT: Pre: Calf is in the calf rearing unit or in a group of type calf.
RESOURCES: Calf List of current groups
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer looks for a group of type calf, which fits the calf's age.
- **If** the group does not exist **then** DEFINE CALF GROUP.

TITLE: Sell cow
GOAL: Record cow was sent to the market to be sold
CONTEXT: Pre: Cow is a discard cow or a male calf and cow has been weighed the day of the sale.
RESOURCES: Cow Date of sale Dairy farm set of cows Sale form
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer saves in the Sale form the date, weight and the complete history of the cow.
- The dairy farmer deletes the cow from the dairy farm set of cows.

TITLE: Send calf to the calf rearing unit
GOAL: Record a calf is sent to the calf rearing unit.
CONTEXT: Pre: The calf is approximately between 1 and 5 days age
RESOURCES: Calf Date Calf rearing unit set of calves
ACTORS: Dairy farmer
EPISODES:

- # The dairy farmer adds the calf to the calf rearing unit set of calves.
- The dairy farmer registers the entry date and the calf identification number.#

TITLE: Send out to pasture
GOAL: Send a group to eat pasture in a plot
CONTEXT: Pre: The group is not empty
RESOURCES: Group Date Period for the new plot New plot Group form
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer saves leaving date for the previous plot.
- The dairy farmer records, in the Group form, the entry date, the period the group is expected to be in the plot, the plot identification and the group identification.

TITLE: Take calf out the calf rearing unit
GOAL: Register the end of a calf's artificial breeding.
CONTEXT: Pre: Calf is between 45 and 60 days age and it is able to eat at least 1 kilogram of balanced food per day.
RESOURCES: Calf Date Calf rearing unit set of calves
ACTORS: Dairy farmer
EPISODES:

- # The dairy farmer records leaving date and identification number of the calf.
- The dairy farmer removes the calf from the set of calves of the calf rearing unit.#

- ASSIGN TO A GROUP of type calf.

TITLE: Vaccinate cow
GOAL: Register the vaccination of a cow.
CONTEXT: Pre: The vaccine has not expired and the cow is a pregnant heifer or a dry cow in its seventh or ninth month of pregnancy and the vaccine is against diarrhoea, or the cow is a female calf of 3-10 months and the vaccine is against brucellosis, or the cow is a calf of 3, 6, 9 or 12 months age and the vaccine is triple.
RESOURCES: Cow Date Vaccine Vaccination form
ACTORS: Dairy farmer
EPISODES:

- The dairy farmer registers, in the Vaccination form, the identification number of the cow, the date and the vaccine serial number and type.

Appendix C

The Specification

C.1 DAIRY_FARM Module

context: FIELDS, COW_GROUPS, BULLS, DAIRY_FARMERS

```
scheme DAIRY_FARM =
  class
    object
      CS : COWS,
      BS : BULLS,
      FS : FIELDS,
      CGS : COW_GROUPS(CS),
      DFS : DAIRY_FARMERS

    type
      Dairy_farm ::
        cows : CS.Cows ↔ chg_cows
        bulls : BS.Bulls ↔ chg_bulls
        fields : FS.Fields ↔ chg_fields
        groups : CGS.Cow_groups ↔ chg_groups
        dairy_farmers :
          DFS.Dairy_farmers ↔ chg_dairy_farmers
        past_cows : CS.Cows ↔ chg_past_cows

    value
      can_goto_group :
        GT.Cow_id × D.Date × Dairy_farm → Bool
      can_goto_group(ci, d, df) ≡
        CGS.can_goto_group(
            ci, d, CS.select_group_for_cow(ci, d, cows(df)),
            cows(df), groups(df)),

      assign_group_to_cow :
        GT.Cow_id × D.Date × Dairy_farm →̃ Dairy_farm
```

```
assign_group_to_cow(ci, d, df) ≡
   chg_cows(
       CS.assign_group_to_cow(
             ci, d,
             CGS.select_group_for_cow(
                 ci, d, cows(df), groups(df)), cows(df)), df)
pre can_goto_group(ci, d, df),

can_breed_artif :
   GT.Cow_id × D.Date × GT.Litres × GT.Quantity ×
   Dairy_farm →
      Bool
can_breed_artif(ci, d, mr, bal, df) ≡
   GT.calf_rearing_unit ∈ groups(df) ∧
   CS.can_breed_artif(ci, d, mr, bal, cows(df)),

breed_artif :
   GT.Cow_id × D.Date × GT.Litres × GT.Balanced ×
   Dairy_farm →̃
      Dairy_farm
breed_artif(ci, d, mr, bal, df) ≡
   chg_cows(
       CS.breed_artif(ci, d, mr, bal, cows(df)), df)
pre can_breed_artif(ci, d, mr, bal, df),

buy_bull :
   GT.Bull_id × D.Date × D.Date × GT.Field_id ×
   GT.Features × Dairy_farm →̃
      Dairy_farm
buy_bull(bi, bd, d, fi, f, df) ≡
   chg_bulls(
       BS.add_bull(bi, bd, d, fi, f, bulls(df)), df)
pre bi ∉ bulls(df) ∧ fi ∈ fields(df),

can_give_birth :
   GT.Cow_id × D.Date × Dairy_farm → Bool
can_give_birth(ci, d, df) ≡
   CS.can_give_birth(ci, d, cows(df)),

next_birth_date :
   GT.Cow_id × D.Date × Dairy_farm →̃ D.Date
next_birth_date(ci, d, df) ≡
   CS.next_birth_date(ci, d, cows(df))
pre can_give_birth(ci, d, df),

dfarm_has_prod_milk : D.Period × Dairy_farm → Bool
```

```
dfarm_has_prod_milk(p, df) ≡
   CGS.has_produced_milk(
       GT.one, p, groups(df), cows(df)) ∨
   CGS.has_produced_milk(
       GT.two, p, groups(df), cows(df)),

d_farm_indiv_prod :
   D.Period × Dairy_farm ~→ GT.Indiv_prod
d_farm_indiv_prod(p, df) ≡
   let
      cows_in_one =
         CGS.cows_in_group(GT.one, groups(df), cows(df)),
      cows_in_two =
         CGS.cows_in_group(GT.two, groups(df), cows(df))
   in
      CS.cows_milk_in_period(
          p, cows_in_one ∪ cows_in_two) /
      real (CS.number_milkings_in_period(p, cows_in_one) +
                CS.number_milkings_in_period(p, cows_in_two))
   end
pre dfarm_has_prod_milk(p, df),

cow_has_prod_milk :
   GT.Cow_id × D.Period × Dairy_farm → Bool
cow_has_prod_milk(ci, p, df) ≡
   CS.has_produced_milk(ci, p, cows(df)),

cow_indiv_prod :
   GT.Cow_id × D.Period × Dairy_farm ~→
      GT.Indiv_prod
cow_indiv_prod(ci, p, df) ≡
   CS.cow_indiv_prod(ci, p, cows(df))
pre cow_has_prod_milk(ci, p, df),

group_has_prod_milk :
   GT.Group_id × D.Period × Dairy_farm → Bool
group_has_prod_milk(gt, p, df) ≡
   CGS.has_produced_milk(gt, p, groups(df), cows(df)),

group_indiv_prod :
   GT.Group_id × D.Period × Dairy_farm ~→
      GT.Indiv_prod
group_indiv_prod(gt, p, df) ≡
   CGS.group_indiv_prod(gt, p, groups(df), cows(df))
pre group_has_prod_milk(gt, p, df),
can_compute_ration : GT.Group_id × Dairy_farm → Bool
can_compute_ration(gt, df) ≡
   CGS.can_compute_ration(gt, groups(df), cows(df)),

compute_ration :
   GT.Group_id × Dairy_farm →̃ GT.Quantity
compute_ration(gt, df) ≡
   CGS.compute_ration(gt, groups(df), cows(df))
pre can_compute_ration(gt, df),

define_calf_group :
   Nat × Nat × Dairy_farm →̃ Dairy_farm
define_calf_group(ds, de, df) ≡
   chg_groups(
       CGS.define_calf_group(ds, de, groups(df)), df)
pre
   GT.calf(ds, de) ∉ groups(df) ∧
   de − ds ≤ K.calves_age_dif,

define_cow_classif :
   GT.Cow_id × D.Date × Dairy_farm →̃ Dairy_farm
define_cow_classif(ci, d, df) ≡
   chg_cows(CS.define_cow_classif(ci, d, cows(df)), df)
pre ci ∈ cows(df),

set_plot :
   GT.Group_id × GT.Field_id × Dairy_farm →
      GT.Plot_id × GT.Size × GT.Location × D.Date ×
      Nat
/∗ dummy value for now ∗/
set_plot(gt, fi, df) ≡
   (1, 0.0, "", D.mk_Date(2003, 1, 1, 3), 1),

can_define_plot :
   GT.Group_id × GT.Field_id × Dairy_farm → Bool
can_define_plot(gt, fi, df) ≡
   fi ∈ fields(df) ∧ gt ∈ groups(df) ∧
   ∼CGS.empty(gt, groups(df), cows(df)),

define_plot :
   GT.Group_id × GT.Field_id × Dairy_farm →̃
      Dairy_farm
define_plot(gt, fi, df) ≡
   let (pi, si, lo, sd, dn) = set_plot(gt, fi, df) in
      chg_fields(
          FS.add_plot(pi, si, lo, sd, dn, fi, fields(df)),
            df)
    end
pre can_define_plot(gt, fi, df),

can_discard_bull : GT.Bull_id × Dairy_farm → Bool
can_discard_bull(bi, df) ≡
    BS.can_discard_bull(bi, bulls(df)),

discard_bull :
    GT.Bull_id × D.Date × GT.Discard_cause ×
    Dairy_farm →̃
        Dairy_farm
discard_bull(bi, d, dc, df) ≡
    chg_bulls(BS.discard_bull(bi, d, dc, bulls(df)), df)
pre can_discard_bull(bi, df),

can_dry_cow :
    GT.Cow_id × D.Date × Dairy_farm → Bool
can_dry_cow(ci, d, df) ≡
    CS.can_dry_cow(ci, d, cows(df)),

dry_cow :
    GT.Cow_id × D.Date × GT.Dried_cause × Dairy_farm →̃
        Dairy_farm
dry_cow(ci, d, dc, df) ≡
    chg_cows(CS.dry_cow(ci, d, dc, cows(df)), df)
pre can_dry_cow(ci, d, df),

can_feed_group :
    GT.Group_id × D.Date × Dairy_farm → Bool
can_feed_group(gt, d, df) ≡
    CGS.can_feed_group(gt, d, groups(df), cows(df)),

feed_group :
    GT.Group_id × D.Date × GT.Corn_sil × GT.Hay ×
    GT.Conc × Dairy_farm →̃
        Dairy_farm
feed_group(gt, d, corn, hay, conc, df) ≡
    chg_groups(
         CGS.feed_group(
                gt, d, corn, hay, conc, groups(df), cows(df)),
         df)
pre can_feed_group(gt, d, df),

can_save_cow_death : GT.Cow_id × Dairy_farm → Bool
can_save_cow_death(ci, df) ≡
```

```
    CS.can_save_cow_death(ci, cows(df), past_cows(df)),

save_cow_death :
    GT.Cow_id × D.Date × GT.Death_cause × Dairy_farm →̃
        Dairy_farm
save_cow_death(ci, d, dc, df) ≡
    let
        (cs, pcs) =
            CS.save_cow_death(
                ci, d, dc, cows(df), past_cows(df))
    in
        chg_past_cows(pcs, chg_cows(cs, df))
    end
pre can_save_cow_death(ci, df),

can_insem_cow :
    GT.Cow_id × D.Date × Dairy_farm → Bool
can_insem_cow(ci, d, df) ≡
    CS.can_insem_cow(ci, d, cows(df)),

insem_cow_artif :
    GT.Cow_id × D.Date × GT.Artif_info × Dairy_farm →̃
        Dairy_farm
insem_cow_artif(ci, d, ai, df) ≡
    chg_cows(CS.insem_cow_artif(ci, d, ai, cows(df)), df)
pre can_insem_cow(ci, d, df),

insem_cow_natural :
    GT.Cow_id × D.Date × GT.Bull_id × Dairy_farm →̃
        Dairy_farm
insem_cow_natural(ci, d, bi, df) ≡
    chg_cows(
         CS.insem_cow_natural(ci, d, bi, cows(df)), df)
pre bi ∈ bulls(df) ∧ can_insem_cow(ci, d, df),

give_birth :
    GT.Cow_id × GT.Calf_sex × D.Date × Dairy_farm →̃
        Dairy_farm
give_birth(ci, csex, d, df) ≡
    chg_cows(CS.give_birth(ci, csex, d, cows(df)), df)
pre can_give_birth(ci, d, df),

can_deparasite_cow :
    GT.Cow_id × D.Date × Dairy_farm → Bool
can_deparasite_cow(ci, d, df) ≡
    CS.can_deparasite_cow(ci, d, cows(df)),
deparasite_cow :
   GT.Cow_id × D.Date × GT.Dep_inf × Dairy_farm ~→
      Dairy_farm
deparasite_cow(ci, d, d_inf, df) ≡
   chg_cows(
       CS.deparasite_cow(ci, d, d_inf, cows(df)), df)
pre can_deparasite_cow(ci, d, df),

can_milk_cow :
   GT.Cow_id × D.Date × Dairy_farm → Bool
can_milk_cow(ci, d, df) ≡
   CS.can_milk_cow(ci, d, cows(df)),

milk_cow :
   GT.Cow_id × D.Date × GT.Litres × Dairy_farm ~→
      Dairy_farm
milk_cow(ci, d, lts, df) ≡
   chg_cows(CS.milk_cow(ci, d, lts, cows(df)), df)
pre can_milk_cow(ci, d, df),

can_weigh_cow :
   GT.Cow_id × D.Date × Dairy_farm → Bool
can_weigh_cow(ci, d, df) ≡
   CS.can_weigh_cow(ci, d, cows(df)),

weigh_cow :
   GT.Cow_id × D.Date × GT.Weight × Dairy_farm ~→
      Dairy_farm
weigh_cow(ci, d, w, df) ≡
   chg_cows(CS.weigh_cow(ci, d, w, cows(df)), df)
pre can_weigh_cow(ci, d, df),

can_detect_heat :
   GT.Group_id × D.Date × (GT.Cow_id × Bool)* ×
   Dairy_farm →
      Bool
can_detect_heat(gt, d, csl, df) ≡
   CGS.can_detect_heat(gt, d, csl, groups(df), cows(df)),

detect_heat :
   GT.Group_id × D.Date × (GT.Cow_id × Bool)* ×
   Dairy_farm ~→
      Dairy_farm
detect_heat(gt, d, csl, df) ≡
   let
```

```
          (cgs, cs) =
              CGS.detect_heat(gt, d, csl, groups(df), cows(df))
      in
          chg_groups(cgs, chg_cows(cs, df))
      end
  pre can_detect_heat(gt, d, csl, df),

  can_detect_pregnancy :
      GT.Cow_id × D.Date × Dairy_farm → Bool
  can_detect_pregnancy(ci, d, df) ≡
      CS.can_detect_pregnancy(ci, d, cows(df)),

  detect_pregnant_cow :
      GT.Cow_id × D.Date × Bool × Dairy_farm ~→
          Dairy_farm
  detect_pregnant_cow(ci, d, preg, df) ≡
      chg_cows(
           CS.detect_pregnant_cow(ci, d, preg, cows(df)), df)
  pre can_detect_pregnancy(ci, d, df),

  can_select_calf_group :
      GT.Cow_id × Dairy_farm → Bool
  can_select_calf_group(ci, df) ≡
      CGS.can_select_calf_group(ci, cows(df)),

  select_calf_group :
      GT.Cow_id × D.Date × Dairy_farm ~→
          GT.Group_id × Dairy_farm
  select_calf_group(ci, d, df) ≡
      let
          (gt, gs) =
              CGS.select_calf_group(
                   ci, d, cows(df), groups(df))
      in
          (gt, chg_groups(gs, df))
      end
  pre can_select_calf_group(ci, df),

  can_sell_cow :
      GT.Cow_id × D.Date × Dairy_farm → Bool
  can_sell_cow(ci, d, df) ≡
      CS.can_sell_cow(ci, d, cows(df), past_cows(df)),

  sell_cow :
      GT.Cow_id × D.Date × Dairy_farm ~→ Dairy_farm
  sell_cow(ci, d, df) ≡
```

```
    let
        (cs, pcs) =
            CS.sell_cow(ci, d, cows(df), past_cows(df))
    in
        chg_past_cows(pcs, chg_cows(cs, df))
    end
pre can_sell_cow(ci, d, df),

can_goto_cru :
    GT.Cow_id × D.Date × Dairy_farm → Bool
can_goto_cru(ci, d, df) ≡
    GT.calf_rearing_unit ∈ groups(df) ∧
    CS.can_goto_cru(ci, d, cows(df)),

send_calf_to_cru :
    GT.Cow_id × D.Date × Dairy_farm ~→ Dairy_farm
send_calf_to_cru(ci, d, df) ≡
    chg_cows(CS.send_calf_to_cru(ci, d, cows(df)), df)
pre can_goto_cru(ci, d, df),

can_send_to_pasture :
    GT.Group_id × D.Date × Nat × GT.Plot_id ×
    GT.Field_id × Dairy_farm →
        Bool
can_send_to_pasture(gt, d, dn, pi, fi, df) ≡
    CGS.can_send_to_pasture(
         gt, d, pi, fi, groups(df), cows(df)) ∧
    FS.is_defined(pi, fi, d, dn, fields(df)),

send_to_pasture :
    GT.Group_id × D.Date × Nat × GT.Plot_id ×
    GT.Field_id × Dairy_farm ~→
        Dairy_farm
send_to_pasture(gt, d, dn, pi, fi, df) ≡
    chg_groups(
         CGS.send_to_pasture(
              gt, d, dn, pi, fi, groups(df), cows(df)), df)
pre can_send_to_pasture(gt, d, dn, pi, fi, df),

can_take_out_cru :
    GT.Cow_id × D.Date × Dairy_farm → Bool
can_take_out_cru(ci, d, df) ≡
    ci ∈ cows(df) ∧
    GT.calf_rearing_unit ∈ groups(df) ∧
    CS.can_take_out_cru(ci, d, cows(df)) ∧
    CS.can_eat_bal(ci, d, cows(df)) ∧
    CGS.can_goto_group(
        ci, d, CS.select_group_for_cow(ci, d, cows(df)),
        cows(df), groups(df)),

take_calf_out_cru :
    GT.Cow_id × D.Date × Dairy_farm →̃ Dairy_farm
take_calf_out_cru(ci, d, df) ≡
    assign_group_to_cow(ci, d, df)
pre can_take_out_cru(ci, d, df),

can_receive_vacc :
    GT.Cow_id × D.Date × GT.Vaccine × Dairy_farm →
        Bool
can_receive_vacc(ci, d, vacc, df) ≡
    CS.can_receive_vacc(ci, d, vacc, cows(df)),

vaccinate_cow :
    GT.Cow_id × D.Date × GT.Vaccine × Dairy_farm →̃
        Dairy_farm
vaccinate_cow(ci, d, vacc, df) ≡
    chg_cows(CS.vaccinate_cow(ci, d, vacc, cows(df)), df)
pre can_receive_vacc(ci, d, vacc, df),

field_hectare_loading :
    GT.Field_id × D.Date × Dairy_farm →̃
        GT.Hect_loading
field_hectare_loading(fi, d, df) ≡
    real (CGS.number_cows_in_field(
                fi, d, groups(df), cows(df))) /
    FS.field_size(fi, fields(df))
pre fi ∈ fields(df),

group_current_plot :
    GT.Group_id × D.Date × Dairy_farm →̃ GT.In_plot
group_current_plot(gt, d, df) ≡
    CGS.group_current_plot(gt, d, groups(df))
pre gt ∈ groups(df),

cow_current_plot :
    GT.Cow_id × D.Date × Dairy_farm →̃ GT.In_plot
cow_current_plot(ci, d, df) ≡
    CGS.group_current_plot(
        cow_group(ci, df), d, groups(df))
pre
    ci ∈ cows_in_group(cow_group(ci, df), df) ∧
    cow_group(ci, df) ∈ groups(df),
can_be_in_group : GT.Cow_id × Dairy_farm → Bool
can_be_in_group(ci, df) ≡
   CS.can_be_in_group(ci, cows(df)),

cow_group : GT.Cow_id × Dairy_farm →̃ GT.Group_id
cow_group(ci, df) ≡ CS.cow_group(ci, cows(df))
pre can_be_in_group(ci, df),

cow_events :
   GT.Cow_id × D.Period × CE.Cow_event_kind ×
   Dairy_farm →̃
      CH.History
cow_events(ci, p, ev_kind, df) ≡
   CS.cow_events(ci, p, ev_kind, cows(df))
pre ci ∈ cows(df),

group_events :
   GT.Group_id × D.Period × GE.Group_event_kind ×
   Dairy_farm →̃
      GH.History
group_events(gt, p, ev_kind, df) ≡
   CGS.group_events(gt, p, ev_kind, groups(df))
pre gt ∈ groups(df),

cows_in_group : GT.Group_id × Dairy_farm →̃ CS.Cows
cows_in_group(gt, df) ≡
   CGS.cows_in_group(gt, groups(df), cows(df))
pre gt ∈ groups(df),

cow_mother : GT.Cow_id × Dairy_farm →̃ GT.Cow_id
cow_mother(ci, df) ≡ CS.cow_mother(ci, cows(df))
pre ci ∈ cows(df),

cow_classif :
   GT.Cow_id × Dairy_farm →̃ GT.Cow_classif
cow_classif(ci, df) ≡ CS.cow_classif(ci, cows(df))
pre ci ∈ cows(df),

cow_history_p :
   GT.Cow_id × D.Period × Dairy_farm →̃ CH.History
cow_history_p(ci, p, df) ≡
   CS.cow_history_p(ci, p, cows(df))
pre ci ∈ cows(df),

is_wf_group_plot : D.Date × Dairy_farm → Bool
```

```
is_wf_group_plot(d, df) ≡
   (∀ gt : GT.Group_id •
      gt ∈ groups(df) ⇒
         let
            plot = group_current_plot(gt, d, df),
            fi = GT.field_id(plot),
            pi = GT.plot_id(plot)
         in
            fi ∈ fields(df) ∧
            (FS.exists_plot(pi, fi, fields(df)) ⇒
                  CGS.is_only_group(
                     gt, d, pi, fi, groups(df)))
         end),

is_wf_cow_mother : D.Date × Dairy_farm → Bool
is_wf_cow_mother(d, df) ≡
   (∀ ci : GT.Cow_id •
      (ci ∈ cows(df) ∨ ci ∈ past_cows(df)) ⇒
         (∀ ev : CH.Event •
               ev ∈
                  cow_events(ci, D.upto(d), CE.birth, df) ⇒
                  let cf = CE.calf_id(CH.event_inf(ev)) in
                     cf ∈ cows(df) ∨
                     cf ∈ past_cows(df)
                  end)),

is_wf_cow_group : D.Date × Dairy_farm → Bool
is_wf_cow_group(d, df) ≡
   (∀ ci : GT.Cow_id •
      (ci ∈ cows(df) ∨ ci ∈ past_cows(df)) ⇒
         (∀ ev : CH.Event •
               ev ∈
                  cow_events(
                     ci, D.upto(d), CE.cow_to_group, df) ⇒
                  let gt = CE.group(CH.event_inf(ev)) in
                     gt ∈ groups(df)
                  end)),

is_wf_cow_insem : D.Date × Dairy_farm → Bool
is_wf_cow_insem(d, df) ≡
   (∀ ci : GT.Cow_id •
      (ci ∈ cows(df) ∨ ci ∈ past_cows(df)) ⇒
         (∀ ev : CH.Event •
               ev ∈
                  cow_events(
                     ci, D.upto(d), CE.insemination, df) ⇒
```

```
                    let
                      insem =
                        CE.insem_classif(CH.event_inf(ev))
                    in
                      case insem of
                        GT.nat_insem(bi) → bi ∈ bulls(df),
                        _ → true
                      end
                    end)),

is_wf_group_to_plot : D.Date × Dairy_farm → Bool
is_wf_group_to_plot(d, df) ≡
  (∀ gt : GT.Group_id •
     gt ∈ groups(df) ⇒
       (∀ ev : GH.Event •
           ev ∈
             group_events(
                gt, D.upto(d), GE.group_to_plot, df) ⇒
             let plot = GE.plot_in(GH.event_inf(ev)) in
               GT.field_id(plot) ∈ fields(df) ∧
               (FS.exists_plot(
                   GT.plot_id(plot), GT.field_id(plot),
                   fields(df)) ∨
                FS.is_past_plot(
                   GT.plot_id(plot), GT.field_id(plot),
                   fields(df)))
             end)),

check_milking_record : Dairy_farm → Bool
check_milking_record(df) ≡
  (∀ ci : GT.Cow_id •
     ci ∈ cows(df) ⇒
       (∀ h1, h2, h3 : CH.History •
           CS.cow_history(ci, cows(df)) = h3 ^ h2 ^ h1 ∧
           h2 ≠ ⟨⟩ ∧
           CE.kind_of(CH.event_inf(h2(len (h2)))) =
             CE.birth ∧
           ∼CH.in_history(CE.cow_dried, h2) ∧
           ∼CH.in_history(CE.death, h2) ∧
           (∀ ev : CH.Event •
              ev ∈ h2 ∧
              CE.is_cow_to_group(CH.event_inf(ev)) ⇒
                CE.group(CH.event_inf(ev)) ≠ GT.dry_cow
           ) ⇒
             CH.check_event_record(
                K.day_milkings, K.hours_tolerance, h2))),
      −− Two more functions similar to check_milking_record
      −− should be written: one to check that each
      −− group receives one feeding in each 24 hours
      −− period and the other to check that groups
      −− of type heifer,one and two have two heat detections
      −− per 24 hours period
      −− more checking functions may be necessary
      is_consistent_dairy_farm :
         Nat × Nat × D.Date × Dairy_farm → Bool
      is_consistent_dairy_farm(max_h, min_h, d, df) ≡
         is_wf_group_plot(d, df) ∧
         is_wf_cow_mother(d, df) ∧ is_wf_cow_group(d, df) ∧
         is_wf_cow_insem(d, df) ∧
         is_wf_group_to_plot(d, df) ∧
         check_milking_record(df)
   end
```

C.2 COW_GROUPS Module

```
context: COW_GROUP, COWS

scheme COW_GROUPS(CS : COWS) =
   class
      object CG : COW_GROUP

      type Cow_groups = GT.Group_id -m-> CG.Cow_group

      value
         add_cow_group :
            GT.Group_id × Cow_groups -~-> Cow_groups
         add_cow_group(gt, cgs) ≡
            cgs † [gt ↦ CG.make_cow_group()]
         pre gt ∉ cgs ∧ ∼is_calf_group(gt),

         is_calf_group : GT.Group_id → Bool
         is_calf_group(gt) ≡
            case gt of
               GT.calf(_, _) → true,
               _ → false
            end,

         define_range : GT.Cow_id × Cow_groups → Nat × Nat
         /∗ dummy value for now ∗/
         define_range(id, cgs) ≡ (0, 0),
define_calf_group :
   Nat × Nat × Cow_groups ⥲ Cow_groups
define_calf_group(ds, de, cgs) ≡
   cgs † [ GT.calf(ds, de) ↦ CG.make_cow_group() ]
pre GT.calf(ds, de) ∉ cgs,

delete_cow_group :
   GT.Group_id × Cow_groups → Cow_groups
delete_cow_group(gt, cgs) ≡ cgs \ {gt},

cows_in_group :
   GT.Group_id × Cow_groups × CS.Cows ⥲ CS.Cows
cows_in_group(gt, cgs, cs) ≡
   if cs = [ ] then [ ]
   else
      let ci = hd cs in
         if CS.is_in_group(ci, gt, cs)
         then
            [ ci ↦ cs(ci) ] †
            cows_in_group(gt, cgs, cs \ {ci})
         else cows_in_group(gt, cgs, cs \ {ci})
         end
      end
   end
pre gt ∈ cgs,

is_female_calf_group :
   GT.Group_id × Cow_groups × CS.Cows → Bool
is_female_calf_group(gt, cgs, cs) ≡
   gt ∈ cgs ∧
   (∀ ci : GT.Cow_id •
       ci ∈ cows_in_group(gt, cgs, cs) ⇒
          CS.is_female_calf(ci, cs)),

has_produced_milk :
   GT.Group_id × D.Period × Cow_groups × CS.Cows →
      Bool
has_produced_milk(gt, p, cgs, cs) ≡
   gt ∈ cgs ∧ (gt = GT.one ∨ gt = GT.two) ∧
   (∃ ci : GT.Cow_id •
       ci ∈ cs ∧ CS.is_in_group(ci, gt, cs) ∧
       CS.has_produced_milk(ci, p, cs)),

group_indiv_prod :
   GT.Group_id × D.Period × Cow_groups × CS.Cows ⥲
```

```
        GT.Indiv_prod
group_indiv_prod(gt, p, cgs, cs) ≡
    CS.cows_milk_in_period(p, cows_in_group(gt, cgs, cs)) /
    real (CS.number_milkings_in_period(
                p, cows_in_group(gt, cgs, cs)))
pre has_produced_milk(gt, p, cgs, cs),

can_select_calf_group : GT.Cow_id × CS.Cows → Bool
can_select_calf_group(ci, cs) ≡
    ci ∈ cs ∧ CS.is_calf(ci, cs),

select_calf_group :
    GT.Cow_id × D.Date × CS.Cows × Cow_groups ~→
        GT.Group_id × Cow_groups
select_calf_group(ci, d, cs, cgs) ≡
    if cgs = [ ]
    then
        let
            (s, e) = define_range(ci, cgs),
            new_groups = define_calf_group(s, e, cgs)
        in
            (GT.calf(s, e), new_groups)
        end
    else
        case hd cgs of
            GT.calf(ds, de) →
                if
                    D.is_in_range(
                        ds, de, CS.C.cow_age_days(d, cs(ci)))
                then (GT.calf(ds, de), cgs)
                else
                    let
                        (s, e) = define_range(ci, cgs),
                        new_groups = define_calf_group(s, e, cgs)
                    in
                        (GT.calf(s, e), new_groups)
                    end
                end,
            _ →
                select_calf_group(ci, d, cs, cgs \ {hd cgs})
        end
    end
pre can_select_calf_group(ci, cs),

empty : GT.Group_id × Cow_groups × CS.Cows → Bool
empty(gt, cgs, cs) ≡
```

```
    gt ∈ cgs ∧ cows_in_group(gt, cgs, cs) = [ ],

can_goto_group :
    GT.Cow_id × D.Date × GT.Group_id × CS.Cows ×
    Cow_groups →
        Bool
can_goto_group(ci, d, gt, cs, cgs) ≡
    ci ∈ cs ∧ gt ∈ cgs ∧
    CS.can_goto_group(ci, d, gt, cs),

select_group_for_cow :
    GT.Cow_id × D.Date × CS.Cows × Cow_groups ~→
        GT.Group_id
select_group_for_cow(ci, d, cs, cgs) ≡
    if CS.is_calf(ci, cs)
    then
        let (gt, gs) = select_calf_group(ci, d, cs, cgs) in
            gt
        end
    else CS.select_group_for_cow(ci, d, cs)
    end
pre
    ci ∈ cs ∧
    (∼CS.is_calf(ci, cs) ∧
     CS.select_group_for_cow(ci, d, cs) ∈ cgs ∨
     CS.is_calf(ci, cs) ∧
     let (gt, gs) = select_calf_group(ci, d, cs, cgs) in
         gt ∈ cgs
     end),

average_weight :
    GT.Group_id × Cow_groups × CS.Cows → GT.Weight
average_weight(gt, cgs, cs) ≡
    let
        cows = cows_in_group(gt, cgs, cs),
        number_cows = card (dom (cows))
    in
        CS.sum_current_weight(cows) / real (number_cows)
    end
pre gt ∈ cgs,

can_compute_ration :
    GT.Group_id × Cow_groups × CS.Cows → Bool
can_compute_ration(gt, cgs, cs) ≡
    gt ∈ cgs ∧ ∼empty(gt, cgs, cs) ∧
    gt ≠ GT.calf_rearing_unit,
compute_ration :
   GT.Group_id × Cow_groups × CS.Cows $\xrightarrow{\sim}$ GT.Quantity
compute_ration(gt, cgs, cs) ≡
   **let**
      p =
         **case** gt **of**
            GT.one → K.ration_one,
            GT.pre_birth_cow → K.ration_two,
            GT.two → K.ration_two,
            GT.dry_cow → K.ration_dry,
            GT.discard_cow → K.ration_discard,
            GT.heifer → K.ration_heifer,
            GT.calf(min, max) →
               **if**
                  min ≥ K.calf_min_age_ration ∧
                  max ≤ K.calf_max_age_ration
               **then** K.ration_calf_min
               **else** K.ration_calf_max
               **end**
         **end**
   **in**
      p ∗ average_weight(gt, cgs, cs) / 100.00
   **end**
**pre** can_compute_ration(gt, cgs, cs),

can_send_to_pasture :
   GT.Group_id × D.Date × GT.Plot_id ×
   GT.Field_id × Cow_groups × CS.Cows →
      **Bool**
can_send_to_pasture(gt, d, pi, fi, cgs, cs) ≡
   gt ∈ cgs ∧ is_only_group(gt, d, pi, fi, cgs) ∧
   ∼empty(gt, cgs, cs) ∧
   CG.can_send_to_pasture(d, cgs(gt)),

is_only_group :
   GT.Group_id × D.Date × GT.Plot_id ×
   GT.Field_id × Cow_groups →
      **Bool**
is_only_group(gt, d, pi, fi, cgs) ≡
   gt ∈ cgs ∧
   (∀ g : GT.Group_id •
       g ∈ cgs ⇒
          g = gt ∨
          CG.in_plot(d, cgs(g)) ≠ GT.mk_In_plot(fi, pi)),
send_to_pasture :
   GT.Group_id × D.Date × Nat × GT.Plot_id ×
   GT.Field_id × Cow_groups × CS.Cows ~→
      Cow_groups
send_to_pasture(gt, d, dn, pi, fi, cgs, cs) ≡
   cgs †
   [ gt ↦ CG.send_to_pasture(d, dn, pi, fi, cgs(gt)) ]
pre can_send_to_pasture(gt, d, pi, fi, cgs, cs),

compute_pasture_eaten :
   GT.Group_id × GT.Ration × Cow_groups × CS.Cows ~→
      GT.Quantity
compute_pasture_eaten(gt, r, cgs, cs) ≡
   let rq = compute_ration(gt, cgs, cs) in
      (rq − GT.total_foods(r))
   end
pre gt ∈ cgs,

is_well_def_ration_one :
   GT.Quantity × GT.Corn_sil × GT.Hay ×
   GT.Balanced × GT.Grain →
      Bool
is_well_def_ration_one(rq, corn, hay, bal, gr) ≡
   corn ≥ K.min_corn_one ∗ rq / 100.0 ∧
   corn ≤ K.max_corn_one ∗ rq / 100.0 ∧
   hay ≥ K.min_hay_one ∗ rq / 100.0 ∧
   hay ≤ K.max_hay_one ∗ rq / 100.0 ∧
   bal + GT.gr_quantity(gr) ≥
      K.min_conc_one ∗ rq / 100.0 ∧
   bal + GT.gr_quantity(gr) ≤
      K.max_conc_one ∗ rq / 100.0,

is_well_def_ration_two :
   GT.Quantity × GT.Corn_sil × GT.Hay ×
   GT.Balanced × GT.Grain →
      Bool
is_well_def_ration_two(rq, corn, hay, bal, gr) ≡
   corn ≥ K.min_corn_two ∗ rq / 100.0 ∧
   corn ≤ K.max_corn_two ∗ rq / 100.0 ∧
   hay ≥ K.min_hay_two ∗ rq / 100.0 ∧
   hay ≤ K.max_hay_two ∗ rq / 100.0 ∧
   bal + GT.gr_quantity(gr) ≥
      K.min_conc_two ∗ rq / 100.0 ∧
   bal + GT.gr_quantity(gr) ≤
      K.max_conc_two ∗ rq / 100.0,
is_well_def_ration_discard :
   GT.Quantity × GT.Corn_sil × GT.Hay ×
   GT.Balanced × GT.Grain →
      Bool
is_well_def_ration_discard(rq, corn, hay, bal, gr) ≡
   corn = 0.0 ∧ hay = 0.0 ∧
   bal + GT.gr_quantity(gr) = 0.0,

is_well_def_ration_calff :
   GT.Quantity × GT.Corn_sil × GT.Hay ×
   GT.Balanced × GT.Grain →
      Bool
is_well_def_ration_calff(rq, corn, hay, bal, gr) ≡
   corn = 0.0 ∧ hay = 0.0 ∧ bal ≥ K.min_bal_calf ∧
   bal ≤ K.min_bal_calf ∧ GT.gr_quantity(gr) = 0.0,

is_well_def_ration_calfm :
   GT.Quantity × GT.Corn_sil × GT.Hay ×
   GT.Balanced × GT.Grain →
      Bool
is_well_def_ration_calfm(rq, corn, hay, bal, gr) ≡
   corn = 0.0 ∧ hay = 0.0 ∧ bal = 0.0 ∧
   GT.gr_quantity(gr) = 0.0,

is_well_def_ration :
   GT.Group_id × GT.Quantity × GT.Corn_sil ×
   GT.Hay × GT.Balanced × GT.Grain × Cow_groups ×
   CS.Cows →
      Bool
is_well_def_ration(gt, rq, corn, hay, bal, gr, cgs, cs) ≡
   gt ∈ cgs ∧ ∼empty(gt, cgs, cs) ∧
   gt ≠ GT.calf_rearing_unit ∧
   if (gt = GT.one ∨ gt = GT.heifer)
   then is_well_def_ration_one(rq, corn, hay, bal, gr)
   elsif
      (gt = GT.two ∨ gt = GT.dry_cow ∨
       gt = GT.pre_birth_cow)
      then
         is_well_def_ration_two(rq, corn, hay, bal, gr)
   elsif gt = GT.discard_cow
      then
         is_well_def_ration_discard(
             rq, corn, hay, bal, gr)
   elsif
      let GT.calf(s, e) = gt in
         (e ≤ K.calf_middle_age ∨
           is_female_calf_group(gt, cgs, cs))
        end
        then
           is_well_def_ration_calff(rq, corn, hay, bal, gr)
     else
        is_well_def_ration_calfm(rq, corn, hay, bal, gr)
     end,

  can_feed_group :
     GT.Group_id × D.Date × Cow_groups × CS.Cows →
        Bool
  can_feed_group(gt, d, cgs, cs) ≡
     gt ∈ cgs ∧ ∼empty(gt, cgs, cs) ∧
     gt ≠ GT.calf_rearing_unit ∧
     CG.can_feed_group(d, cgs(gt)),

  feed_group :
     GT.Group_id × D.Date × GT.Corn_sil × GT.Hay ×
     GT.Conc × Cow_groups × CS.Cows ~→
        Cow_groups
  feed_group(gt, d, corn, hay, conc, cgs, cs) ≡
     let
        ration = GT.mk_Ration(0.0, corn, hay, conc),
        new_r =
           GT.chg_pasture(
              compute_pasture_eaten(gt, ration, cgs, cs),
              ration)
     in
        cgs † [ gt ↦ CG.feed_group(new_r, d, cgs(gt)) ]
     end
  pre can_feed_group(gt, d, cgs, cs),

  can_detect_heat :
     GT.Group_id × D.Date × (GT.Cow_id × Bool)* ×
     Cow_groups × CS.Cows →
        Bool
  can_detect_heat(gt, d, csl, cgs, cs) ≡
     gt ∈ cgs ∧
     (gt = GT.one ∨ gt = GT.two ∨ gt = GT.heifer) ∧
     (∀ (ci, h) : GT.Cow_id × Bool •
         (ci, h) ∈ csl ⇒
            ci ∈ cows_in_group(gt, cgs, cs)) ∧
     CG.can_detect_heat(d, cgs(gt)),

  detect_heat :
     GT.Group_id × D.Date × (GT.Cow_id × Bool)* ×
```

```
    Cow_groups × CS.Cows $\overset{\sim}{\to}$
        Cow_groups × CS.Cows
detect_heat(gt, d, csl, cgs, cs) ≡
    (cgs † [ gt ↦ CG.detect_heat(d, cgs(gt)) ],
     cs † CS.register_heat(d, csl, cs))
pre can_detect_heat(gt, d, csl, cgs, cs),

number_cows_in_group :
    GT.Group_id × Cow_groups × CS.Cows $\overset{\sim}{\to}$ Nat
number_cows_in_group(gt, cgs, cs) ≡
    card (dom (cows_in_group(gt, cgs, cs)))
pre gt ∈ cgs,

is_in_field :
    GT.Field_id × D.Date × GT.Group_id × Cow_groups →
        Bool
is_in_field(fi, d, gt, cgs) ≡
    gt ∈ cgs ∧
    GT.field_id(CG.in_plot(d, cgs(gt))) = fi,

number_cows_in_field :
    GT.Field_id × D.Date × Cow_groups × CS.Cows → Nat
number_cows_in_field(fi, d, cgs, cs) ≡
    if cgs = [ ] then 0
    else
        let gt = hd cgs in
            if is_in_field(fi, d, gt, cgs)
            then
                number_cows_in_group(gt, cgs, cs) +
                number_cows_in_field(fi, d, cgs \ {gt}, cs)
            else number_cows_in_field(fi, d, cgs \ {gt}, cs)
            end
        end
    end,

group_current_plot :
    GT.Group_id × D.Date × Cow_groups $\overset{\sim}{\to}$ GT.In_plot
group_current_plot(gt, d, cgs) ≡
    CG.in_plot(d, cgs(gt))
pre gt ∈ cgs,

group_events :
    GT.Group_id × D.Period × GE.Group_event_kind ×
    Cow_groups $\overset{\sim}{\to}$
        GH.History
group_events(gt, p, ev_kind, cgs) ≡
```

```
            CG.group_events(p, ev_kind, cgs(gt))
        pre gt ∈ cgs
  end
```

C.3 COW_GROUP Module

context: GT, GH

```
scheme COW_GROUP =
  class
    type Cow_group :: history : GH.History ↔ chg_history

    value
      in_plot : D.Date × Cow_group ~→ GT.In_plot
      in_plot(d, cg) ≡
        let
          GE.group_to_plot(pl, n) =
            GH.get_last_ev_info(
                GE.group_to_plot, history(cg))
        in
          pl
        end
      pre GH.in_history(GE.group_to_plot, history(cg)),

      is_in_plot : GT.In_plot × D.Date × Cow_group → Bool
      is_in_plot(pl, d, cg) ≡
        GH.in_history(GE.group_to_plot, history(cg)) ∧
        let
          d1 =
            GH.get_last_ev_date(
                GE.group_to_plot, history(cg)),
          GE.group_to_plot(plo, n) =
            GH.get_last_ev_info(
                GE.group_to_plot, history(cg))
        in
          pl = plo ∧
          ∼GH.event_in_period(
               GE.group_out_plot, D.since(d1), history(cg))
        end,

      make_cow_group : Unit → Cow_group
      make_cow_group() ≡ mk_Cow_group(GH.empty),

      can_send_to_pasture : D.Date × Cow_group → Bool
      can_send_to_pasture(d, cg) ≡
```

```
        if GH.in_history(GE.group_to_plot, history(cg))
        then
          let
            d1 =
              GH.get_last_ev_date(
                 GE.group_to_plot, history(cg)),
            ev =
              GH.get_last_ev_info(
                 GE.group_to_plot, history(cg)),
            n = GE.days_number(ev)
          in
            ~GH.event_in_period(
                GE.group_out_plot, D.since(d1), history(cg)) ∧
            ~GH.event_in_period(
                GE.group_to_plot, D.since(d1), history(cg))
          end
        else true
        end,

      send_to_pasture :
        D.Date × Nat × GT.Plot_id × GT.Field_id ×
        Cow_group $\overset{\sim}{\rightarrow}$
          Cow_group
      send_to_pasture(d, dn, pi, fi, cg) ≡
        if GH.in_history(GE.group_to_plot, history(cg))
        then
          let
            new_gr =
              chg_history(
                 GH.add_event(
                     d, GE.group_out_plot, history(cg)), cg)
          in
            chg_history(
               GH.add_event(
                   d,
                   GE.group_to_plot(GT.mk_In_plot(fi, pi), dn),
                   history(new_gr)), cg)
          end
        else
          chg_history(
             GH.add_event(
                 d,
                 GE.group_to_plot(GT.mk_In_plot(fi, pi), dn),
                 history(cg)), cg)
        end
      pre can_send_to_pasture(d, cg),
can_feed_group : D.Date × Cow_group → Bool
can_feed_group(d, cg) ≡
   GH.in_history(GE.group_to_plot, history(cg)) ∧
   ∼GH.event_in_period(
         GE.feeding, D.since(D.last_midnight(d)),
         history(cg)),

feed_group :
   GT.Ration × D.Date × Cow_group → Cow_group
feed_group(r, d, cg) ≡
   chg_history(
       GH.add_event(d, GE.feeding(r), history(cg)), cg),

can_detect_heat : D.Date × Cow_group → Bool
can_detect_heat(d, cg) ≡
   if D.in_morning(d)
   then
      GH.event_in_period(
          GE.heat_detection, D.since(D.last_midnight(d)),
          history(cg))
   else
      GH.event_in_period(
          GE.heat_detection, D.since(D.last_midday(d)),
          history(cg))
   end,

detect_heat : D.Date × Cow_group ∼→ Cow_group
detect_heat(d, cg) ≡
   chg_history(
       GH.add_event(d, GE.heat_detection, history(cg)),
       cg)
pre can_detect_heat(d, cg),

group_rations : D.Period × Cow_group → GH.History
group_rations(p, cg) ≡
   GH.filter(GE.is_feeding)(history(cg)),

group_events :
   D.Period × GE.Group_event_kind × Cow_group →
      GH.History
group_events(p, ev_kind, cg) ≡
   case ev_kind of
      GE.group_to_plot → group_plots(p, cg),
      GE.feeding → group_feedings(p, cg),
      GE.heat_detection → group_heat_det(p, cg),
            _ → ⟨⟩
          end,

        group_plots : D.Period × Cow_group → GH.History
        group_plots(p, cg) ≡
          GH.evs_in_period(
               p, GH.filter(GE.is_group_to_plot)(history(cg))),

        group_feedings : D.Period × Cow_group → GH.History
        group_feedings(p, cg) ≡
          GH.evs_in_period(
               p, GH.filter(GE.is_feeding)(history(cg))),

        group_heat_det : D.Period × Cow_group → GH.History
        group_heat_det(p, cg) ≡
          GH.evs_in_period(
               p, GH.filter(GE.is_heat_detection)(history(cg)))
  end
```

C.4 GH Module

context: HISTORY, GE

```
object GH :
  HISTORY(
     GE{Group_event for Event_info,
          Group_event_kind for Event_kind})
```

C.5 GE Module

context: GROUP_EVENT

object GE : GROUP_EVENT

C.6 GROUP_EVENT Module

context: K

```
scheme GROUP_EVENT =
  class
    type
      Group_event ==
        group_to_plot(
            plot_in : GT.In_plot, days_number : Nat) |
        group_out_plot |
        feeding(ration : GT.Ration) |
        heat_detection,
    Group_event_kind ==
        group_to_plot |
        group_out_plot |
        feeding |
        heat_detection

value
    kind_of : Group_event → Group_event_kind
    kind_of(gev) ≡
        case gev of
            group_to_plot(_, _) → group_to_plot,
            group_out_plot → group_out_plot,
            feeding(_) → feeding,
            heat_detection → heat_detection
        end,

    is_feeding : Group_event → Bool
    is_feeding(gev) ≡
        case kind_of(gev) of
            feeding → true,
            _ → false
        end,

    is_group_to_plot : Group_event → Bool
    is_group_to_plot(gev) ≡
        case kind_of(gev) of
            group_to_plot → true,
            _ → false
        end,

    is_group_out_plot : Group_event → Bool
    is_group_out_plot(gev) ≡
        case kind_of(gev) of
            group_out_plot → true,
            _ → false
        end,

    is_heat_detection : Group_event → Bool
    is_heat_detection(gev) ≡
        case kind_of(gev) of
            heat_detection → true,
          _ → false
        end
  end
```

C.7 COWS Module

```
context: COW

scheme COWS =
  class
    object C : COW

    type Cows = GT.Cow_id $\overrightarrow{m}$ C.Cow

    value
      add_cow :
        GT.Cow_id × D.Date × GT.Cow_classif × Cows $\xrightarrow{\sim}$
          Cows
      add_cow(ci, birthd, classif, cs) ≡
        cs † [ ci ↦ C.make_cow(birthd, classif) ]
      pre ci ∉ cs,

      delete_cow : GT.Cow_id × Cows → Cows
      delete_cow(ci, cs) ≡ cs \ {ci},

      update_classif :
        GT.Cow_id × GT.Cow_classif × Cows $\xrightarrow{\sim}$ Cows
      update_classif(ci, classif, cs) ≡
        let new_clas = C.chg_classif(classif, cs(ci)) in
          cs † [ ci ↦ new_clas ]
        end
      pre ci ∈ cs,

      is_female_calf : GT.Cow_id × Cows → Bool
      is_female_calf(ci, cs) ≡
        ci ∈ cs ∧ C.is_female_calf(cs(ci)),

      can_milk_cow : GT.Cow_id × D.Date × Cows → Bool
      can_milk_cow(ci, d, cs) ≡
        ci ∈ cs ∧ C.can_milk_cow(d, cs(ci)),

      milk_cow :
        GT.Cow_id × D.Date × GT.Litres × Cows $\xrightarrow{\sim}$ Cows
      milk_cow(ci, d, lts, cs) ≡
        cs † [ ci ↦ C.milk_cow(d, lts, cs(ci)) ]
```

```
pre can_milk_cow(ci, d, cs),

can_dry_cow : GT.Cow_id × D.Date × Cows → Bool
can_dry_cow(ci, d, cs) ≡
   ci ∈ cs ∧ ∼C.is_dried(cs(ci)),

dry_cow :
   GT.Cow_id × D.Date × GT.Dried_cause × Cows -~→
      Cows
dry_cow(ci, d, dc, cs) ≡
   cs † [ ci ↦ C.dry_cow(d, dc, cs(ci)) ]
pre can_dry_cow(ci, d, cs),

can_weigh_cow : GT.Cow_id × D.Date × Cows → Bool
can_weigh_cow(ci, d, cs) ≡
   ci ∈ cs ∧ C.can_weigh_cow(d, cs(ci)),

weigh_cow :
   GT.Cow_id × D.Date × GT.Weight × Cows -~→ Cows
weigh_cow(ci, d, w, cs) ≡
   cs † [ ci ↦ C.weigh_cow(d, w, cs(ci)) ]
pre can_weigh_cow(ci, d, cs),

can_deparasite_cow :
   GT.Cow_id × D.Date × Cows → Bool
can_deparasite_cow(ci, d, cs) ≡
   ci ∈ cs ∧ C.can_deparasite_cow(d, cs(ci)),

deparasite_cow :
   GT.Cow_id × D.Date × GT.Dep_inf × Cows -~→ Cows
deparasite_cow(ci, d, di, cs) ≡
   cs † [ ci ↦ C.deparasite_cow(d, di, cs(ci)) ]
pre can_deparasite_cow(ci, d, cs),

can_insem_cow : GT.Cow_id × D.Date × Cows → Bool
can_insem_cow(ci, d, cs) ≡
   ci ∈ cs ∧ C.can_insem_cow(d, cs(ci)),

insem_cow_artif :
   GT.Cow_id × D.Date × GT.Artif_info × Cows -~→
      Cows
insem_cow_artif(ci, d, ai, cs) ≡
   cs † [ ci ↦ C.insem_cow_artif(d, ai, cs(ci)) ]
pre can_insem_cow(ci, d, cs),

insem_cow_natural :
   GT.Cow_id × D.Date × GT.Bull_id × Cows →̃ Cows
insem_cow_natural(ci, d, bi, cs) ≡
   cs † [ ci ↦ C.insem_cow_natural(d, bi, cs(ci)) ]
pre can_insem_cow(ci, d, cs),

can_detect_pregnancy :
   GT.Cow_id × D.Date × Cows → Bool
can_detect_pregnancy(ci, d, cs) ≡
   ci ∈ cs ∧ C.can_detect_pregnancy(d, cs(ci)),

detect_pregnant_cow :
   GT.Cow_id × D.Date × Bool × Cows →̃ Cows
detect_pregnant_cow(ci, d, preg, cs) ≡
   cs † [ ci ↦ C.detect_pregnant_cow(d, preg, cs(ci)) ]
pre can_detect_pregnancy(ci, d, cs),

can_save_cow_death : GT.Cow_id × Cows × Cows → Bool
can_save_cow_death(ci, cs, pcs) ≡
   ci ∈ cs ∧ ∼C.is_dead_cow(cs(ci)) ∧ ci ∉ pcs,

save_cow_death :
   GT.Cow_id × D.Date × GT.Death_cause × Cows × Cows →̃
      Cows × Cows
save_cow_death(ci, d, dc, cs, pcs) ≡
   let new_cow = C.save_cow_death(d, dc, cs(ci)) in
      (delete_cow(ci, cs), pcs † [ ci ↦ new_cow ])
   end
pre can_save_cow_death(ci, cs, pcs),

can_receive_vacc :
   GT.Cow_id × D.Date × GT.Vaccine × Cows → Bool
can_receive_vacc(ci, d, vacc, cs) ≡
   ci ∈ cs ∧ C.can_receive_vacc(d, vacc, cs(ci)),

vaccinate_cow :
   GT.Cow_id × D.Date × GT.Vaccine × Cows →̃ Cows
vaccinate_cow(ci, d, vacc, cs) ≡
   cs † [ ci ↦ C.vaccinate_cow(d, vacc, cs(ci)) ]
pre can_receive_vacc(ci, d, vacc, cs),

can_sell_cow :
   GT.Cow_id × D.Date × Cows × Cows → Bool
can_sell_cow(ci, d, cs, pcs) ≡
   ci ∈ cs ∧ C.can_sell_cow(d, cs(ci)) ∧
   ci ∉ pcs,
sell_cow :
   GT.Cow_id × D.Date × Cows × Cows $\xrightarrow{\sim}$
      Cows × Cows
sell_cow(ci, d, cs, pcs) ≡
   **let** new_cow = C.sell_cow(d, cs(ci)) **in**
      (delete_cow(ci, cs), pcs † [ci ↦ new_cow])
   **end**
**pre** can_sell_cow(ci, d, cs, pcs),

can_give_birth : GT.Cow_id × D.Date × Cows → **Bool**
can_give_birth(ci, d, cs) ≡
   ci ∈ cs ∧ C.can_give_birth(d, cs(ci)),

next_birth_date :
   GT.Cow_id × D.Date × Cows $\xrightarrow{\sim}$ D.Date
next_birth_date(ci, d, cs) ≡
   C.next_birth_date(d, cs(ci))
**pre** can_give_birth(ci, d, cs),

has_produced_milk :
   GT.Cow_id × D.Period × Cows → **Bool**
has_produced_milk(ci, p, cs) ≡
   ci ∈ cs ∧ C.has_produced_milk(p, cs(ci)),

cow_indiv_prod :
   GT.Cow_id × D.Period × Cows $\xrightarrow{\sim}$ GT.Indiv_prod
cow_indiv_prod(ci, p, cs) ≡
   C.cow_indiv_prod(p, cs(ci))
**pre** has_produced_milk(ci, p, cs),

cows_milk_in_period : D.Period × Cows → GT.Litres
cows_milk_in_period(p, cs) ≡
   **if** cs = [ ] **then** 0.0
   **else**
      **let** ci = **hd** cs **in**
         C.milk_in_period(p, cs(ci)) +
         cows_milk_in_period(p, cs \ {ci})
      **end**
   **end**,

number_milkings_in_period : D.Period × Cows → **Nat**
number_milkings_in_period(p, cs) ≡
   **if** cs = [ ] **then** 0
   **else**
      **let** ci = **hd** cs **in**
         C.number_milkings_in_period(p, cs(ci)) +
          number_milkings_in_period(p, cs \ {ci})
        end
      end,

    is_in_group : GT.Cow_id × GT.Group_id × Cows → Bool
    is_in_group(ci, gt, cs) ≡
      ci ∈ cs ∧ C.is_in_group(gt, cs(ci)),

    is_calf : GT.Cow_id × Cows → Bool
    is_calf(ci, cs) ≡ ci ∈ cs ∧ C.is_calf(cs(ci)),

    can_be_in_group : GT.Cow_id × Cows → Bool
    can_be_in_group(ci, cs) ≡
      ci ∈ cs ∧ C.can_be_in_group(cs(ci)),

    cow_group : GT.Cow_id × Cows →̃ GT.Group_id
    cow_group(ci, cs) ≡ C.cow_group(cs(ci))
    pre can_be_in_group(ci, cs),

    can_goto_group :
      GT.Cow_id × D.Date × GT.Group_id × Cows → Bool
    can_goto_group(ci, d, gt, cs) ≡
      ci ∈ cs ∧ C.can_goto_group(d, gt, cs(ci)),

    select_group_for_cow :
      GT.Cow_id × D.Date × Cows →̃ GT.Group_id
    select_group_for_cow(ci, d, cs) ≡
      C.select_group_for_cow(d, cs(ci))
    pre
      ci ∈ cs ∧
      (C.is_heifer(cs(ci)) ∨ C.is_dairy_cow(cs(ci))),

    assign_group_to_cow :
      GT.Cow_id × D.Date × GT.Group_id × Cows →̃ Cows
    assign_group_to_cow(ci, d, gt, cs) ≡
      cs † [ ci ↦ C.assign_group_to_cow(d, gt, cs(ci)) ]
    pre can_goto_group(ci, d, gt, cs),

    define_cow_classif :
      GT.Cow_id × D.Date × Cows →̃ Cows
    define_cow_classif(ci, d, cs) ≡
      cs † [ ci ↦ C.define_cow_classif(d, cs(ci)) ]
    pre ci ∈ cs,

    can_goto_cru : GT.Cow_id × D.Date × Cows → Bool
    can_goto_cru(ci, d, cs) ≡
    ci ∈ cs ∧ C.can_goto_cru(d, cs(ci)),

send_calf_to_cru :
    GT.Cow_id × D.Date × Cows $\overset{\sim}{\to}$ Cows
send_calf_to_cru(ci, d, cs) ≡
    cs † [ ci ↦ C.send_calf_to_cru(d, cs(ci)) ]
pre can_goto_cru(ci, d, cs),

can_take_out_cru : GT.Cow_id × D.Date × Cows → Bool
can_take_out_cru(ci, d, cs) ≡
    ci ∈ cs ∧ C.can_take_out_cru(d, cs(ci)),

can_breed_artif :
    GT.Cow_id × D.Date × GT.Litres × GT.Quantity ×
    Cows →
        Bool
can_breed_artif(ci, d, mr, bal, cs) ≡
    ci ∈ cs ∧ C.can_breed_artif(d, mr, bal, cs(ci)),

breed_artif :
    GT.Cow_id × D.Date × GT.Litres × GT.Quantity ×
    Cows $\overset{\sim}{\to}$
        Cows
breed_artif(ci, d, mr, bal, cs) ≡
    cs † [ ci ↦ C.breed_artif(d, mr, bal, cs(ci)) ]
pre can_breed_artif(ci, d, mr, bal, cs),

can_eat_bal : GT.Cow_id × D.Date × Cows → Bool
can_eat_bal(ci, d, cs) ≡
    ci ∈ cs ∧ C.can_eat_bal(d, cs(ci)),

register_heat :
    D.Date × (GT.Cow_id × Bool)$^*$ × Cows → Cows
register_heat(d, csl, cs) ≡
    if csl = ⟨⟩ then cs
    else
        let (ci, heat) = hd csl in
            if
                ci ∈ cs ∧
                C.can_register_heat(d, heat, cs(ci))
            then
                cs † [ ci ↦ C.register_heat(d, heat, cs(ci)) ] ∪
                register_heat(d, tl csl, cs)
            else register_heat(d, tl csl, cs)
            end
        end
```

```
   end,

define_cow_id : Unit → GT.Cow_id
/∗ dummy value for now ∗/
define_cow_id() ≡ 0,

give_birth :
   GT.Cow_id × GT.Calf_sex × D.Date × Cows ⥲ Cows
give_birth(ci, csex, d, cs) ≡
   let calf_id = define_cow_id() in
      cs †
      [ ci ↦ C.give_birth(d, calf_id, csex, cs(ci)) ] ∪
      [ calf_id ↦ C.make_calf(csex, d) ]
   end
pre can_give_birth(ci, d, cs),

cow_mother : GT.Cow_id × Cows ⥲ GT.Cow_id
cow_mother(ci, cs) ≡
   let cim = hd cs in
      if C.is_mother(ci, cs(ci), cs(cim)) then cim
      else cow_mother(ci, cs \ {cim})
      end
   end
pre ci ∈ cs ∧ cs ≠ [ ],

cow_events :
   GT.Cow_id × D.Period × CE.Cow_event_kind × Cows ⥲
      CH.History
cow_events(ci, p, ev_kind, cs) ≡
   C.cow_events(p, ev_kind, cs(ci))
pre ci ∈ cs,

cow_classif : GT.Cow_id × Cows ⥲ GT.Cow_classif
cow_classif(ci, cs) ≡ C.cow_classif(cs(ci))
pre ci ∈ cs,

cow_history_p :
   GT.Cow_id × D.Period × Cows ⥲ CH.History
cow_history_p(ci, p, cs) ≡ C.cow_history_p(p, cs(ci))
pre ci ∈ cs,

cow_history : GT.Cow_id × Cows ⥲ CH.History
cow_history(ci, cs) ≡ C.history(cs(ci)),

sum_current_weight : Cows → GT.Weight
sum_current_weight(cs) ≡
```

```
            if cs = [ ] then 0.0
            else
               let ci = hd cs in
                  C.current_weight(cs(ci)) +
                  sum_current_weight(cs \ {ci})
               end
            end
   end
```

C.8 COW Module

```
context: CH

scheme COW =
   class
      type
         Cow ::
            birthday : D.Date
            cow_classif : GT.Cow_classif ↔ chg_classif
            history : CH.History ↔ chg_history

      value
         make_cow : D.Date × GT.Cow_classif → Cow
         make_cow(d, c_class) ≡ mk_Cow(d, c_class, CH.empty),

         default_heifer_info : GT.Heifer_info =
            GT.mk_Heifer_info(""),/* dummy value for now */

         is_calf : Cow → Bool
         is_calf(c) ≡
            case cow_classif(c) of
               GT.calf(_) → true,
               _ → false
            end,

         calf_location : Cow ~→ GT.Calf_location
         calf_location(c) ≡
            if CH.in_history(CE.cow_to_group, history(c))
            then
               let
                  CE.cow_to_group(gt) =
                     CH.get_last_ev_info(
                        CE.cow_to_group, history(c))
               in
                  GT.with_group(gt)
```

```
          end
      else GT.with_mother
      end
  pre is_calf(c),

  is_female_calf : Cow → Bool
  is_female_calf(c) ≡
      case cow_classif(c) of
          GT.calf(c_inf) →
              case GT.calf_sex(c_inf) of
                  GT.female_calf(_) → true,
                  GT.male_calf → false
              end,
          _ → false
      end,

  is_heifer : Cow → Bool
  is_heifer(c) ≡
      case cow_classif(c) of
          GT.heifer(_) → true,
          _ → false
      end,

  is_dairy_cow : Cow → Bool
  is_dairy_cow(c) ≡
      case cow_classif(c) of
          GT.dairy(_) → true,
          _ → false
      end,

  is_discard_cow : Cow → Bool
  is_discard_cow(c) ≡
      case cow_classif(c) of
          GT.dairy(dc_info) →
              case GT.dairy_classif(dc_info) of
                  GT.discard → true,
                  _ → false
              end,
          _ → false
      end,

  is_milking_cow : Cow → Bool
  is_milking_cow(c) ≡
      case cow_classif(c) of
          GT.dairy(c_inf) →
              case GT.dairy_classif(c_inf) of
```

```
                GT.milking(_) → true,
                _ → false
            end,
        _ → false
    end,

is_empty_cow : Cow → Bool
is_empty_cow(c) ≡
    case cow_classif(c) of
        GT.dairy(c_inf) →
            case GT.dairy_classif(c_inf) of
                GT.milking(mc_inf) →
                    case mc_inf of
                        GT.empty → true,
                        _ → false
                    end,
                _ → false
            end,
        _ → false
    end,

is_post_birth_cow : Cow → Bool
is_post_birth_cow(c) ≡
    case cow_classif(c) of
        GT.dairy(c_inf) →
            case GT.dairy_classif(c_inf) of
                GT.milking(mc_inf) →
                    case mc_inf of
                        GT.post_birth → true,
                        _ → false
                    end,
                _ → false
            end,
        _ → false
    end,

is_early_pregnant_cow : Cow → Bool
is_early_pregnant_cow(c) ≡
    case cow_classif(c) of
        GT.dairy(c_inf) →
            case GT.dairy_classif(c_inf) of
                GT.milking(mc_inf) →
                    case mc_inf of
                        GT.early_pregnant → true,
                        _ → false
                    end,
            _ → **false**
          **end**,
        _ → **false**
      **end**,

    is_dry_cow : Cow → **Bool**
    is_dry_cow(c) ≡
      **case** cow_classif(c) **of**
        GT.dairy(c_inf) →
          **case** GT.dairy_classif(c_inf) **of**
            GT.dry(_) → **true**,
            _ → **false**
          **end**,
        _ → **false**
      **end**,

    is_pre_birth_cow : Cow → **Bool**
    is_pre_birth_cow(c) ≡
      **case** cow_classif(c) **of**
        GT.dairy(c_inf) →
          **case** GT.dairy_classif(c_inf) **of**
            GT.dry(dc_inf) →
              **case** dc_inf **of**
                GT.pre_birth → **true**,
                _ → **false**
              **end**,
            _ → **false**
          **end**,
        _ → **false**
      **end**,

    is_non_pre_birth_cow : Cow → **Bool**
    is_non_pre_birth_cow(c) ≡
      **case** cow_classif(c) **of**
        GT.dairy(c_inf) →
          **case** GT.dairy_classif(c_inf) **of**
            GT.dry(dc_inf) →
              **case** dc_inf **of**
                GT.non_pre_birth → **true**,
                _ → **false**
              **end**,
            _ → **false**
          **end**,
        _ → **false**
      **end**,
cow_age_months : D.Date × Cow → Nat
cow_age_months(d, c) ≡ D.months_since(birthday(c), d),

dairy_cow_age : D.Date × Cow $\overset{\sim}{\to}$ Nat
dairy_cow_age(d, c) ≡
    D.months_since(first_birth_date(c), d) / 12
pre is_dairy_cow(c),

cow_age_days : D.Date × Cow → Nat
cow_age_days(d, c) ≡ D.days_since(birthday(c), d),

is_pregnant : D.Date × Cow → Bool
is_pregnant(d, c) ≡
    (is_heifer(c) ∨ is_dairy_cow(c)) ∧
    ∼is_on_heat(d, c) ∧
    CH.in_history(CE.preg_detection, history(c)) ∧
    let
        d =
            CH.get_last_ev_date(
                CE.preg_detection, history(c))
    in
        CE.pregnant(
            CH.get_last_ev_info(
                CE.preg_detection, history(c))) = true ∧
        ∼CH.event_in_period(
            CE.birth, D.since(d), history(c))
    end,

is_on_heat : D.Date × Cow → Bool
is_on_heat(d, c) ≡
    (is_heifer(c) ∨ is_dairy_cow(c)) ∧
    ∼is_pregnant(d, c) ∧
    CH.in_history(CE.heat, history(c)) ∧
    let d = CH.get_last_ev_date(CE.heat, history(c)) in
        CH.event_in_period(
            CE.heat, D.last_n_hours(K.heat_period, d),
            history(c))
    end,

pregnancy_month : D.Date × Cow $\overset{\sim}{\to}$ Nat
pregnancy_month(d, c) ≡
    D.months_since(last_insem_date(c), d)
pre is_pregnant(d, c),

milked : D.Date × Cow → Bool
milked(d, c) ≡
```

```
    if D.in_morning(d)
    then
        CH.event_in_period(
             CE.milking, D.since(D.last_midnight(d)),
             history(c))
    else
        CH.event_in_period(
             CE.milking, D.since(D.last_midday(d)),
             history(c))
    end,

can_milk_cow : D.Date × Cow → Bool
can_milk_cow(d, c) ≡
    is_milking_cow(c) ∧ ∼milked(d, c),

milk_cow : D.Date × GT.Litres × Cow →̃ Cow
milk_cow(d, lts, c) ≡
    chg_history(
         CH.add_event(d, CE.milking(lts), history(c)), c)
pre can_milk_cow(d, c),

end_lactation : D.Date × Cow → Bool
end_lactation(d, c) ≡
    is_milking_cow(c) ∧
    CH.in_history(CE.birth, history(c)) ∧
    D.months_since(last_birth_date(c), d) ≥
        K.lact_period,

current_weight : Cow →̃ GT.Weight
current_weight(c) ≡
    let
        weight =
            CE.weight(
                 CH.get_last_ev_info(CE.weigh, history(c)))
    in
        weight
    end
pre CH.in_history(CE.weigh, history(c)),

last_heat_date : Cow →̃ D.Date
last_heat_date(c) ≡
    CH.get_last_ev_date(CE.heat, history(c))
pre CH.in_history(CE.heat, history(c)),

first_birth_date : Cow →̃ D.Date
first_birth_date(c) ≡
```

```
    CH.date(
        CH.get_first_event(
              cow_births(D.since(birthday(c)), c)))
**pre** CH.in_history(CE.birth, history(c)),

last_birth_date : Cow $\stackrel{\sim}{\rightarrow}$ D.Date
last_birth_date(c) ≡
    CH.get_last_ev_date(CE.birth, history(c))
**pre** CH.in_history(CE.birth, history(c)),

last_insem_date : Cow $\stackrel{\sim}{\rightarrow}$ D.Date
last_insem_date(c) ≡
    CH.get_last_ev_date(CE.insemination, history(c))
**pre** CH.in_history(CE.insemination, history(c)),

is_dried : Cow → **Bool**
is_dried(c) ≡
    CH.event_in_period(
        CE.cow_dried, D.since(birthday(c)), history(c)),

dry_cow : D.Date × GT.Dried_cause × Cow $\stackrel{\sim}{\rightarrow}$ Cow
dry_cow(d, dc, c) ≡
    **let**
        new_c1 =
            chg_history(
                CH.add_event(d, CE.cow_dried(dc), history(c)),
                c),
        new_c2 =
            assign_group_to_cow(d, GT.dry_cow, new_c1)
    **in**
        set_discard_classif(d, new_c2)
    **end**
**pre** ∼is_dried(c),

can_weigh_cow : D.Date × Cow → **Bool**
can_weigh_cow(d, c) ≡
    ∼CH.event_in_period(
            CE.weigh, D.last_n_months(1, d), history(c)) ∨
    (is_discard_cow(c) ∨ ∼is_female_calf(c)) ∧
    ∼CH.event_in_period(
            CE.weigh, D.since(D.last_midnight(d)),
            history(c)),

weigh_cow : D.Date × GT.Weight × Cow $\stackrel{\sim}{\rightarrow}$ Cow
weigh_cow(d, w, c) ≡
    chg_history(
          CH.add_event(d, CE.weigh(w), history(c)), c)
    pre can_weigh_cow(d, c),

    can_deparasite_cow : D.Date × Cow → Bool
    can_deparasite_cow(d, c) ≡
      is_calf(c) ∧
      (is_in_group(GT.calf_rearing_unit, c) ∨
       ∼CH.event_in_period(
             CE.deparasitation,
             D.last_n_months(K.deparas_period, d),
             history(c))) ∨
      is_heifer(c) ∧ ∼is_pregnant(d, c) ∧
      ∼CH.event_in_period(
            CE.deparasitation,
            D.last_n_months(K.deparas_period, d), history(c)),

    deparasite_cow : D.Date × GT.Dep_inf × Cow →̃ Cow
    deparasite_cow(d, di, c) ≡
      chg_history(
          CH.add_event(d, CE.deparasitation(di), history(c)),
          c)
    pre can_deparasite_cow(d, c),

    correct_he_weight : Cow → Bool
    correct_he_weight(c) ≡
      current_weight(c) ≥
        K.he_weight_per ∗ K.dairy_weight,

    can_insem_cow : D.Date × Cow → Bool
    can_insem_cow(d, c) ≡
      (is_post_birth_cow(c) ∧
       len (cow_heats(D.since(last_birth_date(c)), c)) ≥ 1 ∨
       is_heifer(c) ∧ correct_he_weight(c)) ∧
      len (cow_inseminations(
                  D.last_n_months(K.insem_months, d), c)) <
        K.insem_limit ∧ is_on_heat(d, c),

    insem_cow_artif :
      D.Date × GT.Artif_info × Cow →̃ Cow
    insem_cow_artif(d, ai, c) ≡
      chg_history(
          CH.add_event(
                d, CE.insemination(GT.artif_insem(ai)),
                history(c)), c)
    pre can_insem_cow(d, c),
insem_cow_natural :
   D.Date × GT.Bull_id × Cow $\xrightarrow{\sim}$ Cow
insem_cow_natural(d, bi, c) ≡
   chg_history(
       CH.add_event(
            d, CE.insemination(GT.nat_insem(bi)),
            history(c)), c)
pre can_insem_cow(d, c),

can_be_pregnant_cow : D.Date × Cow → Bool
can_be_pregnant_cow(d, c) ≡
   (is_heifer(c) ∨ is_dairy_cow(c)) ∧
   CH.in_history(CE.heat, history(c)) ∧
   CH.event_in_period(
       CE.insemination, D.since(last_heat_date(c)),
       history(c)),

can_detect_pregnancy : D.Date × Cow → Bool
can_detect_pregnancy(d, c) ≡
   can_be_pregnant_cow(d, c) ∧
   CH.in_history(CE.insemination, history(c)) ∧
   D.days_since(
       d,
       CH.get_last_ev_date(CE.insemination, history(c))) ≥
      K.preg_detect_period,

detect_pregnant_cow : D.Date × Bool × Cow $\xrightarrow{\sim}$ Cow
detect_pregnant_cow(d, preg, c) ≡
   if preg
   then
      if is_dairy_cow(c)
      then
         let
            new_c1 =
               chg_history(
                   CH.add_event(
                        d, CE.preg_detection(true), history(c)),
                   c),
            new_c2 =
               assign_group_to_cow(d, GT.two, new_c1)
         in
            define_cow_classif(d, new_c2)
         end
      else
         chg_history(
             CH.add_event(
                    d, CE.preg_detection(true), history(c)), c)
        end
    else
        chg_history(
            CH.add_event(
                d, CE.preg_detection(false), history(c)), c)
    end
pre can_detect_pregnancy(d, c),

is_dead_cow : Cow → Bool
is_dead_cow(c) ≡
    CH.event_in_period(
        CE.death, D.since(birthday(c)), history(c)),

save_cow_death :
    D.Date × GT.Death_cause × Cow →̃ Cow
save_cow_death(d, dc, c) ≡
    chg_history(
        CH.add_event(d, CE.death(dc), history(c)), c)
pre ∼is_dead_cow(c),

vacc_received :
    GT.Vacc_type × D.Period × Cow → Bool
vacc_received(vt, p, c) ≡
    ∼(∃ ev : CH.Event •
            ev ∈
                CH.evs_in_period(
                    p, CH.filter(CE.is_vaccination)(history(c))) ∧
            let vacc = CE.vaccine(CH.event_inf(ev)) in
                GT.vacc_type(vacc) = vt
            end),

can_receive_vacc : D.Date × GT.Vaccine × Cow → Bool
can_receive_vacc(d, vacc, c) ≡
    case GT.vacc_type(vacc) of
        GT.triple →
            is_calf(c) ∧
            ∼vacc_received(
                GT.triple,
                D.last_n_months(K.triple_vacc_period, d), c) ∧
            (cow_age_months(d, c) = 3 ∨
             cow_age_months(d, c) = 6 ∨
             cow_age_months(d, c) = 9 ∨
             cow_age_months(d, c) = 12),
        GT.diarrhoea →
            is_heifer(c) ∧ is_pregnant(d, c) ∨
            is_dry_cow(c) ∧
            (pregnancy_month(d, c) = 7 ∧
             ~vacc_received(
                   GT.diarrhoea, D.last_n_months(1, d), c) ∨
             pregnancy_month(d, c) = K.pregnancy_period ∧
             ~vacc_received(
                   GT.diarrhoea, D.last_n_months(1, d), c)),
         GT.brucellosis →
            is_female_calf(c) ∧
            cow_age_months(d, c) ≥ 3 ∧
            cow_age_months(d, c) ≤ 10 ∧
            ~vacc_received(
                  GT.vacc_type(vacc), D.since(birthday(c)), c)
     end ∧ ~GT.expired(d, vacc) ∧
     ~CH.event_in_period(
            CE.vaccination, D.since(D.last_midnight(d)),
            history(c)),

vaccinate_cow : D.Date × GT.Vaccine × Cow ~→ Cow
vaccinate_cow(d, vacc, c) ≡
    chg_history(
        CH.add_event(d, CE.vaccination(vacc), history(c)),
        c)
pre can_receive_vacc(d, vacc, c),

can_sell_cow : D.Date × Cow → Bool
can_sell_cow(d, c) ≡
    (~is_female_calf(c) ∨ is_discard_cow(c)) ∧
    CH.event_in_period(
        CE.weigh, D.since(D.last_midnight(d)), history(c)) ∧
    ~CH.event_in_period(
            CE.cow_sale, D.since(birthday(c)), history(c)),

sell_cow : D.Date × Cow ~→ Cow
sell_cow(d, c) ≡
    chg_history(
        CH.add_event(d, CE.cow_sale, history(c)), c)
pre can_sell_cow(d, c),

next_birth_date : D.Date × Cow ~→ D.Date
next_birth_date(d, c) ≡
    D.add_n_months(
        last_insem_date(c), K.pregnancy_period)
pre can_give_birth(d, c),

sum_litres : CH.History → GT.Litres
```

```
sum_litres(h) ≡
   if h = ⟨⟩ then 0.0
   else
      let lts = CE.litres(CH.event_inf(hd h)) in
         lts + sum_litres(tl h)
      end
   end,

milk_in_period : D.Period × Cow → GT.Litres
milk_in_period(p, c) ≡
   sum_litres(
       CH.evs_in_period(
            p, CH.filter(CE.is_milking)(history(c)))),

has_produced_milk : D.Period × Cow → Bool
has_produced_milk(p, c) ≡
   CH.event_in_period(CE.milking, p, history(c)),

number_milkings_in_period : D.Period × Cow → Nat
number_milkings_in_period(p, c) ≡
   len (cow_milkings(p, c)),

cow_indiv_prod : D.Period × Cow ~→ GT.Indiv_prod
cow_indiv_prod(p, c) ≡
   milk_in_period(p, c) /
   real (number_milkings_in_period(p, c))
pre has_produced_milk(p, c),

is_in_group : GT.Group_id × Cow → Bool
is_in_group(gt, c) ≡ gt = cow_group(c),

can_goto_group : D.Date × GT.Group_id × Cow → Bool
can_goto_group(d, gt, c) ≡
   if
      CH.in_history(CE.cow_to_group, history(c)) ∧
      gt =
         CE.group(
            CH.get_last_ev_info(
                 CE.cow_to_group, history(c)))
   then false
   else true
   end,

assign_group_to_cow :
   D.Date × GT.Group_id × Cow ~→ Cow
assign_group_to_cow(d, gt, c) ≡
```

```
    chg_history(
        CH.add_event(d, CE.cow_to_group(gt), history(c)),
        c)
pre can_goto_group(d, gt, c),

can_be_in_group : Cow → Bool
can_be_in_group(c) ≡
    ∼is_calf(c) ∨
    is_calf(c) ∧ calf_location(c) ≠ GT.with_mother,

cow_group : Cow ~→ GT.Group_id
cow_group(c) ≡
    CE.group(
        CH.get_last_ev_info(CE.cow_to_group, history(c)))
pre can_be_in_group(c),

can_be_in_one : D.Date × Cow → Bool
can_be_in_one(d, c) ≡
    is_post_birth_cow(c) ∧
    D.is_in_period(
        last_birth_date(c), D.last_n_days(7, d)),

can_be_in_two : D.Date × Cow → Bool
can_be_in_two(d, c) ≡
    is_in_group(GT.one, c) ∧
    D.months_since(last_birth_date(c), d) ≥ 3 ∨
    is_dairy_cow(c) ∧ is_pregnant(d, c) ∧
    pregnancy_month(d, c) < 7,

can_be_in_pre_birth : D.Date × Cow → Bool
can_be_in_pre_birth(d, c) ≡
    (is_dry_cow(c) ∨ is_heifer(c) ∧ is_pregnant(d, c)) ∧
    D.is_in_range(
        15, 20, D.days_since(next_birth_date(d, c), d)),

can_be_in_dry : D.Date × Cow → Bool
can_be_in_dry(d, c) ≡
    is_early_pregnant_cow(c) ∧
    pregnancy_month(d, c) ≥ 7,

can_be_in_discard : Cow → Bool
can_be_in_discard(c) ≡ is_discard_cow(c),

select_group_for_cow : D.Date × Cow ~→ GT.Group_id
select_group_for_cow(d, c) ≡
    if can_be_in_one(d, c) then GT.one
    elsif can_be_in_two(d, c) then GT.two
    elsif can_be_in_pre_birth(d, c)
        then GT.pre_birth_cow
    elsif can_be_in_dry(d, c) then GT.dry_cow
    elsif can_be_in_discard(c) then GT.discard_cow
    else GT.heifer
    end
pre is_heifer(c) $\vee$ is_dairy_cow(c),

define_cow_classif : D.Date $\times$ Cow $\rightarrow$ Cow
define_cow_classif(d, c) $\equiv$
    if is_female_calf(c) $\wedge$ cow_age_months(d, c) $\geq$ 12
    then set_heifer_classif(d, c)
    elsif
        (is_heifer(c) $\vee$ is_dairy_cow(c)) $\wedge$
        CH.in_history(CE.birth, history(c)) $\wedge$
        D.days_since(d, last_birth_date(c)) $\leq$ 7
        then set_post_birth_classif(d, c)
    elsif
        is_pregnant(d, c) $\wedge$ is_post_birth_cow(c) $\wedge$
        D.months_since(last_birth_date(c), d) $\geq$ 3
        then set_early_pregnant_classif(d, c)
    elsif is_pregnant(d, c) $\wedge$ pregnancy_month(d, c) = 7
        then set_dry_classif(d, c)
    elsif
        is_dry_cow(c) $\wedge$
        D.is_in_range(
             15, 20, D.days_since(next_birth_date(d, c), d))
        then set_pre_birth_classif(d, c)
    elsif
        is_post_birth_cow(c) $\wedge$ $\sim$is_pregnant(d, c) $\wedge$
        len (cow_inseminations(
                        D.since(last_birth_date(c)), c)) =
            K.insem_limit
        then set_empty_classif(d, c)
    elsif is_dried(c) then set_discard_classif(d, c)
    else c
    end,

set_heifer_classif : D.Date $\times$ Cow $\overset{\sim}{\rightarrow}$ Cow
set_heifer_classif(d, c) $\equiv$
    chg_classif(GT.heifer(default_heifer_info), c)
pre is_female_calf(c) $\wedge$ cow_age_months(d, c) $\geq$ 12,

set_post_birth_classif : D.Date $\times$ Cow $\overset{\sim}{\rightarrow}$ Cow
set_post_birth_classif(d, c) $\equiv$
```

```
        if is_heifer(c)
        then
            chg_classif(
                GT.dairy(
                    GT.mk_Dairy_info(GT.milking(GT.post_birth))),
                c)
        else
            let GT.dairy(d_inf) = cow_classif(c) in
                chg_classif(
                    GT.dairy(
                        GT.chg_dairy_classif(
                            GT.milking(GT.post_birth), d_inf)), c)
            end
        end
    pre
        (is_heifer(c) ∨ is_dairy_cow(c)) ∧
        CH.in_history(CE.birth, history(c)) ∧
        D.days_since(d, last_birth_date(c)) ≤ 7,

    set_early_pregnant_classif : D.Date × Cow ~→ Cow
    set_early_pregnant_classif(d, c) ≡
        let GT.dairy(d_inf) = cow_classif(c) in
            chg_classif(
                GT.dairy(
                    GT.chg_dairy_classif(
                        GT.milking(GT.early_pregnant), d_inf)), c)
        end
    pre
        is_pregnant(d, c) ∧ is_post_birth_cow(c) ∧
        D.months_since(last_birth_date(c), d) ≥ 3,

    set_dry_classif : D.Date × Cow ~→ Cow
    set_dry_classif(d, c) ≡
        let GT.dairy(d_inf) = cow_classif(c) in
            chg_classif(
                GT.dairy(
                    GT.chg_dairy_classif(
                        GT.dry(GT.non_pre_birth), d_inf)), c)
        end
    pre is_pregnant(d, c) ∧ pregnancy_month(d, c) = 7,

    set_pre_birth_classif : D.Date × Cow ~→ Cow
    set_pre_birth_classif(d, c) ≡
        let GT.dairy(d_inf) = cow_classif(c) in
            chg_classif(
                GT.dairy(
                    GT.chg_dairy_classif(
                        GT.dry(GT.pre_birth), d_inf)), c)
    end
pre
    is_dry_cow(c) ∧
    D.is_in_range(
        15, 20, D.days_since(next_birth_date(d, c), d)),

set_empty_classif : D.Date × Cow →~ Cow
set_empty_classif(d, c) ≡
    let GT.dairy(d_inf) = cow_classif(c) in
        chg_classif(
            GT.dairy(
                GT.chg_dairy_classif(
                    GT.milking(GT.empty), d_inf)), c)
    end
pre
    is_post_birth_cow(c) ∧ ∼is_pregnant(d, c) ∧
    len (cow_inseminations(
                D.since(last_birth_date(c)), c)) =
        K.insem_limit,

set_discard_classif : D.Date × Cow →~ Cow
set_discard_classif(d, c) ≡
    let GT.dairy(d_inf) = cow_classif(c) in
        chg_classif(
            GT.dairy(
                GT.chg_dairy_classif(GT.discard, d_inf)), c)
    end
pre is_dried(c),

is_with_mother : D.Date × Cow → Bool
is_with_mother(d, c) ≡
    is_calf(c) ∧
    CH.in_history(CE.calf_with_mother, history(c)) ∧
    ∼CH.in_history(CE.calf_to_cru, history(c)),

can_goto_cru : D.Date × Cow → Bool
can_goto_cru(d, c) ≡
    is_calf(c) ∧ cow_age_days(d, c) ≥ 5 ∧
    is_with_mother(d, c) ∧
    ∼CH.event_in_period(
        CE.calf_to_cru, D.since(birthday(c)), history(c)),

add_photo : D.Date × Cow →~ Cow
/∗ dummy value for now ∗/
```

```
add_photo(d, c) ≡ c,

send_calf_to_cru : D.Date × Cow $\xrightarrow{\sim}$ Cow
send_calf_to_cru(d, c) ≡
   if is_female_calf(c)
   then
      chg_history(
          CH.add_event(d, CE.calf_to_cru, history(c)),
          add_photo(d, c))
   else
      chg_history(
          CH.add_event(d, CE.calf_to_cru, history(c)), c)
   end
pre can_goto_cru(d, c),

can_take_out_cru : D.Date × Cow → Bool
can_take_out_cru(d, c) ≡
   is_calf(c) $\wedge$
   is_in_group(GT.calf_rearing_unit, c) $\wedge$
   cow_age_days(d, c) $\geq$ K.min_age_out_cru $\wedge$
   cow_age_days(d, c) $\leq$ K.max_age_out_cru $\wedge$
   can_eat_bal(d, c),

can_breed_artif :
   D.Date × GT.Litres × GT.Quantity × Cow → Bool
can_breed_artif(d, mr, bal, c) ≡
   is_calf(c) $\wedge$
   is_in_group(GT.calf_rearing_unit, c) $\wedge$
   mr $\geq$ K.min_milk_cru $\wedge$ mr $\leq$ K.max_milk_cru $\wedge$
   bal $\leq$ 1.0 $\wedge$
   ~CH.event_in_period(
         CE.artif_breeding, D.since(D.last_midnight(d)),
         history(c)),

breed_artif :
   D.Date × GT.Litres × GT.Quantity × Cow $\xrightarrow{\sim}$ Cow
breed_artif(d, mr, bal, c) ≡
   chg_history(
       CH.add_event(
            d, CE.artif_breeding(mr, bal), history(c)), c)
pre can_breed_artif(d, mr, bal, c),

can_eat_bal : D.Date × Cow → Bool
can_eat_bal(d, c) ≡
   is_calf(c) $\wedge$
   is_in_group(GT.calf_rearing_unit, c) $\wedge$
```

```
    CH.in_history(CE.artif_breeding, history(c)) ∧
    let
        b =
            CE.bal(
                CH.get_last_ev_info(
                    CE.artif_breeding, history(c)))
    in
        b ≥ 1.0
    end,

can_register_heat : D.Date × Bool × Cow → Bool
can_register_heat(d, heat, c) ≡
    heat ∧ ∼is_on_heat(d, c),

register_heat : D.Date × Bool × Cow ~→ Cow
register_heat(d, heat, c) ≡
    chg_history(CH.add_event(d, CE.heat, history(c)), c)
pre can_register_heat(d, heat, c),

can_give_birth : D.Date × Cow → Bool
can_give_birth(d, c) ≡
    is_pregnant(d, c) ∧
    pregnancy_month(d, c) = K.pregnancy_period ∧
    ∼CH.event_in_period(
            CE.birth,
            D.last_n_months(K.pregnancy_period, d),
            history(c)),

give_birth :
    D.Date × GT.Cow_id × GT.Calf_sex × Cow ~→ Cow
give_birth(d, calf_id, csex, c) ≡
    let
        new_c1 =
            chg_history(
                CH.add_event(d, CE.birth(calf_id), history(c)),
                c),
        new_c2 = assign_group_to_cow(d, GT.one, new_c1)
    in
        set_post_birth_classif(d, new_c2)
    end
pre can_give_birth(d, c),

make_calf : GT.Calf_sex × D.Date → Cow
make_calf(csex, d) ≡
    mk_Cow(
        d, GT.calf(GT.mk_Calf_info(csex)),
        CH.add_event(d, CE.calf_with_mother, CH.empty)),

check_weight : GT.Weight × GT.Weight → Bool
/* dummy value for now */
check_weight(w1, w2) ≡ true,

cow_events :
   D.Period × CE.Cow_event_kind × Cow → CH.History
cow_events(p, ev_kind, c) ≡
   case ev_kind of
      CE.birth → cow_births(p, c),
      CE.heat → cow_heats(p, c),
      CE.preg_detection → cow_pregnancies(p, c),
      CE.insemination → cow_inseminations(p, c),
      CE.vaccination → cow_vaccinations(p, c),
      CE.deparasitation → cow_deparasitations(p, c),
      CE.weigh → cow_weighs(p, c),
      CE.cow_to_group → cow_groups(p, c),
      CE.milking → cow_milkings(p, c),
      CE.artif_breeding → cow_artif_breedings(p, c),
      _ → ⟨⟩
   end,

cow_weighs : D.Period × Cow → CH.History
cow_weighs(p, c) ≡
   CH.evs_in_period(
        p, CH.filter(CE.is_weigh)(history(c))),

cow_vaccinations : D.Period × Cow → CH.History
cow_vaccinations(p, c) ≡
   CH.evs_in_period(
        p, CH.filter(CE.is_vaccination)(history(c))),

cow_deparasitations : D.Period × Cow → CH.History
cow_deparasitations(p, c) ≡
   CH.evs_in_period(
        p, CH.filter(CE.is_deparasitation)(history(c))),

cow_births : D.Period × Cow → CH.History
cow_births(p, c) ≡
   CH.evs_in_period(
        p, CH.filter(CE.is_birth)(history(c))),

cow_inseminations : D.Period × Cow → CH.History
cow_inseminations(p, c) ≡
   CH.evs_in_period(
        p, CH.filter(CE.is_insemination)(history(c))),

cow_heats : D.Period × Cow → CH.History
cow_heats(p, c) ≡
    CH.evs_in_period(
        p, CH.filter(CE.is_on_heat)(history(c))),

cow_milkings : D.Period × Cow → CH.History
cow_milkings(p, c) ≡
    CH.evs_in_period(
        p, CH.filter(CE.is_milking)(history(c))),

cow_pregnancies : D.Period × Cow → CH.History
cow_pregnancies(p, c) ≡
    CH.evs_in_period(
        p, CH.filter(CE.is_preg_detection)(history(c))),

cow_groups : D.Period × Cow → CH.History
cow_groups(p, c) ≡
    CH.evs_in_period(
        p, CH.filter(CE.is_cow_to_group)(history(c))),

cow_artif_breedings : D.Period × Cow → CH.History
cow_artif_breedings(p, c) ≡
    CH.evs_in_period(
        p, CH.filter(CE.is_artif_breeding)(history(c))),

in_lactation_period : D.Date × Cow → Bool
in_lactation_period(d, c) ≡ is_milking_cow(c),

dried_date : Cow ~→ D.Date
dried_date(c) ≡
    CH.get_last_ev_date(CE.cow_dried, history(c))
pre is_dried(c),

is_mother : GT.Cow_id × Cow × Cow → Bool
is_mother(ci, c, cm) ≡
    is_dairy_cow(cm) ∧
    (∃ ev : CH.Event •
        ev ∈ cow_births(D.since(birthday(c)), cm) ∧
        let cf_id = CE.calf_id(CH.event_inf(ev)) in
            cf_id = ci
        end),

cow_history_p : D.Period × Cow → CH.History
cow_history_p(p, c) ≡ CH.evs_in_period(p, history(c))
```

```
end
```

C.9 CH Module

context: HISTORY, CE

```
object CH :
  HISTORY(
    CE{Cow_event for Event_info,
        Cow_event_kind for Event_kind})
```

C.10 CE Module

context: COW_EVENT

```
object CE : COW_EVENT
```

C.11 COW_EVENT Module

context: K

```
scheme COW_EVENT =
  class
    type
      Cow_event ==
        birth(calf_id : GT.Cow_id) |
        heat |
        preg_detection(pregnant : Bool) |
        insemination(insem_classif : GT.Insem_classif) |
        death(cause : GT.Death_cause) |
        vaccination(vaccine : GT.Vaccine) |
        deparasitation(dep_inf : GT.Dep_inf) |
        weigh(weight : GT.Weight) |
        cow_to_group(group : GT.Group_id) |
        calf_to_cru |
        calf_with_mother |
        cow_sale |
        cow_dried(d_cause : GT.Dried_cause) |
        milking(litres : GT.Litres) |
        artif_breeding(
          milk_repl : GT.Litres, bal : GT.Quantity),
      Cow_event_kind ==
        birth |
        heat |
        preg_detection |
        insemination |
        death |
        vaccination |
        deparasitation |
        weigh |
        cow_to_group |
        calf_to_cru |
        calf_with_mother |
        cow_sale |
        cow_dried |
        milking |
        artif_breeding

value
    kind_of : Cow_event → Cow_event_kind
    kind_of(cev) ≡
        case cev of
            birth(_) → birth,
            heat → heat,
            preg_detection(_) → preg_detection,
            insemination(_) → insemination,
            death(_) → death,
            vaccination(_) → vaccination,
            deparasitation(_) → deparasitation,
            weigh(_) → weigh,
            cow_to_group(_) → cow_to_group,
            calf_to_cru → calf_to_cru,
            calf_with_mother → calf_with_mother,
            cow_sale → cow_sale,
            cow_dried(_) → cow_dried,
            milking(_) → milking,
            artif_breeding(_, _) → artif_breeding
        end,

    is_vaccination : Cow_event → Bool
    is_vaccination(cev) ≡
        case kind_of(cev) of
            vaccination → true,
            _ → false
        end,

    is_insemination : Cow_event → Bool
    is_insemination(cev) ≡
    case kind_of(cev) of
        insemination → true,
        _ → false
    end,

is_on_heat : Cow_event → Bool
is_on_heat(cev) ≡
    case kind_of(cev) of
        heat → true,
        _ → false
    end,

is_deparasitation : Cow_event → Bool
is_deparasitation(cev) ≡
    case kind_of(cev) of
        deparasitation → true,
        _ → false
    end,

is_preg_detection : Cow_event → Bool
is_preg_detection(cev) ≡
    case kind_of(cev) of
        preg_detection → true,
        _ → false
    end,

is_milking : Cow_event → Bool
is_milking(cev) ≡
    case kind_of(cev) of
        milking → true,
        _ → false
    end,

is_weigh : Cow_event → Bool
is_weigh(cev) ≡
    case kind_of(cev) of
        weigh → true,
        _ → false
    end,

is_birth : Cow_event → Bool
is_birth(cev) ≡
    case kind_of(cev) of
        birth → true,
        _ → false
    end,
      is_cow_to_group : Cow_event → Bool
      is_cow_to_group(cev) ≡
        case kind_of(cev) of
          cow_to_group → true,
          _ → false
        end,

      is_artif_breeding : Cow_event → Bool
      is_artif_breeding(cev) ≡
        case kind_of(cev) of
          artif_breeding → true,
          _ → false
        end
  end
```

C.12 FIELDS Module

```
context: FIELD

scheme FIELDS =
  class
    object F : FIELD

    type Fields = GT.Field_id $\overrightarrow{m}$ F.Field

    value
      add_field :
        GT.Field_id × GT.Location × GT.Size ×
        GT.Pasture × Fields $\xrightarrow{\sim}$
          Fields
      add_field(fi, floc, fsize, fpast, fs) ≡
        fs † [ fi ↦ F.make_field(floc, fsize, fpast) ]
      pre fi ∉ fs,

      exists_plot :
        GT.Plot_id × GT.Field_id × Fields → Bool
      exists_plot(pi, fi, fs) ≡
        fi ∈ fs ∧ F.exists_plot(pi, fs(fi)),

      add_plot :
        GT.Plot_id × GT.Size × GT.Location × D.Date ×
        Nat × GT.Field_id × Fields $\xrightarrow{\sim}$
          Fields
      add_plot(pi, si, lo, sd, dn, fi, fs) ≡
```

```
            fs † [ fi ↦ F.add_plot(pi, si, lo, sd, dn, fs(fi)) ]
        pre ∼exists_plot(pi, fi, fs),

        update_past :
            GT.Field_id × GT.Pasture × Fields →̃ Fields
        update_past(fi, fpast, fs) ≡
            let new_past = F.chg_pasture(fpast, fs(fi)) in
                fs † [ fi ↦ new_past ]
            end
        pre fi ∉ fs,

        is_defined :
            GT.Plot_id × GT.Field_id × D.Date × Nat × Fields →
                Bool
        is_defined(pi, fi, d, dn, fs) ≡
            fi ∈ fs ∧ F.is_defined(fs(fi), pi, d, dn),

        field_size : GT.Field_id × Fields →̃ GT.Size
        field_size(fi, fs) ≡ F.size(fs(fi)) pre fi ∈ fs,

        is_past_plot :
            GT.Plot_id × GT.Field_id × Fields → Bool
        is_past_plot(pi, fi, fs) ≡
            fi ∈ fs ∧ pi ∈ F.past_plots(fs(fi))
  end
```

C.13 FIELD Module

```
context: PLOTS

scheme FIELD =
  class
    object PS : PLOTS

    type
      Field ::
        field_location : GT.Location
        size : GT.Size
        pasture : GT.Pasture ↔ chg_pasture
        plots : PS.Plots ↔ chg_plots
        past_plots : PS.Plots ↔ chg_past_plots

    value
      make_field :
        GT.Location × GT.Size × GT.Pasture → Field
      make_field(floc, fsize, fpast) ≡
         mk_Field(floc, fsize, fpast, PS.empty, PS.empty),

      exists_plot : GT.Plot_id × Field → Bool
      exists_plot(pi, f) ≡ pi ∈ plots(f),

      is_defined :
         Field × GT.Plot_id × D.Date × Nat → Bool
      is_defined(f, pi, d, dn) ≡
         exists_plot(pi, f) ∧
         PS.is_defined(plots(f), pi, d, dn),

      add_plot :
         GT.Plot_id × GT.Size × GT.Location × D.Date ×
         Nat × Field ~→
            Field
      add_plot(pi, si, lo, sd, dn, f) ≡
         chg_plots(
             PS.add_plot(pi, si, lo, sd, dn, plots(f)), f)
      pre pi ∉ plots(f),

      delete_plot : GT.Plot_id × Field ~→ Field
      delete_plot(pi, f) ≡
         let
            (new_pps, new_ps) =
               PS.delete_plot(pi, plots(f), past_plots(f))
         in
            chg_plots(new_ps, chg_past_plots(new_pps, f))
         end
      pre pi ∈ plots(f) ∧ pi ∉ past_plots(f)
   end
```

C.14 PLOTS Module

```
context: PLOT

scheme PLOTS =
   class
      object P : PLOT

      type Plots = GT.Plot_id →m P.Plot

      value
         empty : Plots = [ ],
        add_plot :
            GT.Plot_id × GT.Size × GT.Location × D.Date ×
            Nat × Plots $\xrightarrow{\sim}$
                Plots
        add_plot(pi, lo, si, sd, dn, ps) ≡
            ps † [ pi ↦ P.mk_Plot(si, lo, sd, dn) ]
        pre pi ∉ ps,

        is_defined :
            Plots × GT.Plot_id × D.Date × Nat → Bool
        is_defined(ps, pi, d, dn) ≡
            pi ∈ ps ∧ P.is_defined(ps(pi), d, dn),

        delete_plot :
            GT.Plot_id × Plots × Plots $\xrightarrow{\sim}$ Plots × Plots
        delete_plot(pi, ps, pps) ≡
            (pps † [ pi ↦ ps(pi) ], ps † ps \ {pi})
        pre pi ∈ ps ∧ pi ∉ pps
    end
```

C.15 PLOT Module

```
context: GT

scheme PLOT =
    class
        type
            Plot ::
                plot_location : GT.Location
                size : GT.Size
                starting : D.Date
                days : Nat ↔ chg_days

        value
            is_defined : Plot × D.Date × Nat → Bool
            is_defined(pl, d, dn) ≡
                D.later(d, starting(pl)) ∧
                D.later(
                    D.add_n_days(starting(pl), days(pl)),
                    D.add_n_days(d, dn))
    end
```

C.16 BULLS Module

```
context: BULL

scheme BULLS =
   class
      object B : BULL

      type Bulls = GT.Bull_id $\overrightarrow{m}$ B.Bull

      value
         add_bull :
            GT.Bull_id × D.Date × D.Date × GT.Field_id ×
            GT.Features × Bulls $\tilde{\rightarrow}$
               Bulls
         add_bull(bi, birthd, pd, fi, bf, bs) ≡
            bs †
            [ bi ↦ B.mk_Bull(birthd, pd, fi, bf, B.current) ]
         pre bi ∉ bs,

         update_location :
            GT.Bull_id × GT.Field_id × Bulls $\tilde{\rightarrow}$ Bulls
         update_location(bi, fi, bs) ≡
            let new_loc = B.chg_location(fi, bs(bi)) in
               bs † [ bi ↦ new_loc ]
            end
         pre bi ∈ bs,

         update_features :
            GT.Bull_id × GT.Features × Bulls $\tilde{\rightarrow}$ Bulls
         update_features(bi, bf, bs) ≡
            let new_fe = B.chg_features(bf, bs(bi)) in
               bs † [ bi ↦ new_fe ]
            end
         pre bi ∈ bs,

         can_discard_bull : GT.Bull_id × Bulls → Bool
         can_discard_bull(bi, bs) ≡
            bi ∈ bs ∧ ∼B.is_discarded_bull(bs(bi)),

         discard_bull :
            GT.Bull_id × D.Date × GT.Discard_cause × Bulls $\tilde{\rightarrow}$
               Bulls
         discard_bull(bi, d, dc, bs) ≡
            bs † [ bi ↦ B.discard_bull(d, dc, bs(bi)) ]
         pre can_discard_bull(bi, bs)
```

```
  end
```

C.17 BULL Module

```
context: GT

scheme BULL =
  class
    type
      Bull_status ==
        current |
        discarded(date : D.Date, cause : GT.Discard_cause),
      Bull ::
        birthday : D.Date
        purchase_date : D.Date
        location : GT.Field_id ↔ chg_location
        features : GT.Features ↔ chg_features
        status : Bull_status ↔ chg_status

    value
      is_discarded_bull : Bull → Bool
      is_discarded_bull(b) ≡
        case status(b) of
          discarded(_, _) → true,
          _ → false
        end,

      discard_bull :
        D.Date × GT.Discard_cause × Bull ~→ Bull
      discard_bull(d, dc, b) ≡
        chg_status(discarded(d, dc), b)
      pre ∼is_discarded_bull(b)
  end
```

C.18 DAIRY_FARMERS Module

```
context: DAIRY_FARMER

scheme DAIRY_FARMERS =
  class
    object DF : DAIRY_FARMER

    type
```

```
      Dairy_farmers =
         GT.Dairy_farmer_id $\overrightarrow{m}$ DF.Dairy_farmer

   value
      add_dfarmer :
         GT.Dairy_farmer_id × GT.Salary × Dairy_farmers $\xrightarrow{\sim}$
            Dairy_farmers
      add_dfarmer(dfi, sal, dfs) ≡
         dfs † [ dfi ↦ DF.mk_Dairy_farmer(sal, {}) ]
      pre dfi ∉ dfs,

      update_salary :
         GT.Dairy_farmer_id × GT.Salary × Dairy_farmers $\xrightarrow{\sim}$
            Dairy_farmers
      update_salary(dfi, sal, dfs) ≡
         let new_sal = DF.chg_salary(sal, dfs(dfi)) in
            dfs † [ dfi ↦ new_sal ]
         end
      pre dfi ∈ dfs,

      add_empl :
         GT.Dairy_farmer_id × GT.Employee × Dairy_farmers $\xrightarrow{\sim}$
            Dairy_farmers
      add_empl(dfi, empl, dfs) ≡
         dfs † [ dfi ↦ DF.add_empl(empl, dfs(dfi)) ]
      pre dfi ∈ dfs ∧ empl ∉ DF.employees(dfs(dfi)),

      delete_empl :
         GT.Dairy_farmer_id × GT.Employee × Dairy_farmers $\xrightarrow{\sim}$
            Dairy_farmers
      delete_empl(dfi, empl, dfs) ≡
         dfs † [ dfi ↦ DF.delete_empl(empl, dfs(dfi)) ]
      pre dfi ∈ dfs ∧ empl ∈ DF.employees(dfs(dfi)),

      delete_dfarmer :
         GT.Dairy_farmer_id × Dairy_farmers $\xrightarrow{\sim}$
            Dairy_farmers
      delete_dfarmer(dfi, dfs) ≡ dfs † dfs \ {dfi}
      pre dfi ∈ dfs
end
```

C.19 DAIRY_FARMER Module

context: GT

```
scheme DAIRY_FARMER =
  class
    type
      Dairy_farmer ::
        salary : GT.Salary ↔ chg_salary
        employees : GT.Employee-set ↔ chg_employee

    value
      add_empl :
        GT.Employee × Dairy_farmer ⥲ Dairy_farmer
      add_empl(empl, df) ≡
        chg_employee({empl} ∪ employees(df), df)
      pre empl ∉ employees(df),

      delete_empl :
        GT.Employee × Dairy_farmer ⥲ Dairy_farmer
      delete_empl(empl, df) ≡
        chg_employee(employees(df) \ {empl}, df)
      pre empl ∈ employees(df)
  end
```

C.20 K Module

context: CONSTANTS

object K : CONSTANTS

C.21 CONSTANTS Module

context: GT

```
scheme CONSTANTS =
  class
    type
      Hours_tolerance = {| ht : Nat • ht < D.hours_per_day
          |}

    value
      calf_weight_atbirth : GT.Weight = 40.0,
      calf_weight_2months : GT.Weight = 60.0,
      heifer_weight : GT.Weight = 350.0,
      dairy_weight : GT.Weight = 550.0,
      discard_weight : GT.Weight = 580.0,
        calves_age_dif : Nat = 60,
        calf_min_age_ration : Nat = 60,
        calf_max_age_ration : Nat = 120,
        calf_middle_age : Nat = 180,
        he_weight_per : Real = 0.64,
        weight_variation : Real = 0.10,
        pregnancy_period : Nat = 9,/*in months*/
        lact_period : Nat = 7,/*in months*/
        heat_period : Nat = 12,/*in hours*/
        preg_detect_period : Nat = 60,/*in days*/
        discard_age : Nat = 4,/*in years*/
        insem_limit : Nat = 4,
        insem_months : Nat = 3,/*in months*/
        triple_vacc_period : Nat = 3,/*in months*/
        deparas_period : Nat = 2,/*in months*/
        post_birth_period : Nat = 3,/*in months*/
        min_age_cru : Nat = 5,/*in days*/
        min_age_out_cru : Nat = 45,/*in days*/
        max_age_out_cru : Nat = 60,/*in days*/
        max_balanced : Real = 1.0,
        min_milk_cru : Real = 4.0,
        max_milk_cru : Real = 5.0,
        ration_one : Real = 3.5,
        ration_two : Real = 3.0,
        ration_dry : Real = 2.0,
        ration_discard : Real = 2.0,
        ration_heifer : Real = 3.0,
        ration_calf_min : Real = 2.2,
        ration_calf_max : Real = 2.5,
        min_corn_one : Real = 25.0,
        max_corn_one : Real = 30.0,
        min_hay_one : Real = 10.0,
        max_hay_one : Real = 15.0,
        min_conc_one : Real = 30.0,
        max_conc_one : Real = 35.0,
        min_corn_two : Real = 25.0,
        max_corn_two : Real = 30.0,
        min_hay_two : Real = 10.0,
        max_hay_two : Real = 15.0,
        min_conc_two : Real = 15.0,
        max_conc_two : Real = 20.0,
        min_bal_calf : Real = 35.0,
        max_bal_calf : Real = 45.0,
        day_milkings : Nat,
        hours_tolerance : Hours_tolerance
    end
```

C.22 GT Module

```
context: GENERAL_TYPES

object GT : GENERAL_TYPES
```

C.23 GENERAL_TYPES Module

```
context: D

scheme GENERAL_TYPES =
  class
    type
      Cow_classif ==
        calf(info : Calf_info) |
        heifer(info : Heifer_info) |
        dairy(info : Dairy_info),
      Calf_info :: calf_sex : Calf_sex ↔ chg_sex,
      Heifer_info :: h_info : Text, /*reserved for later development*/
      Dairy_info ::
        dairy_classif : Dairy_classif ↔ chg_dairy_classif,
      Dairy_classif ==
        discard |
        dry(dry_classif : Dry_classif) |
        milking(milking_classif : Milking_classif),
      Dry_classif == pre_birth | non_pre_birth,
      Milking_classif == post_birth | early_pregnant | empty,
      Calf_location ==
        with_mother | with_group(group : Group_id),
      Calf_sex == male_calf | female_calf(photo : Photo),
      Cow_id = Int,
      Group_id ==
        one |
        two |
        pre_birth_cow |
        discard_cow |
        dry_cow |
        heifer |
        calf(min_days : Nat, max_days : Nat) |
        calf_rearing_unit,
      Quantity = Real,
      Hect_loading = Real,
      Weight = {| w : Real • w ≤ 700.0 |},
      Litres = {| l : Real • l ≥ 0.0 |},
      Indiv_prod = Litres,
Field_id = **Nat**,
Photo = **Text**, /*reserved for later development*/
Plot_id = **Nat**,
Bull_id = **Nat**,
Dairy_farmer_id = **Nat**,
Salary = **Nat**,
Employee = **Text**, /*reserved for later development*/
Brand = **Text**, /*reserved for later development*/
Features = **Text**, /*reserved for later development*/
Size = **Real**,
Location = **Text**, /*reserved for later development*/
Death_cause = **Text**,
Dried_cause = **Text**,
Discard_cause = **Text**,
Substance = **Text**, /*reserved for later development*/
Dose = **Nat**,
Vaccine_id = **Nat**,
In_plot :: field_id : Field_id   plot_id : Plot_id,
Dep_inf :: substance : Substance   dose : Dose,
Vacc_type == triple | diarrhoea | brucellosis,
Vaccine ::
   vacc_id : Vaccine_id
   vacc_type : Vacc_type
   expiration_date : D.Date,
Artif_info = **Nat**,
Insem_classif ==
   artif_insem(info : Artif_info) |
   nat_insem(bull : Bull_id),
Gr_type = **Text**,
Balanced = Quantity,
Hay = Quantity,
Corn_sil = Quantity,
Pasture = Quantity,
Grain ::
   gr_type : Gr_type ↔ chg_gr_type
   gr_quantity : Quantity ↔ chg_gr_quantity,
Conc ::
   balanced : Balanced ↔ chg_bal_quantity
   grain : Grain ↔ chg_grain,
Ration ::
   pasture : Pasture ↔ chg_pasture
   corn_sil : Corn_sil ↔ chg_corn
   hay : Hay ↔ chg_hay
   conc : Conc ↔ chg_conc
```

value

```
      expired : D.Date × Vaccine → Bool
      expired(d, vacc) ≡ D.later(expiration_date(vacc), d),

      total_foods : Ration → Quantity
      total_foods(r) ≡
        corn_sil(r) + hay(r) + balanced(conc(r)) +
        gr_quantity(grain(conc(r)))
  end
```

C.24 HISTORY Module

context: D, EVENT_INFO

```
scheme HISTORY(E : EVENT_INFO) =
  class
    type
      Event :: date : D.Date   event_inf : E.Event_info,
      /∗ the latest event is at the front ∗/
      History = {| h : Event* • is_ordered(h) |}

    value
      is_ordered : Event* → Bool
      is_ordered(l) ≡
        (∀ id1 : Int •
            id1 ∈ inds l ⇒
              (∀ id2 : Int •
                  id2 ∈ inds l ⇒
                    id1 < id2 ⇒
                      ∼D.later(date(l(id2)), date(l(id1))))),

      empty : History = ⟨⟩,

      add_event :
        D.Date × E.Event_info × History ~→ History
      add_event(d, ei, hist) ≡
        let event = mk_Event(d, ei) in ⟨event⟩ ^ hist end
      pre ∼in_history(E.kind_of(ei), hist),

      get_first_event : History ~→ Event
      get_first_event(hist) ≡
        if tl hist = empty then hd hist
        else get_first_event(tl hist)
        end
      pre hist ≠ empty,
get_last_event : E.Event_kind × History $\overset{\sim}{\to}$ Event
get_last_event(evkind, history) ≡
   **if** E.kind_of(event_inf(**hd** history)) = evkind
   **then hd** history
   **else** get_last_event(evkind, **tl** history)
   **end**
**pre** in_history(evkind, history),

get_last_ev_info :
   E.Event_kind × History $\overset{\sim}{\to}$ E.Event_info
get_last_ev_info(evkind, history) ≡
   event_inf(get_last_event(evkind, history))
**pre** in_history(evkind, history),

get_last_ev_date : E.Event_kind × History $\overset{\sim}{\to}$ D.Date
get_last_ev_date(evkind, history) ≡
   date(get_last_event(evkind, history))
**pre** in_history(evkind, history),

filter : (E.Event_info → **Bool**) → History → History
filter(f)(h) ≡
   **if** h = ⟨⟩ **then** ⟨⟩
   **else**
      **if** f(event_inf(**hd** h))
      **then** ⟨**hd** h⟩ ^ filter(f)(**tl** h)
      **else** filter(f)(**tl** h)
      **end**
   **end**,

evs_in_period : D.Period × History → History
evs_in_period(p, h) ≡
   **if** h = ⟨⟩ **then** ⟨⟩
   **elsif** D.is_in_period(date(**hd** h), p)
      **then** ⟨**hd** h⟩ ^ evs_in_period(p, **tl** h)
   **else** evs_in_period(p, **tl** h)
   **end**,

in_history : E.Event_kind × History → **Bool**
in_history(ek, h) ≡
   (∃ ev : Event •
       ev ∈ h ∧ E.kind_of(event_inf(ev)) = ek),

event_in_period :
   E.Event_kind × D.Period × History → **Bool**
event_in_period(ek, p, h) ≡
   (∃ ev : Event •
          ev ∈ h ∧ E.kind_of(event_inf(ev)) = ek ∧
          D.is_in_period(date(ev), p)),

      duration : History → Nat
      duration(h) ≡
        if h = ⟨⟩ then 0
        else D.diff(date(hd (h)), date(h(len (h))))
        end,

      check_event_record : Nat × Nat × History → Bool
      check_event_record(nro_ev, hs_tolerance, h) ≡
        (duration(h) ≥ D.hours_per_day + hs_tolerance ⇒
          len (h) ≥ nro_ev) ∧
        (duration(h) ≤ D.hours_per_day − hs_tolerance ⇒
          len (h) ≤ nro_ev)
  end
```

C.25 EVENT_INFO Module

```
scheme EVENT_INFO =
  class
    type Event_kind, Event_info

    value
      kind_of : Event_info → Event_kind
  end
```

C.26 D Module

context: DATE

object D : DATE

C.27 DATE Module

```
scheme DATE =
  class
    type
      Year = Nat,
      Month = {| m : Nat • m ≥ 1 ∧ m ≤ 12 |},
      Day = {| d : Nat • d ≥ 1 ∧ d ≤ 31 |},
      Hour = {| h : Nat • h ≥ 0 ∧ h ≤ 23 |},
    Date ::
        year : Year
        month : Month
        day : Day
        hour : Hour,
    Period ==
        closed(start : Date, finish : Date) |
        since(starting : Date) |
        upto(ending : Date) |
        point(now : Date)

value
    hours_per_day : Nat = 24,

    later : Date × Date → Bool /∗ first > second ∗/
    later(d1, d2) ≡
        year(d1) > year(d2) ∨
        year(d1) = year(d2) ∧
        (month(d1) > month(d2) ∨
         month(d1) = month(d2) ∧
         (day(d1) > day(d2) ∨
            day(d1) = day(d2) ∧ hour(d1) > hour(d2))),

    /∗ number of months between Date and current Date
    y1 − y2 ≤ 1
    ∗/
    months_since : Date × Date ~→ Nat
    months_since(d1, d2) ≡
        let
            y1 = year(d1),
            m1 = month(d1),
            y2 = year(d2),
            m2 = month(d2)
        in
            if y1 = y2 then m2 − m1 else 12 − m1 + m2 end
        end
    pre ∼later(d1, d2),

    is_in_period : Date × Period → Bool
    is_in_period(d, p) ≡
        case p of
            closed(s, f) → ∼(later(d, f) ∨ later(s, d)),
            since(s) → ∼(later(s, d)),
            upto(f) → ∼(later(d, f)),
            point(sf) → ∼(later(d, sf) ∨ later(sf, d))
        end,
months_back : Date × Nat → Date
months_back(d, n) ≡
   let
      y = year(d), m = month(d), a = day(d), h = hour(d)
   in
      if m − n > 0 then mk_Date(y, m − n, a, h)
      else
         let
            y1 =
               if (n − m) \ 12 = 0 then y − (n − m) / 12
               else y − (n − m) / 12 + 1
               end,
            m1 = 12 − (n − m − n − m / 12 ∗ 12)
         in
            mk_Date(y1, m1, a, h)
         end
      end
   end,

days_back : Date × Nat → Date
days_back(d, n) ≡
   let
      y = year(d), m = month(d), a = day(d), h = hour(d)
   in
      if a − n > 0 then mk_Date(y, m, a − n, h)
      else
         let
            ms =
               if (a − n) \ 30 = 0 then (a − n) / 30
               else (a − n) / 30 + 1
               end,
            d1 = months_back(d, ms),
            a1 = 30 − (n − a − n − a / 30 ∗ 30)
         in
            mk_Date(year(d1), month(d1), a1, h)
         end
      end
   end,

/∗ only to go back up to 24 hours∗/
hours_back : Date × Nat → Date
hours_back(d, n) ≡
   let
      y = year(d), m = month(d), a = day(d), h = hour(d)
   in
```

```
          if h − n > 0 then mk_Date(y, m, a, h − n)
          else
             let
                m1 = if a = 1 then m − 1 else m end,
                a1 = a − 1,
                h1 = 24 − (n − (24 − h))
             in
                mk_Date(y, m1, a1, h1)
             end
          end
       end,

    last_midnight : Date → Date
    last_midnight(d) ≡
       let
          y = year(d), m = month(d), a = day(d), h = hour(d)
       in
          mk_Date(y, m, a, 0)
       end,

    last_midday : Date → Date
    last_midday(d) ≡
       let
          y = year(d), m = month(d), a = day(d), h = hour(d)
       in
          mk_Date(y, m, a, 12)
       end,

    in_morning : Date → Bool
    in_morning(d) ≡
       let h = hour(d) in h ≥ 0 ∧ h ≤ 12 end,

    last_n_months : Nat × Date → Period
    last_n_months(n, d) ≡ since(months_back(d, n)),

    last_n_days : Nat × Date → Period
    last_n_days(n, d) ≡ since(days_back(d, n)),

    last_n_hours : Nat × Date → Period
    last_n_hours(n, d) ≡ since(hours_back(d, n)),

    add_n_months : Date × Nat → Date
    add_n_months(d, n) ≡
       let
          y = year(d), m = month(d), a = day(d), h = hour(d)
       in
```

```
        if m + n ≥ 12
        then
           let y1 = y + (m + n) / 12, m1 = (m + n) \ 12 in
              mk_Date(y1, m1, a, h)
           end
        else mk_Date(y, m + n, a, h)
        end
     end,

  add_n_days : Date × Nat → Date
  add_n_days(d, n) ≡
     let
        y = year(d), m = month(d), a = day(d), h = hour(d)
     in
        if m + n ≥ 30
        then
           let
              d1 = add_n_months(d, (m + n) / 30),
              a1 = (m + n) \ 30
           in
              mk_Date(y, month(d1), a1, h)
           end
        else mk_Date(y, m, a + n, h)
        end
     end,

  is_in_range : Nat × Nat × Nat → Bool
  is_in_range(sr, er, n) ≡ n ≥ sr ∧ n ≤ er,

  /* both dates in the same year */
  days_since : Date × Date →̃ Nat
  days_since(d1, d2) ≡
     let
        y1 = year(d1),
        m1 = month(d1),
        a1 = day(d1),
        y2 = year(d2),
        m2 = month(d2),
        a2 = day(d2)
     in
        if
           m2 >
              m1 +
              1 /* d1 is at least two months greater than d2
              */
        then (m2 − (m1 + 1)) ∗ 30 + (30 − a1) + a2
```

```
            elsif m2 > m1 then (30 − a1) + a2
            else a2 − a1
            end
         end
      pre ∼later(d1, d2),

      diff : Date × Date → Nat
      diff(d1, d2) ≡ 1
end
```