Identifying and Fixing Ambiguities in, and Semantically Accurate Formalisation of, Behavioural Requirements

Thuy Nguyen[1], Imen Sayar[2], Sophie Ebersold[2], and Jean-Michel Bruel[2]

[1]EDF (Electricité De France), retired, ORCID 0009-0005-9992-176X
[2]University of Toulouse, CNRS-IRIT

May 2, 2025

Abstract

To correctly and accurately formalise requirements expressed in natural language, *ambiguities* must first be identified and then fixed. This paper focuses on *behavioural requirements* (i.e., requirements related to dynamic aspects and phenomena). Its first objective is to show, based on a practical, public case study, that the disambiguation process *cannot be fully automated*: even though natural language processing (NLP) and machine learning (ML) might help in the *identification* of ambiguities, *fixing* them often requires a deep, application-specific *understanding* of the reasons of being, nature and boundaries of the system of interest, of the composition and characteristics of its environment, of which trade-offs between conflicting objectives are acceptable, and of what is achievable and what is not; it may also require arduous negotiations between stakeholders. Such an understanding and consensus-making ability is not in the reach of current tools and technologies, and will likely remain so for a long while.

Beyond ambiguity, requirements are often marred by various other types of defects that could lead to wholly unacceptable consequences. In particular, operational experience shows that requirements *inadequacy* (whereby, in some of the situations the system could face, what is required is woefully inappropriate or what is necessary is left unspecified) is a significant cause for systems failing to meet expectations. The second objective of this paper is to propose a semantically accurate behavioural requirements formalisation format enabling *tool-supported requirements verification*, notably with *simulation*. Such support is necessary for the engineering of large and complex *cyber-physical* and *socio-technical* systems to ensure first that the specified requirements indeed reflect the true intentions of their authors, and second that they are adequate for all the situations the system could face. To that end, the paper presents an overview of the BASAALT (*Behaviour Analysis and Simulation All Along systems Life Time*) systems engineering method, and of FORM-L (*FOrmal Requirements Modelling Language*), its supporting language which aims at representing as accurately and completely as possible the semantics expressed in the original, natural language behavioural requirements, and is markedly different from languages intended for software programming.

The paper shows that generally, semantically accurate formalisation is not a simple *paraphrasing* of the original natural language requirements: additional elements are often needed to fully and explicitly reflect all that is implied in natural language. To provide such complements for the case study presented in the paper, we followed different *formalisation patterns*, i.e., sequences of formalisation steps.

For this paper, to avoid being skewed by what a particular automatic tool can and cannot do, BASAALT and FORM-L were applied manually. Still, the lessons learned could be used to specify and develop NLP tools that could *assist* the disambiguation and formalisation processes. However, more studies are needed to determine whether an exhaustive set of formalisation patterns can be identified to support the formalisation process.

1 Introduction

In many industry sectors, system failures can have large and sometimes wholly unacceptable human, societal, environmental, and economic consequences. Several studies have been made to draw lessons from the operational experience of actual systems. One of their common conclusions is that defects in requirements are a significant cause of systems failing to meet expectations. That is well-documented in sectors where safety is of importance (see, for example, [OEC12, EPR08, EPR15]).

To improve its requirements engineering (RE) practices, particularly concerning the specification of behavioural requirements (i.e., requirements related to dynamic aspects and phenomena occurring and evolving in time, including functions, performance, cost and workloads; in what follows, term *requirements* stands for *behavioural requirements*), a set of criteria was developed at EDF for selecting an RE method and a supporting formal modelling language suitable for its industrial systems. Unfortunately, a survey found no good candidates [Azz19]. Eventually, the set of criteria matured into BASAALT (*Behaviour Analysis & Simulation All Along systems Life Time*), a model and simulation-based systems engineering (SE) *method*, and into FORM-L (*FOrmal Requirements Modelling Language*), the modelling *language* that supports BASAALT [Ngu23b].

BASAALT and FORM-L are based on experience gained in the engineering of a large number and a wide variety of actual systems and projects: new power plants, refurbishment of existing power plants, backup electric power systems, cooling systems, HVAC (heating, ventilation and air conditioning) systems, autonomous race cars, landing gear systems, smart multi-energy power grids, inspection drones, etc. They can support the complete system life cycle, from initial conceptual studies to design, manufacturing, construction and installation at site, operation, maintenance, renovation and finally, deconstruction. Contrary to what its name suggests, FORM-L can be used to model not only *requirements* but also all other aspects of dynamic behaviour, including *assumptions* and *solutions* to requirements.

This paper illustrates some aspects of BASAALT and FORM-L with the requirements of *k3* [Lan23b, Lan23a], one of the case studies collected in the [Lan23c] public repository. Each case in the repository provides a set of natural language (English) requirements, with *k3* specifying requirements for the information system of the Nursing Department of a teaching institution. The challenge set by the repository authors is to formalise these natural language requirements automatically.

However, for this paper, to avoid being skewed by what a particular automatic tool can and cannot do, BASAALT and FORM-L were applied manually. Before proceeding to formalisation, one must first identify and fix the almost inevitable *ambiguities* of the natural language requirements. This paper shows that the disambiguation process *cannot be fully automated*: even though tools might help in the *identification* of ambiguities, *fixing* them often requires a deep, application-specific *understanding* of the reasons of being, nature and boundaries of the system, of the composition and characteristics of its environment, of which of the possible trade-offs between conflicting objectives are acceptable, and of what is achievable and what is not; it may also require arduous negotiations between stakeholders. Though such an understanding and consensus-making ability is not within the reach of current tools and technologies and will likely remain so for a long while, the lessons learned from the case study can be used to specify and develop NLP (Natural Language Processing) tools to *assist* the disambiguation process, at least for the English language.

This paper also shows what it means to *formalise* requirements, from a BASAALT and FORM-L perspective. Contrary to most formalisation methods, their main objective is not to provide an initial seed to software design and implementation, but to enable the tool support (in particular *behavioural simulation*) necessary to rigorous *assessment and verification of requirements semantics*, to help ensure freedom from defects such as *inadequacy* (whereby what is required is woefully inappropriate in some of the situations the system could face), *over-ambition* (whereby what is required is not absolutely necessary but leads to unwarranted complexity and unacceptable risks) and *contradiction* (whereby the satisfaction of some requirements necessarily implies the violation of some others). As is shown in the paper, that implies not only *semantically accurate modelling*, but also, in many cases, providing necessary *information and contextual elements* missing and / or implicit in the original requirements. The *k3* case study shows that such complements can be provided by the application of a limited number of *formalisation patterns* (i.e., sequences of formalisation steps). More studies are needed to determine whether a complete set of formalisation patterns can be identified to support the formalisation process in the general case.

Section 2 presents an overview of the BASAALT method and of the FORM-L language. It also compares them with other, better-known methods, in particular with KAOS [RI07] and Event-B [JR10]. Section 3 classifies the various types of ambiguity that can be found in natural language requirements and gives *k3*-based examples for each type. Section 4 presents and discusses the FORM-L modelling of some representative *k3* requirements. The formalisation of the full set can be found in [Ngu23a]. Finally, the proposed method and its application to *k3* are discussed in Section 5, highlighting the differences and complementarity with other methods. We conclude in Section 6. Finally, in Annex A, two tables summarise the FORM-L notations used in the paper.

The *k3* case study was given as an input to the authors of this paper, but as BASAALT and FORM-L are based on very general principles and concepts honed and experimented on a very wide variety of systems and requirements, any other case study of the repository would likely lead to similar conclusions.

2 BASAALT and FORM-L

The BASAALT / FORM-L pair [Ngu23b] mainly addresses systems behaviour and dynamic phenomena. As it proposes a very general approach, this brief presentation is not a complete one, and not all what is presented here has been used in the *k3* case study.

2.1 How BASAALT / FORM-L Differs from Most Other RE Methods

The targets of BASAALT / FORM-L are *cyber-physical systems* (*CPS*, i.e., systems involving computing, networking and physics), *socio-technical systems* (*STS*, i.e., technical systems also involving human and organisational aspects) and *systems of systems* (*SoS*, i.e., systems composed of relatively independent but interacting systems, each having its own, separate life cycle). In the following, when used without qualification, term 'system' stands for *CPS* / *STS* / *SoS*.

Many RE methods [RI07, JR10, Lam09, WB13, LK22, Lef10, Lar12, Coc01, Poh16, PA10, KS98, RR12, ZJ97, Boo15, Gli22, Mey22] and standards ([IEE90, CKS11, Boo15]) are issued from *software engineering*, which often leaves aside aspects that are essential to systems. However, these aspects are also important for pure software or for software-based systems, for which one needs to consider the environment within which they will be tested, distributed, operated and maintained, which always has human and physical components: failing to take them fully into account can lead to inadequate or missing requirements, with sometimes significant or even unacceptable adverse consequences.

In particular, BASAALT / FORM-L addresses:

- *Timing*, which is essential for real-life systems, for which 'too late' often means 'failure'. Thus:
  - "*After event E, action A shall be performed*" will not do since it allows *A* to occur minutes, hours, days, years, or aeons after *E*.
  - "*When event E, action A shall be performed*" will not do either, as no system can react instantaneously.
- *Physical quantities* such as temperature or pressure, which represent *continuous states* that come in addition to possible discrete *functional states*. For physical quantities, like for timing, one always needs to specify margins:
  - "*When pressure > 10 bars do A otherwise do B*" will not do, as no physical system can be infinitely precise.

  Also, to avoid ambiguity, physical quantities must be expressed in terms of *physical units*. For example, NASA's *Mars Climate Orbiter* suffered a catastrophic failure [Boa13] because the need for a necessary unit conversion was not obvious and was not performed.
- *Randomness*, which in physical and socio-technical systems may be due to phenomena such as hardware failures, variability of manufacturing and human behaviour, noise or external events.
- *Non-engineered interactions*, which may be caused by unwanted side-effects even during normal operation (e.g., heating due to geographical proximity or vibration due to physical connections), failure propagation, human errors or malicious attacks.

Often, notably in software engineering, *requirements specification* is considered as a *phase* in the SE process, different and separate from the *design of solutions*. For large and complex systems, *this cannot be so*. Such systems are *recursive*: subsystems are often full-fledged systems of their own, and the design of a system consists of RE for its subsystems: therefore, new requirements are elicited and specified not only as design progresses (see Fig. 1) but also when complex activities during construction, operation and deconstruction require well-engineered solutions. Thus, BASAALT is both an RE and an SE method. This is also the case of MOFLT [Bou21], the Airbus method: requirements are central and

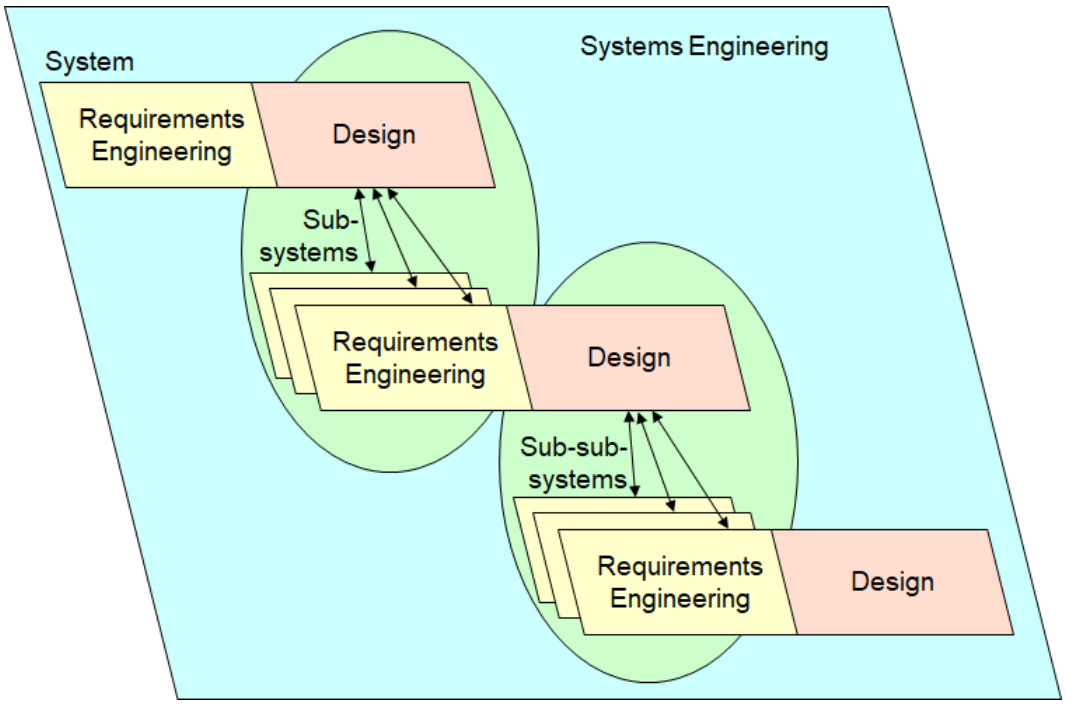

Figure 1: RE is intrinsically intermingled with SE

cover all the development refinements from Missions to Operational, Functional, Logical and Technical elements.

Another important difference with many RE methods is that BASAALT / FORM-L cares not only about the form and appearance of requirements but also of their semantics to verify that they are *unambiguous*, *adequate* with respect to overall objectives, neither *over-ambitious* nor *over-specified*, and last but not least, *achievable*.

KAOS

In addition to what is already mentioned, one important difference between KAOS [vL04, RI07] and BASAALT / FORM-L is the fact that *goals* and *requirements* in KAOS are expressed only in natural language, whereas in FORM-L, all *properties* (including *objectives*, *requirements*, *assumptions*, *guarantees* and *guards*) can be expressed both in natural language and in a formal format enabling the extensive tool support (in particular, but not only, *simulation* and / or *formal verification*) necessary to reveal and correct the inevitable defects in specified requirements (see Section 2.2.1).

One distinctive feature of KAOS is the progressive refinement of initial *goals* into detailed *requirements*. BASAALT / FORM-L also proposes such a progressive refinement from top-level, natural language, often abstract, sometimes unachievable *objectives* into concrete, achievable and verifiable technical *requirements*. However, the distinction between goals and requirements in KAOS is not the same as the distinction between objectives and requirements in FORM-L: whereas in KAOS, an objective is something that is expected of the system but not yet assigned to a single agent, in BASAALT / FORM-L, an objective is something that is desired but not absolutely required: generally, *probabilistic* requirements are used to specify limits to failure to satisfy objectives.

Also, property refinement in FORM-L is richer and more precise than in KAOS. In KAOS, a refinement is expressed with the decomposition of a goal into sub-goals and requirements. In BASAALT / FORM-L, different types of property refinement are possible:

- *Formalisation* gives a formal expression to a property that was up-to-then specified only in natural language.
- *Concretisation* associates an *abstract* property (expressed in terms that can be neither observed nor computed) with one or more *concrete* properties (expressed in terms that can all be either observed

or computed). It is generally used to refine initial, high-level and fuzzy objectives into detailed, precise and technical requirements. The concretisation must be *stronger* than the property being concretised, i.e., satisfaction of the concretisation must imply satisfaction of the concretised.

- *Substitution* associates an *unachievable* property with one or more supposedly *achievable* properties. The substitution is usually *weaker* than the property being substituted, i.e., satisfaction of the substitution does not necessarily imply satisfaction of the substituted.

Considering that a property can be in one of three states (*not tested*, *satisfaction* or *violation*), FORM-L has three built-in operators to combine properties:

- *AND* is in *satisfaction* when all arguments are, in *violation* when any argument is, in *not tested* otherwise.
- *weak OR* is in *satisfaction* when any argument is, in *violation* when all arguments are, in *not tested* otherwise.
- *strong OR* is in *satisfaction* when any argument is while none are in *violation*, in *violation* when all arguments are, in *not tested* otherwise.

Lastly, whereas KAOS is essentially graphical, BASAALT / FORM-L is currently essentially textual, and whereas modelling modularity in KAOS mainly relies on different types of models (goal models, object models, responsibility models, operation models, ...), in BASAALT / FORM-L it relies on step-by-step *model extension* (see Section 2.2.3), with only one type of *source model* containing and integrating all necessary information. Specialised KAOS-style *derived models* could be generated from source models by appropriate tools.

Event-B

Event-B [JR10] is a formal method that enables the gradual construction of correct formal models by means of a proof activity. This method is an extension of the classic B language [Abr96]. Based on set theory and first-order logic, this method is used to develop formal models for reactive systems and sequential and distributed algorithms. It can also be used to develop systems combining software, hardware and user interfaces. The development of Event-B models can be carried out with the Rodin (Rigorous Open Development Environment for Complex Systems) open-source platform [ABHV06].

Like Event-B, FORM-L is based on a formal notation. However, the Event-B notation is essentially a mathematical notation facilitating *theorem proving*: it is not necessarily well-mastered by average engineers. On the contrary, the English-like FORM-L notation (with possibly syntactic variants for other natural languages) is designed to facilitate *comprehension* with minimal training by all concerned participants. Thus, whereas Event-B is mainly used for the most *critical software parts* of highly critical systems, the ambition of BASAALT / FORM-L is to support the engineering of *complete*, *complex* systems, mainly with *simulation*.

Unlike Event-B, BASAALT / FORM-L does not inherently embed the notion of *proof*. Otherwise, all behaviours that can be expressed in Event-B can also be expressed in FORM-L. The reverse is not true, as FORM-L has an explicit notion of *time*, with possibly overlapping time intervals and time margins. Example (see Section 2.3.7 for an explanation of the FORM-L syntactic highlighting used in this paper):

```
external Event eRequest
  "Randomly occurring service requests"
  is deferred;
Event eAcknowledgement (eRequest)
  "The parameter of this event is the service request occurrence to be
   acknowledged"
  is deferred;
Requirement acknowledge
  "Each service request occurrence shall be acknowledged within 10 seconds"
  is after eRequest within 10*s
     achieve eAcknowledgement (bop);
     // bop (beginning of period) is the eRequest occurrence that opened
     // the 10-second time interval
```

In this example, each `eRequest` occurrence opens a 10-second time interval within which it must be acknowledged. When there is less than 10 seconds between two consecutive `eRequest` occurrences, their respective time intervals will overlap. Indeed, at any instant, there could be any number of overlapping such intervals.

2.2 Overview of the BASAALT / FORM-L Method

2.2.1 Defects in Requirements

With BASAALT / FORM-L, models express *requirements* on the system of interest, *assumptions* made on the environment of that system, and *solutions* to meet requirements. One important goal of verification is to make sure that the contemplated solutions satisfy the specified requirements in the framework of the assumptions made. However, experience shows that defects in requirements and assumptions are also a significant cause of systems failing to meet objectives.

Besides *ambiguity*, *inadequacy* is a major worry. It occurs when in some situation the system could face, what is specified is inappropriate or when what is necessary is not specified. It can have unacceptable consequences. Example: the *Cranbrook Manoeuvre* [Net78] is an aircraft accident in 1978 that killed 42 people; it was in large part caused by a thrust reverser requirement that was adequate in most situations but was totally inappropriate in the situation of the accident.

2.2.2 Tool-Supported Requirements Verification

It is a well-accepted fact in software engineering that any significant piece of programming needs rigorous, extensive testing and verification. Industry experience shows that it should be the same for requirements and assumptions: for a system worthy of the name, defects in requirements and assumptions are almost inevitable and need to be detected and corrected. To do so, one needs to consider the individual and collective meaning of specified requirements and assumptions, addressing not just form and appearance but also intentions and semantics.

In this endeavour, purely manual methods are useful but not sufficiently effective, much like for software, for which no one would be content just with reviews and inspections. As testing of the actual system is possible only very late in the engineering process, BASAALT / FORM-L relies on modelling and simulation for early verification and validation of requirements and assumptions.

FORM-L models can be simulated with tools like Stimulus [1], a stochastic, constraint-based test case generator: at each activation, it automatically generates a new test case consistent with assumptions and definitions, and automatically verifies, for each requirement, whether it has been challenged (*tested*), and if so, whether it has been *satisfied* or *violated* (see Fig. 3 and 4). This way, with minimal effort, large numbers of cases can be explored right from the very early stages of the engineering process.

Automatic test case generation is not only a means to save human effort: it is also a means to *open-mindedly* explore the set of situations the system could face (as specified by assumptions and definitions), with minimum bias and preconceptions. As each engineering step introduces new elements, human test case authors may fail to take all of them into account and often concentrate on normal cases, leaving large swathes of the situation set unexplored, unanticipated and unexpected.

As situations are combinatorial and often impossible to explore exhaustively by simulation, formal verification (not studied for BASAALT / FORM-L yet) could be considered.

2.2.3 Modelling Modularity and Model Extension

In the modelling of large and complex systems, *modularity* is a necessity. With BASAALT / FORM-L, it is supported by the notion of *model extension*, whereby a new model adds information to existing models without having to modify them, as illustrated in Fig. 2.

Model extension serves several important purposes and can, in particular, be used to keep track of:

- The step-by-step nature (in time) of the systems engineering process. That is shown in Fig. 2 by model extension along the *horizontal axis*. In the figure, white boxes represent non-formal or semi-formal FORM-L modelling, whereas green boxes represent formal FORM-L modelling. Hatched

[1] https://www.3ds.com/products-services/catia/products/stimulus/

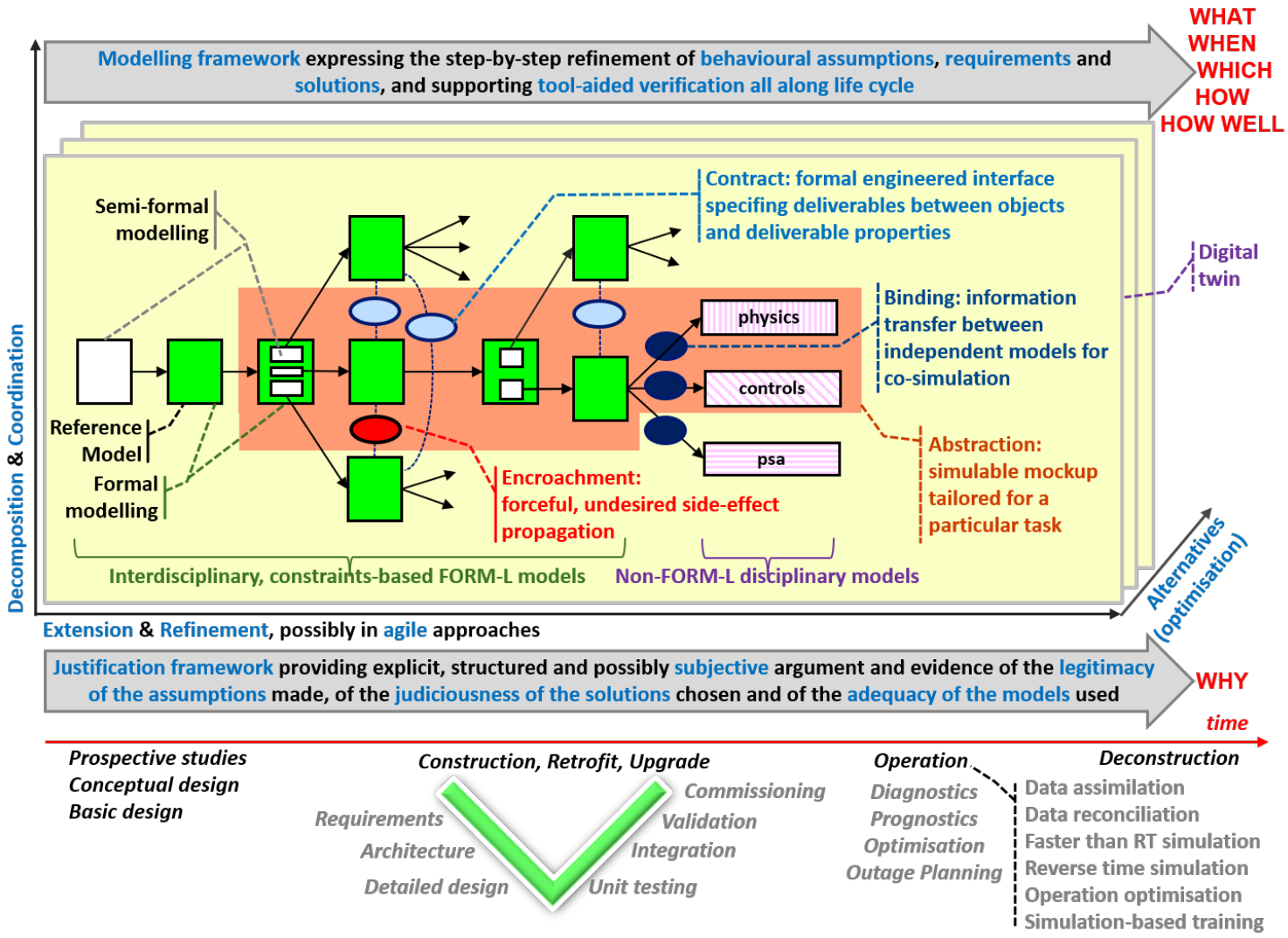

Figure 2: The BASAALT step-by-step refinement and decomposition method, as supported by model extension

boxes represent non-FORM-L, discipline-specific models that can be developed once the design of solutions is sufficiently detailed.

- The contributions of the *various participants* (teams, engineering disciplines, stakeholders, organisations) involved, each having their own viewpoint, expectations and requirements. That is shown in Fig. 2 along the *vertical axis*.
- The gradual *decomposition* of the system into sub-systems. That is also shown in Fig. 2 along the *vertical axis*.
- The *alternatives solutions* considered and assessed while looking for optimal solutions. That is shown in Fig. 2 along the *diagonal axis*.
- *Scenario models* used to guide the random test case generator toward test cases of interest. That is not explicitly shown in Fig. 2.

Interfaces are an essential ingredient of modularity and are shown in Fig. 2 as coloured ovals:

- Light blue ovals represent *contracts*, i.e., desired, engineered interfaces.

- Red ovals represent *encroachments*, i.e., undesired side effects that nonetheless need to be taken into consideration.
- Deep blue ovals represent so-called *bindings*, which enable information exchange between FORM-L models that were developed independently from one another (enabling, for example, the use of off-the-shelf product and solution models provided by independent organisations) or with non-FORM-L, discipline-specific models. (*psa* stands for *probabilistic safety assessment*.)

In this paper, the engineering process starts with a number of natural language statements that are then formalised into rigorous models that can be assessed and verified by simulation. Aspects for which there is not enough information are *deferred* to future extension models.

2.2.4 The BASAALT Requirements Verification Process

The requirements verification process presented here is the one recommended by BASAALT. It has not been fully applied in this paper since the *k3* case study has been set independently of BASAALT.

Reference Model

BASAALT proposes a rigorous method for requirements verification. In particular, it suggests to start with a so-called *reference model*. Its role is to:

- Identify the *system of interest* and, if necessary, its boundaries.
- Identify the entities that interact with the system and constitute its *environment*. They could be other systems, passive components or structures, human agents, or the physical environment (e.g., the atmosphere). Other entities could be added later, in future extension models, when new engineering disciplines are involved and need to take account of additional effects, or when more is decided and known about the system and its operation.
- Model the *situations* the system might face: they could be due to *internal system states* (normal and abnormal), the *states and behaviour* (normal and abnormal) of the various entities of its *environment*, and the *operational objectives* assigned to the system at any instant.
- State the key *assumptions* of the system regarding its environment, possibly depending on situation.
- Specify the top-level *requirements* placed on the system and which constitute its reasons of being, also possibly depending on situation.

This model is generally developed in two steps. The first step is semi-formal, requirements and assumptions being expressed in natural language. It is represented by the leftmost white box of Fig. 2 and must be verified by reviews and inspections. The second step is fully formal and is represented by the leftmost green box of Fig. 2. Being formal, it benefits from aids provided by available support tools. In particular, even though it is a constraint-based, non-deterministic model, a *stochastic test case generator* can be used to generate random test cases consistent with assumptions and definitions. The simulation data flow is illustrated by Fig. 3. One can then check whether, in each case, the behaviour is indeed as was intended. As this checking is essentially manual, the reference model should be as simple as possible and should focus on what is essential.

Extension Models

Each subsequent extension model represents the decisions made at an individual engineering step in the form of additional definitions, requirements and assumptions. Like the reference model, it can be verified by stochastic simulation against the requirements of the models it extends, including, ultimately, the reference model. To this end, its own requirements are treated as assumptions: supposing that they are satisfied, will the requirements of the models being extended be satisfied? (See Fig. 4.)

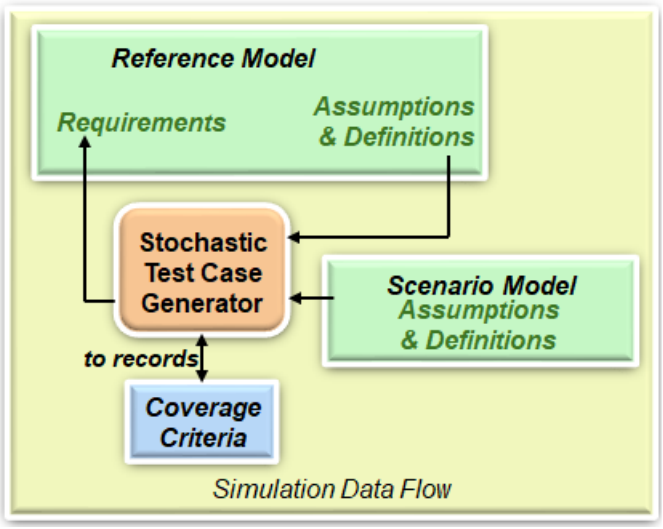

Figure 3: Simulation data flow to verify the adequacy of the reference model

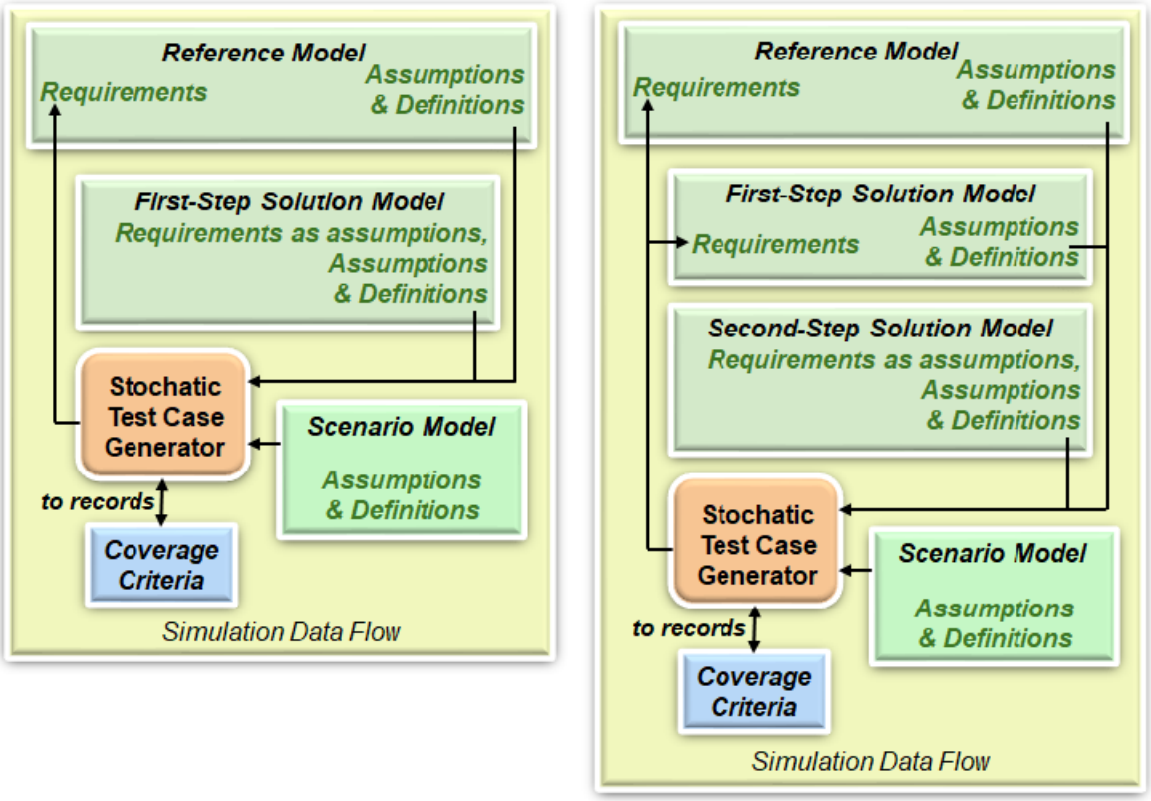

Figure 4: Simulation data flows to verify a first-step solution model against the requirements of the reference model (left) and a second-step solution model against the requirements of the first-step solution model and of the reference model (right)

2.2.5 Traceability

With BASAALT / FORM-L, traceability is obtained not only with the notions of extension and refinement but also with the notions of *concretisation* (whereby an initial fuzzy and ambiguous requirement is gradually re-expressed into more precise requirements) and *substitution* (whereby an initially overambitious

or unachievable requirement is gradually replaced by actually achievable ones). Another strength of the method is that, in addition to the modelling framework offered by FORM-L, it also includes a *justification framework* [ISO15] to explain the WHY of engineering decisions (see also Fig. 2).

Keeping track of these transformations is important for many reasons, but in particular:

- Natural language statements are often ambiguous, but they may be more apt at conveying true intentions than formally specified ones, which are more precise but also more verbose and more difficult to understand.
- Initial statements are often incomplete and fail to address all necessary aspects, but they express what is essential, which could be difficult to identify when eventually all the missing aspects are addressed.

2.3 Overview of the FORM-L Language

FORM-L [Ngu23b, Ngu19, Ngu18] is a domain-specific language [BEG+21] dedicated to properties (e.g., assumptions, objectives and requirements) and behavioural modelling. It supports the notions of *behavioural item*, *determiner*, *instruction*, *time domain*, *interface*, *statement* and *model* that are described below.

2.3.1 Behavioural Items

As the name implies, dynamic phenomena and behaviours are modelled with *behavioural items*, i.e., *objects* and *classes*. Very classically, classes are templates for objects. An object may be an instance of multiple classes.

An object bundles together the *features* (in the form of embedded objects) that characterise a real-world entity. It may declare and define features of its own in addition to those specified by its classes.

To reflect the dynamic composition of some systems, in particular of *SoS*, in addition to named *permanent objects*, *dynamic objects* may be created and deleted at any time along each operational case. All objects, including dynamically created objects, are necessarily included in some *sets*. In particular, to each class corresponds the implicitly defined set of all its instances.

An object can be *valued* or *non-valued*. A *valued object* is also an implicit function of time: at any instant along an operational case, it has one and only one *value*. Valued objects that take other objects as parameters are called *functions*. The nature of the value depends on whether the object is:

- A *variable*. Very classically, there are four predefined *variable classes*: `Boolean`, `Real`, `Integer`, and `String`. One can also define *enumerated classes*: an instance of such a class is a *finite state automaton*.
- An *event*, in which case it represents a fact of interest that may occur, possibly multiple times, and the duration of which is neglected. Each time that fact occurs is an *occurrence*. The value of an event is its set of past occurrences. `Event` is the only predefined *event class*.
- A *property*, in which case it expresses *constraints* on objects identified by name or by *selectors*, at instants or time intervals specified by a *temporal locator* (see Section 2.3.3). Its value is its state (*not tested*, *satisfaction* or *violation*) along an operational case. There are six predefined *property classes*: `Property` (neutral), `Assumption` (never in *violation*), `Objective` (desired but not necessarily achievable and required), `Guarantee` (in *contracts*), `Requirement` (must be satisfied) and `Guard` (defining the validity domain of a model). A property is often declared as a feature of the behavioural item that is responsible for it.
- A *set*. A *set of objects* is a finite, non-ordered collection of objects of the same class. A *set of values* is a finite, non-ordered collection of values or value intervals of the same nature. In both cases, the value of the set is its *membership*, which may vary along an operational case. There are no predefined *set classes*.

`Object` is the only predefined *non-valued class*.

New classes may be created by *extending* existing classes. In particular, `Real` can be extended to represent physical and non-physical quantities (e.g., `Mass`, `Length` or `Money`).

An *attribute* is a piece of information built-in in the FORM-L language that can be used to observe and/or control an object. For example, a variable has, among others, attributes `value`, `previous` and `next`. A `Real` also has, among others, attributes `quantity`, `unit` and `derivative`.

2.3.2 Determiners

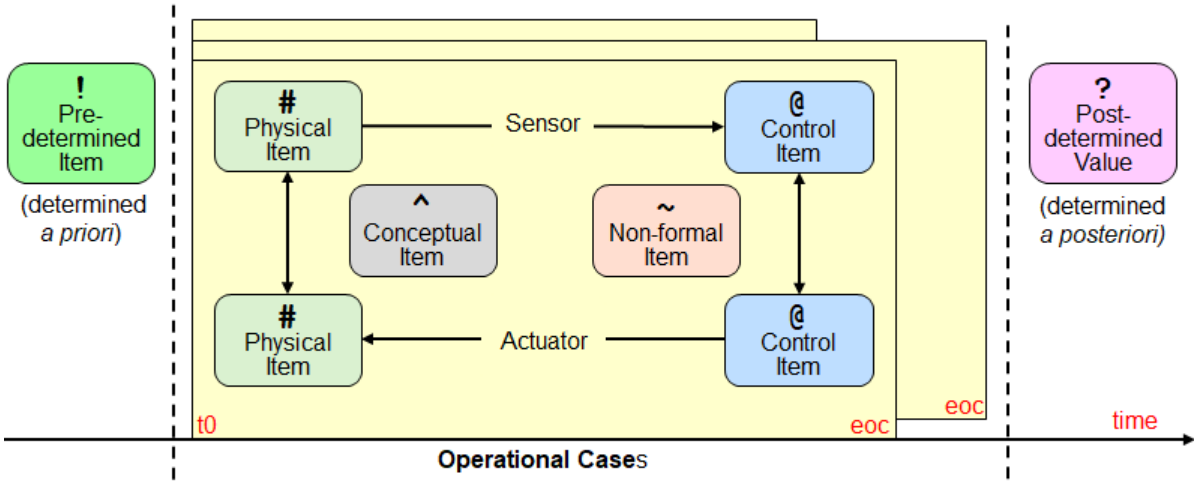

Figure 5: The six FORM-L determiners. `t0` is the beginning of an operational case, and `eoc` its end.

A *determiner* is an optional piece of information that may be attached to a behavioural item. It does not affect behaviour *per se*, but helps understand the intentions of model authors. Determiners are denoted with specific marks (!, #, @, ˜ , ?, ˆ), see Fig. 5.

The **!** determiner indicates that an object, the definition of which is currently *deferred*, will be fully determined during design and has exactly the same behaviour in all operational cases. Constants are generally but not necessarily predetermined (some can be determined only *a posteriori* and have a **?** determiner). A predetermined item is not necessarily constant.

The # determiner indicates that the object is a so-called *physical object*, whereas the @ determiner indicates that the object is a so-called *control object*. Physical objects represent actual entities and observable and measurable aspects of the *real, physical world*. Control objects represent the '*beliefs*' or the '*desires*' that some parts of the system or its environment may have on the physical world and are obtained either by measurement of physical aspects or by calculation based on other control, physical or predetermined objects.

The ˆ determiner indicates that an item represents an *abstract, conceptual notion* that cannot be observed, measured or calculated. Even though random generation of legitimate cases may be used, contrary to concrete objects (i.e., physical or control objects), they do not and will never have a fully deterministic definition. The item may be used to express high-level properties, but as these would also be conceptual, they will need, at some point in the engineering process, to be associated (through the *concretisation* attribute) with concrete properties (i.e. with a # or @ determiner).

The ˜ determiner applies to properties expressed only in natural language and that will not be formalised. Such items are provided for information to readers but do not participate in simulation or formal verification and cannot be used in formal expressions. They must be *redeclared* with other determiners if model authors eventually decide to formalise them.

The **?** determiner applies to *post-determined* constants, i.e., values that are unknown during the whole course of individual operational cases and that can be determined only by the analysis of the outcomes of large, appropriate sets of cases. This is, for example, the case of system-level failure probabilities. These values may be used to express objectives and requirements, but these can themselves be determined only *a posteriori*.

2.3.3 Instructions

Instructions define the behaviour of behavioural items. They are of two kinds: *elementary* and *composite*.

Elementary Instructions

Elementary instructions are composed of three parts: an *action*, optional *selectors*, and an optional *temporal locator*.

Temporal locators place actions precisely in time: *discrete temporal locators* (*DTL*) specify finite numbers of discrete instants, whereas *continuous temporal locators* (*CTL*) specify finite numbers of possibly overlapping time intervals (see Fig. 6).

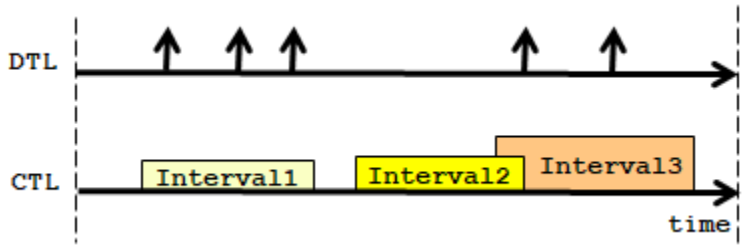

Figure 6: Discrete (DTL) and Continuous (CTL) Temporal Locators

Selectors are natural companions for sets: they are known in mathematics as *universal* or *existential quantifiers*.

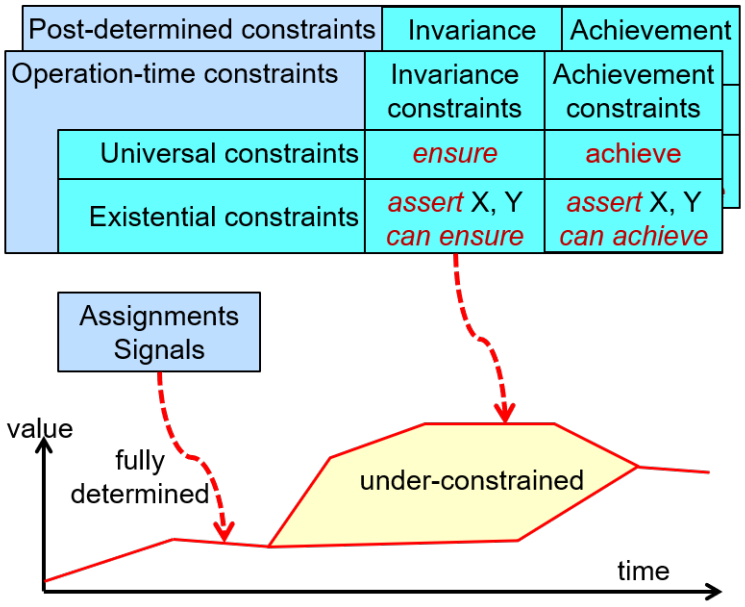

Figure 7: Action of assignments, signals and constraints on valued items

There are different kinds of actions (see Fig. 7):

- An *assignment* specifies a definite value for an item attribute, which is then *fully constrained*.
- A *signal* specifies the occurrence or absence of occurrence of an event, which is then also *fully constrained*.
- A *deletion* eliminates a dynamically created object.
- A *constraint* specifies a Boolean condition to be satisfied. Contrary to assignments, it generally allows multiple values for the attributes concerned, which are then *under-constrained*. *Contradiction*

occurs when for some attribute, no value can satisfy all the specified constraints: that attribute is then *over-constrained* and the model needs to be corrected.

FORM-L has different kinds of constraints:

- It distinguishes *operation-time constraints* (which can be evaluated in the framework of each individual operational case) from *post-determined constraints* (which can be evaluated only by considering the outcomes of large numbers of operational cases). *Probabilistic* and *minimisation* / *maximisation* constraints are typically *post-determined*.
- It also distinguishes *universal constraints* (which must be satisfied in all operational cases) from *existential constraints* (which need to be satisfied in at least one operational case). The latter are used to express *capabilities* such as "*For each legitimate behaviour of all objects other than X, there shall be a legitimate behaviour of X such that constraint C is satisfied*". (A legitimate behaviour is a behaviour consistent with all assumptions and definitions.)
- Under the scope of a *CTL*, FORM-L also distinguishes *invariance constraints* (which must be satisfied all along each time interval) from *achievement constraints* (which only need to be satisfied after some point within each time interval).

Composite Instructions

Composite instructions are used to precisely coordinate multiple instructions in time. This could be done by specifying adequate temporal locators for each. However, to highlight the nature of the coordination, to simplify the expression of the coordinated instructions and thus facilitate understanding and avoid modelling errors, FORM-L offers five temporal coordination mechanisms:

- *Sequence* places two or more instructions one after the other.
- *Concurrence* places the beginning of two or more instructions at the same instant.
- *Iteration* repeats an instruction.
- *Selection* selects one instruction among several, based on Boolean or probabilistic criteria.
- *Time exclusion* facilitates the avoidance of contradictions (e.g., different values being assigned at the same time to the same attribute): the effective temporal locator for an individual instruction in a time exclusion chain is the one specified, but excluding all the instants and time intervals determined by the temporal locators of the instructions preceding it in the chain.

With composite instructions, one can express processes much like in languages such as BPMN [ISO13], and also engineering processes properties.

2.3.4 Time Domains

Currently, in FORM-L, there is one and only one *continuous (Newtonian) time domain* where human activities and physical processes occur. By default, a behavioural item is in the continuous time domain, but one can place it in a *discrete time domain* by specifying its `clock` attribute, which is an event, the occurrences of which determine the discrete instants of the time domain.

2.3.5 Statements

A FORM-L *statement* can be a *declaration* (which states the existence and nature of an item), a *definition block* (which adds to the contents and / or behaviour of an item) or an *instruction*.

An instruction is generally part of a behavioural item definition, but when it coordinates multiple behavioural items (which is often the case of composite instructions), it may be preferable to have it as a "free-floating" instruction.

An item may be declared multiple times, provided there are no contradictions. However, keyword `redeclared` may be used to alter an existing declaration, within authorised limits.

A definition may be given in multiple definition blocks providing progressive *refinements*:

- A first model may declare an item but give no definition. A class with no definition has no features other than those of its super-classes. An object with no definition has no features other than those of its classes and behaves randomly within the limits defined by its classes.
- An extension model may specify a *definition block* giving a partial definition with additional features and assignments and/or constraints at certain times.
- A further extension model may specify an additional definition block with other additional features, assignments and constraints, provided there are no contradictions.
- And so on.

2.3.6 Models

A FORM-L *model* is a named series of statements that generally represents a certain viewpoint on a system of interest (represented by an object or class declared with keyword `main`) and its environment (represented by objects and classes declared with keyword `external`).

A *partial model* is a named model subset that is used to organise and provide structure to a large model.

2.3.7 Syntactic Highlighting

In this paper:

- FORM-L examples and models are presented in coloured boxes.
- Natural language comments are in red (e.g., `"A comment"`).
- Reserved words are highlighted according to their category:
  - Names of predefined and defined classes are in dark green italics: e.g., `Boolean` or `Duration`.
  - Names of predefined operators and of operators defined in the `Standard` model are in light green italics: e.g., `or` or `mutates`.
  - Predefined constants and identifiers, and constants defined in the `Standard` model are in red italics: e.g., `year` or `me`.
  - *Attribute* are in light grey italics: e.g., `value` or `concretisation`.
  - Keywords related to *temporal locators* are in light blue italics: e.g., `when` or `after`.
  - Keywords related to *selectors* are in magenta italics: e.g., `for all` or `such that`.
  - Keywords related to *actions* are in brown italics: e.g., `ensure` or `achieve`.
  - The other keywords are in dark blue italics: e.g., `begin` or `end`.

The FORM-L notations used in this paper are summarised in Appendix A.

3 Ambiguities Classification

More often than not, statements expressed in natural language are *ambiguous*. This is due to the fact that, for the sake of communication "efficiency" and concision, human authors (speakers and writers) tend not to explicitly express what is "obvious". The human audience (listeners and readers) relies on *context* to determine what authors want to convey. They also rely on *common sense*, i.e., shared implicit, sometimes cultural, knowledge and beliefs. When context is lacking or insufficient, or when the implicit assumptions of the audience are not the same as the authors', incorrect interpretations can (and do) ensue.

That particularly applies to natural language requirements: context is rarely sufficient or may be missing altogether, and common sense is often not that common. The *k3* requirements, whether from file *k3data.txt* [Lan23a] or file *k3ucs.txt* [Lan23b], are no exception.

The first step in the formalisation process is thus to identify ambiguities as best and as thoroughly as possible and, for each, to list the possible options. Then, one needs to decide which are the correct

ones. In some cases, the right option is obvious to a human reader, but not necessarily so to a software tool lacking 'common sense'. In others, several possible options are equally plausible, even to humans endowed with common sense. As an arbitrary selection could fail to meet real intentions and satisfy actual needs, selecting the right option is then necessarily a manual, possibly long and arduous process involving requirements authors and various other participants such as stakeholders, domain experts and system designers.

For the *k3* case study, as neither requirements authors nor any other participants could be consulted, we selected options that seemed the most plausible to us, but sometimes that selection had to be somewhat arbitrary. Nonetheless, all clear enough options can be modelled in FORM-L, and a few examples are given to show that options were not retained just because they could be modelled. In some cases, as no obvious interpretation could be figured out, definitions are *deferred* to some future *extension model* when clarification can be obtained.

Note: In the following, to facilitate referencing of the individual requirements of the case study, a name is given to each. Those in *k3data.txt* are named *d1*, *d2*, ..., *d9*. Those in *k3ucs.txt* are named *r1*, *r2*, ..., *r26*.

As one of the ambitions of this paper is to propose an approach for *tool-assisted* ambiguity detection, it classifies ambiguities in requirements into five main categories.

3.1 System Ambiguity

System ambiguity occurs when it is unclear what the *system of interest* (to which the requirements apply) is, what its *nature* and *boundaries* are, what belongs to it, what belongs to its *environment*, and what is assumed regarding that environment. In the *k3* case study, *k3data.txt* and *k3ucs.txt* express requirements without any introductory text stating what the system of interest and its environment are.

Thus, one first essential step in the disambiguation process is the explicit identification of the system of interest and the explicit characterisation of its nature and boundaries, and of the environment within which it is placed and with which it interacts.

From the requirements, a human reader can guess that *k3* is about an *information system* and can also identify a number of elements constituting its environment, in particular human users (e.g., *program administrators* and *nursing staff members*) and human organisations (e.g., the *Nursing Department*). That interpretation could be modelled in FORM-L as follows:

```
1 Model K3Data begin
2   main Object system "An information management system";
3
4   external Class User "of the system";
5   external User{} programAdministrators;
6   external User{} nursingStaffMembers;
7   external Object nursingDepartment;
8 end K3Data;
```

Keyword *main* in line 2 indicates that the item (an object or a class) being declared (here, object `system`) is the item of interest of the model. Similarly, keyword *external* in lines 4, 5, 6 and 7 indicates that the instances of class *User*, and objects `programAdministrators`, `nursingStaffMembers` and `nursingDepartment` are not part of the system of interest but of its environment.

Curly brackets `{}` in lines 5 and 6 indicate that the objects being declared are sets (i.e., finite, unordered collections of instances of a given class). Here, `programAdministrators` and `nursingStaffMembers` are both sets of instances of class *User*.

That is only one of the possible interpretations: another would be to consider that the system is not merely an information system but a socio-technical system that also encompasses human users and organisations. The modelling would then be as follows:

```
1 Model K3Data begin
2   Class User "of the information system";
3   main Object system "A socio-technical system" begin
```

```
    Object  informationSystem;
    User{} programAdministrators;
    User{} nursingStaffMembers;
    Object  nursingDepartment;
    ...
  end system;
end K3Data;
```

In the following, it is the first interpretation that is retained.

That point being made, it is also not clear whether the information system is for the exclusive use of the *Nursing Department* or is a *University*-wide information system, with *k3data.txt* and *k3ucs.txt* specifying only the *Nursing Department* requirements. In the following, it is the second option that is retained.

3.2 Lexical Ambiguity

Lexical ambiguity occurs when the exact meaning of some terms denoting notions important to the application being considered is unclear. Such a term can be a single word, e.g., a noun, a verb, an adjective or an adverb. It can also be an expression, e.g., a nominal group (i.e., a noun with one or more adjectives) or a verbal group (i.e., a verb with one or more adverbs).

For human readers having at their disposal only the requirements of *k3data.txt* and *k3ucs.txt*, and thus lacking the background context of the *k3* case study, understanding the precise meaning of many of the terms used and how they relate to the actual world and to each other is not obvious.

3.2.1 Term *Class*

In the *k3* case study, term *class* appears multiple times. Even when restricted to an educational context, according to the *Merriam-Webster* dictionary, it could mean:

- A body of students meeting regularly to study the same subject (e.g., 'several students in the class are absent today'). That would be modelled as:

```
Class Student;
Class _Class "Group of students" extends Student{};
```

  An underscore is used in identifier `_Class` to distinguish it from FORM-L keyword `Class`. One can also note the pair of curly brackets `{}` indicating that the class being extended is 'set of `Student`s'.

- A period during which such a body meets (e.g., 'there shall be no class this morning'). That would be modelled as:

```
Class Date;
Class _Class begin
  Date      beginning "When it begins";
  Duration  _duration "How long it lasts";
  Student{} students  "of the class";
end _Class;
```

  An underscore is used in identifier `_duration` to distinguish it from FORM-L keyword `duration`.

- A course of instruction (e.g., 'they are doing well in their algebra class'). That would be modelled as:

```
Class _Class "Course of instruction" begin
  String subject "of the course";
end _Class;
```

- A set of students or alumni whose year of graduation is the same (e.g., 'the class of 1999'). That would be modelled as:

```
1 Class Year;
2 Class Student begin
3   Year graduationYear;
4 end Student;
5
6 Student{} class (Year y) "Students graduating the same year"
7   is all x of Student such that x.graduationYear = y;
```

In the framework of an expression, the name of a FORM-L class stands for the implicitly declared set of all its instances. Thus, in line 7, `Student` denotes the set of all instances of class `Student`.

3.2.2 Term *Cohort*

Other terms have no clear, generally accepted meaning in an educational context. For example, also according to *Merriam-Webster*, term *cohort* could mean:

- Companion, colleague.
- A group of individuals having a statistical factor (such as age or class membership) in common in a demographic study.
- One of 10 divisions of an ancient Roman legion.
- A group of warriors or soldiers.

It is only by reading and carefully analysing the complete set of requirements speaking of *cohort*:

> *d6: Classes for a given cohort shall not conflict with regards to the time and day that they are offered.*
>
> *r2: Program Administrators and Nursing Staff Members shall be able to add a new cohort to the system identified by start month and year.*
>
> *r6: The system shall be able to display a report of needed classes for a given quarter for all cohorts of all programs for Program Administrators/Nursing Staff Members planning purposes.*
>
> *r15: The system shall be able to display a printable summary for individual cohorts which will include the students enlisted, the Program of study, sequence of classes, cohort progress through the program, and timeline of completion.*
>
> *r18: The system shall be able to display a printable summary for individual nursing students, which will include (but not be limited to) student name, student ID, admission date, classes, credits, GPA, and the cohort that the student is enrolled in.*
>
> *r20: The system will notify affected parties when changes occur affecting cohorts, including but not limited to changes to the sequence for a cohort's program of study and changes to a given week's schedule (lab cancelled this week due to instructor illness).*
>
> *r22: Program Administrators/Nursing Staff Members shall have the ability to modify information relating to cohorts, including cohort identifier, program of study, preferred sequence of classes and quarters that a cohort will be taking specific classes.*

that one can surmise that a *cohort* is a group of *students* following the same *program of study* at the same time.

3.2.3 Term *Student*

Some nouns sometimes appear with one or more adjectives, sometimes without any. For example, noun *student* appears in many requirements without any adjective, but some others use nominal group *nursing students*. Is the set of *nursing students* the same as the set of *students*? The FORM-L data model would then be:

```
1 Class Student;
```

Alternatively, *nursing students* could be a strict subset of *students*, some *students* not being *nursing students*. The FORM-L data model would then be:

```
1 Class Student;
2 Class NursingStudent extends Student;
```

The choice needs to be made in consistency with deciding whether the system of interest is dedicated to the specific needs of the *Nursing Department* or not. As it was decided earlier that the system is a *University*-wide information system, we have to consider that not all *students* are *nursing students*.

3.2.4 Actual World vs. Information On It

Another lexical ambiguity concerns how a term relates to the real, physical world. For example, term *student* could refer to an actual human being, like in:

> *d9: The system shall contain contact information for all people relevant to the system, including students, ...*

However, it could also refer to the information the system keeps on a student, such in:

> *r10: The system shall allow a Program Administrator or Nursing staff member to remove a student from a clinical lab section.*

In FORM-L, *determiners* can be used to fix such ambiguities, as in the following:

```
1 external Class #User
2   "Actual human being, in the environment of, and user of, the system";
3 Class ProgramAdministrator extends User;
4
5 Class @Person
6   "Information on people relevant to the system";
7 Class Student extends Person;
```

In line 1, the # determiner indicates that class `User` and its instances represent actual human beings. Keyword `external` indicates that the instances are not part of the system but of its environment. In line 3, class `ProgramAdministrator` 'inherits' the determiner of `User`.

In line 5, the @ determiner indicates that class `Person` and its instances do not represent actual human beings but information the system has on human beings. In line 7, class `Student` 'inherits' the determiner of `Person`.

3.3 Syntactic Ambiguity

Syntactic ambiguity occurs when a natural language sentence can be grammatically parsed in different ways. For example, in "*Alice took a photo of Bob with his dog*", it is clear that the dog is Bob's, and common sense tells us that "*Alice took a photo of (Bob with his dog)*" is more likely than "*Alice (took a photo of Bob) with his dog*". On the contrary, in '*Andrew took a photo of Bob with his dog*', it is not clear whether the dog is Andrew's or Bob's, and without any additional information, both options are equally plausible.

Syntactic ambiguities in the *k3* requirements are less numerous than lexical ones and generally easier to fix with some common sense. For example, in:

d4: A Program of Study shall consist of a program name and listing of required classes (both clinical and non-clinical) that must be completed.

common sense tells us that "*program name and listing of required classes that must be completed*" should be parsed as "*(program name) and (listing of (required classes) that must be completed)*", and not, for example, as "*((program name) and (listing of ((required classes)) that must be completed*".

However, not all syntactic ambiguities in *k3* are that easy to fix. For example, in:

r1: Program Administrators and Nursing Staff Members shall be able to add clinical classes or sections to a sequence of classes.

"*clinical classes or sections*" could be parsed as "*(clinical classes) or sections*" or as "*clinical (classes or sections)*". Both options are plausible, and choosing one or the other might depend on decisions made regarding term *section* (see Section 4.1.2).

3.4 Semantic Ambiguity

Semantic ambiguity occurs when the exact meaning of commonly used words, which generally do not need to be explicitly defined, is unclear within the framework of a given sentence. For example, in "*Alice and Bob shall be able to do it*", the word *and* can be interpreted either as "*Alice and Bob together*" or as "*each of them individually*".

It also sometimes occurs with the use of plural or singular.

For example, in "*Program Administrators shall be able to ...*", even when there is a clear definition of term "*Program Administrator*", the plural form could be interpreted in different manners: it could mean "*Each individual Program Administrator*", "*Some individual Program Administrators*", or "*All Program Administrators together, as a whole, single body*" (like in "*Every four years, citizens shall be able to elect the president of the nation*"). After initial declaration:

```
Class PA "Program Administrators";
```

the modelling in FORM-L of the three options would be, respectively:

```
Requirement r1 is
  for all x of PA
  assert x can achieve ...;
```

```
Requirement r1 is
  for some x of PA
  assert x can achieve ...;
```

```
Requirement r1 is
  assert PA can achieve ...;
```

Similarly, "*A Program Administrator shall be able to ...*", could mean "*One particular Program Administrator shall be able to ...*", or "*Each individual Program Administrator shall be able to ...*".

The fixing of semantic ambiguity may imply the splitting of an original requirement into two or more requirements, as is the case for *k3* requirements *r1* (see Section 4.2.6) and *r20* (see Section 4.2.8).

3.5 Silence

Silence occurs when one or more pieces of information essential to precise, unambiguous requirements are missing. For example, in '*When event E occurs, the system shall perform action A*', no timing is specified, even though no system can react instantaneously. Good practice dictates that *response time limits* should be specified. Similarly, in '*The temperature shall not exceed 50 degrees Celsius*', *where* in space that temperature is taken is not specified.

Some level of silence might be acceptable in natural language requirements, for which one expects and can tolerate a certain level of informality (only up to a certain point, though). That is absolutely not the case in formal requirements, for which one expects utmost precision: otherwise, what would be the purpose of fuzzy formal requirements?

In the *k3* case study, no time limits are given in any of the requirements specifying an action or an outcome required of the system. Such limits are necessary in formal requirements. Otherwise, during validation tests, system developers can always pretend that one just has to wait a little more to observe the required action or outcome.

Other key issues in the *k3* case study are also left unaddressed, such as sizing (how many items of each kind, e.g., *classes*, *students*, ..., the system must be able to handle). However, as they will need new, additional requirements that one can assume will be specified later in the engineering process, they will not be discussed here.

4 Formal Modelling of the *k3* Case Study

The *k3* requirements are formalised in two FORM-L models:

- `K3Data`.
- `K3Requirements`.

Note: Sometimes, an identifier is used before it is declared. This is, for example, the case of `programOfStudy` , which is used in line 7 of the box in Section 4.1.3, but is declared only later, in the definition of *`Cohort`* in Section 4.1.6. Such forward references are allowed in FORM-L, and one of the missions of support tools is to verify that each identifier is used within the scope of an appropriate declaration.

4.1 The K3Data Model

The `K3Data` model, in file *K3Data.fml*, is the top-level model: it provides an interpretation for the nouns and nominal groups used in the *k3* requirements in the form of a consistent set of class and object declarations and definitions, which will be used in the formal modelling of the requirements.

4.1.1 Overview

The `K3Data` model is declared and defined as follows:

```
1 Model K3Data extends Standard begin
2   partial Model CoreClassesDecl "Declaration of k3 core classes";
3   partial Model CoreObjectsDecl "Declaration of k3 core objects";
4   partial Model AuxClassesDecl  "Declaration of k3 auxiliary classes";
5   partial Model FunctionsDecl   "Declaration of k3 ancillary functions";
6   partial Model CoreClassesDef  "Definition  of k3 core classes";
7 end K3Data;
```

In the engineering of *cyber-physical systems*, it is necessary to represent physical quantities (e.g., time, length, mass, etc.) according to a system of units. In line 1, `Standard` is a pre-existing model that declares and defines the classes and constants supporting the SI International System of Units [BIP19]. It also defines commonly used monetary units, basic physical and mathematical constants, and commonly used mathematical operators. Even though the *k3* system is an *information system*, its formal requirements need to refer to time and thus need such a model.

As seen in Section 3.2, many of the nouns and nominal groups at the core of the *k3* case study suffer from *lexical ambiguity*: for each, a general dictionary like *Merriam-Webster* gives several possible interpretations. The exact meaning of the specified requirements (their semantics) depends in large part on the one chosen for each term.

In `K3Data`, one plausible interpretation has been chosen for each term based on all the clues that could be found in the complete set of requirements. That proved to be a rather difficult task, and it is doubtful that an automatic tool could do it by itself.

The data model is divided into several partial models.

4.1.2 Declaration of the *k3* Core Classes

Partial model `CoreClassesDecl` just *declares* the *classes* associated with some of the nouns at the heart of the *k3* case study. The other nouns at the heart of the case study are associated with *objects* (see Section 4.1.3). In FORM-L, a *declaration* just states the existence and nature of an item and possibly whether a class is an extension of other classes, whereas a *definition* specifies its contents and/or its behaviour. A *natural language comment* is attached to some of the classes (red text enclosed in double quotes) to indicate how it was interpreted.

Term *Section*

The *k3* requirements use term *clinical lab section* 9 times, *lab section* once, *clinical section* once, and *section* alone 3 times. It is not clear whether *section* is always used with the same meaning, in particular in requirement *r25*, which speaks of *department/section*. *r25* aside, it is also not clear (*lexical ambiguity*, see Section 3.2) whether:

- The four terms represent the same notion, *lab section*, *clinical section* and *section* being just short cuts for *clinical lab section*.
- Each term represents a different, specific notion. If so, one can surmise that *section* is a super notion to the other three, but what would be the difference between these other three, and how would they be related?
- Term *section* represents one notion, and the other three a unique extension notion, *clinical section* and *lab section* being just shortcuts for *clinical lab section*.

Other interpretations are also possible. In the following, it is the third option that has been retained as it was deemed the most plausible in the absence of any other information. It is also consistent with the decision to consider the system as a *University*-wide information system.

The exact meaning of *section* still needs to be clarified. Among the many interpretations proposed by the *Merriam-Webster* dictionary for word *section*, only one makes sense in an educational context and for the *k3* requirements: *one of the classes formed by dividing the students taking a course*. Thus, the formal modelling could be:

```
1 Class @Section
2   "One of the groups formed by dividing the students registered for a given
3    class. Members of the same section attend the same training sessions for
4    that class"
5   extends Student{};
6 Class ClinicalLabSection extends Section;
```

The second sentence of the comment is based on the wording of requirement *d7*:

> *d7: A clinical lab section shall include the clinical site name, the class instructor, day and time of the lab.*

However, that wording is itself marred by *lexical ambiguity*. In particular, *day* could be interpreted in many different ways:

- A day of the week (e.g., Monday).
- A day of the month (e.g., the 20th of each month).
- A specific day of a specific year (e.g., February 20th, 1998).

The above interpretation for *section* is consistent with the first interpretation for *day*, but one can note that in *d7*, *day* is singular, which excludes two or more training sessions per week for a given class. That constraint seems unnecessary and unlikely in real life. And requirement *r1* sows additional doubts:

> *r1: Program Administrators and Nursing Staff Members shall be able to add clinical classes or sections to a sequence of classes.*

An alternative interpretation for *section* could be based on the third interpretation for *day* and be as follows:

```
Class @Section
  "Training session for a class held at a specific date (day and time)";
Class @_Class
  "Set of sections constituting a complete course on a given topic";
```

In the following, it is that interpretation that is retained.

Term *Class*

As seen previously, term *class* has many possible interpretations (*lexical ambiguity*, see Section 3.2). Based on the wording of some of the requirements, such as requirement *r5*:

> *r5: Program Administrators and Nursing Staff Members shall have the ability to specify which classes are required for a Program of Study.*

it is interpreted as *a course of instruction on a given subject* (e.g., algebra, anatomy or literature). The teaching of such a course of instruction is spread over multiple sessions, each of which which, as discussed just above, is taught to a *section*.

The set of *classes* that a student must complete to obtain a degree is a *program of study*.

That course of instruction is taught periodically (e.g., every year, quarter or semester) for new incoming groups of students: a group of students following the same *program of study* at the same time is called a *cohort*.

Model

Other core terms of the *k3* case study are also marred by *lexical ambiguity*, but a similar interpretation approach can be applied. As a detailed discussion for each would be tedious, only the resulting class declarations are presented here. For some, an indication of which requirements were at the origin of the interpretation retained is given in the comments.

```
CoreClassesDecl begin
  abstract Class @Section                          // r1, d7
    "Training session for a class held at a specific date (day and time)";
  Class Lecture             extends Section;       // d2
  Class ClinicalLabSection  extends Section;

  abstract Class @_Class
    "Set of sections constituting a complete course on a given topic";
  Class NonClinicalClass extends _Class;           // d1
  Class ClinicalClass    extends _Class;           // d1

  Class @ProgramOfStudy                            // d4
    "Set of classes required to obtain a degree";

  Class @SequenceOfClasses                         // r1
    "Set of classes, all being taught during the same specified quarter.
     Can also include sections not belonging to any class";
```

```
18
19    abstract Class @Person                              // d9
20      "Information in the system about a person relevant to the system";
21    Class Student      extends Person;                  // d9
22    Class Instructor   extends Person;                  // d9
23    Class StaffMember  extends Person;                  // d9
24    Class ClinicalSiteAdministrator extends Person;     // d9
25
26    Class @Cohort                                       // r18
27      "Set of all students registered to the same Nursing program of study,
28       for the same quarter"
29      extends Student{};
30
31    Class @Department "or Department Section";
32
33    Class @ClinicalSite;
34
35    external abstract Class #User "of the system";
36    Class PA "Program Administrators" extends User;
37    Class NSM "Nursing Staff Members" extends User;
38  end CoreClassesDecl;
```

4.1.3 Declaration of the *k3* Core Objects

Now that the core classes are declared, we can declare the objects that are part of the *k3* system of interest.

```
1   CoreObjectsDecl begin
2     main Object #system "A University-wide information management system" begin
3       Department       nursingDepartment "One of the Departments";
4       ProgramOfStudy{} nursingPrograms "Programs of the Nursing Department";
5       Cohort{}         nursingCohorts
6         "Cohorts following a program of study of the Nursing Department"
7         is all x of Cohort such that x.programOfStudy in nursingPrograms;
8       Student{}        nursingStudents
9         is all x of Student such that x.cohort in nursingCohorts;
10    end system;
11  end CoreObjectsDecl;
```

The *k3* natural language requirements are given without any introductory text indicating what the system of interest is and what constitutes its environment. To fix this *system ambiguity* (see Section 3.1), an object named `system` is declared and then defined: all other *k3* core objects are by default *features* of `system`.

Regarding this combined declaration and definition:

- In line 2, keyword *main* indicates that `system` is the object of interest of the model.
- Also in line 2, *Object* is a predefined class built in the FORM-L language. It specifies no features. An instance of a class has all the features and behaviour elements specified by that class (here, none) but can also specify features and behaviour elements of its own, like those declared in lines 3 to 9.
- As the definition block for `system` declares more than one feature, they are enclosed within a *begin end* pair (lines 2 and 10).
- These features are those that are known at that stage of the engineering process. Later, as more will be known about it, some *extension models* can *refine* that definition and add new features.

- `nursingCohorts` (in line 5) and `nursingStudents` (in line 8) are sets that are completely defined by a single expression, introduced by keyword *is*.

Sets are a key ingredient of the FORM-L language:

- Operator *in* in lines 7 and 9 stands for symbols $\in$ and $\subset$ of set theory.
- Line 7 is the FORM-L equivalent of mathematical expression $\{x \in Cohort | x.programOfStudy \in nursingPrograms\}$
- Similarly, line 9 is the FORM-L equivalent of mathematical expression $\{x \in Student | x.cohort \in nursingCohorts\}$

4.1.4 Declaration of *k3* Auxiliary Classes

Other nouns and nominal groups used in the expression of *k3* requirements have a lesser importance in the understanding of the case study. Most of them are simply declared in this first initial model. Keyword *deferred* means that their definition must and will be provided in some future *extension* models addressing implementation issues.

```
1  AuxClassesDecl begin
2    Class @Information "in a report"                                  is deferred;
3    Class @RoomType                                                   is deferred;
4    constant Class @Name "of an item managed by the system"           is deferred;
5    constant Class @Id "of an item managed by the system"             is deferred;
6    Class @Credit "of a student"                                      is deferred;
7    Class @GPA "of a student, whatever that is"                       is deferred;
8    constant Class @Need "of an item managed by the system"           is deferred;
9    Class @Date
10     "from which one can derive year, month, day of month and time of day"
11                                                                     is deferred;
12   Class Month "First day of a month at 00:00"     extends Date is deferred;
13   Class Year "First day of a year at 00:00"       extends Date is deferred;
14   Class Quarter "First day of a quarter at 00:00" extends Date is deferred;
15
16   Class @EMail                                                      is deferred;
17   Class @PhoneNumber                                                is deferred;
18   Class @Address                                                    is deferred;
19
20   Class @EReport "issued on request"          extends Event     is deferred;
21   Class @ENotification "issued to warn users" extends Event     is deferred;
22 end AuxClassesDecl;
```

The *value* attribute of an *Event* indicates when a fact of interest (the duration of which is neglected) has occurred. *Event extensions* like *EReport* in line 20 and *ENotification* in line 21 can specify features providing additional information for each occurrence.

4.1.5 Declaration of *k3* Ancillary Functions

The following declares three ancillary functions that will be used in the definition of some of the *k3* core classes and in the formalisation of some of the requirements.

```
1  FunctionsDecl begin
2    Duration quarterDuration (Quarter q) is deferred;
```

In line 2, class *Duration* is defined in the `Standard` model as an *extension* of built-in class *Real* to support the notion of time.

```
1   Boolean overlap (Date date1, Duration d1, Date date2, Duration d2)
2     "Whether two time intervals overlap"
3     is deferred;
```

```
1   Boolean inQuarter (Date d, Quarter q) is
2     overlap (d, 1*s, q, quarterDuration (q));
3 end FunctionsDecl;
```

In line 2, *s* is a constant *Duration* of 1 second defined in the Standard model.

4.1.6 Definition of the *k3* Core Classes

These definitions declare and sometimes define the features of each of the *k3* core classes and also of some auxiliary classes, as could be inferred from the analysis of the requirements.

```
1 CoreClassesDef begin
2   Class _Class begin
3     Name name;                  // d2, d3
4     Id id;                      // r17
5     Need{} lectureRoomNeeds; // d2, d3
6     Section{} sections is all x of Section such that x.class = me;
7     Student{} students is all x of Student such that me in x.classes;
8   end _Class;
```

In lines 6 and 7, keyword *me* denotes the object being defined, or, in a class definition, as is the case here, the instance of the class.

```
1   NonClinicalClass begin
2     Need{} instructorNeeds; // d2
3   end NonClinicalClass;
```

```
1   ClinicalClass begin         // d3
2     Need{} clinicalSiteNeeds;
3     Need{} lectureInstructorNeeds;
4     Need{} clinicalLabInstructorNeeds;
5   end ClinicalClass;
```

```
1   Section begin               // d7
2     _Class       class "to which the section belongs";
3     Date         beginning "When it begins";
4     Duration     @_duration "How long it lasts";
5     Student{}    students "registered to the section";
6     Information contactInformation;
7   end Section;
```

In line 4, feature *_duration* has a determiner because *Duration* is predefined in the Standard model without one.

```
1   ClinicalLabSection begin
2     Name          name;
3     Instructor    instructor; // d7
4     ClinicalSite site;
5     Department    department;
6   end ClinicalLabSection;
```

```
ProgramOfStudy begin   // d4
  Name      name;
  _Class{} requiredClasses;
end ProgramOfStudy;
```

```
SequenceOfClasses begin
  Section{} sections;
  _Class{}  classes is sections.class;
end SequenceOfClasses;
```

```
Cohort begin
                     value is all x of Student such that x.cohort = me;
  Name               name;
  Id                 id;
  ProgramOfStudy     programOfStudy;
  Month              startMonth;
  SequenceOfClasses  prefSequence;
  Guard              consistency1 is
    ensure prefSequence.classes = programOfStudy.requiredClasses;
  Guard              consistency2 is
    ensure AND (prefSequence.quarters >= startMonth);
end Cohort;
```

In lines 8 and 10, the *Guard*s are properties which, when violated, indicate that the model is no longer valid.

In line 11, *AND* is an operator that applies to a set of Booleans: it returns the *and* of all the members of that set.

```
Person begin
  Name         name;
  Id           id;
  EMail        eMail;       // d9
  PhoneNumber phoneNumber;// d9
end Person;
```

```
Student begin
  Date                  admissionDate;
  Boolean               @partTime: default is false;
  Cohort                cohort "of the student";
  SequenceOfClasses     sequenceOfClasses:
    default is cohort.preferredSequence;
  _Class{}              classes "of the student" is sequenceOfClasses.classes;
  ClinicalClass{}       clinicalClasses is classes inter ClinicalClass;
  NonClinicalClass{}    nonClinicalClasses is classes inter NonClinicalClass;
  Section{}             sections is all x of classes.sections
                                    such that me in x.students;
  ClinicalLabSection{} clinicalLabSections is
    sections inter ClinicalLabSection;
  Lecture{}             lectures is sections inter Lecture;
end Student;
```

```
Department begin
  _Class{} classes "offered by this Department";
```

```
3   end Department;
```

```
1   ClinicalSite begin          // d8
2     Name         name;
3     Person       contactPerson;
4     Information  contactInformation;
5     Address      address;
6   end ClinicalSite;
```

```
1   EReport begin
2                  occurrence is specific;
3     Information{} contents   is specific;
4   end EReport;
```

```
1   ENotification begin
2              occurrence               is specific;
3     Person{} persons "to be notified" is specific;
4   end ENotification;
5 end CoreClassesDef;
```

4.2 The K3Requirements

The `K3Requirements` model, in file *K3Requirements.fml*, extends the `K3Data` model and formally specifies each of the 9 + 26 requirements found in *k3data.txt* and *k3ucs.txt*.

4.2.1 Requirements Model Overview

The overall structure of the `K3Requirements` model is as follows:

```
1  Model K3Requirements extends K3Data begin
2    refined system begin
3      // k3data Requirements
4      partial Model D1;
5      partial Model D2;
6      ...
7      partial Model D9;
8
9      // k3ucs Requirements
10     partial Model R1;
11     partial Model R2;
12     ...
13     partial Model R26;
14   end system;
15
16   constant Duration !shortDelay      is deferred;
17   constant Duration !reasonableDelay is deferred;
18 end K3Requirements;
```

Line 1 specifies that model `K3Requirements` is an *extension* of model `K3Data`: that means that `K3Requirements` includes all the contents of `K3Data` and then adds new modelling elements of its own. Here, the main addition is a refinement of `system`, the object of interest of `K3Data`, which is by default also the object of interest of `K3Requirements`.

A *partial Model* is declared for each of the requirements. Partial models are used to organise large models or large definitions (as it is the case here) into more manageable subsets.

In line 16, *constant Duration* !shortDelay is declared. It will be used in the formalisation of many requirements to specify a time limit within which a required action must be completed and thus to fix some cases of *silence* (see Section 3.5). The ! determiner indicates that even though shortDelay is *deferred* for now, it will be given a definite value during design.

Similarly, in line 17, *constant Duration* !reasonableDelay is declared. It will be used to specify a time limit in the formalisation of requirements mandating notifications to system users.

The following presents only a few selected requirements: presenting them all would be very tedious since only a limited number of *formalisation patterns* have been applied.

4.2.2 Requirement *d1*

d1: A class shall be either a non-clinical class or a clinical class.

Ambiguities

Lexical ambiguity affecting term *class* has already been fixed in Section 4.1.2.

Modelling

This requirement is modelled in a partial model D1.

```
1 system.D1 begin
2   Requirement d1
3     "A class shall be either a non-clinical class or a clinical class"
4     is ensure ClinicalClass union NonClinicalClass = _Class
5           and ClinicalClass inter NonClinicalClass = {};
6 end system.D1;
```

The formal definition of requirement *d1* is straightforward and can be explained as follows:

- It starts by declaring a *Requirement* bearing the same name (*d1*) and with the original natural language text as a comment. That will be so for each of the *k3* requirements.
- As no *temporal locator* is specified, the *constraint* must be satisfied at all times.
- In the framework of an expression, the *name of a class* denotes the set of all its instances, as is the case in lines 4 and 5 for *ClinicalClass*, *NonClinicalClass* and *_Class*.
- In line 5, the curly brackets pair denotes the empty set.

4.2.3 Requirement *d2*

d2: A non-clinical class shall specify the course name, lecture room requirements and instructor needs.

Ambiguities

Lexical ambiguity affecting term *class* has already been fixed in Section 4.1.2. Lexical ambiguity also affects term *course*, which appears only in requirements *d2* and *d3*: it was decided to consider that a *course* is a *class*.

Modelling

This requirement is modelled in a partial model `D2`.

```
1  system.D2 begin
2    Requirement ~d2
3      "A non-clinical class shall specify the course name, lecture room
4       requirements and instructor needs"
5    : concretisation is {dd2};
6
7    Requirement ~dd2
8    // Implemented in class definitions
9      "A non-clinical class shall specify the name of the class, the needs of the
10      lecture room of the class, and the needs of the instructor of the class";
11 end system.D2;
```

The ~ determiner in line 2 indicates that requirement `d2` will not be formalised directly. Here, it is because it needs first to be clarified. The `concretisation` attribute in line 5 indicates that the clarification is provided by requirement `dd2`, the natural language comment of which adds precision to the one of `d2`.

The ~ *determiner* in line 7 indicates that `dd2` will not be formalised either. Here, it is because that is done directly in the data model: see, in Section 4.1.6, lines 3 and 5 of the definition of `_Class`, and line 2 of the definition of `NonClinicalClass`.

4.2.4 Requirement *d5*

> *d5: The report of needed classes shall include (but not be limited to) classes to be offered, number of sections needed, number of labs needed, and room types needed.*

Ambiguities

Lexical ambiguity affecting terms *class* and *section* has already been fixed in Section 4.1.2. Lexical ambiguity also affects term *lab*, which appears only in requirements *d5* and *r16*. Term *clinical lab* also appears in requirement *r17*. As there is not enough information in the complete *k3* case study to make a decision, the fixing will need to be made in a future *extension model* that will provide a definition for `infoOnNbLabsNeededByNursingDep`, which is declared as `deferred` in line 18.

The requirement also suffers from *system ambiguity* (see Section 3.1): it is *a priori* not clear which entity is concerned by the report, the *Nursing Department* or the *University*. Here, as it is considered that the system of interest is a *University-wide information system*, but that the specified requirements address only the needs of the *Nursing Department*, it was decided that the report concerns the *Nursing Department*.

Modelling

This requirement is modelled in a partial model `D5`.

```
1  system.D5 begin
2    Requirement ~d5
3      "The report of needed classes shall include (but not be limited to)
4       classes to be offered, number of sections needed, number of labs
5       needed, and room types needed"
6    : concretisation is {dd5};
7
8    Requirement ~dd5
9      "The report of needed classes shall include (but not be limited to)
10      classes to be offered by the Nursing Department, with the corresponding
11      number of sections needed, number of labs needed, and room types needed"
12   : substitution is dd5s;
```

```
13
14    Requirement dd5s
15      "The system shall be able to produce, when requested, a report of needed
16       classes that shall include (but not be limited to) classes to be offered
17       by the Nursing Department, with the corresponding number of sections
18       needed, number of labs needed, and room types needed";
19
20    external Event @eReqForReportOnNeededClasses is deferred;
21
22    EReport        eReportOnNeededClasses        is deferred;
23
24    Information    infoOnClassesToBeOffered      is deferred;
25    Information    infoOnNbSectionsNeeded        is deferred;
26    Information    infoOnNbLabsNeeded            is deferred;
27    Information    infoOnRoomTypesNeeded         is deferred;
28
29    dd5s is
30      after eReqForReportOnNeededClasses within shortDelay
31      ensure one eReportOnNeededClasses
32             such that {infoOnClassesToBeOffered,
33                        infoOnNbSectionsNeeded,
34                        infoOnNbLabsNeeded,
35                        infoOnRoomTypesNeeded} in contents;
36  end system.D5;
```

Concerning the reason of being of `dd5s`, see discussion in Section 5.2.4. Its formalisation can be explained as follows:

- In line 20, `eReqForReportOnNeededClasses` is the event that signals requests for a new report on needed classes. Keyword *external* indicates that it is not issued by the system of interest, but by its environment.
- In line 22, `eReportOnNeededClasses` is the event that marks the issuance of the requested report.
- Lines 24 to 27 declare the *Information* pieces to be included *a minima* in the report.
- Lines 30 to 31 require a new issuance of the report within a short delay after each request.
- Lines 32 to 35 specify that the `contents` of each occurrence of the report (`contents` is a feature of `eReportOnNeededClasses` that is a set of *Information*) shall include the pieces of information declared in lines 24 to 27.

This formalisation is consistent with the "*shall include (but not be limited to)*" part of the requirement.

4.2.5 Requirement *d6*

> *d6: Classes for a given cohort shall not conflict with regards to the time and day that they are offered.*

Ambiguities

Lexical ambiguity affecting terms *class* and *cohort* has already been fixed in Section 4.1.2. Lexical ambiguity also affects verb *conflict*. Here, it was decided to consider that it just means 'overlap in time', but that is probably too simplistic since when classes are given at different locations, students need time to move from one location to another.

Modelling

This requirement is modelled in a partial model `D6`.

```
1  system.D6 begin
2    Requirement ~d6
3      "Classes for a given cohort shall not conflict with regards to the time and
4       day that they are offered"
5    : concretisation is {dd6};
6
7    Requirement dd6
8      "The sequence of classes of any student of any cohort shall not
9       conflict with regards to the time and day that they are offered"
10     is for all x of Cohort
11        for all y of x
12        for all s1,s2 of y.sequenceOfClasses.sections
13        such that s1.identity <> s2.identity
14        ensure not overlap (s1.beginning, s1._duration,
15                            s2.beginning, s2._duration);
16 end system.D6;
```

In lines 10 and 11, "`for all x in Set`" is the FORM-L equivalent of the mathematical expression "$\forall x \in Set$". Similarly, in line 12, "`for all s1, s2 in Set`" is the FORM-L equivalent of mathematical expression "$\forall s1, s2 \in Set$".

In line 13, one compares the `identity` attributes of `s1` and `s2` to avoid checking the overlap of a `section` with itself.

4.2.6 Requirement *r1*

> *r1: Program Administrators and Nursing Staff Members shall be able to add clinical classes or sections to a sequence of classes.*

Ambiguities

Lexical ambiguity affecting terms *class* and *section* has already been fixed in Section 4.1.2, but as mentioned in Section 3.4, words *and* and *or*, and the use of *plural* for *Program Administrator*, *Nursing Staff Member*, *class* and *section*, introduce *semantic ambiguity* (see Section 3.4).

Also, as mentioned in Section 3.3, the requirement also suffers from *syntactic ambiguity*.

Lastly, the requirement is a *capability requirement*, whereby some human agents must be able to use the system of interest to perform a given task: however, as no time limit is specified (meaning that even if it takes years and decades to perform the task, the requirement would still be satisfied), the requirement suffers from *silence* (see Section 3.5).

Modelling

This requirement is modelled in a partial model `R1`.

```
1  system.R1 begin
2    Requirement ~r1
3      "Program Administrators and Nursing Staff Members shall be able to add
4       clinical classes or sections to a sequence of classes"
5    : concretisation is {rr1a, rr1b, rr1c, rr1d};
6
7    Requirement rr1a
8      "Each Program Administrator, each Nursing Staff Member shall be able,
9       within a short delay, to add new sections to the system";
10   Requirement rr1b
11     "Each Program Administrator, each Nursing Staff Member shall be able,
```

```
12      within a short delay, to add new sequences of classes to the system";
13    Requirement rr1c
14      "Each Program Administrator, each Nursing Staff Member shall be able,
15       within a short delay, to add any clinical classes already in the
16       system to any sequence of classes also already in the system"
17    Requirement rr1d
18      "Each Program Administrator, each Nursing Staff Member shall be able,
19       within a short delay, to add any sections already in the system to
20       any sequence of classes also already in the system";
21
22    // rr1a
23    external Event eAddSections
24      "Decision to add new sections to the system"
25    begin
26      occurrence is deferred;
27      Section {} sections "to be added";
28    end eAddSections;
29
30    rr1a is
31      for all x of PA union NSM
32      after eAddSections within shortDelay
33      assert x can achieve bop.sections in Section;
34
35    // rr1b
36    external Event eAddSequencesOfClasses
37      "Decision to add new sequences of classes to the system"
38    begin
39      occurrence is deferred;
40      SequenceOfClasses{} sequencesOfClasses "to be added";
41    end eAddSequencesOfClasses;
42
43    rr1b is
44      for all x of PA union NSM
45      after eAddSequencesOfClasses within shortDelay
46      assert x can achieve bop.sequencesOfClasses in SequenceOfClasses;
47
48    // rr1c
49    external Event eAddClinicalClassesToSequence
50      "Decision to add clinical classes to a sequence of classes"
51    begin
52      occurrence is deferred;
53      ClinicalClass {} clinicalClasses "to be added"
54      : Guarantee g1 is when occurrence ensure me in ClinicalClass;
55      SequenceOfClasses sequenceOfClasses
56      : Guarantee g1 is when occurrence ensure me in SequenceOfClasses;
57    end eAddClinicalClassesToSequence;
58
59    rr1c is
60      for all x of PA union NSM
61      after eAddClinicalClassesToSequence within shortDelay
62      assert x can achieve bop.clinicalClasses in bop.sequenceOfClasses.classes;
63
64    // rr1d
65    external Event eAddSectionsToSequence
66      "Decision to add sections to a sequence of classes"
67    begin
```

```
68      occurrence is deferred;
69      Section{} sections "to be added"
70      : Guarantee g1 is when occurrence ensure me in system.sections;
71      SequenceOfClasses sequenceOfClasses
72      : Guarantee g1 is when occurrence ensure me in system.sequencesOfClasses;
73    end eAddSectionsToSequence;
74
75    rr1d is
76      for all x of PA union NSM
77      after eAddSectionsToSequence within shortDelay
78      assert x can achieve bop.sections in bop.sequenceOfClasses.sections;
79  end system.R1;
```

Line 5 adds precision to requirement *r1* with the specification of two missing requirements (`rr1a` and `rr1b`) and two less ambiguous ones (`rr1c`, and `rr1d`). These are described in natural language in lines 7 to 20.

Strictly speaking, `rr1a` and `rr1b` are extra: they were deemed necessary because no other *k3* requirements address the creation in the system of *sections* and of *sequences of classes*, Whereas requirement *r3* does address the creation of *classes* (and thus of *clinical classes*), requirement *r8* addresses the creation of just *clinical lab sections*: with the interpretation retained for *sections* and *clinical lab sections*, and if we parse *clinical classes or sections* as *(clinical classes) or sections*, then `rr1a` is necessary.

Each of `rr1a`, `rr1b`, `rr1c`, and `rr1d` is then formally modelled following the same pattern:

- First, an *external Event* is declared and defined:
  - Keyword *external* indicates that it is not issued by the system of interest, but by its environment.
  - Each of its occurrences marks an intention and decision to add or modify some pieces of information in the `system`.
  - Its features specify the pieces of information to be entered and those to be modified.
- Then the requirement is formally modelled, specifying
  - *WHO* (*all* `x` *of PA union NSM*). Reminder: *PA* is the set of all *Program Administrators*, and *NSM* the set of all *Nursing Staff Members*. The modelling specifies that any of them is individually concerned by the *WHAT*.
  - *WHEN* (*after* `event` *within* `shortDelay`). This expression is a *continuous temporal locator* (*CTL*) that specifies possibly overlapping time periods. Each begins when a decision is made (as marked by the occurrences of the *event*) and lasts for `shortDelay`.
  - *WHAT* (*assert* `x` *can achieve* `condition`). Keyword *achieve* indicates that the requirement is an *achievement requirement*, where the required `condition` may not be satisfied at the beginning of the time periods specified by the *WHEN* part, but must become and remain so during each period.
    Expression *assert* `x` *can achieve* indicates that the requirement is also a *capability requirement*, specifying that there must exist at least one authorised behaviour of `x` such that the `condition` is achieved. In natural language, expressions such as "*x shall be able to*", "*x shall have the ability to*" or "*the system shall allow x to*", where *x* is *external* to the system of interest are hints that a requirement is indeed a capability requirement.
    Lastly, keyword *bop* (which stands for "*beginning of period*") denotes the event occurrence that triggers the beginning of the time period.

4.2.7 Requirement *r15*

r15: The system shall be able to display a printable summary for individual cohorts which will include the students enlisted, the Program of study, sequence of classes, cohort progress through the program, and timeline of completion.

Ambiguities

Lexical ambiguity affecting terms *cohort*, *student* and *class* has already been fixed in Section 4.1.2, but as mentioned in Section 3.4, the use of *plural* for *cohort* and *student* introduces *semantic ambiguity*. Also, the requirement is a *capability requirement*, whereby the system of interest must be able to perform a given task: however, as no time limit is specified (meaning that even if it takes years and decades to perform the task, the requirement would still be satisfied), the requirement suffers from *silence* (see Section 3.5).

Modelling

This requirement is modelled in a partial model `R15`.

```
1  system.R15 begin
2    Requirement ~r15
3      "The system shall be able to display a printable summary for individual
4       cohorts, which will include the students enlisted, the Program of study,
5       sequence of classes, cohort progress through the program, and timeline of
6       completion"
7    : concretisation is {rr15};
8
9    Requirement rr15
10     "The system shall be able to display a printable summary for any cohort
11      in the system."
12     "That summary shall include all the students enlisted in the cohort."
13     "That summary shall include the program of study of the cohort."
14     "That summary shall include the sequence of classes of the cohort."
15     "That summary shall include the progress of the cohort through its program
16      of study."
17     "That summary shall include the timeline of completion of the cohort.";
18
19   external Event @eRequestForCohortSummary (Cohort c) is deferred;
20
21   Information infoOnStudents (Cohort cohort) is deferred;
22   Information infoOnProgram  (Cohort cohort) is deferred;
23   Information infoOnClasses  (Cohort cohort) is deferred;
24   Information infoOnProgress (Cohort cohort) is deferred;
25   Information infoOnTimeline (Cohort cohort) is deferred;
26
27   EReport eReport is deferred;
28
29   rr15 is
30     after eRequestForCohortSummary within shortDelay
31     achieve eReport such that {infoOnStudents (bop.cohort),
32                                infoOnProgram  (bop.cohort),
33                                infoOnClasses  (bop.cohort),
34                                infoOnProgress (bop.cohort),
35                                infoOnTimeline (bop.cohort)
36                               } in contents;
37 end system.R15;
```

Here, one can see a *pattern* similar to, but slightly different from, the one used for requirement *r1*:

- In lines 2 to 6, the original requirement *r15* is first modelled as is, with a `~` determiner indicating that it will not be formalised directly but through a concretisation requirement named *rr15*.
- In lines 9 to 17, requirement *rr15* is declared with natural language comments that fix some of the ambiguities of *r15*.

- In line 19, external event `eRequestForCohortSummary` is declared and defined to mark the instants where the system is requested to display the printable summary. Its `cohort` parameter specifies for which *cohort* that summary is requested.
- In lines 21 to 25, the pieces of information that must be in the report are then declared. Their precise definitions are *deferred* to a future extension model.
- In line 27, event `eReport` (of class *EReport*) is declared to mark the instants where the system issues the printable summary.
- Finally, in lines 29 to 36, requirement *rr15* is formally defined with a *WHEN* part and a *WHAT* part.

4.2.8 Requirement *r20*

> *r20: The system will notify affected parties when changes occur affecting cohorts, including but not limited to changes to the sequence for a cohort's program of study and changes to a given week's schedule (lab cancelled this week due to instructor illness).*

Ambiguities

Lexical ambiguity affecting terms *cohort* and *lab* has already been fixed in Section 4.1.2, but there might be a possible lexical ambiguity with the word *change*: if a change to the program of study of a cohort concerns multiple elements of the program, should affected parties receive a single global notification or a notification for each elementary change? Here, for the sake of simplicity, it is the second option that has been retained.

Also, as mentioned in Section 3.4, the use of *plural* for *cohort* and *change* introduces *semantic ambiguity*.

Lastly, the requirement is an *achievement requirement*, whereby the system of interest must at certain times be able to perform a given task, but no time limit is specified (*silence*, see Section 3.5).

Modelling

This requirement is modelled in a partial model `R20`.

```
1  system.R20 begin
2    Requirement ~r20
3      "The system will notify affected parties when changes occur affecting
4       cohorts, including but not limited to changes to the sequence for a
5       cohort's program of study and changes to a given week's schedule (lab
6       cancelled this week due to instructor illness)"
7    begin
8      concretisation is deferred;
9      Guard !g20 is ensure {rr20a, rr20b} in concretisation;
10   end r21;
11
12   Cohort: ENotification eProgramOfStudyChanged is deferred;
13
14   Requirement rr20a
15     "The system shall notify all persons in the system affected by any change
16      in the program of study for any cohort in the system"
17     is for all x of Cohort
18        after x.programOfStudy mutates within reasonableDelay
19        achieve eProgramOfStudyChanged;
20
21   Cohort: ENotification eScheduleChanged is deferred;
22   Section{} sections is UNION Cohort.prefSequence.sections;
23
```

```
24    Requirement rr20b
25      "The system shall notify all persons in the system affected by any change
26       in the schedule for any cohort in the system"
27      is after (OR sections.beginning MUTATES) or (OR sections._duration MUTATES)
28         within reasonableDelay
29         achieve x.eScheduleChanged;
30  end system.R20;
```

A formalisation pattern similar to the one of *r1* is applied here, but one can note that:

- In line 8, attribute `concretisation` is not assigned a value but is subject to a constraint: it must contain `rr20a` and `rr20b`, but other concretisation requirements could be added later. This is to reflect the "*including but not limited to*" part of requirement `r20`.
- In lines 12 and 21, two notification events, `eProgramOfStudyChanged` and `eScheduleChanged` are declared for each instance of class `Cohort`.
- In line 18, post-fixed operator `mutates` is applicable to any object. It produces an event occurring each time the value or any feature of the object changes.
- In line 27, the post-fixed operator `MUTATES` is applicable only to sets of objects. It produces a set of events, one for each set member, occurring when the value or any feature of the set member changes.
- Also in line 27, pre-fixed operator `OR` is applicable only to a set of Booleans or a set of events: it performs a global `or` of all the set members.

5 Discussion

5.1 Lessons Learned for Tool-Assisted Diusambiguation

The tool assistance suggested in this Section for identifying and fixing ambiguities in requirements follows the classification presented in Section 3 and illustrated with the *k3* requirements. It applies to the *English* language. It could probably be extended to other *Indo-European* languages, *mutatis mutandis*, but that will need to be confirmed. For *non-Indo-European* languages, specific studies and possibly different approaches will be necessary.

5.1.1 System Ambiguity

Often, as in the case in the *k3* case study, requirements make use of term *system*, which most likely refers to the system of interest to which the requirements apply. However, is it used in exactly the same sense everywhere it appears? More often than not, requirements specification is a collective and collaborative endeavour involving multiple authors who may have slightly different understandings of what the system of interest is.

For example, in the case of the *k3* study, some authors may understand the system as a *University*-wide information system, whereas others may understand it as a *Nursing Department*-only information system.

In some cases, term *system* is not used, and one has to explicitly state what the system of interest is.

What is not part of the system of interest but of its environment needs also to be clearly identified, which is not the case in the *k3* case study. That information is important for a clear understanding of the boundaries of the system.

A tool, in interaction with requirements authors, could make sure that:

- There is one and only one noun or nominal group that is used to denote the system of interest.
- It has an explicit definition.

- All nouns or nominal groups used to denote entities of the system environment are identified as such.
- Each has an explicit definition.
- All other nouns or nominal groups are parts or features of the system of interest or of some entity of the environment.

5.1.2 Lexical Ambiguities

A noun may be associated with one or more *determining adjectives* (i.e., adjectives that profoundly affect its meaning). For example, an *agent* is usually a *person*, but a *chemical agent* is not: *agent* (as a *person*) and *chemical agent* are two very different notions.

A *noun* may also be associated with one or more *qualifying adjectives* (i.e., adjectives that just add nuance to the meaning of the noun, as in "*a young person*").

For nouns, a tool could do the following (questions being asked to and answered by human users):

- Identify each noun and nominal group occurring in the requirements. ML could help decide whether an *adjective* is a *determining* or a *qualifying adjective*, but in some cases, user input may be necessary.
- For each noun:
  - Do all its occurrences have the same meaning?
  - If not, how should each different meaning be named so that each noun has one and only one meaning?
  - Are there other nouns used with the same meaning?
  - If so, which unique noun should be used?
  - Does it represent a *core notion* of the application?
  - If so, it needs to be precisely understood and defined: list the standard definitions (if any) given by a general dictionary.
  - In case of multiple definitions, suggest (possibly with the help of ML) which are the most / less likely, based on the context offered by the available inputs and the ambiguity fixing decisions already made.
  - Is one of them appropriate? If so, which one?
  - If not, what definition should be used?
- Are there multiple nominal groups for that noun? If so, how are they related in a phylogenetic tree?
- Do some nominal groups need an explicit definition?

A similar approach could also be needed for *verbs* and *verbal groups*.

5.1.3 Syntactic Ambiguities

In *English*, syntactic ambiguity could result from the use of *pronouns* (e.g., *she*), *possessive determiners* (e.g., *her*) and *possessive pronouns* (e.g., *hers*). It could also result from the use of *adjectives* and of words such as *and*, as in "*young cats and dogs*": should that be parsed "*(young cats) and dogs*" or "*young (cats and dogs)*"?

Detection of syntactic ambiguities is (or should be) at the heart of NLP and will not be addressed further here.

A tool could do the following (questions being asked to and answered by human users):

- Identify requirements with syntactic ambiguities.
- For each such requirement:
  - List the possible parsing options.
  - Suggests (possibly with the help of ML) which are the most / less likely, based on the context offered by the available inputs and the ambiguity fixing decisions already made.
  - Which of them is appropriate?

5.1.4 Semantic Ambiguities

Some commonly used terms, such as *and* and *or*, and the use of plural, are highly ambiguous, as shown in Section 3.4.

A tool could:

- Have a list of ambiguous common terms, and for each, their possible interpretations.
- Identify each of their occurrences in the requirements.
- Identify each use of plural in the requirements.
- For each occurrence of either:
  - List the possible interpretations.
  - Suggests (possibly with the help of ML) which are the most / less likely, based on the context offered by the available inputs and the ambiguity fixing decisions already made.
  - Which of them is appropriate?

5.1.5 Silence

The identification of all that is missing in a set of requirements is most likely beyond the reach of an NLP tool. Indeed, in the BASAALT method, it is one of the very purposes of formalisation to enable simulation so that, among other defects, one can identify and fix silences.

However, possibly with the help of ML, a tool could help identify silence in individual requirements, such as those mentioned in Section 3.5.

5.2 Formalisation Patterns

Section 4.2 shows the formalisation of a few typical *k3* requirements and demonstrates that semantically accurate requirements formalisation is not just a mere paraphrasing of natural language statements: for each of the examples shown, a different sequence of formalisation steps (hereafter called a *formalisation pattern*) had to be applied. Any automatic formalisation tool would need to proceed in a similar way and would have, for each requirement, to first determine an appropriate pattern.

5.2.1 *d1* and *d2*

Strictly speaking, these two requirements are not *behavioural requirements* but *data modelling requirements*. However, whereas *d2* is formalised in a single step as a pure data modelling requirement represented only by the class structure in the `K3Data` model, the formalisation pattern for *d1* has an additional step specifying a behavioural requirement that adds precision to the class structure and constrains how that structure could be completed or modified in future refinements.

The *d2* formalisation pattern is also applicable to several other *k3* requirements not shown in Section 4.2, such as *d3* and *d4*.

5.2.2 *d6*

The formalisation pattern used for *d6* is the simplest and most straightforward one: after disambiguation, the natural language statement is directly translated into a formal requirement without the need for any extraneous step.

5.2.3 *r1*

The disambiguation process for *r1* concretised it with two less ambiguous natural language requirements, *rr1c* and *rr1d*. Then, the first step of the formalisation pattern adds two natural language requirements that are deemed missing, *rr1a* and *rr1b*.

rr1a, *rr1b*, *rr1c* and *rr1d* are then formalised separately. They all are *capability requirements* involving actions performed by human users external to the system of interest. Their formalisation follows the same steps:

- An *event*, not present in the natural language statement, is introduced to signal decisions (made externally to the system) to undertake the action related to the requirement.
- That event carries all the pieces of information characterising the decision and the action to be undertaken.
- A time limit, also not present in the natural language statement, is specified for the achievement of the action.
- The constraint specifies that the outcomes of the action must be consistent with what had been decided.

That pattern is also applicable to several other *k3* requirements not shown in Section 4.2, such as *r2*, *r3*, *r4*, *r5*, *r7*, *r8*, *r9*, *r10*, *r11*.

5.2.4 *d5* and *r15*

d5 and *r15* both specify pieces of information that must be included *a minima* in reports produced by the system of interest, but they are expressed slightly differently:

- *d5: The report of needed classes shall include ...*
- *r15: The system shall be able to display a printable summary for individual cohorts, which will include ...*

Whereas *d5* is expressed with what seems to be an *invariance constraint*, *r15* is expressed with a *capability constraint*. However, they are semantically very similar, and *d5* could be expressed much like *r15*:

The system shall be able to display reports of needed classes, which will include ...

This expression is a little more precise than, and thus preferable to, the original one since strictly speaking, the original one does not mandate the ability to produce reports of needed classes and could be interpreted as:

When reports of needed classes are displayed, they shall include ...

with the *when* never occurring because the ability is not required.

The formalisation pattern for *d5* thus begins with a *substitution* step, and then is the same as the one for *r15*, which is similar to but slightly different from the one for *r1*:

- An *event*, not present in the natural language statement, is introduced to signal report requests (made externally to the system).
- A time limit, also not present in the natural language statement, is specified for the production of the report.
- The constraint specifies that report contents include the required pieces of information.

That pattern is also applicable to *k3* requirement *r18* (not shown in Section 4.2).

5.2.5 *r20*

Like for *r1*, the disambiguation process for *r20* concretised it with two less ambiguous natural language requirements *rr20a* and *rr20b*. However, the difference here is that these two do not constitute the complete concretisation, but a concretisation *a minima*: other similar requirements could be added later.

Another difference is that *r20* does not specify a reaction of the system to an external request but an action to be decided by the system following some internal changes. Though with the current set of *k3* requirements, these changes result from external requests, one cannot guarantee that that will remain so with future *k3* refinements.

The pattern for *rr20a* and *rr20b* is also slightly different from the one for *r1*:

- An *event*, not present in the natural language statement, is introduced to signal the performance of the action by the system.
- As the action is a notification to users, the features of the event (which are `deferred` for now) specify who is to be notified and what pieces of information must be provided.
- A time limit, also not present in the natural language statement, is specified for the performance of the action, but its value is not the same as the one specified for system *reactions* to external requests.

That pattern is also applicable to *k3* requirement *r21* (not shown in Section 4.2).

5.3 How FORM-L Modelling Differs from Other Modelling Results

The main difference between the approach proposed here and those that can be found in the provided repository[2] concerns its objectives.

We do not aim at generating *software code* or *use case* skeletons but at producing *accurate and simulable formal models*. Models and simulation results can be reviewed and assessed by the authors of the natural language requirements (and possibly other stakeholders) so they can verify that: a) the formal requirements correspond to what they intended; b) they will not lead to undesirable or unacceptable behaviour in any situations the system could face. To that end, a strong focus is placed on:

- Understanding semantics, and as a consequence, the need to identify and fix ambiguities.
- Modelling of semantics as accurately and as completely as possible, and as a consequence, the need for a modelling language such as FORM-L.

In particular:

- Behaviours are not modelled with *use cases*, which at best specify behaviour in specific, individual cases, but with *properties* based on *temporal locators* and *constraints*, and specifying the envelopes in time and values of assumed or required behaviours.
- Pieces of information that are completely missing in the natural language requirements but are nonetheless necessary and could be intuited (e.g., a clear indication of the system of interest, its environment, and time limits) have been added.
- We do not follow the frequently applied approach in software RE of separating *functional requirements* from *non-functional requirements*: for real-life systems, non-functional aspects such as timing and time limits, accuracy and reliability are intrinsic aspects of a requirement.
- When not enough information is available to make an educated guess on what is meant, definitions are explicitly `deferred` to indicate that the model is currently incomplete and will need some *extension* models before it can be simulated. When that information cannot be obtained rapidly, the extension can be a *scenario model* providing temporary, plausible definitions.

We do not claim that the choices we made for fixing the ambiguities are the right ones. Indeed, the simulation's very purpose is to show unambiguously to the authors of the requirements and other stakeholders what are the consequences of the choices. They can then decide whether they agree or not with them. They can also decide whether the behaviours implied by their requirements are what they expect and want. If not, requirements will need to be amended.

6 Conclusion

Requirements documents are often full of imperfections, including ambiguities, inadequacies, over-specifications and contradictions. This paper illustrates the formalisation of the data [Lan23a] and behaviour [Lan23b] requirements of the *k3* case study. The process presented in this paper shows that NLP tools cannot detect silences and lack sufficient understanding to make the right imperfection-fixing

[2] See https://github.com/kevinlano/RequirementsFormalisation/tree/main/formalisationResults

decisions. It also shows how the BASAALT method and the FORM-L language can be used to identify and fix ambiguities in these requirements and then to faithfully formalise the original natural language requirements into simulable requirements models. Simulation can then be used to identify and fix imperfections other than ambiguities, providing subsequent engineering activities with "clean", rigorously verified requirements.

One of the limitations of what is presented in this paper is that BASAALT and FORM-L have been applied manually, first to identify and correct ambiguities in the initial requirements and then to formally model the corrected requirements. As NLP tools could greatly facilitate that process, future work will focus on developing such NLP tools, at least for the English language.

References

[ABHV06] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for event-b. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering*, pages 588–605, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[Abr96] Jean-Raymond Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 1996.

[Azz19] Azzouzi E., Jardin A., Mhenni F. A Survey on Systems Engineering Methodologies for Large Multi-Energy Cyber-Physical Systems. *13th Annu. Int. Syst. Conf. SysCon 2019 - Proc*, April 2019.

[BEG+21] Jean-Michel Bruel, Sophie Ebersold, Florian Galinier, Manuel Mazzara, Alexandr Naumchev, and Bertrand Meyer. The role of formalism in system requirements. *ACM Computing Surveys*, 54(5):1–36, 2021.

[BIP19] BIPM. The International System of Units (SI), 2019.

[Boa13] Mars Climate Orbiter Mishap Investigation Board. Mars Climate Orbiter Mishap Investigation Board - Phase I Report. Nimble Books LLC, 2013.

[Boo15] Goodwill Books. A guide to the business analysis body of knowledge(BABOK Guide). *v3, IIBA*, 2015.

[Bou21] Fabien Bouffaron. Airbus MBSE Framework : Model Execution of System Architectures (MOFLT). In *MBSE Cyber Experience Symposium 2021 - JAPAN*, ONLINE, Japan, September 2021.

[CKS11] Mary Beth Chrissis, Mike Konrad, and Sandra Shrum. *CMMI for development: guidelines for process integration and product improvement*. Pearson Education, 3rd edition, 2011.

[Coc01] Alistair Cockburn. *Writing effective use cases*. Pearson Education India, 2001.

[EPR08] EPRI (Electric Power Research Institute). Operating experience insights on common-cause failure in digital instrumentation & control systems. *TR 1016731*, December 2008.

[EPR15] EPRI (Electric Power Research Institute). Severe nuclear accidents: lessons learned for instrumentation & control and human factors. *TR 3002005385*, December 2015.

[Gli22] M Glinz. A glossary of requirements engineering terminology. Version 2.0.1. International Requirements Engineering Board (IREB). Technical report, IREB, 2022.

[IEE90] IEEE. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[ISO13] ISO/IEC. Information technology — Object Management Group Business Process Model and Notation. *19510*, 2013.

[ISO15] ISO/IEC. Systems and Software Assurance: Assurance Case. *15026-2*, 2015.

[JR10] Abrial JR. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[KS98] Gerald Kotonya and Ian Sommerville. *Requirements engineering: processes and techniques*. Wiley, 1998.

[Lam09] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 1st edition, 2009.

[Lan23a] Kevin Lano. k3 data. https://github.com/kevinlano/RequirementsFormalisation/tree/main/casestudies/k3data.txt, 2023.

[Lan23b] Kevin Lano. k3 ucs. https://github.com/kevinlano/RequirementsFormalisation/tree/main/casestudies/k3ucs.txt, 2023.

[Lan23c] Kevin Lano. Requirements formalisation. https://github.com/kevinlano/RequirementsFormalisation/tree/main/, 2023.

[Lar12] Craig Larman. *Applying UML and patterns: an introduction to object oriented analysis and design and interative development*. Pearson Education India, 2012.

[Lef10] Dean Leffingwell. *Agile software requirements: lean requirements practices for teams, programs, and the enterprise*. Addison-Wesley, 2010.

[LK22] Phillip A Laplante and Mohamad H Kassab. *Requirements engineering for software and systems*. CRC press, 2022.

[Mey22] Bertrand Meyer. *Handbook of Requirements and Business Analysis*. Springer, 2022.

[Net78] Aviation Safety Network. The Cranbrook Manoeuvre. https://aviation-safety.net/database/record.php?id=19780211-0, 1978.

[Ngu18] Thuy Nguyen. An Improved Approach to Traceability in the Engineering of Complex Systems. *2018 IEEE International Systems Engineering Symposium (ISSE)*, pages 1–6, 2018.

[Ngu19] Thuy Nguyen. Formal Requirements and Constraints Modelling in FORM-L for the Engineering of Complex Socio-Technical Systems. In *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW)*, pages 123–132, 2019.

[Ngu23a] Thuy Nguyen. k3 case study requirements in FORM-L. https://github.com/CoCoVaD/k3-Case-Study, 2023.

[Ngu23b] Thuy Nguyen. The BASAALT systems engineering method, and FORM-L, its supporting language. https://github.com/CoCoVaD/basaalt, 2023.

[OEC12] OECD-NEA. COMPuter-based Systems Important to Safety (COMPSIS) project: final report. *NEA/CSNI/R(2012)12*, July 2012.

[PA10] Shari Lawrence Pfleeger and Joanne M Atlee. *Software engineering: theory and practice*. Pearson Education India, 2010.

[Poh16] Klaus Pohl. *Requirements engineering fundamentals: a study guide for the certified professional for requirements engineering exam*. Rocky Nook, Inc., 2016.

[RI07] Respect-IT. A KAOS Tutorial, 2007.

[RR12] Suzanne Robertson and James Robertson. *Mastering the requirements process: Getting requirements right*. Addison-Wesley, 2012.

[vL04] A. van Lamsweerde. Goal-oriented requirements enginering: a roundtrip from research to practice [enginering read engineering]. In *Proceedings. 12th IEEE International Requirements Engineering Conference, 2004.*, pages 4–7, 2004.

[WB13] Karl Wiegers and Joy Beatty. *Software Requirements*. Microsoft, 3rd edition, 2013.

[ZJ97] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM transactions on Software Engineering and Methodology (TOSEM)*, 6(1):1–30, 1997.

A Summary of FORM-L Notations Used in this Paper

FORM-L Notations	*Comment*
Declarations	State the existence and nature of an item
Model <Name>	Declaration of a top-level model
Model <$Name_1$> *extends* <$Name_2$>	Declaration of an extension model, which can extend more than one model
partial Model <Name>	Partial model declaration
Library <Name>	Library declaration, to access to all what is defined in the library
Class <Name>	Class declaration, implicitly extending predefined class *Object*
Class <$Name_1$> *extends* <$Name_2$>	Declaration of an extension class, which can extend more than one class
Object Boolean Integer Real String Event Property Objective Requirement Assumption Guarantee Guard	Predefined classes
<Class> <name>	Object declaration, instance of <Class>
<Class> {} <name>	Declaration of a set of instances of <Class>
<Class> <name> (<parameters>)	Function declaration, returning an instance of <Class>
<Class> {} <name> (<parameters>)	Function declaration, returning a set of instances of <Class>
main	Object or class of interest of the model
external	Object or class in the environment of the item of interest
abstract	Class that cannot be instantiated directly
! # @ ^ ~ ?	Determiners (see Section 2.3.2)
Definitions	Specify the composition and behaviour of an item
<declaration> *begin* <features> *end*	Definition of an item with multiple features
<declaration> : <feature>	Definition of an item with a single feature
<declaration> *is deferred*	Indicates that the definition is to be provided in some extension model

Figure 8: FORM-L notations used in this paper - Part 1

FORM-L Notations	*Comment*
Continuous Time Locators (CTL)	Specify finite numbers of possibly overlapping time intervals
after <event>	Time intervals beginning at each occurrence of <event> and lasting until the end of the test case
after <event> *within* <duration>	Time intervals beginning at each occurrence of <event> and lasting <duration>
after $\langle event_1 \rangle$ *until* $\langle event_2 \rangle$	Time intervals beginning at each occurrence of $\langle event_1 \rangle$ and lasting until the first following occurrence of $\langle event_2 \rangle$
Discrete Time Locators (DTL)	Specify finite numbers of instants
when <event>	
Selectors	Indicate which set members are concerned
for all <name> *of* <set>	Universal quantifier without filtering condition
for all <name> *of* <set> *such that* <cond>	Universal quantifier with a filtering condition
for some <name> *of* <set>	Existential quantifier without filtering condition
for some <name> *of* <set> *such that* <cond>	Existential quantifier with a filtering condition
Actions	
achieve <cond>	Achievement constraint
ensure <cond>	Invariant constraint
assert <name> *can achieve* <cond>	Achievement capability constraint
assert <name> *can ensure* <cond>	Invariant capability constraint
<valued_object> *is* <expr>	Global assignment to a <valued_object>
Expressions	
<expr> *in* <set>	<expr> $\in$ <set>
$\langle set_1 \rangle$ *in* $\langle set_2 \rangle$	$\langle set_1 \rangle \subseteq \langle set_2 \rangle$
$\langle set_1 \rangle$ *union* $\langle set_2 \rangle$	Union of two sets
UNION <set of sets>	Union of all the sets in the <set of sets>
all <name> *of* <set> *such that* <cond>	Set of all members of <set> for which <cond> is true
for all <name> *of* <set> : <expr>	Set obtained by evaluating <expr> for each member of <set>
for all <name> *of* <set> *such that* <cond> : <expr>	Set obtained by evaluating <expr> for each member of <set> for which <cond> is true
me	Object being defined
bop	In an action and / or selector under the scope of a CTL, denotes the event occurrence at the beginning of the time interval
value	Attribute denoting the current value of a valued-object
identity	Attribute that uniquely identifies each object

Figure 9: FORM-L notations used in this paper - Part 2