REFERENCE MANUAL FOR THE FIGARO PROBABILISTIC MODELLING LANGUAGE

Marc Bouissou, Sybille Humbert, Jean-Christophe Houdebine

Version E - January 2019

Summary

This document is aimed at those who wish to become familiar with the Figaro language and its capabilities (chapters 1 and 2), and also those who need to write or modify knowledge bases. It uses numerous examples to illustrate the syntax of the language, defined formally in the document "Syntax of the Figaro modelling language" and associated semantics. It also provides a formal definition for the semantics of the Figaro language of order 0 which is the starting point for all processing treatments.

Once the reader has absorbed the contents of this document he will be able to use just the shorter document "Syntax of the Figaro modelling language".

- The first chapter describes the objectives of the language and explains why this language has been developed.
- The second chapter describes the principles of systems modelling in Figaro. In particular, it describes the use of rules in a Figaro model in forward and backward chaining modes.
- The third chapter provides a formal definition of the semantics of the language using a few simple mathematical concepts.
- The fourth chapter describes the lexical elements of the language.
- The fifth chapter mainly describes the syntax of the expressions used by Figaro.
- The sixth chapter describes in detail the syntax of defining classes of objects in Figaro.
- The seventh chapter describes advanced possibilities of the language.
- The eighth chapter details the possibilities of model documentation.
- The ninth chapter covers the automatic detection of model inconsistencies and provides advice for writing knowledge bases whose consistency is guaranteed by construction.
- Finally, the appendix contains a list of evolutionary changes made to the language since 1995, and an index of the main notions handled in this manual.

This document constitutes the fifth version of the Figaro language reference manual. Modifications carried out since the third version include, *inter alia*:

- on the one hand, changes due to evolutions in the language necessitated by the KB3 software with the introduction of new information associated with failures, and the new system object idea that replaced what was written in the so-called GLOBAL pseudo-class. Moreover, functions that can be exploited only through Monte Carlo simulation have been added following the creation of a new quantifying tool, known as YAMS, based on this method.
- and on the other hand, the addition of a chapter providing a formal definition of the semantics of the language plus another describing various methods for avoiding or detecting inconsistencies in the models. These chapters cover the same ground as two articles published in the 2002 ESREL conference.

Table of contents

Preliminary discussion

This document is aimed at those who wish to become familiar with the Figaro language and its capabilities (chapters 1 and 2), and also those who have to write or modify knowledge bases. It describes in detail the syntax of the language and associated semantics.

The second version of the manual (rev. B) incorporates feedback obtained mainly from operational projects, KB3 and TOPASE, and from the experience of teaching Figaro at the University of Rouen. In addition, it includes clarification of certain concepts of the Figaro language; this clarification was made while rewriting in C++ the instantiation and verification of knowledge bases.

The third version of the manual (rev. C) describes the language in its stable form during development of version 3 of KB3 before 2005.

This fifth version (rev. E) relates to the status of the language in 2016; it makes significant alterations and additions with respect to rev. C (see summary).

Conventions used in this manual

This manual contains descriptions of syntax for the various expressions in the Figaro language. This notation uses the following metacharacters:

Metacharacter	**Description**	**Example**
[ ]	Indicates that the information between the square brackets is optional.	[OF_CLASS class]
|	Is used to express an alternative.	A | AN

This manual uses the following typographical conventions:

`Courier`

Shows that the corresponding lines represent a Figaro program excerpt.

`Courier bold`

Highlights the difference between the user's identifiers and key words of the language in the examples. Key words are always bold.

Italic

Shows that the word refers to an object field or to a class of an example.

1. Introduction

Figaro is a Domain Specific Language created in 1989 for modelling systems in order to carry out operational safety studies and, by extension, probabilistic calculations on discrete time-continuous stochastic models. The aim of this chapter is to present a quick overview of the language's field of application and its main objectives.

1.1. Context and historical background

1.1.1. What is a dependability analysis?

The purpose of a dependability analysis on a system is to assess the system's reliability, safety, availability or productivity [1], [3]. Normally, such a study has three phases.

In the first phase, the reliability engineer in charge of the study has to **model the system** to be studied, i.e. he has to construct a representation of the system such that it meets the objectives of the analysis. Modelling consists primarily of describing the system (topology, components, system boundaries) and formalising the assumptions used in the investigation (at what level of detail are the components described, what are the failure modes for each component, what modelling is adopted to describe behaviours such as the propagation of fluid or flow of electric current).

In the second phase, the reliability engineer **constructs reliability models** appropriate to the system investigation, using the model and assumptions made in the first phase. There are several reliability models that can be used [1], [3] (Fault tree, Markov Graph, Petri Net, BDMP [Boolean logic Driven Markov Process], etc.). Each of these models enables a certain type of analysis to be undertaken. For example:

- A Fault Tree model allows a very detailed representation of a system's logic, but cannot include interdependencies among components;
- A Markov Graph model allows dependencies between components but is limited by the explosive growth in the number of states to consider.

Each type of model therefore has its limitations, and is associated with a number of assumptions to be satisfied. For example, a Fault Tree model presupposes that the components of the system are independent. The choice of the reliability model used for the investigation of a particular system is the responsibility of the reliability engineer in charge of the study.

In the final phase, the reliability models constructed are **quantified by appropriate calculation codes** to obtain the expected results concerning the system's reliability, safety, availability or productivity.

The process of undertaking a dependability analysis is therefore a multidisciplinary activity, drawing on an understanding of how the system functions and on knowledge of the reliability aspects of systems modelling and the mathematical content of model processing methods.

Normally, modelling the system and establishing reliability models are carried out "manually" by the reliability engineer. The results of the analysis are included in a document describing:

- the modelling of the system (described in reasonable detail);
- the reliability models constructed;
- the results obtained.

1.1.2. Towards automation

Since 1986 EDF has had to successfully undertake several Probabilistic Safety Analyses (PSA) on nuclear power plants, each PSA requiring a large number of dependability analyses of elementary systems. Moreover, since 1995 demands have emerged at EDF to introduce dependability analysis techniques in the systems design process.

The work carried out within these projects demonstrated the limitations of the standard manual approach in undertaking dependability analyses of systems.

A lack of traceability and transparency in the investigations

The reliability engineer's reasoning in constructing a reliability model often contains much that is implicit. This lack of transparency gives rise to two types of problem.

- Problems in updating the studies

  For anyone not involved in the study, it is often difficult to assimilate the reliability models such as to allow them to evolve. This problem arises in particular in the context of the nuclear PSA for which the constructed models must be maintained over tens of years.

- Problems in distributing the analyses to those who are not reliability engineers.

  As we have already emphasised, a reliability analysis depends on different areas of understanding, which means calling on different participants (system designers, operators). It is important that those participants who are not reliability engineers should understand and validate the assumptions and results without necessarily understanding in detail the reliability models that have been constructed.

Problems in guaranteeing the uniformity of studies within the same project

For the successful conclusion of probabilistic investigations of large complex systems (nuclear PSA for example) it is necessary to carry out and incorporate a large number of studies on elementary systems. These studies are frequently undertaken by different participants and it is difficult to guarantee their consistency and uniformity (choice of level of detail, how failure modes are taken into account, etc.).

Study timescales

A system analysis carried out "by hand" is a major undertaking that requires long timescales. These timescales make it hard to integrate such analyses with systems design processes (an iterative process requiring the "rapid" comparison of systems architectures).

Hence, to compensate for the limitations of manual dependability analyses, research has been carried out since 1986 at EDF to automate these investigations. The objective of these research programmes was to establish tools to:

- **model** systems simply,

- **formalise, trace and automate** the reasoning of the reliability engineer in order to construct reliability models based on the modelling of a particular system.

1.1.3. Automation procedure

When a system dependability analysis is carried out, a certain amount of information has to be collected about each component within the system. Hence, for a motor-driven pump for example, what is the pump's electrical power distribution busbar, what is the initial state of the pump (stopped, running), etc.?

To model a component, associated with it (implicitly or otherwise) are a certain number of properties. It is clear that these properties are often **common to all components of the same class**. (All motor-driven pumps in thermohydraulic systems have an electrical supply and an initial state).

Thus, all dependability analyses relating to the same class of systems (thermohydraulic systems, electrical systems, etc.) may use the same **generic model** to describe the systems.

Accordingly, in order to automate the dependability analyses it seems natural to capitalise on generic component models in **knowledge bases** (see Fig. 1). For example, a knowledge base dedicated to thermohydraulic systems contains descriptions of components such as pumps, valves, check-valves, etc.

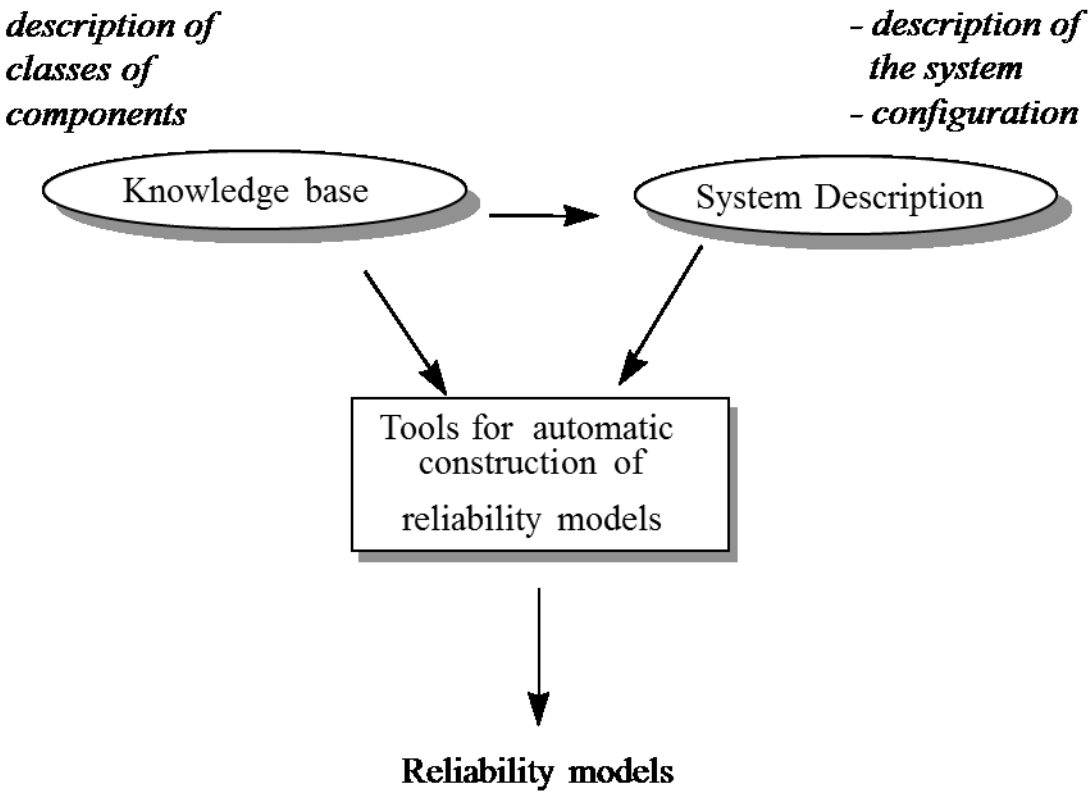

Figure 1. Automation procedure

Following this approach, the **description of the system** to be analysed (modelling the topology of the system and the components within the system) is made by drawing on generic component models from the knowledge base; these generic models are instantiated depending on where they reside in the topology of the system and on their unique characteristics.

The required reliability models are then generated automatically using the **model of the system**, which consists of the **knowledge base** and the **description of the system** to be studied.

Remark: The knowledge represented in a knowledge base naturally depends on the class of systems to be analysed, but often also on the type of dependability analysis that we seek to automate (on the type of reliability model that we wish to generate automatically using the knowledge base).

1.2. Why is a specific modelling language needed?

In implementing this process of automating systems dependability analyses, the first problem is that of finding a **language** that the reliability engineer can use for simply representing his knowledge of a system.

This language must meet the following requirements:

- The knowledge must be **"easy"** for the reliability engineer to represent. In other words, the language chosen must be simple for the reliability engineer to grasp. It must allow him to manipulate simply the concepts necessary for formalising his reasoning (incorporating fully any ideas related to reliability).

  * The language must allow generic knowledge to be described (knowledge bases);

* It must provide a simple formalism for describing particular systems based on generic modelling of the knowledge base. It must be possible for this formalism to be associated with a graphical interface (description of the topology and components of the system);

* It must be able to model systems with the **same** formalism, regardless of the reliability models to be constructed. In other words, this language must be a generalisation of current reliability models (Fault tree, Markov Graph, Petri net, etc.).

- The models built using this language (knowledge bases and system descriptions) must be **intelligible** to non-specialists.

- The formalism must be **capable of evolution** so that it can be extended to system modelling for types of study other than dependability analyses (maintenance, diagnosis, etc.).

- It must have a **formal definition** so as to allow mathematical proof of properties: behaviour of the model defined uniquely and independently of the tools used to manipulate it, an absence of inconsistencies, a finite state space, a guarantee that the model can return to its initial state, etc.

- Finally, all models constructed using this language must be **maintainable**.

* It must be possible to easily alter the knowledge within a model without necessarily having a view of the whole; this points towards a declarative language;

* Knowledge must be **structured** to enable rapid access to the information sought, and **factorised** to avoid repetition of identical pieces of information;

* The language must be associated with debugging tools.

An analysis of the existing computer languages that are commercially available shows that none of them is suitable for all of these requirements. General languages such as ADA, C++, Java, Python, etc. are programming, rather than modelling, languages, and do not directly satisfy the constraints on the intelligibility of the models and the ease with which they can be grasped by the reliability engineer. Also, these languages are "imperative", i.e. they execute program instructions in accordance with the sequence defined by the control instructions (if, do, while etc.); this is appropriate for a simulation, but not for constructing fault trees. PROLOG works in a very different way: it is associated with a "backward chaining" reasoning mechanism that seeks values to give variables in order to satisfy various predicates. This type of reasoning would undoubtedly be very practical in constructing fault trees, but programming in PROLOG is rather special and few are skilled in its use. Furthermore, there is a risk that simulation in PROLOG would be difficult. Generally speaking, all these computer languages are too general to give rise to proofs of properties such as those discussed in chapter 9.

Therefore, in order to address the problem of modelling systems in the field of dependability analyses, EDF has developed its own modelling language, **Figaro**. This initiative was unique in 1989, but its example has been followed by much work since then. One essential advantage Figaro has over its competitors is its great stability; thus models and treatment tools (which make up the KB3 tools platform) have been able to capitalise on this since the 1990s.

The widespread distribution of these tools outside EDF can only take place if their common support is well known, well documented, and freely accessible. Hence the Figaro language is freely available and this document is accessible to the public. The long-term hope is that, as with the Modelica language for physically modelling hybrid systems, tools of diverse origins (commercial or free) using the Figaro language will coexist together.

2. Principles of modelling in Figaro

2.1. Classes and Objects

Figaro is a hybrid language in the sense that it is both an object oriented language (one which manages classes and objects) and a language inspired by artificial intelligence techniques (one which manages mechanisms of inference over rules). This section presents a general overview of the main ideas behind the "Object" part of Figaro. The "Artificial intelligence" part will be presented in due course.

2.1.1. Definitions

Class: A description common to all the components of the systems being studied that have the same behaviour and characteristics. Each class is identified by a name and contains a series of characteristics that are the characteristics common to the components it describes.

Object: Any component of a system. Each object is identified by a particular name.

In Figaro, each object in a system belongs to a class. We say also that the object of a class is an **instance of that class**. The objects in a particular system that belong to the same class C form the **set** of objects of the class C in the system.

An object inherits the characteristics of its class, making them more precise by using information specific to a given system. This information is written in bold in the example below.

Example: the object S002PO is an instance of the class 'pump':
Object S002PO
Characteristics: initial state = **stop**,
power supply busbar = **busbar T1**

The characteristics describing each CLASS or each OBJECT are value fields or rules. **Value fields** define the properties of objects, in particular the different variables that will characterise the state of an object.

Example of CLASS:

```
CLASS      pump;
ATTRIBUTE    (* paragraph defining the state variables of objects of class pump *)
    flowrate DOMAIN REAL REAL;
    state  DOMAIN 'stop' 'working';
```

Examples of OBJECTS:

```
OBJECT S002PO IS_A pump;
ATTRIBUTE
    flowrate = 0;
    state = 'stop';

OBJECT S001PO IS_A pump;
ATTRIBUTE
    flowrate = 5000;
    state = 'working';
```

As we shall see later in this chapter, **rules** define the dynamic behaviour of objects, i.e. the rules governing the change in the state variables of objects over the course of time (e.g. starting one pump to back up another, etc.).

2.1.2. The structure of a Figaro model

Now that we have defined the ideas of "class" and "object", we can describe **how** the Figaro language is used to automate the dependability studies of systems. The principle is as follows:

- For a category of systems (thermohydraulic systems, electrical systems, etc.), each CLASS of component belonging to the category of systems being studied is described in Figaro in a **knowledge base**. A Figaro knowledge base therefore contains a description model common to all the systems in the category. This is a set of CLASS(es).

- To study a particular system, we develop a **description of that system** using the knowledge base:

  * Each object in the system is created as an instance of a class in the knowledge base.

  * For each object, values specific to the system are attributed to each value field characteristic of the class this to which this object belongs.

The association (knowledge base + set of objects in the system) constitutes a complete model of the system to be investigated. This model is known as the **Figaro model** for the system.

Processing then allows the Figaro model to be used to automatically generate the required reliability models.

2.2. Models of order 1 and order 0

The Figaro model described in the previous section is said to be a "model of order 1", echoing the vocabulary used in artificial intelligence to designate the rule bases that apply to sets of objects that have not been defined explicitly.

A behavioural rule is most often common to all the objects described by the same class. Hence the Figaro language can describe, for each class from a knowledge base, rules that are generic to all the objects described by the class (factorisation of knowledge).

Any use of this model must proceed via an instantiation operation that will result in a "model of order zero" containing only those rules applicable to well defined unique variables for the modelled system. This instantiation operation, possible because the number of objects that make up the system is fixed and will not change over the course of a simulation, has several advantages:

- A certain number of tests will be made just once to simplify the rules, even eliminating some completely if their conditions are always assessed as FALSE.
- The model of order 0 will be easy to compile efficiently to allow rapid processing.
- The semantics of the order 0 model can be formally and simply defined (see this definition in chapter 3).
- It is possible to make additional consistency checks on a model of order 0.

There are only objects In a Figaro model of order 0. This is a "flat" model with no hierarchy.

The remainder of chapter 0 describes the language of order 1, except where order 0 is mentioned explicitly.

2.3. Inheritance between classes

Inheritance allows information to be factorised and hence to create more concise knowledge bases where the information is better structured.

Take the example of a thermohydraulic system, which contains both motorised pumps and turbopumps. These two classes of pump have common characteristics that can be judiciously combined. We shall therefore define a generic class (that of pumps) and two specialised classes (those of motorised pumps and turbopumps respectively).

```
CLASS      pump;
ATTRIBUTE
    flowrate DOMAIN REAL;
    state  DOMAIN 'stop' 'working' ;

CLASS      motorised_pump EXTENDS pump;
ATTRIBUTE
    motor_power DOMAIN INTEGER;

CLASS      turbo_pump EXTENDS pump;
ATTRIBUTE
    turbine_capacity DOMAIN INTEGER;
```

The first class defines the common pump characteristics.

The second and third classes inherit the first, i.e. they have the same general characteristics (flowrate and state) to which they add their own characteristics (motor power and turbine capacity).

The class *motorised_pump* therefore comprises a *pump* part and a *motorised_pump* part. An instance of this class, i.e. an object of class *motorised_pump* should initialise the three fields (flowrate, state and motor power). An object of type *pump* has just the flowrate and state fields. Given that an object of class *motorised_pump* "contains" an object of class *pump*, we assume that an object of class *motorised_pump* is also an instance of *pump.*

It is possible to redefine in a sub-class certain characteristics of the class from which it inherits. For example, the class *motorised_pump* could redefine the behaviour of a pump or some of its characteristics (e.g. to single out the KIND slot of an interface or add an enumerated value to a domain), etc. This is known as overriding. The overriding mechanism allows a field or behaviour to be redefined in a lower class (i.e. a class that inherits). This enables a generic characteristic to be singled out. For example, another possible state of a motorised pump is the start-up demand. We would therefore have:

```
CLASS      motorised_pump EXTENDS pump;
ATTRIBUTE
    motor_power DOMAIN INTEGER;
    state DOMAIN 'working' 'stop' 'start-up demand' ;
```

In this example, we override the DOMAIN of the *state* attribute. Any characteristic or rule having a name can be overridden.

This little example gave the main aspects of the idea of inheritance. These are as follows:

- Factorisation of information in upper classes (those that pass down the inheritance). Several attributes and rules are **pooled** in the same class.

- Specification of attributes and behaviours in the lower classes (those that inherit). Attributes and rules are **added** in specialised classes.

- Redefinition of certain types of behaviour in lower classes. The characteristics of the mother classes are overridden.

This approach is characteristic of object oriented languages.

It is possible in Figaro that a class inherits from several classes: this is known as multiple inheritance. The class may take advantage of each of the characteristics of the inherited classes. The new class then represents the **union** of the characteristics of the inherited classes. It can then add characteristics or redefine them as we have seen. In the following example, the class *motorised_pump* is a pump and an electrical component.

```
CLASS      pump;
    ATTRIBUTE
    flowrate DOMAIN REAL;
    state  DOMAIN 'stop' 'working' ;

CLASS      electrical_component;
    ATTRIBUTE
    motor_power DOMAIN INTEGER;

CLASS      motorised_pump EXTENDS pump electrical_component;
```

Nevertheless, certain problems can arise when two characteristics inherited from different classes have the same name: it would be impossible to know where this characteristic came from.

```
CLASS pump;
    ATTRIBUTE
    state DOMAIN 'working' 'stop' ;

CLASS motorised_equipment;
    ATTRIBUTE
    state DOMAIN 'fast_running' 'slow_running' 'stop' ;

CLASS motorised_pump EXTENDS pump motorised_equipment;
(* What is the DOMAIN of the state ATTRIBUTE? *)
```

The rules for managing the overriding and inheritance mechanisms implemented in Figaro are fully explained in document [5] which describes the syntax of the language in detail. In this manual it will be sufficient to set out the principles of these mechanisms and the associated prohibitions (see Advanced notions).

2.4. Communication between objects

A system is a set of interacting components. To ensure the functions of the system, the components are mutually **linked** to one another. Three main types of link can be distinguished:

- Topological links: components are connected one to another in order to propagate the main flows of the system. For example: pipework to carry fluid in a thermohydraulic system, cables to carry current in electrical systems;
- Functional links: the operation of a component within the system is dependent on other components. For example: a backup pump for another one should start only when the normal pump has failed;
- Links between the components of the system and the support systems. For example: a Low Voltage busbar provides the power for a pump's motor.

Modelling the links between the objects in a system is an important point in developing a model, since the behaviour of each object depends on the behaviour of the objects with which it is linked.

When we have a particular system to be modelled, we know the structure of the system and can therefore easily represent the links between the objects in the system. On the other hand, when we develop a generic model for a category of systems (description of classes in a knowledge base) we only know the existing relationships (potential or mandatory) between the classes. For example, an arbitrary object of class *motorised_pump* is linked with one, and only one, object of class *busbar*.

The Figaro language allows both of the following:

- to represent, for each class described in a knowledge base, the potential or mandatory relationships between an arbitrary object of that class and the objects of other classes (writing a generic model), or even of the same class.

- "to give precise information" about these relationships for each object in a particular system (to instantiate these relationships) in order to precisely define the set of objects in the system that are related to that particular object.

The links between the objects in a system are represented in Figaro by **interface fields**.

2.4.1. Interface field of a class

When we define a class, we have to define the **relationships** that exist between some arbitrary object of this class and other objects. More precisely, the interface fields of a class enable definition of:

- the existing relationships with the other objects, whether of the same or different classes,

- the cardinality of each relationship, stating in particular whether this relationship is mandatory or otherwise (see §2.4.3.2).

In a knowledge base (definition of classes), the interface fields **define the relationships** between **classes**. Each interface field contains the **identifier for a relationship**.

Example: a pump is fed via a busbar.

```
CLASS motorised_pump;

INTERFACE elec_supply KIND busbar CARDINAL 1;
```

A relationship defined between two classes is not symmetrical. In the example above, it is essential to know which busbar is feeding a pump, but in order to know which pumps are being fed via a given busbar we will have to run through the entire set of pumps looking for those that have that particular busbar in their *elec_supply* interface. Hence it is important to make the correct choices when defining the interfaces in a knowledge base.

2.4.2. Interface field of an object

In a system description (definition of objects), the interface fields relate precise sets of objects with one another. In other words, the **relationships defined** in classes are **instantiated** in objects.

Example 1: pump p1 is fed via the low voltage busbar bt1.

```
OBJECT p1 IS_A motorised_pump;

INTERFACE elec_supply = bt1;
```

Example 2: pump p1 is connected upstream to two valves v1 and v2, and downstream to valve v3.

```
OBJECT p1 IS_A pump;

INTERFACE     upstream = v1 v2;

              downstream = v3;
```

For a particular object, each interface field is a list of objects, i.e. it contains the names of objects.

It is very important to note that an object's interface fields **must** be constant fields in Figaro. This means that the existing links between the objects in a system cannot change over the life of that system. In other words, Figaro rules will not allow an object to be added to an interface or an object to be removed from an interface.

An object cannot be interfaced with itself.

2.4.3. Relationships and sets

2.4.3.1. Relationship identifier

It is important to note that a relationship's identifier can be considered as the name of a set (in the mathematical sense). Consider the two previous examples:

* If x is an arbitrary object of class *motorised pump*, *elec_supply OF x* denotes the set of objects of class *busbar* that are related to x in an arbitrary system described using the knowledge base. In the example above, *elec_supply OF p1* denotes the singleton {bt1}.

* If x is an arbitrary object of class *hydraulic_component*, *upstream OF x* denotes the set of objects of class *hydraulic_component* that are connected upstream to x in an arbitrary system described using the knowledge base. In the example above *upstream OF p1* denotes the set {v1, v2}.

2.4.3.2. Cardinality

When we define a relationship between a class A and a class B in a knowledge base, it is important to specify the constraints on the cardinality of the set of objects of class B that will be related to some arbitrary object of class A. In particular, this allows us to specify whether or not the relationship is mandatory. Figaro provides several possibilities, illustrated using the following example:

```
CLASS component;

    INTERFACE upstream KIND component;
```

- If no restrictions have been placed on the *upstream* relationship, that means that a component may have no component upstream (the relationship is **not mandatory**) or that it may have several, unlimited in number. The cardinality of the set of objects of class *component* upstream of an object of class *component* is indeterminate (the set may be empty).

```
CLASS component;
    INTERFACE upstream KIND component;
```

Remark: *we could also write:*

```
CLASS component;
INTERFACE upstream KIND component CARDINAL 0 TO INFINITY;
```

- If we specify that the cardinality of the relationship *upstream* is n, that means that a component **MUST** have n components upstream. The cardinality of the set of objects of class *component* upstream of an object of class component is n.

```
CLASS component;

    INTERFACE upstream KIND component CARDINAL 3;
```

In this case, if n=1 we say that the interface is **single-valued**. This particular case is important since it allows the rules to be written with a simpler syntax than for the general case.

- If we specify that the cardinality of the upstream relationship is contained within the interval (p, q), that means that a component MUST have between p and q components upstream. The cardinality of the set of objects of class *component* upstream of an object of class *component* lies between p and q.

```
CLASS component ;

    INTERFACE upstream KIND component CARDINAL 2 TO 8;
```

In this case, if p = 0, the relationship is not mandatory.

2.4.4. Use of interface fields

Interface fields will be used to model the influence of objects related to a component on the behaviour of that component.

Hence, as we shall see when we present the Figaro rules, declaring class B in an interface of class A (i.e. setting class B in relation to class A) will allow the characteristics of class B to be used to model the behaviour of class A.

Hence the definition of the interfaces between the classes described in a knowledge base is an important step in the design of a knowledge base.

2.4.5. Modelling hardware links between objects

The graphical representation of a system associated with a Figaro description comprises nodes (icons) and links (lines, with various attributes of thickness, colour, types of arrow, etc.). Each graphical object has to have its correspondence in Figaro. However, there exist two types of situation, depending on whether the links have just been declared or whether they contain fields to which values have been assigned, even behavioural rules in the same way as the nodes.

When two objects are related to one another in physical reality by some piece of hardware (pipework, cable, etc.) it should be noted that two types of modelling are conceivable for these links, depending on the objectives of the modelling process.

If these physical links have specific characteristics that are useful in modelling the behaviour of the system, then standard Figaro classes must be defined for these links.

> For example, if for a class of electrical systems the characteristics of the cables (impedance, length, etc.) are important for the model, then an *electric_cable* class will be defined in the knowledge base. This class will itself be interfaced with the other classes in the model (each cable will have a component upstream and downstream).

If the physical links do not have specific characteristics that are useful in modelling the behaviour of the system, it will be sufficient to declare these classes as "empty" (no Figaro content) for these links. However, they will be associated with instructions for filling in the interfaces of the nodes on which they will be connected in the XML configuration file for the KB3 graphical interface.

> For example, if for a class of thermohydraulic systems we assume the pipes are perfect, the links simply provide a graphical representation of the system even though the flow propagates directly from a hydraulic component represented by a node to the components declared in its *downstream* interface field. This interface will be filled in automatically by KB3 during a system's graphical input when the user draws the links.

2.5. Fields of classes and objects

A few notes on vocabulary: we use the word "field" rather than attribute or property to avoid confusion with the vocabulary of UML models (see § 2.6 for the correspondence between UML and Figaro concepts).

As we have previously seen, a class is described by fields that are common to objects of that class and by rules that we shall examine in the next section.

The fields characteristic of a class or object can be of three kinds:

* **Constant fields** represent the characteristics of objects that do not change over the course of the system's life (e.g. length of a connector cable). These fields are single-valued.

* **The interface fields** of a class define the potential relationships between an object of this class and objects of other classes, even of the same class. **The interface fields of an object** provide information about the objects in a system in relation to that object. These are object fields (i.e. containing the names of objects) which may be multivalued.

* **State variables** characterise the state of objects, and can change over the course of the system's life. These fields are single-valued.

It is important to note that, in Figaro, **interface fields are constant fields**. In other words, it is not possible in Figaro to model processes that might alter the relationships between the objects in a system.

As a simple example, take the description of a class *electric_cable* characterised by its length.

Example :
```
CLASS electric_cable;
CONSTANT
length DOMAIN REAL;
(*the domain of definition of the constant is the set of real numbers*)
```

Using the definition of this class we can create a particular instance of *electric_cable* to describe a system by assigning the pertinent value for the system being studied to the *length* field:

Example:
```
OBJECT cable1 IS_A electric_cable;
CONSTANT
       length = 25;
```

2.6. Link between Figaro, UML and SysML concepts

It is easy to make a correspondence between the main concepts of Figaro and those most commonly used in a meta-model described in UML or SysML. The table below summarises this correspondence.

UML Concept	***SysML Concept***	***Figaro Concept***
Class	Block	Class
Object	Part	Object
Inheritance	Inheritance	Inheritance
Attribute	Value	Constant, attribute, effect, failure
Relationship	Flow and Port	Interface

We have not established any correspondence between the concepts of operation in UML and SysML with

Figaro's rules, although in both cases it is a matter of describing the behaviour of classes. The rules do not behave like methods that are called using arguments in the manner of what we find in object-oriented languages.

Despite this restriction, it is easy to use a UML or SysML graphical editor to represent all the structural elements of a Figaro knowledge base metamodel. Such a graphical representation could be a fruitful source of discussion for constructing a new knowledge base or for providing a summary representation of the contents of an existing base.

2.7. Figaro and units

As far as managing the units in Figaro models is concerned, the choice was very simple: they are never given explicitly. This has the advantage of simplifying the syntax, but at the same time the user has to know what he/she is doing, and must write all Figaro expressions using numbers that correspond to a consistent system of units. Fortunately, in the vast majority of models there is just one unit, viz. time. Hence if the unit chosen is the hour, this means that the user must express failure and repair rates in the exponential distributions as hr^{-1} and the parameters in the C_T distributions as hours, as well as the mission times set out in the calculation tools.

2.8. Figaro rules

In the Figaro language, the dynamic behaviour (i.e. over the course of time) of objects is modelled in the form of rules. To properly understand what a rule is in Figaro, we have to understand the **general principle of systems modelling** used in a Figaro model. This section provides simple explanations with many examples. Section 3 will provide a formal definition of the semantics of the language, the operational basis of the tools that are founded on Figaro.

2.8.1. Principle of systems modelling

Model state and event

The state of an object at some given instant is defined as the set of values taken at that instant by the state variables of the object. The **state of a system** at some given instant is completely defined by the state of the objects that constitute the Figaro model at that instant.

The state of the system will change as a function of time as a consequence of the **events** to which the system might be subjected.

The occurrence of an event is conditioned by the current state of the system. In other words, an event can only occur if the state variables of the model satisfy a certain condition.

If pump P1 is running

Then the event: "Pump operational fault" might occur.

An event corresponds to a random phenomenon that occurs according to some particular probability distribution, or to some deterministic phenomenon that occurs at the end of a set period of time (if the condition governing the occurrence of the event is maintained during that period).

If pump P1 is running,

Then the event: "Pump operational fault" might occur in accordance with an exponential probability distribution of parameter 1e-3 hr^{-1}.

Hence the Figaro language is able to model Markov processes (the occurrence of an event or transition depends **uniquely** on the current state of the system) or semi-Markov processes (the occurrence of an event depends on the current state of the system and on the time that has passed in that state). This is very similar to what happens in a stochastic Petri net.

Consequences of an event on the system

An event occurring on a system component will alter the component's state, which also means altering the state variables associated with that component. For example, the occurrence of the event "operational fault in P1" alters a P1 state variable which changes from "running" to "failure".

Alteration of the state of the component affected by the event will trigger a whole series of consequences on the other components in the system and on the system itself (The effects of an event on a particular component will "propagate" throughout the system). We can cite the two following examples as illustrations:

If pump P1 has failed,
Then the feed path of fluid passing through P1 has been lost.

If pump P1 has failed,
Then the backup pump associated with P1 will be started.

Life cycle of a system

We assume that the life of a system consists of a sequence of cycles, with each cycle broken down into two stages:

- When the system is in a given state, certain events may occur. Initially, the occurrence of an event on a component will have **direct and local** consequences on the state of that component.

- Secondly, any modification of the state of the component variables will have consequences for the other components in the system, **propagating the effects** of the event to obtain a new state of the system.

Once the system is in this new state, other events may occur.

So as to model the behaviour of systems according to this principle, Figaro uses a rule-based formalism. There are two types of rule in Figaro:

- **Occurrence rules** enable events that can happen to an object to be modelled: conditions of occurrence of the event depending on the state of the system, the associated probability distribution and consequences on the state of the component.

- **Interaction rules** allow the consequences of an event to be propagated throughout the system by expressing the relationships between the state of an object and the states of other objects in the system.

Before examining how these rules are used in a Figaro model, we shall define their formalism more precisely.

2.8.2. Formalism of a Figaro rule

The formalism of a Figaro rule differs according to whether we are dealing with an occurrence rule or an interaction rule.

2.8.2.1. Form of an occurrence rule

An occurrence rule models the influence on the system of an event (we also speak of a transition) that can happen to the objects in a system. Schematically, these rules have the form:

```
IF <condition>
MAY_OCCUR <event>
INDUCING <actions>
```

The condition is a Boolean expression that expresses the condition the state of the system must satisfy in order for an event to happen to an object. (If the condition has the value TRUE, the event can occur). Computer scientists call this type of condition the event's (or transition's) guard.

Actions describe the consequences of the event on the state variables affected by the event.

Example of an occurrence rule:

```
CLASS pump;
ATTRIBUTE
    state DOMAIN 'stop' 'startup' 'running' 'failure';
CONSTANT
    lambda DOMAIN REAL DEFAULT 1e-3;
OCCURRENCE
    IF state = 'running'
    MAY_OCCUR
    TRANSITION operational_failure
          DIST EXP ( lambda )
          INDUCING state <-- 'failure';
```

This occurrence rule that is **generic** to the objects of the class *pump* can be read as:

*Let **O** be some arbitrary object of class pump,*

*If the state of **O** is "running",*

*Then the event "operational failure" will occur in **O** at the end of some random time sampled from an exponential distribution of parameter lambda, causing the state variable of **O** to become "failure".*

2.8.2.2. Form of an interaction rule

An interaction rule models the logical relationships between the state of an object and the state of the other objects in the system. Schematically, these rules have the form:

```
IF <condition>
THEN <actions>
ELSE <actions>
```

For these rules, the condition (Boolean expression introduced by IF) causes a set of actions affecting the state variables of the objects in the system to be triggered (If the condition has the value TRUE, the actions introduced by THEN are executed; if the condition has the value FALSE, the actions introduced by ELSE are executed.

Example of an interaction rule:

```
CLASS pump;
ATTRIBUTE
    state DOMAIN 'stop' 'startup' 'running' 'failure';
INTERFACE
    backup_pump KIND pump CARDINAL 1;
INTERACTION
    IF state = 'failure'
    THEN
          state OF backup_pump <-- 'startup';
```

In this example the set of objects of class pump in relation to *backup_pump* with some arbitrary pump is a set of cardinality 1. Thus, *state OF backup_pump* denotes the state of the unique pump backing up some arbitrary pump. This rule of interaction generic to objects in the class *pump* can be read as:

Let O be an arbitrary object of class pump,

If the state of O is "failure",

Then the state of the unique object of class pump having the relation backup_pump with O becomes "startup".

2.8.2.3. Condition and actions of a rule, use of interface fields

In principle, the behaviour of objects in a class depends mainly on the state of these objects and on the state of the objects from classes with which they are related (interfaced classes). Hence for a given class:

- the condition of a rule is a Boolean expression using the properties of the class and the properties of the classes that are interfaced,

- the actions of a rule affect the variables of the class and the variables of the interfaced classes. (By definition, the actions of an occurrence rule uniquely affect the variables in the class).

To illustrate this, we shall give a few examples showing how the interface fields of a class are used to write the behavioural rules for this class.

Example 1

Take the example of objects of class *pump*. In this example, the class *pump* is specialised into two sub-classes: pumps that operate in normal time (class *normal_pump*) and pumps that operate as backups to normal pumps (class *backup_pump*).

Each object of class *backup_pump* is there to back up one or more normal pumps. In other words, each object of class *backup_pump* is linked to one or more normal pumps. The behavioural rule that we wish to model is as follows:

If all the pumps backed up by the same backup pump have failed, then the system demands that this backup pump be started.

Given that the behaviour of objects of class *backup_pump* depend on the state of the objects of class normal pump, a relationship must be created between the classes *backup_pump* and *normal_pump*. This relationship is identified by the interface field *backed_up* and has cardinality 1 to INFINITY.

```
CLASS pump;
ATTRIBUTE
    failure DOMAIN BOOLEAN;


CLASS normal_pump EXTENDS pump;


CLASS backup_pump EXTENDS pump;
ATTRIBUTE
    start_demand DOMAIN BOOLEAN;

INTERFACE
    backed_up KIND normal_pump CARDINAL 1 TO INFINITY;

INTERACTION
    IF FOR_ALL x A backed_up WE_HAVE failure OF x = TRUE
    THEN start_demand <-- TRUE;
```

This interaction rule means:

Let O be some arbitrary object of class backup_pump,
If $\forall x \in$ {normal pumps backed up by O}, x has failed,
Then, there is a demand for O to start.

In this example, the condition of the rule depends on the characteristics of the objects of a class that is interfaced with the class *backup_pump.* The action of the rule affects only the objects in the class described. **In other words, this interaction rule enables the state of an object in the class to be modified depending on the state of the objects in the interfaced classes.**

Example 2

In this new example we assume that each normal pump in the system is backed up by one or more backup pumps. The behavioural rule that we wish to model is as follows:

If a normal pump has failed, then the system demands that all the associated backup pumps start.

```
CLASS pump;
ATTRIBUTE
    failure DOMAIN BOOLEAN;


CLASS backup_pump EXTENDS pump;
ATTRIBUTE
    start_demand DOMAIN BOOLEAN;
```

```
CLASS normal_pump EXTENDS pump;
INTERFACE
    backup KIND backup_pump CARDINAL 1 TO INFINITY;
INTERACTION
IF failure = TRUE[1]
THEN
FOR_ALL x A backup DO start_demand OF x <--TRUE[2];
```

This interaction rule means:

Let O be some arbitrary object of class normal_pump,
If O has failed,
Then, ∀x ∈ {backup_pump backing up O}, there is a demand that x should start.

In this example, the condition of the rule depends only on the characteristics of the objects in the class described. The action of the rule affects objects of the class *backup_pump* declared in the interface *backup*. **In other words, this rule of interaction enables the state of objects in interfaced classes to be modified depending on the state of the objects in the class described.**

Potential problems

The examples above show that the definition and use of interfaces must be carried out with great care so as not to create any inconsistencies. For example, problems can arise if there are two different relationships between two different classes.

In the following simple example, class t1 is related to class t2 via interface r12, and similarly class t2 is related to class t1 via interface r21. The rule "Rule _1" alters the state of an object of class t1 depending on the state of the interfaced class t2 object. The rule "Rule _2" alters the state of an object of class t2 depending on the state of the interfaced class t1 object. In this example the two rules are inconsistent since they allow no conclusion as to the state of the objects of these two classes (see § 8 on consistency).

```
CLASS t1;
ATTRIBUTE
    a DOMAIN BOOLEAN;
INTERFACE
    r12 KIND t2 CARDINAL 1;
INTERACTION
Rule_1
IF b(r12) = TRUE
THEN a <-- FALSE
ELSE a <-- TRUE;

CLASS t2;
ATTRIBUTE
    b DOMAIN BOOLEAN;
INTERFACE
    r21 KIND t1 CARDINAL 1;
INTERACTION
Rule_2
IF a(r21) = TRUE
THEN b <-- TRUE
ELSE b <-- FALSE;
```

Single-valued and multi-valued interfaces

[1] This condition `failure = TRUE` could also be written simply as `failure`.

[2] This assignment `demand_start OF x <--TRUE` could also be written simply as `demand_start`.

It is important to note that the use of a single-valued interface (of cardinality 1) differs from that of a multi-valued interface. As we shall see more clearly later in this manual, a single-valued interface field is used as a normal field while a multi-valued interface field can be managed only by quantifiers (THERE_EXISTS, THERE_EXISTS AT_LEAST, FOR_ALL), "iterators" (FOR_ALL, GIVEN), or complex operators (SUM FOR_ALL, PRODUCT FOR_ALL), etc.).

2.8.3. Principles for use of rules

Using a Figaro model amounts to applying the set of rules in the knowledge base to the objects describing the system to be studied.

There are two possible types of application for the Figaro rules, corresponding to two types of treatment: forward chaining and backward chaining.

In all cases, use is made of the instantiated model of order 0 as we saw in § 2.2.

2.8.3.1. Application in forward chaining

Treatment in forward chaining is a cyclic process that takes place in two stages (see Figure 2):

a) Triggering of an event (we also say triggering or firing a transition) on an object in the system;

b) Propagation of the effects of the event on the system by cyclic application of the interaction rules until the state stabilises.

a) Triggering an event

When the system is in a given state, certain events can occur. This means that the conditions of certain occurrence rules for certain objects are satisfied. The software program chooses a[3] rule for an object and applies it (i.e. executes the actions induced by the event), thus altering the object's state variables. The system is then in a state we shall describe as **incomplete**, since only the consequences of the event on the object affected are taken into account. An incomplete state is not a real state of the system.

b) Application of the interaction rules

In order to propagate the consequences of the event throughout the system, the treatment then applies the interaction rules, and proceeds as follows:

The treatment applies the interaction rules object by object.

For each object, the treatment takes the "first" rule of the object and determines its applicability depending on the current state of the system (initially, this state is the incomplete state obtained after firing a transition of an occurrence rule). If the rule is applicable then the treatment applies it, calculates the system's new current state and moves on to the next object rule.

When all the rules of an object have been examined, the treatment moves on to the next object.

Remark: It is very important to note that a new system state is calculated after each application of an interaction rule for an object. So, if an object rule is not initially applicable, it may become applicable after the first rules have been applied.

When the treatment has passed once into each of the interaction rules, a new system state is obtained. However, this is not necessarily a complete state corresponding to a real state of the system: the rules are again applied from the first to the last (in the same order as in the first "application"). If, at the end of two successive cycles the state obtained is the same, then we say that the process has converged and we can stop the inference. This means that we have reached a "fixed point".

Example:

[3] We shall see later that in certain cases, several events can be applied simultaneously.

Take the simple example of 3 thermohydraulic components A, B and C connected in series. We have the following interaction rule in the knowledge base:

```
CLASS component;
INTERFACE
    upstream KIND component;
ATTRIBUTE
    fed DOMAIN BOOLEAN;
INTERACTION
    r1
    IF FOR_ALL x AN upstream WE_HAVE fed OF x = FALSE
    THEN fed <-- FALSE;
```

This generic rule r1 means:

Let O be some arbitrary object of class component,
If $\forall x \in$ {components upstream of O}, x is not fed,
Then O is not fed.

The facts base as initialised by the user is as follows:

```
fed OF A = TRUE
fed OF B = TRUE
fed OF C = TRUE
```

We assume than an event that has occurred in the system has given rise to 'fed OF A = FALSE'. The incomplete state of the system after this event is then:

```
fed OF A = FALSE
fed OF B = TRUE
fed OF C = TRUE
```

The rule of interaction r1 instantiated on object B is as follows:

```
OBJECT B;
INTERACTION
    r1
    IF fed OF A = FALSE
    THEN fed OF B = FALSE;
```

Rule r1 for object B therefore applies. After application, the new system state is:

```
fed OF A = FALSE
fed OF B = FALSE
fed OF C = TRUE
```

Taking account of this new state, rule r1 instantiated on object C applies (it was not so initially). The new state obtained for the system is:

```
fed OF A = FALSE
fed OF B = FALSE
fed OF C = FALSE
```

The interaction rules allow the effects of an event to be propagated from rule to rule. **Hence these rules should be invoked as often as is necessary** so that all the consequences of an event may be taken into account. When the fixed point has been reached[4], that means that all the consequences of the event have been taken into account.

After passing into the interaction rules we get a new model state (complete state) for which **other events can potentially occur**.

Example:

[4] Such a fixed point does not always exist. See § 9 on model consistency to ensure its existence.

Take the previous example, assuming that the system examines object C before object B. In this case, during the first pass only the rule instantiated on object B is applied, and the state obtained after the first pass is:

```
fed OF A = FALSE
fed OF B = FALSE
fed OF C = TRUE
```

During the second pass into the interaction rules, the instantiated rule on object C will be applied, and we obtain the same final state as before.

Remark on the order of application of the interaction rules

It is very important to note that the **order of the interaction rules must have no consequential effect on the result obtained** (This is true in our example where, regardless of the order in which the objects are taken into account, the result obtained is the same. The only difference is the number of rule "cycles" necessary to stabilise the state).

Hence the designer of knowledge bases should be aware of the **commutativity of the interaction rules**, since he/she cannot control the order in which the rules are applied by the inference engine.

For complex cases, we shall see further on in this manual (see § 7.5) that it is possible to introduce a partial order into the application of the interaction rules by combining these into subsets of rules (steps) and applying each subset one after another. Chapter 8 is dedicated to model consistency and explains the precautions to be taken in the use of the steps.

Using a Figaro model with forward chaining

Using a Figaro model with forward chaining (the most "natural") allows all the states of a system to be explored as well as the events linking these states. This type of implementation can therefore be used:

- In interactive simulation mode: the user himself chooses the sequence of transitions he wants to simulate and can therefore see the effects on the system at each transition.
- To construct the system's state transition matrix and thereby calculate the probability of each state (this is only possible if the model is Markovian, i.e. if all transitions are instantaneous or exponentially distributed).
- To explore the **sequences** of the state graph so as to calculate the probability of the sequences leading to a failure state in the system (A sequence is a series of events leading from the initial state to some particular state).
- To carry out a Monte Carlo simulation (see §3.4).

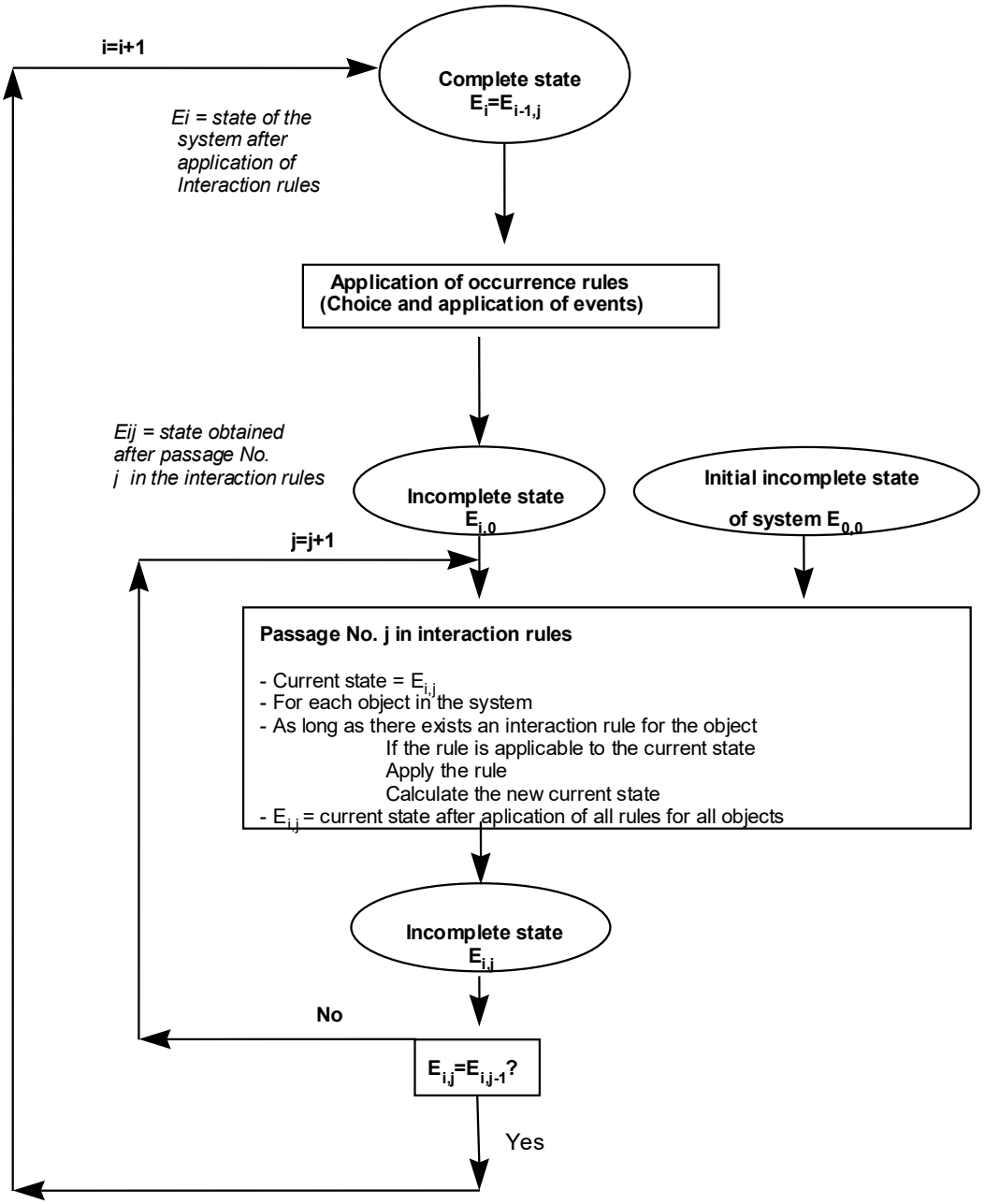

Figure 2. Flow diagram showing the principle of forward chaining in Figaro

2.8.3.2. Application in backward chaining mode

Application of the Figaro rules in backward chaining mode is used to generate a **Fault tree** based on:

- the Figaro 0 model associated with the system to be investigated,
- a description of the undesirable event that we seek to study, defined by occurrence fact of type: equality of a model variable with a given value.

  Example: A Figaro model representing a thermohydraulic system contains objects that represent thermohydraulic components. In particular, each object is described by a Boolean variable *fed* which defines whether or not the object is being fed with fluid. If object B1 represents the aim of the system to be fed, the undesirable event can be described by the fact *fed (B1) = FALSE*.

To construct the Fault tree, the processing proceeds in two stages:

a) construction of a "causal tree" (consisting uniquely of logic gates) using backward chaining on the interaction rules in the Figaro model,

b) transformation of the "causal tree" into a Fault tree by using the occurrence rules in the model or indeed the reliability models associated with the failures to create probabilistic failure models at the leaves of the tree.

a) Construction of the "causal tree" via backward chaining on the interaction rules.

To construct the "causal tree" associated with the undesirable event, the interaction rules in the model are applied in backward chaining mode.

"The causal tree" associated with an undesirable event is a tree that sets out the logical relationships between the undesirable event and basic events. A basic event is something that cannot be explained in the form of other facts occurring in the interaction rules. It corresponds to:

* either a failure in a system component (e.g. "failure on demand of the pump p1 = TRUE"),
* or an expression representative of the state of the system (e.g. "position of valve v1= open").

In order to construct the causal tree associated with an undesirable event, the following process is applied:

Starting from the undesirable event to be explained, we select all the interaction rules (RV1, ..., RVn) that pertain to this fact (whose actions make this fact satisfied), as well as all the interaction rules (RF1, ..., RFp) for which the conclusion is incompatible with the fact (whose actions give a different value to the variable).

The reasoning applied is as follows: in order for the fact to be satisfied, it is necessary and sufficient that one of the rules RV1...RVn (conditions CV1...CVn) applies and that none of the rules RF1...RFp (conditions CF1...CFp) applies. Then a first part of the tree is constructed as follows:

Undesirable event = (CV1 or CV2….or CVn) and not (CF1 or CF2 … or CFp)

Backward chaining continues by applying the same reasoning to each of the undeveloped conditions of the tree.

Remark: It is important to note that, for the treatment in backward chaining mode to be applicable, the interaction rules must, in particular, satisfy the following restrictions:

- the condition of a rule must be a simple Boolean expression combining via operators atomic Boolean expressions of type:
  "variable (object) = value",
- the actions of a rule must be of type "variable (object) <-- value".

Chaining stops when, for each leaf of the tree:

- no interaction rule concludes on the fact represented by the leaf or on some incompatible fact,
- the leaf is already existent in the causal tree.

"The causal tree" obtained is therefore a logical tree that incorporates the logical gates AND, OR, k/n and possibly negations, and where the leaves of the tree are simple Boolean expressions (tests on the value of a variable).

b) Transformation of the Causal tree into a Fault tree

To transform the causal tree into a Fault tree, the treatment uses the model's occurrence rules and the reliability models associated with failures. For each leaf in the causal tree:

- If there exists an applicable occurrence rule (i.e. for which the condition is satisfied for the initial state of the model) whose actions satisfy the fact associated with the leaf, the leaf is replaced by the event of the occurrence rule and, depending on its type (instantaneous or time-delayed), an appropriate leaf model is created.

  Example:

  A tree leaf contains the fact "failure in pump P1 = TRUE" (*fail OF P1 = TRUE*). The Figaro model contains the following occurrence rule for the class pump.

```
CLASS pump;
ATTRIBUTE
   fail DOMAIN BOOLEAN ;
CONSTANT
   initial_state DOMAIN 'running' 'stop' ;
OCCURRENCE
   IF initial_state = 'running'
   MAY_OCCUR
          FAULT fail
          LABEL "failure in operation"
          DIST EXP (lambda_pump)
          INDUCING fail <-- TRUE;
```

  This occurrence rule is applicable for object P1 which is initially running. The treatment therefore replaces the leaf in the causal tree with the basic event described by the rule above, associated with which is a failure rate *lambda_pump*.

- If the leaf in the tree corresponds to the fact that a FAILURE (see §6.2.4.4) is TRUE and if a reliability model exists that is associated with this failure, then this model is put directly into the leaf. This second possibility has been introduced to generate fault trees with varied, sometimes complex, leaf models; it would have been impossible to recognize all types of leaf models on the basis of the occurrence rules. In some sense this mechanism competes with the previous one. Hence we have to choose which one we wish to implement, and if necessary use distinct groups of rules (see § 7.4) for implementing the model in forward and backward chaining modes; thus we can use the occurrence rules for forward chaining and the reliability models associated with failures for backward chaining.

- If there is no occurrence rule giving rise to the fact associated with the leaf, and if this fact is not a failure associated with a reliability model, then the associated Boolean expression is evaluated according to the initial state of the model and the causal tree is simplified depending on the result obtained. The simplification rules used are simple: a TRUE leaf under an AND gate is eliminated, a TRUE leaf under an OR gate makes the OR gate TRUE, etc.

  Example:

  Take the example above and assume that no occurrence rule in the model gives rise to *def (P1) = TRUE*. In this case, if the value of the variable *def* is TRUE for *P1* in the initial state of the system, then the leaf is set to TRUE and the tree is simplified in consequence.

The fault tree thus enables an **undesirable state** of the system (state defined by a constraint of the form "a given variable has a given value") to be explained by the basic events (component failures) that can occur in the system from a given **initial state**. This is a complete initial state, obtained by an application of the interaction rules in forward chaining mode on the basis of the initial state defined by the user, assumed in all cases as incomplete.

3. Formal definition of the Figaro 0 language

The Figaro language of order 1 is complex, and it is therefore difficult to give a formal definition for it: such a definition would be unreasonably long and hard to read. On the other hand, it is possible to give a simple formal definition for the Figaro language of order 0.

Insofar as the language of order 1 is just a convenient means for generating models of order 0, these being the starting point for any treatment, it is sufficient to define formally the language of order 0.

In order to keep the presentation simple, we will initially sidestep the possibility offered by Figaro of defining the Dirac probability distributions that model the deterministic initiation of events at the end of fixed time periods, or those distributions (such as the uniform distribution over an interval or the triangular distribution) whose measure is zero over certain intervals included within $[0,+\infty[$. The majority of models do not invoke these types of distribution that have a qualitative effect on behaviour (the existence or otherwise of certain sequences of transitions between states). This restriction means that we will not need to detail the time-dependent behaviour of the automaton defined by a Figaro 0 model: we will be content to define the states towards which the automaton might evolve from a given state, and if the change of state is instantaneous or takes place after some non-zero time. Only in section 3.4 will we treat particular cases by introducing a schedule, a relatively complex idea and irrelevant for common cases. On the other hand, the distinction between instantaneous states and non-instantaneous (or "tangible") states is a **necessity** in all models for giving the language sufficient modelling power. These two types of state are not treated in the same way, as we shall see in the section that defines how the automaton can evolve from some given state.

3.1. Elements of the model

Let *L* be a set of probability distributions belonging to one of the following two categories: continuous distributions on $[0,+\infty[$, and discrete distributions associating probabilities with a finite number of values.

A model in Figaro 0 is a 7-tuple *(Ξ, O, T, I, σ, Y_0, V_0)* consisting of the following elements:

Ξ is the Cartesian product $E_1 \otimes E_2 ... \otimes E_n$ of the domains of the components of *X*, a finite vector of state variables *(x_1, x_2, … x_n)* whose value defines the state of the model at some instant *t*. Each state variable x_i is either a Boolean variable, an integer, a real number, or a so-called "enumerated" variable that can take values in a finite set E_i. *X* is the concatenation *V*, *Y* of two vectors *V* and *Y*. *V* groups the so-called "**essential**" variables and *Y* groups the so-called "**deduced**" variables.

V_0 is the initial value of *V*, and Y_0 is the so-called "reinitialisation" value of *Y*. ***Y* is a function of *V* and Y_0 defined using the function *I* described later**.

T is the set of **transition groups** in the model. A **transition** is an application of *Ξ* in *Ξ*, which maps any state vector to some other state vector, in general obtained by modification of a small number of essential variables. A transition group is a doublet whose elements are a set of so-called "tied" transitions, and a probability distribution from *L*. The transition group contains a single transition if its distribution is not a discrete distribution. If the distribution is discrete, the transition group contains as many transitions as the distribution has values. A transition represents the local change of state of a component in the system being modelled, e.g. the appearance of a failure mode. *A group of transitions associated with a discrete distribution models the various possible outcomes of a non-deterministic process that is assumed to be instantaneous*; for example, the various results of the throw of a die, or the choice between successful start-up of a component or otherwise. By abuse of language, we shall combine the idea of "transition group" and "transition" for groups containing just one transition. The idea of a group of tied transitions is a novel feature of the Figaro language with respect to the set of formalisms intended to describe non deterministic automata, starting with Petri nets. The importance of this will become apparent later when we describe how the automaton can evolve from a state for which *several instantaneous transition groups* are applicable.

O is a function of *Ξ* in $\mathcal{P}(T)$. In practice, *O* is defined by the set of so-called "**occurrence rules**". These rules associate a (possibly empty) set of transition groups with some arbitrary state *X* of the model. The transitions belonging to the groups of *O*(X) are said to be **"valid"** in state *X*.

I is a function of *Ξ* in *Ξ*, which maps any state vector *X* **for which *I* is defined** to another state vector. In practice the function *I* is defined by the so-called "**interaction rules**" in the Figaro model (and perhaps a **system of linear equations**), and this function **could depend on the order σ of the rules**.

Function I is defined as the composition of a finite set of functions of Ξ in Ξ, denoted $I_0, I_1, \ldots I_P$. In other words, $I(X) = I_p(I_{p-1}(\ldots I_0(X)\ldots))$.

The model's interaction rules are combined as "steps" (see § 7.5), corresponding to the different functions $I_1, \ldots I_P$. I_0 is a particular function for reinitialising the deduced variables: $I_0(V, Y) = V, Y_0$

3.2. Inference carried out in the interaction rules

The function corresponding to each step is defined by a **deterministic automaton** whose operation is that of a simple inference engine. Let $\{R_k\}$ be the set of rules of a given step (to simplify the notation we shall omit the step index). Rule R_k is a function of Ξ in Ξ which maps any state vector X to another state vector $R_k(X)$. In practice, each rule consists of a **condition** calculated by a Boolean function of state X that allows its application, and **actions**, which are most of the time assignment instructions for state variables, by constant values or values calculated from X. There is another possible type of action, for the moment used only in certain knowledge bases for electrical systems: the solution of a system of linear equations, used to calculate a new value of X.

The inference engine applies the rules (R_k) cyclically, *arranged according to order* σ until the same value of X has been obtained at the end of two successive iterations of the rules.

Applying a rule amounts to assessing its condition and, if true, carrying out the corresponding action. In particular, if the condition is not realised, vector X does not change.

The way the interaction rules work allows great flexibility in the modelling process, in particular with regard to the propagation of fluid or flow of electricity in a system. The first versions of the Topase knowledge base (developed by EDF) allowed current and voltage to be calculated at any point in a circuit according to Ohm's law; later these calculations were carried out much more quickly and accurately by solving a system of linear equations.

3.3. Semantics of the complete model

Given that the various elements of a Figaro 0 model have been described, it is easy to define the semantics of the Figaro 0 language. These semantics equate to the operation of the following **random** automaton:

Initialisation:

An initial **incomplete** state is defined by the user who specifies V_0 (sole restriction: respect the domain of V). Then $X_0 = I(V_0, Y_0)$ is calculated, which is the initial complete state of the model.

Note in passing that this way of defining the initial state of the system allows a number of thorny problems in a complex model to be solved: the risk of defining an inconsistent state (e.g. there are two electrical components in series; the current is zero in one but not in the other), or even that the user finds it impossible to define the initial state (e.g. this might be the case if he has to give the currents and voltages at all points in an electrical circuit). Hence defining the initial state is easy and safe for the user: he has few variables to initialise, and the ones there are have a clear meaning and are easy to determine (such as the position of a valve, the number of components in service at the initial instant, etc.).

Change from a current state X:

$O(X)$ defines the applicable transition groups (we also say valid) from X.

- If this set is empty, the current state is absorbing.
- If it is reduced to a group containing a single transition t, the only state the automaton can reach, from state X, is $I(t(X))$.
- If it contains several groups of transitions, there are two cases to be considered (the breakdown that follows constitutes a partition of the set of possible situations):
  - $O(X)$ contains **only** time-delayed transitions, associated with continuous distributions. In this case, state X is said to be "tangible" and the next state that the automaton will take will be $I(t(X))$, where t is some arbitrary transition in $O(X)$ (non-determinism).
  - $O(X)$ contains the groups $G1, G2.., Gn$ of transitions associated with discrete distributions. State X is then said to be "instantaneous", and the continuous distribution transitions are ignored. In this case, the next state taken by the automaton will be $I(t1 \circ t2 \circ \ldots tk (X))$, an expression in which ti represents some arbitrary choice (non-determinism) of transition in the group Gi: $ti \in Gi$. The order in which the transitions are applied (in other words, the choice of indices for the groups) is not given. This can create problems

of consistency, which will be discussed in chapter 8.

3.4. Semantics of a model with arbitrary probability distributions

It is easy to see that, when a model has two concurrent transitions A and B whose conditions are validated at the same time, and with Dirac distributions having parameters Delay_A < Delay_B, only the sequence A, B can be observed. The parameters of the distributions therefore have an effect on the **qualitative** behaviour of the model; this was not the case in the previous sub-sections.

Where we are dealing with arbitrary probability distributions, the only means of explaining the model's semantics is by describing the algorithm that uses the model to carry out a Monte Carlo simulation. This algorithm makes use of the previously defined functions for the rules of occurrence and interaction.

It is based on managing a schedule: this is a list of transitions associated with precise times. Formally, a schedule is defined as S= ((τ1, t1), (τ2, t2), ….(τn, tn)). The second elements in the doublets are the transitions, while the first elements are the associated times. Events are ordered in increasing time: $\tau 1 \leq \tau 2 \leq ...\tau n$.

To simulate a "history" of the model over time interval [0, T_m], the following algorithm is used:

```
Create an empty schedule S

Initialise the variable that represents the simulated time: t = 0

Calculate the complete initial state X= I(V0,Y0)

Calculate the complete initial state X = I(V0,Y0)

While t ≤ Tm:

     1 Calculate O(X): list of groups of valid transitions

     2 If some groups of instantaneous transitions are valid:

            Draw one transition per group at random (taking account of the
               discrete distributions) and apply it. X becomes X'

            Calculate X=I(X') to propagate the effects of the instantaneous
               transitions

            Go back to 1

     4 Update the schedule:

       Draw at random the firing instants of the valid transitions that are
         not already in S, and insert them into S

       Remove from S the transitions that are no longer valid

     5 Find the transition (τmin, tmin) corresponding to the smallest date >t
         in S (if there isn't one, the history terminates via an absorbing
         state)

     6 Apply tmin. X becomes X'

     7 Calculate X=I(X') to propagate the effects of tmin

     8 Set t = τmin  // the date of the transition becomes the current time
```

Looking at this algorithm in detail, we see that there are situations that demand further explanation.

Just as several groups of instantaneous transitions can be validated at the same time, in step 5 of the algorithm

several delayed transitions can occur at exactly the same instant if they are defined by Dirac distributions. We then calculate X' by adding up the cumulative effects of the transitions before propagating the effects at step 7.

What happens with a transition whose distribution parameters change without there being any interruption of the transition's conditions of validity? Several choices can be justified, which are further clarified in article [12]. The choices made for Figaro are as follows:

- For any distribution other than Dirac, a new date is drawn depending on the new parameters, and replaces the one previously associated with the transition in the schedule.
- For the Dirac distribution, the user is left with two choices (this choice must be made in the calculating tools).
    a) the date present in the schedule is kept.
    b) the date present in the schedule is replaced by the date on which the parameter changed plus the new value of the parameter.

These options are not necessarily appropriate for what the user may want to model. He is still able to perform his own calculations by using the variable CURRENT_DATE and the mathematical functions associated with the probability distributions in order to calculate, within the interaction rules, a delayto put into a Dirac distribution (with option b); this is a little complicated, however.

Fortunately, in the very common case of the exponential distribution, no question arises: since this distribution models an absence of memory, there is just one possible option, which involves recalculating the date of the transition (as is specified for Figaro).

"Memory" distributions

Imagine that an object has been placed on a conveyor belt at t=0. We can use the speed of the belt to calculate that it will arrive at some destination at time t1. Hence we can model its arrival by a transition with a Dirac distribution. But what if the belt is stopped from time to time? We can see the advantage of a transition having a memory; this transition will be initiated when the *sum of the times during which the transition was valid* gets to t1.

```
CLASS object_on_belt;
DIST_PARAMETER t1 DEFAULT 10;
ATTRIBUTE in_motion DOMAIN BOOLEAN DEFAULT TRUE;
OCCURRENCE
    IF in_motion     (* this attribute may alternatively be TRUE and FALSE
    depending on the state of the system *)
    MAY_OCCUR
    TRANSITION arrival DIST C_T_M (t1)
    INDUCING in_motion <-- FALSE;
```

This easily understandable case may be generalised to any distribution **other than the exponential** (which by definition is memoryless). This why all the names of the distributions except the EXP distribution have a "double" designating the corresponding distribution with memory, obtained by adding the suffix "_M". The way all these distributions work is the same: the transition is fired when the sum of the times during which the transition is valid reaches the number drawn at random when it was first validated. The counter is reset to zero during the firing. Thus we can easily model a motor which wears out only when running by using a Weibull distribution with memory.

4. Lexical elements of the Figaro language

This chapter describes the various basic rules to be followed in writing a correct Figaro model from a lexicographical point of view. This document describes only the English version of Figaro. For the French version please refer to the companion document which describes all the syntax in detail, in both French and English.

4.1. Alphabet

All ASCII (7 bit) characters are acceptable. The syntactical analyser does not allow accented French characters (except for comments).

A few basic definitions are given below:

- Word **separators** are: space, tab, return and page-break. Any number of these can occur.

- A **letter** is one of the following characters: a...z A...Z _ (underscore).

- A **digit** is one of the following characters: 0...9.

- The **end of a field description** and the end of a **rule description** are represented by a semi-colon ';'.

- The semi-colon is also used as a high-level separator to:

  * mark the end of a description of the "field" CLASS (see CLASS),
  * separate the steps defined for a knowledge base (see STEPS_ORDER),
  * separate the groups defined for a knowledge base (see GROUP_NAMES),
  * separate the systems of equations defined for a knowledge base (see SYSTEM_NAMES).

- The separator for the elements in a **list** is a comma ','. Lists are used mainly:

  * to define several actions for the same rule (see Action),
  * to assign a set of values to the slot EDITION in a field (see EDITION),
  * to define a set of identifiers for the same STEP (see STEP in Advanced notions).

- The separators for the elements of an **enumeration** are space or tab. Enumerations are used mainly:

  * to define the values of an enumerated DOMAIN (see DOMAIN),
  * to define the set of GROUPS a given rule belongs to (see GROUP in Advanced notions),
  * to define in a system description (list of objects), the set of objects connected to the same object in an interface,
  * to define, for a multi-valued heritage, the set of classes inherited by the same CLASS (see EXTENDS and Inheritance).

- A **character string** is written between double quotes (") and may contain spaces. Character strings are used only to define the label of a field (see LABEL).

- a **symbolic (or alphanumeric) value** is written between apostrophes (') and may contain spaces. Symbolic values are used only to define the different values of an enumerated domain (see DOMAIN).

- A **comment** starts with '(*' and finishes with '*)'. Any character is permissible between the two delimiters. Comments can be nested.
Example:

```
CLASS valve
      EXTENDS component;  (* the ";" denotes the end of a CLASS field *)
ATTRIBUTE
      state  EDITION       VISIBLE, MODIFIABLE, NOT MANDATORY
                           (* list *)
             LABEL  "state of valve"
                           (* character string *)
             DOMAIN 'open' 'closed' 'unknown'
                           (* enumeration of symbolic values *)
             DEFAULT 'open';  (* the ";"denotes the end of the
                                 description of the field state*)
      (* comment
            (* nested comment *)
       *)
```

Remark: A carriage return is not permissible in a symbolic value (between '). On the other hand, carriage return is permissible in comments and in character strings (between ").

Remark: The maximum length of a string or symbolic value is 200 characters.

4.2. Numerical values

- **Boolean constants** are: TRUE and FALSE (capitals).

- **Numerical constants** (or numbers) are of two types: integer or real. An integer is a number having several digits (preceded if necessary by the sign '-'). A real number has the following form:

```
<integer part>.<real part>e<power of ten>
```

The power of ten is a positive or negative integer. The integer part is a positive or negative integer. The real part is an unsigned integer.

```
Examples of numerical constants:
          25           (* integer *)
          -135         (* negative integer *)
          256.012      (* real number *)
          -58.5e-5     (* negative real number with a power of ten *)
```

4.3. Identifiers

An identifier gives a name to a class, object, field, special operator or rule.

There are two ways to describe an identifier:

* Either by a sequence of letters and digits always starting with a letter and containing no space (the character _ is often used to make the identifier easier to read).

* Or by a sequence of letters and digits bracketed by vertical bars '|'. In this case, the name of the identifier may start with a digit and may contain spaces and tabs (carriage return is not permitted).

The maximum length of an identifier is 200 characters.

Examples:

```
ABCDEF (* permissible *)
abcdef (* permissible. These two symbols are different *)
A1234        (* permissible *)
12ABCD (* not permissible !!! Will be read as 12 ABCD *)
|12 AbDEF|   (* permissible *)
```

Remark: To ensure that the Figaro model can be easily read, it is preferable not to use identifiers that are the same as keywords, and not to use two identical identifiers to represent different data.

Example not to be followed:

```
CLASS component;
ATTRIBUTE
    attribute DOMAIN INTEGER;
    (* NOT RECOMMENDED: attribute (the field)and ATTRIBUTE (keyword)*)
```

4.4. Keywords

Keywords in Figaro must be written in capitals, otherwise they will be taken as variable identifiers.

4.4.1. Operators

Meaning	Figaro
assignment operator	<--
Boolean operator, equality assignment in Figaro 0	=
Boolean operator not equals	<>
Boolean operator strictly less than	<
Boolean operator less than or equals	<=
Boolean operator strictly greater than	>
Boolean operator greater than or equals	>=
Multiplication operator	*
Division operator	/
Addition operator	+
Subtraction operator	-
modulo (a % b denotes the remainder of integer division of a by b)	%
power (a**b denotes a to the power of b)	**
Boolean negation	NOT
Boolean operator AND	AND
Boolean operator OR	OR

4.4.2. Keywords

The table below is an exhaustive list sorted alphabetically. The length of this list should not be daunting since it contains words that are frequently used at the very heart of the language, and words that have highly specific uses, especially those used to define probability distributions and their associated parameters. To aid the reader, the following colour code is used:

Blue: mathematical functions, e.g. COS for cosine;

Violet: names of probability distributions, e.g. WEI or WEIBULL for the Weibull distribution;

Green: names of probability distribution parameters.

Table of all Figaro keywords (excluding operators defined by symbols):

A (= AN)	EXTENDS	LOG	RELIABILITY_DATA
ALPHA	FAILURE	MANDATORY	REPAIR
ALREADY_REALIZED	FALSE	MAX	REPAIRS
AN (= A)	FAULT	MAXIMUM	REPLACES
AND	FLOOR	MAY_OCCUR	ROLE
AT_LEAST	FOR_ALL	MIN	SIN
ATTRIBUTE	FORMULA	MINIMUM	SOLVE_SYSTEM
BETA	FRECHET	MODEL_F	SQRT
BOOLEAN	FRECHET_M	MODEL_FROZEN	STANDARD
C_T	FREQUENCE	MODEL_G	STATE_TIME
C_T_M	GAMMA	MODEL_GLM	STEP
CARDINAL	GAMMA_NO_DETECTION	MODEL_GLTM	STEPS_ORDER

CEIL	GAMMA_RECONFIG	MODEL_PET	SUCH_THAT
CLASS	GAMMA_TEST	MODEL_REPLACED	SUM
CONDITION	GIVEN	MODEL_RT	SYSTEM_NAMES
CONFIGURATION	GROUP	MODEL_WB	SYSTEM_OBJECT
CONSTANT	GROUP_NAMES	MODIFIABLE	T_INIT_TEST
CONSTANT_TIME	GUMBEL	MT	T_INTER_TEST
COS	GUMBEL_M	MU	T_TEST
CURRENT_DATE	IF	MYSELF	T0
CYCLE	INCLUDED_IN	NINT	TAN
CYCLE_M	INDUCING	NORMAL	TEMPORARY
DEFAULT	INFINITY	NORMAL_M	THEN
DEFAULT_MODEL	INS	NOT	THERE_EXISTS
DESCRIPTION	INSTANTANEOUS	OBJECT	TO
DESIGN	INTEGER	OCCURRENCE	TRANSITION
DIST	INTEGRAL	OF	TRIANG
DIST_PARAMETER	INTERACTION	OF_CLASS	TRIANG_M
DO	INTERFACE	OF_TERMS	TRUE
DOMAIN	IS_A	OR	UNAVAILABILITY
EDITION	KB_DESCRIPTION	OR_ELSE	UNI
EFFECT	KIND	PARETO	UNI_M
ELSE	LABEL	PARETO_M	VERIFYING
EQUATION	LAMBDA	POINT	VISIBLE
EQUATION_SYSTEM	LAMBDA_TEST	POINT_M	WE_HAVE
ERL	LGN	PRODUCT	WEI
ERLANG_M	LGN_M	RAND	WEI_M
EXP	LINEAR	REAL	WEIBULL
EXPONENTIAL	LN	REFERENCE	WITHIN
		REINITIALISATION	WORKING

Please note that the keywords THERE_EXISTS, FOR_ALL, EXTENDS have replaced the keywords (respectively) IT_EXISTS, FOR_ANY, KIND_OF. This change was made at the end of 2018, but for compatibility reasons, the Figaro tools still accept the deprecated versions of these keywords.

5. Expressions

In this part we shall begin by defining precisely what the variables in the Figaro language are, and how to access them. Then we will define the syntax of the expressions, starting with simple expressions (similar to those of standard programming languages) and finishing with expressions using complex operators that are specific to Figaro (quantifiers, complex Boolean and numerical operators).

5.1. Preliminary observations

5.1.1. Value fields and global variables

Expressions written in Figaro will mainly use two types of variable:

* descriptive fields of classes and objects (attributes, constants, etc.),
* global variables: these are descriptive fields of "system objects" (see § 7.2).

Value fields are fields used to describe the characteristics of a class or some particular object.

```
CLASS rectangle;
CONSTANT
    length DOMAIN REAL;
    width DOMAIN REAL;
    area DOMAIN REAL DEFAULT (length*width);
```

In this example, the expression *length*width* is an expression using two constants of class rectangle to define a third one. The order of the declarations is not important. The only restriction to be observed is to avoid creating a circular loop of dependencies, since that will generate an error message.

5.1.2. Defining a set

5.1.2.1. Ideas of sets and tied variables

As we shall see in the remainder of this part, Figaro allows the use of quantifiers and complex numerical operators that are used on **sets** (these could be empty: see § 5.1.2.6) in order to carry out tests on a series of objects. In a knowledge base, this capability is essential in order to write **generic rules** that model the behaviour of an arbitrary object of a class.

A **logical quantifier** is defined as a symbol that allows the use of a variable defined on a particular set. There are two types of quantifier in Figaro:

* the universal quantifier: $\forall$ x $\in$ set E,
* the existential quantifier: $\exists$ x $\in$ set E.

Similarly, the **complex numerical operators** sum, product, minimum and maximum allow calculations on a set of variables:

- $\sum_{x \in set} f(x)$
- $\prod_{x \in set} f(x)$
- same principle for minimum and maximum.

In Figaro, quantifiers and complex numerical operators are used only on **sets of objects**. The quantifier (or operator) will be used with a **tied variable** representing an arbitrary object in the set.

Example

```
CLASS pump;
    ATTRIBUTE
          failure DOMAIN BOOLEAN;

CLASS normal_pump EXTENDS pump;

CLASS backup_pump EXTENDS pump;
    ATTRIBUTE
          start_request DOMAIN BOOLEAN;
    INTERFACE
          backed_up KIND normal_pump CARDINAL 1 TO INFINITY;
    INTERACTION
    IF FOR_ALL x A backed_up WE_HAVE failure OF x = TRUE
    THEN start_request <-- TRUE;
```

This interaction rule means:

Let O be some arbitrary object of class backup_pump
If $\forall$ x $\in$ {normal pumps backed up by O}, x has failed, (x is a tied variable)
Then there is a request for O to start

To use quantifiers or complex numerical operators in Figaro it is therefore necessary to define sets of objects. In the remainder of this part we shall see that there are two main ways to define a set of objects in Figaro.

5.1.2.2. Using an interface field to define a set

The first way to define a set of objects in relation to an object in a class is to use an **interface field** of the class.

Example 1:

```
CLASS component;
```

```
ATTRIBUTE
    linked DOMAIN BOOLEAN;
INTERFACE
    upstream KIND component;
INTERACTION
    IF THERE_EXISTS x AN upstream SUCH_THAT linked OF x = TRUE
    THEN linked <-- TRUE;
```

If O is some arbitrary object of class component, *upstream* denotes the set of components linked with O by the upstream relationship.

In this example, the set defined for the variable tied to the quantifier 'THERE_EXISTS' is defined by **the identifier of an interface field**. This way of defining a set of objects with an interface is the method most often used in Figaro. The following two examples illustrate some more complex set definitions.

Example 2:

```
CLASS X;

CLASS Y;
INTERFACE
    i1 KIND X;

CLASS Z;
INTERFACE
    i2 KIND Y CARDINAL 1;
INTERACTION
    IF THERE_EXISTS x AN i1 OF i2 ....;
```

If O is some arbitrary object of class Z, the expression *i1 OF i2* denotes the set of objects of class X in relation i1 to **the unique** object of class Y in relation i2 with O.

In this example, the set defined for the variable tied to the quantifier 'THERE_EXISTS' is defined by **'interface field identifier' OF 'set of cardinality 1'.**

Example 3:

```
CLASS X;

CLASS Y;
    i1 KIND X;

CLASS Z;
INTERFACE
    i2 KIND Y;
INTERACTION
    IF THERE_EXISTS y AN i2 SUCH_THAT (THERE_EXISTS x AN i1 OF y SUCH_THAT..)...;
```

If O IS some arbitrary object of class Z:

* the expression i2 denotes the set of objects of class Y in relation i2 with O,
* the expression i1 OF y denotes the set of objects of class X in relation i1 to y, an element of the previous set.

In this example, the set defined for the variable related to the second quantifier 'THERE_EXISTS' is defined by **the 'interface field identifier' OF 'tied variable'.**

5.1.2.3. Using a class to define a set

There is another way to access a set of objects. It is sometimes beneficial in Figaro to be able to address all the objects of a **particular class**. This is why Figaro allows the use of a class name to define a set.

To define a set using a class name, we use a syntax of the form:

∀ x AN OBJECT OF_CLASS 'class name'

Example:

```
CLASS component;

CLASS pump;
INTERACTION
IF FOR_ALL x AN OBJECT OF_CLASS component WE_HAVE ...
THEN ... ;
```

The expression *AN OBJECT OF_CLASS component* denotes the set of instances of class *component* in a system.

5.1.2.4. Limit of Figaro

The sets of objects handled in Figaro are **constant sets**.

As we have previously mentioned, the links between the objects in a system (interface field of objects) do not change over the course of the system's life. In other words, the rules of Figaro cannot add or remove an object to or from an interface.

Similarly, it is impossible in Figaro to add to the system an instance of a class (or to delete an instance of a class).

So, whether we use interface fields or class names to define a set of objects, the set will be constant over the course of the system's life.

5.1.2.5. Restriction of a set

Sometimes it so happens that we wish to **restrict** the set of objects in an interface or the set of objects of a class. It is therefore possible in Figaro to use constraints to reduce the number of objects in a set.

There are two optional constraints in Figaro introduced by the keywords OF_CLASS and VERIFYING. These two constraints can be used together or separately. They apply to the definition set of the quantifier or complex operator.

The constraint OF_CLASS:

This constraint restricts the class of objects in the set. **The constraint can only be used when the set is defined with an interface field.** The objects in an interface are all of a particular class, specified in the declaration by the keyword KIND. However, it can be beneficial to reduce this set of objects to a subset. As an example, take the following hierarchy of classes:

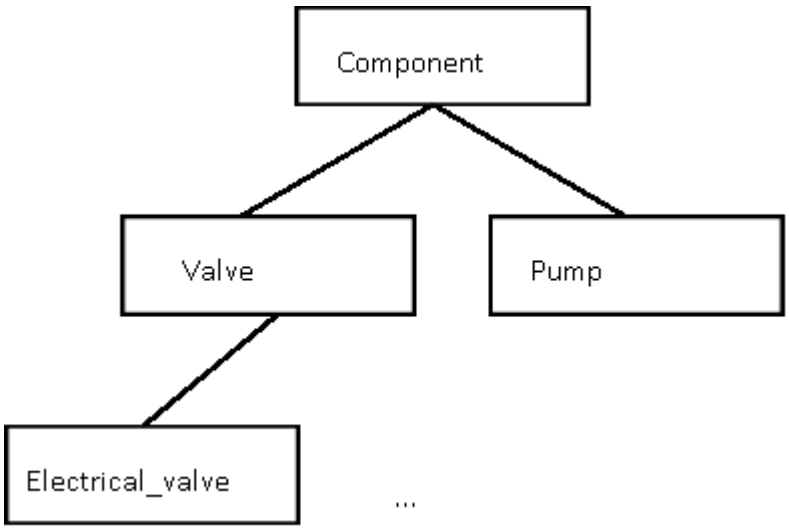

Figure 3. Example of hierarchy of classes

Component is a generic class inherited by the classes *valve* and *pump*. *Electrical_valve* is a class that specifies the class *valve*. Thus, if the interface class is *component*, every instance of the 4 classes above may be included in the interface. Since these classes inherit from *component*, they are components. The advantage of the constraint OF_CLASS then becomes apparent.

The following example reveals the use of this constraint: the class *valve* is a sub-class of *component* and the objects interfaced with the class *pump* are of class *component*. In the rule we are only interested in objects of class *valve* that are interfaced:

```
CLASS component;
CLASS valve EXTENDS component;
CLASS pump EXTENDS component;
    INTERFACE
          upstream KIND component;
    INTERACTION
    IF FOR_ALL x AN upstream OF_CLASS valve WE_HAVE ...
    THEN ... ;
```

If O is some arbitrary object of class pump the expression *upstream OF_CLASS valve* denotes the set of objects of class valve (sub-class of component) in relation upstream with O.

The constraint VERIFYING:

This constraint restricts the objects in a set according to some condition. The idea is simple: the set is reduced to a subset that contains only objects satisfying this condition.

The condition that follows the keyword VERIFYING must therefore be a Boolean expression (which returns TRUE or FALSE).

In order for the defined set to be a constant set of objects, the expression following the keyword VERIFYING must be a **constant expression**, i.e. dependent only on constant fields.

The condition may contain an arbitrary number of Boolean expressions (including complex ones) linked by the logical operators (OR, AND or NOT).

In the following example, we are interested only in interfaced objects for which the flowrate exceeds a particular value:

```
CLASS pump;
CONSTANT
    flowrate DOMAIN REAL;

CLASS component;
INTERFACE
    upstream KIND pump;
INTERACTION
IF FOR_ALL x AN upstream VERIFYING flowrate OF x > 3000 WE_HAVE ...
THEN ... ;
```

If O is some arbitrary object of class component, *AN upstream VERIFYING flowrate OF x > 3000* denotes the set of pumps related to O by the upstream relationship for which the flowrate is strictly greater than 3000.

5.1.2.6. Empty sets

It is very important to note that, in order to maintain maximum flexibility in using the language, expressions involving quantifiers or complex numerical operators must give a result (rather than an error) even when the set to which these constructions relate is empty.

For the operators FOR_ALL, THERE_EXISTS, SUM and PRODUCT, the principle is the same: the result is the identity element of the operation, which is consistent with the fact that if we go from 0 to one element in the set, we go from the identity element to the value corresponding to the single element in the set.

If we were to apply the same principle to the minimum and maximum, we should take $+\infty$ and $-\infty$ respectively. Since infinite values cannot be represented, we have chosen arbitrarily to make the result zero in both cases. In summary, we have the following values:

FOR_ALL x AN element of an empty set WE_HAVE...	TRUE
THERE_EXISTS x AN element of an empty set SUCH_THAT	FALSE
SUM FOR_ALL x AN element of an empty set OF_TERMS	0
PRODUCT FOR_ALL x AN element of an empty set OF_TERMS	1
MINIMUM FOR_ALL x AN element of an empty set OF_TERMS	0
MAXIMUM FOR_ALL x AN element of an empty set OF_TERMS	0

Table1: values returned by complex operators applied to empty sets

We must therefore beware of these special cases of empty sets and, to avoid any unexpected model behaviour, we must use one of the following techniques:

- Ensure that the system models do not contain this particular case by ensuring, for example, that the cardinality of an interface is at least 1 (see 2.4.3.2).
- Write the expression in such a way as to explicitly take into account cases where the set is empty. In the following example, we want to set the effect (see 6.2.4.5) 'linked' of the class being described as true only if an instance of this class has objects in its upstream interface and if these objects are themselves all linked. In the particular case where there is no upstream, 'linked' must remain unchanged.

  Example:

```
INTERACTION
       IF( CARDINAL OF upstream <> 0 ) AND
         ( FOR_ALL x AN upstream WE_HAVE linked OF x = TRUE )
       THEN linked <-- TRUE;
```

5.1.3. Access to a field of class or object

5.1.3.1. Access to fields of the class described

To describe a CLASS, when we need to write an expression using a field of objects of the class, we need only name the field.

```
CLASS pump;
ATTRIBUTE
    state DOMAIN 'stop' 'running';
INTERACTION
    IF state = 'stop'
    THEN ....;
```

If O is some arbitrary object of class pump, expression *state = 'stop'* means *state of O = 'stop'*.

5.1.3.2. Access to the fields of a variable tied to a set of objects

To access the objects in a set, we always use a tied variable that represents an object in the set (this variable will successively take all the values in the set). To access the fields of objects represented by the tied variable, we use the operator **OF** or brackets.

Syntax:

```
field OF tied_variable
field (tied_variable)
```

The operator's mechanism is simple. For each object name substituted into the variable it returns the value of the object field. If the field does not exist, an error is generated.

```
CLASS pump;
INTERFACE
    backup KIND component;
ATTRIBUTE
    state DOMAIN 'running' 'stop';
INTERACTION
    IF THERE_EXISTS x A backup SUCH_THAT state OF x = 'stop'
    THEN ... ;
```

5.1.3.3. Access to the field of an object defined by a set of cardinality 1

If we wish to access an object in a set of cardinality 1, it is not necessary to use a tied variable. The object is accessed simply by naming the set, like for *backup* in the following example:

```
CLASS pump;
INTERFACE
    backup KIND pump CARDINAL 1;
ATTRIBUTE
    state DOMAIN 'running' 'stop';
INTERACTION
    IF state OF backup = 'stop'
    THEN ....;
```

5.1.3.4. Access to the fields of a named object

In a facts base (description of the system), the operator **OF** is used to access the fields of a named object.

Syntax:

```
field OF object_name
field  (object_name)
```

This operator uses an object's name to access the associated field. The operator's mechanism is simple:

- It checks whether the object is properly defined in the facts base, and if the object does not exist, the operator generates an error.

- If the object exists, it returns the field value of the object in question. If the field does not exist, an error is generated.

Example:

```
    flowrate OF v1
```

5.1.3.5. Access to global variables

This is a special case of what was described in the previous section. The major difference between the fields of system objects and the other object fields is that reference can be made directly to those in the model **classes**.

```
CLASS global;
    ATTRIBUTE
          repairable DOMAIN BOOLEAN DEFAULT TRUE;

SYSTEM_OBJECT global_parameters IS_A global;

CLASS pump;
    ATTRIBUTE
          state DOMAIN 'running' 'stop' 'failure';
    OCCURRENCE
          IF state = 'failure' AND repairable(global_parameters)
          MAY_OCCUR REPAIR.................;
```

5.2. Simple expressions

"Simple expressions" is taken to mean expressions without complex operators specific to the Figaro language.

Let us recall certain definitions:

1. An expression is a combination **of elementary expressions**.

```
Example of an expression:
a = (b + d) * ((r + z) >= 3.5 AND NOT q)
```

2. Generally speaking, an elementary expression consists of an operator and one or two operands.

```
Examples of elementary expressions:
b + d
r + z
NOT q
a = e
```

3. An elementary expression is said to be numeric if the operator is arithmetic. Otherwise, the expression is said to be logical.

```
Examples of numeric expressions:
b + d
r + z
Examples of logical expressions:
NOT q
a = e
```

4. The **domain** of an operand is the set of possible values of the operand. There are 4 possible domains in Figaro: INTEGER, REAL, BOOLEAN and enumerated. If a field should be assigned a value outside its domain, Figaro generates an error.

Examples of fields with their domain:

```
CLASS pump;
CONSTANT
    a DOMAIN INTEGER
       DEFAULT 3;
    b DOMAIN REAL
       DEFAULT -12e-3 ;
    c DOMAIN BOOLEAN
       DEFAULT TRUE;
    d DOMAIN 'running' 'stop'  (* enumerated domain *)
       DEFAULT 'stop' ;
```

We shall examine the various operations available in Figaro.

5.2.1. Arithmetic operations

Operators:

The available arithmetic operators are the usual ones:

```
+   addition
-   subtraction
*   multiplication
/   division
**  exponentiation
%   modulo
```

Operands:

These are binary operations, i.e. they require two operands. These operands may be integer or real. The subtraction operator is unary if the left operand is an operator or if there is no left operator (e.g. -3). In this case, it acts only on the right-hand operator.

```
Examples:
```

```
5 - 15        (* operator - binary *)
5 + -15       (* operator - unary: the result is the same *)
1e-4 * 123
5 % 3         (* 5 modulo 3: the result is 2 *)
```

Domains:

If two operands do not have the same domain in an arithmetic expression, the result of the expression is a value whose domain is the "largest" domain of the two operands, given that:

```
domain(real) > domain (integer) > domain (Boolean)
```

Hence the domain of the following expression

```
1.8e-4 * 123
```

is real since 1.8e-4 is real and 123 is an integer. So the expression is that of the largest domain, i.e. real.

It might be of interest to use Boolean operators in an arithmetic operation. In such a case the Boolean values are converted into integer values (or real values, should the need arise):

TRUE is replaced by **1**
FALSE is replaced by **0**

Hence:

TRUE * (5 > 6)* 3
is equal to (because of *, TRUE is converted)
1 * **FALSE** * 3
is equal to (because of *, FALSE is converted)
1 * 0 * 3

The transformation of a Boolean value into an integer or real number is particularly useful for defining complicated expressions using "indicator functions", a well-known trick among mathematicians.

For example, the following expression determines the fine to be paid after a minor_infraction (17€), a moderate_infraction (100€), or an infraction that is neither minor nor moderate (1000€):

```
17 * minor_infraction + 100 * moderate_infraction +
1000 * (NOT (minor_infraction OR moderate_infraction))
```

5.2.2. Boolean operators

Boolean operators (i.e. those that return the Boolean result TRUE or FALSE) are split into two categories: comparison operators and logical operators.

Comparison operators:

```
=   equals
<>  notequals
<   less than
<=  less than or equal to
>   greater than
>=  greater than or equal to
```

Operands:

Comparison operators are binary. Their operands are integers, or reals, or uniquely for the operators = and <>, symbolic values (between '), Booleans or objects.

Examples:

```
5 >= 6   (* always FALSE *)
```

```
a < 12.4 (* a can be only integer or real *)
state = 'stop' (* TRUE or FALSE depending on the value of the state variable *)

(* In the following example, x is a tied variable representing an object
contained in the upstream interface. We wish to test whether x is different from
a system object called source *)
… x AN upstream VERIFYING x <> source …
```

Domains:

The result is always Boolean. If the operands have different domains, the domain of the operand having the smallest domain will be transformed into the domain of the operand having the largest domain, only if the domains are INTEGER or REAL. Otherwise Figaro will generate an error.

Example:

```
5 >= 6.2e-3         (* 5 is transformed into a real number *)
```

Comparison between symbolic and numeric operands is not permitted. No conversion is possible.

Examples:

```
'stop' < 'running'  (* not permitted *)
'stop' < 5          (* not permitted *)
```

Logical operators:

OR
NOT
AND

Operands:

Only the negation operator is unary (it takes just one operand on the right-hand side). The others are binary. The operands can only be Boolean. Each of these operations returns a Boolean value.

Examples:

```
breakage OR short_circuit
  (* the variables must be Boolean *)
NOT open
```

Domains:

Calculating the domains presents no problem since everything is Boolean. It is clear, therefore, that any logical operation between non-Boolean operands will generate an error. Figaro cannot convert a non-Boolean into a Boolean.

```
Examples:
    a < 10 OR 123(* not permitted *)
    a < 10 AND b > 4    (* OK *)
    breakage OR NOT ( a < 13)
                        (* OK if breakage is a Boolean variable
                           and a is an integer or real variable *)
```

5.2.3. Priority between operators

The table below shows the evaluation priorities that apply to the operators. An expression in Figaro is calculated in order of operator priority. I.e. Figaro will start by evaluating elementary expressions with those operators having the highest priority down to those expressions having lower priority. Where the priorities are the same, the left-most operator has priority.

**	Higher priority
* /	
+ -	.
= > >= < <= <>	.
NOT	.
AND	.
OR	Lower priority

The use of brackets helps to manage the order in which an expression is evaluated. To avoid getting the meaning of an expression wrong, it is better to put in too many brackets than too few!

Examples:

```
10 * a - 3 / 2
    (* is evaluated as (10*a) - (3/2) *)
NOT p OR a <= 3 * 2
    (* is evaluated as (NOT p) OR (a <= (3*2)) *)
2/3*4
    (* is evaluated as (2/3)*4 *)
2-3+4
    (* is evaluated as (2-3)+4 *)
```

5.3. Quantifiers and "iterators"

One of the great strengths of the Figaro language is its ability to define highly generic models thanks to expressions that are valid over sets of objects in which the number of elements is unknown before instantiation on a particular system. This is possible because of the quantifiers and "iterators" (this neologism denotes an operator that performs the same role as a loop in a programming language).

As we have already seen, a **logical quantifier** is defined as a symbol that enables a variable defined on a set of objects to be used. Figaro includes quantifiers corresponding to the following keywords: THERE_EXISTS, THERE_EXISTS AT_LEAST and FOR_ALL. Quantifiers are used in Boolean expressions defining the **conditions** for rules.

If we need to define an **action** over all the elements in a set, we will use the "iterator" FOR_ALL.

Finally (rarely, it has to be said) we sometimes need to create (when we instantiate a model at order 0) a set of rules involving the objects in a set, or even all the tuples of objects that belong to the cardinal product of several sets. To this end, we shall use the "iterator" GIVEN.

5.3.1. The quantifier THERE_EXISTS

Syntax:

```
THERE_EXISTS variable
          A | AN set_name
          [ OF_CLASS    class ]
          [ VERIFYING   condition ]
          SUCH_THAT Boolean_expression
```

This operator corresponds to the quantifier ∃ in predicate logic. It tests whether **at least one** object in the set satisfies a particular condition (slot SUCH_THAT) which is represented by a Boolean expression:

```
THERE_EXISTS x A <set of objects> SUCH_THAT <condition>
```

This operator's mechanism is as follows:

- If the set is non-empty, the Boolean expression from the condition SUCH_THAT is evaluated for each object in the set. If at least one result is TRUE (logical OR between the results), the result of the operator is TRUE. Otherwise (all the results are FALSE), the result of the operator is FALSE.

- If the set is empty, the result is FALSE.

A Boolean expression can be a combination of simple or complex Boolean expressions.

Remark: The tied variable is related to the operator. It can therefore be used only:

- in the constraints that restrict the quantifier's definition set,

- in the condition SUCH_THAT.
but not in the remainder of the rule.

Example:

```
CLASS valve;
CONSTANT
    flowrate DOMAIN REAL;
ATTRIBUTE
    position DOMAIN 'open' 'closed' DEFAULT 'open';

CLASS pump;
INTERFACE
    upstream KIND valve;
INTERACTION
IF THERE_EXISTS x AN upstream
          SUCH_THAT (flowrate OF x > 1000 AND position OF x = 'open')
THEN ... ;
```

The previous example can be written differently, using theconstraint VERIFYING. The flowrate field of class valve is a constant (if it were an attribute, it would be rejected by the parser).

Example:

```
CLASS valve;
    CONSTANT
          flowrate DOMAIN REAL;
    ATTRIBUTE
          position DOMAIN 'open' 'closed' DEFAULT 'open';

CLASS pump;
    INTERFACE
          upstream KIND valve;
    INTERACTION
    IF THERE_EXISTS x   AN upstream
                        VERIFYING flowrate OF x > 1000
                        (* use of constraint VERIFYING *)
                        SUCH_THAT (position OF x = 'open')
    THEN ... ;
```

5.3.2. The quantifier FOR_ALL

Syntax:

```
FOR_ALL variable
          A | AN        set_name
          [ OF_CLASS    class ]
          [ VERIFYING   condition ]
```

```
        WE_HAVE Boolean_expression
```

This operator corresponds to the quantifier ∀ in predicate logic. It tests whether **all** objects in the set satisfy a particular condition (facet WE_HAVE) which is represented by a Boolean expression:

```
FOR_ALL x A <set of objects> WE_HAVE <condition>
```

This operator's mechanism is as follows:

- If the set is non-empty, the Boolean expression from the condition WE_HAVE is evaluated for each object in the set. If all the results are TRUE (logical AND between the results), the result of the operator is TRUE. Otherwise (at least one of the results is FALSE), the result of the operator is FALSE.

- If the set is empty, the result is TRUE. Take care with this particular case, since if it is forgotten it could be a source of error!

Example:

```
CLASS pump;
    ATTRIBUTE
        state DOMAIN 'stop' 'running' DEFAULT 'running';

CLASS backup_pump;
    INTERFACE
        backed_up_pump KIND pump;
    INTERACTION
    IF FOR_ALL x A backed_up_pump
                WE_HAVE state OF x = 'stop'
    THEN ... ;
```

5.3.3. The quantifier THERE_EXISTS AT_LEAST

Syntax:

```
THERE_EXISTS AT_LEAST   integer_expression
                        variable
                        INCLUDED_IN   set_name
                        [ OF_CLASS    class ]
                        [ VERIFYING   condition ]
                        SUCH_THAT Boolean_expression
```

This operator corresponds to the quantifier *k* over *n*. It tests whether **at least k** (expressed by the integer expression) objects in the set satisfy a particular condition (facet SUCH_THAT) which is represented by a Boolean expression:

```
THERE_EXISTS AT_LEAST k x INCLUDED_IN <set of objects> SUCH_THAT <condition>
```

This operator's mechanism is as follows:

- If the set is non-empty (i.e. $n \neq 0$), the integer expression representing the value *k* is evaluated.

  * If $k = 0$, the result is TRUE.

  * If $k > n$, the result is FALSE.

  - If $0 < k \leq n$, the Boolean expression from the condition SUCH_THAT is evaluated for each object in the set. If at least *k* expressions are evaluated as TRUE, the result is TRUE. Otherwise (at least $n - k$ expressions are evaluated as FALSE) the result is FALSE.

- If the set is empty (i.e. $n = 0$), the result is FALSE.

Example:

```
CLASS pump;
    ATTRIBUTE
          state DOMAIN 'stop' 'running' DEFAULT 'running';

CLASS backup_pump;
    ATTRIBUTE
          K      DOMAIN INTEGER
                 DEFAULT 2;
    INTERFACE
          backed_up_pump KIND pump;
    INTERACTION
    (* The backup pump starts only if AT_LEAST K pumps
       are stopped *)
    IF THERE_EXISTS AT_LEAST K x     INCLUDED_IN backed_up_pump
                                     SUCH_THAT state OF x = 'stop'
    THEN ... ;
```

5.3.4. Combination of quantifiers

When using nested complex operators, the result that Figaro produces will not always be obvious. To cope with this issue, it is possible to consider any quantifier as corresponding to a succession of conditions bound by a logical operator. The results of evaluating the quantifier conditions are bound:

- by an OR for the quantifier THERE_EXISTS

- by an AND for the quantifier FOR_ALL.

When nesting quantifiers, it can be advantageous to substitute the quantifiers by a combination of expressions on the objects of an example to get an idea of the result (this equates to mentally making an order 0 Figaro instantiation).

For example, if we consider the following sets of objects:

```
X = { x1 x2 }
Y = { y1 y2 y3 }
```

and the following rule:

```
IF ∃ x AN X / (∀ y A Y WE_HAVE x <> y) THEN ...
```

this rule is equivalent to:

```
IF (∀ y A Y WE_HAVE x1 <> y) OR (∀ y A Y WE_HAVE x2 <> y) THEN ...
```

which is equivalent to

```
IF (x1 <> y1 AND x1 <> y2 AND x1 <> y3) OR
   (x2 <> y1 AND x2 <> y2 AND x2 <> y3) THEN ...
```

This substitution makes it easy to get an idea of the result.

Particular attention must be paid to the scope of the tied variables when nesting complex operators. The scope of a tied variable is reduced to within the quantifier that defines the set to which this variable belongs.

The following example leads to an error since the variable x (defined in the operator THERE_EXISTS) is not defined in the quantifier FOR_ALL.

Example:

```
IF THERE_EXISTS  x AN X              (* end of operator THERE_EXISTS *)
```

```
AND FOR_ALL y A Y WE_HAVE x <> y  (* ERROR: x is not defined! *)
```

We must write:

```
IF THERE_EXISTS x  AN X SUCH_THAT

   FOR_ALL y A Y WE_HAVE x <> y      (* x is defined in THERE_EXISTS *)
```

5.3.5. The iterator FOR_ALL

This iterator specifies a series of actions in the slots INDUCING of an occurrence rule, THEN and ELSE of an interaction rule. This will be presented in the section covering the actions of Figaro rules (see § 6.3.2.2).

Example:

```
IF state OF grid = 'working'
THEN FOR_ALL x A busbar
     DO state OF x <-- 'powered';
```

5.3.6. The iterator GIVEN

This iterator creates (when a model is instantiated at order 0) a set of rules applicable to all the objects in a set, or to all the tuples of objects belonging to a Cartesian product of sets. This will be presented in the section covering the actions of Figaro rules (see § 6.3.4).

Example:

```
GIVEN x A backup_pump
IF state(x) = 'stop' AND needed(x)
THEN state(x) <-- 'startup';
```

5.3.7. Complete knowledge base with quantifiers

The following mini knowledge base (of just 45 lines) completely defines the semantics of generalised stochastic Petri nets. It has the advantage of illustrating the use of quantifiers, access to a variable through an interface chain and the CARDINAL function. We also see how, as is the case for generalised stochastic Petri nets, it is possible to have a transition whose distribution parameter is an attribute that is liable to change when the state of the system changes. The semantics of this type of transition are described in §3.4. Finally, this knowledge base is representative of so-called "abstract" bases since the classes correspond, not directly to physical components as in all the examples considered so far, but to the elements of a graphical model.

```
(* Definition of generalised stochastic Petri nets in Figaro *)
CLASS place;
    ATTRIBUTE marking DOMAIN INTEGER DEFAULT 0;

CLASS arc;
    CONSTANT weight DOMAIN INTEGER DEFAULT 1;

CLASS input_arc EXTENDS arc;
    INTERFACE start KIND place  CARDINAL 1;
              end KIND transition  CARDINAL 1;

CLASS output_arc EXTENDS arc;
    INTERFACE start KIND transition  CARDINAL 1;
              end KIND place  CARDINAL 1;

CLASS inhibitor_arc EXTENDS input_arc;

CLASS transition;
    INTERFACE upstream   KIND input_arc;
              downstream  KIND output_arc;
              inhibitor   KIND inhibitor_arc;

CLASS exponential_transition EXTENDS transition;
    INTERFACE pl KIND place CARDINAL 0 TO 1;

    DIST_PARAMETER lambda DEFAULT 1.e-3;
    ATTRIBUTE calculated_lambda DOMAIN REAL DEFAULT 0.0;

    OCCURRENCE
    IF ( FOR_ALL x AN upstream WE_HAVE marking ( start OF x ) >= weight OF x )
           AND
       ( FOR_ALL y AN inhibitor WE_HAVE marking ( start OF y ) < weight OF y )
    MAY_OCCUR TRANSITION firing DIST EXP ( calculated_lambda )
       INDUCING
            ( FOR_ALL x A downstream DO marking ( end OF x )
                    <--  marking ( end OF x ) + weight OF x ) ,
            ( FOR_ALL x AN upstream DO marking ( start OF x )
                    <--  marking ( start OF x ) + weight OF x ) ;

    INTERACTION
    (* If a place is declared in pl interface, then
      calculated_lambda is equal to its marking*lambda.
      otherwise it is fixed and equal to lambda *)
    IF CARDINAL(pl) = 1
    THEN FOR_ALL x A pl DO calculated_lambda <-- marking(x) * lambda
    ELSE calculated_lambda <-- lambda;
```

5.4. Boolean operators

There are four Boolean operators: an operator for managing sets (INCLUDED_IN), an operator for managing expressions (AT_LEAST...WITHIN) and two failure testing operators (WORKING and FAILURE).

5.4.1. The operator INCLUDED_IN

Syntax:

```
variable INCLUDED_IN set_name
```

This operator tests whether an object belongs to a set. To this end, the object variable is specified on the left of the operator, and on the right, a set of objects is represented by an interface variable or a class name. In the case of a class name, we simply test whether the object is of the class described or if the object is of a class that inherits from the class specified.

If the object appears in the set of objects, the result will be TRUE. Otherwise, it will be FALSE. If the set is empty, the result is FALSE.

In the following example, we test whether all the pumps in the system are included in the object's interface.

Example:

```
CLASS component;
INTERFACE
    upstream KIND pump;
INTERACTION
IF FOR_ALL x AN OBJECT OF_CLASS pump
             WE_HAVE x INCLUDED_IN upstream
THEN ...
```

Remark: It is not possible to impose set constraints on the operator INCLUDED_IN.

5.4.2. The WORKING operator

Syntax:

```
WORKING [ OF object ]
WORKING [ ( object ) ]
```

The WORKING operator tests whether an object is working, i.e. whether all the FAILURE object variables are FALSE (see FAILURE (as state variable), § 6.2.4.4). The WORKING operator enables access to an object as follows:

- If no object has been specified, the operator assumes that it is the object in which the operator is used.

- The object on which the operator has to act may be identified either by a tied variable or by a set name of cardinality 1, or by a particular object name.

The operator carries out the following operations:

- It collects all the FAILURE fields of the object.

- It tests whether all these (Boolean) fields are FALSE. If this is so, the result of the operation is TRUE. Otherwise (at least one of the FAILURE fields is TRUE), the result is FALSE.

- If the object has no FAILURE field, the result is TRUE

The following example tests whether a pump is both free from failures and fed by at least one source.

Example:

```
CLASS pump;
INTERFACE
    upstream KIND source;
FAILURE
    pump_failure;
EFFECT fed;
INTERACTION
    IF WORKING AND
     THERE_EXISTS x AN  upstream SUCH_THAT fed OF x
    THEN fed;
```

5.4.3. The FAILURE operator

Syntax:

```
FAILURE [ OF object ]
FAILURE [ ( object ])]
```

The FAILURE operator is the complement of the WORKING operator. It tests whether at least one of the attributes FAILURE (see FAILURE (as a state variable), § 6.2.4.4) of the object is TRUE. That being so, it returns the value TRUE. The FAILURE operator provides access to an object as follows:

- If no object has been specified, the operator assumes that it is the object in which the operator is used.

- The object on which the operator has to act may be identified either by a tied variable or by a set name of cardinality 1, or by a particular object name.

The operator carries out the following operations:

- It collects all the object's FAILURE fields.
- It tests whether at least one of these (Boolean) fields is TRUE. If this is so, the result of the operation is TRUE. Otherwise (all the FAILURE fields are FALSE), the result is FALSE.
- If the object has no FAILURE field, the result is FALSE.

The following example concerns a backup pump that starts only if the backed-up pump has failed.

Example:

```
CLASS backup_pump;
INTERFACE
    backed_up_pump KIND pump CARDINAL 1;
ATTRIBUTE
    state DOMAIN 'stop' 'running' 'start_request'
      DEFAULT 'stop' ;
INTERACTION
    (* Attempt to start if the backed-up pump has failed *)
    IF WORKING AND state = 'stop'
       AND FAILURE ( backed_up_pump )
    THEN
          state  <--  'start_request';
```

5.4.4. The operator AT_LEAST ... WITHIN

Syntax:

```
AT_LEAST integer_expression WITHIN ( Boolean_expressions_list )
```

This operator tests whether at least *k* expressions in the list are satisfied. To this end, the value of *k* is given by an integer expression and the list of Boolean expressions to be tested are separated by commas. If at least *k* expressions are evaluated as TRUE, the result will be TRUE. Otherwise, it will be FALSE. If the value of *k* is 0, the result is TRUE. If the value of *k* is greater than the number of expressions, the result is FALSE.

In the following example, we test whether at least 2 of the interfaced components have failed.

Example:

```
CLASS component;
INTERFACE
    component1 KIND component CARDINAL 1;
    component2 KIND component CARDINAL 1;
    component3 KIND component CARDINAL 1;
INTERACTION
IF AT_LEAST 2 WITHIN (  FAILURE(component1),
                        FAILURE(component2),
                        FAILURE(component3),
THEN ...
```

5.5. Complex numeric operators

Figaro also features numeric operators said to be "complex" since they act upon sets, i.e. interface fields or class names. These operators are used mainly for writing equations in Figaro (see the Equations chapter).

Remark: Just as for complex Boolean operators, it is possible to specify constraints on the set of objects treated by the operator (OF_CLASS and VERIFYING) so as to treat just one subset of objects.

There are five complex numeric operators: CARDINAL, SUM, PRODUCT, MINIMUM, MAXIMUM.

5.5.1. The CARDINAL operator

Syntax:

```
CARDINAL   OF interface_field
CARDINAL   ( interface_field)
```

The CARDINAL operator returns the number of elements in a set. Applied to an interface, it returns the number of objects that the interface field contains.

- If the interface is single-valued, the result is 1.
- If the interface contains no objects, the result is 0.

It is possible to get the CARDINAL of the interface of a neighbouring object using the operator OF or ( ).

The following example models the behaviour of an AND logic gate. We check whether the logic gate AND is related to at least two components in *upstream*. We also check that all the objects in the upstream interface of class *logic_gate,* have a downstream interface with just one element.

Example:

```
CLASS component;

CLASS logic_gate EXTENDS component;
INTERFACE
    downstream KIND component;

CLASS AND_gate EXTENDS logic_gate;
INTERFACE
    upstream KIND component;
ATTRIBUTE
    state DOMAIN 'valid' 'not valid' DEFAULT 'valid';
INTERACTION
    IF ( CARDINAL OF upstream < 2 )
    OR ( THERE_EXISTS x AN upstream OF_CLASS logic_gate
                       SUCH_THAT CARDINAL OF downstream OF x <> 1 )
    THEN   state <-- 'not valid' ;
```

Remark: This operator should not be confused with the CARDINAL slot, used for the description of an object's interface.

5.5.2. The SUM operator

Syntax:

```
SUM FOR_ALL variable A | AN set_name
                           [ OF_CLASS class ]
                           [ VERIFYING condition ]
        OF_TERMS numeric_expression
```

The SUM operator sums over the expressions in a set of objects.

It uses a tied variable defined over a set of objects (slot FOR_ALL) and for each object in the set it successively adds the value of the numeric expression (slot OF_TERMS) for that object:

```
SUM FOR_ALL x A <set of objects> OF_TERMS <numeric expression>
```

Example:

```
CLASS macro_component;
ATTRIBUTE
    failure_rate DOMAIN REAL DEFAULT 0 ;
INTERFACE
    element KIND component;
INTERACTION
    THEN failure_rate <-- SUM FOR_ALL x AN element OF_TERMS
        (lambda OF x * WORKING OF x);
```

The operator's mechanism is as follows:

- If the set is non-empty, the numeric expression from the condition OF_TERMS is evaluated for each object in the set. The result from the operator is the sum of the results of the expressions.

- If the set is empty, the result is 0. Don't overlook this special case!

The facet OF_TERMS specifies what calculations we wish to carry out and add. In order to be added up, the expression must be numeric.

5.5.3. The PRODUCT operator

Syntax:

```
PRODUCT FOR_ALL variable A | AN set_name
                           [ OF_CLASS class ]
                           [ VERIFYING condition ]
        OF_TERMS numeric_expression
```

The PRODUCT operator generates a product of expressions in a set of objects.

It uses a tied variable defined over a set of objects (slot FOR_ALL) and it calculates the product of the values of the numeric expression given in the facet OF_TERMS for all the objects in the set :

```
PRODUCT FOR_ALL x A <set of objects> OF_TERMS <numeric expression>
```

The following example shows the use of this operator in the default initialisation of an attribute.

Example:

```
CLASS calculation ;
INTERFACE
    upstream KIND component;
ATTRIBUTE
    value DOMAIN INTEGER
          DEFAULT PRODUCT FOR_ALL x AN upstream
                                        OF_TERMS ( a OF x ) + 5;
```

The operator's mechanism is as follows:

- If the set is non-empty, the numeric expression from the condition OF_TERMS is evaluated for each object in the set. The result from the operator is the product of the results of the expressions.

- If the set is empty, the result is 1. Don't overlook this special case!

The slot OF_TERMS specifies what calculations we wish to carry out and multiply. In order to be multiplied, the expression must be numeric. It may also be Boolean since the operator used to link the results is the multiplication operator, which can convert Booleans to integers.

5.5.4. The MAXIMUM and MINIMUM operators

Syntax:

```
MAXIMUM | MINIMUM
       FOR_ALL  variable A | AN  set_name
                        [ OF_CLASS class ]
                        [ VERIFYING condition ]
       OF_TERMS numeric_expression
```

These operators respectively calculate the maximum or minimum of expressions over a set of objects.

They use a tied variable defined over a set of objects (slot FOR_ALL) and they calculate the maximum (respectively: minimum) of the values of the numeric expression given in the slot OF_TERMS for all the objects in the set:

```
MAXIMUM FOR_ALL x A <set of objects> OF_TERMS <numeric expression>
```

The following example shows the use of this operator in the default initialisation of a constant.

Example:

```
CLASS calculation ;
INTERFACE
    upstream KIND component;
CONSTANT
    greatest_flowrate DOMAIN INTEGER
           DEFAULT MAXIMUM FOR_ALL x AN upstream
                        OF_TERMS flowrate OF x;
```

The operator's mechanism is as follows:

- If the set is non-empty, the numeric expression from the slot OF_TERMS is evaluated for each object in the set. The result from the operator is the maximum (respectively: minimum) of the results of the expressions.

- If the set is empty, the result is 0. Don't overlook this special case!

The slot OF_TERMS specifies what calculations we wish to carry out and compare. In order to be compared, the expression must be numeric.

5.6. Special operators:

Figaro contains special operators such as mathematical functions (e.g. exp, sin, cos, tan, max, etc.) or variously distributed random number generators.

The list of special operators can be extended according to need. This list is itemised in the note that defines the Figaro syntax.

The use of such operators follows the rules below:

```
operator ( expression )
or
operator ( expression_list )
```

Remark: Expressions in a list of expressions are separated by commas.

Examples:

```
EXP ( 2 + (a * 3) )
MAX ( a, b, c ) (* NB here the keyword is MAX and not MAXIMUM *)
RAND_UNI (a, b) (* each call to this function produces a random number uniformly
distributed between the real numbers a and b *)
```

5.7. The operator ?=

The operator ?= is very powerful since it can do the same thing as an operator of type "case" or "switch" in a programming language, or a set of nested instructions "if then else".

Syntax:

```
N ?= (Value if N=1, value if N=2…, value otherwise)
  (* N is an integer expression *)
```

The result of this expression has the same type as that of the values between brackets (which must all have the same type).

Examples:

```
size ?= ('small', 'medium', 'large', 'non-existent') (*  size goes from 1 to 3 *)

(size = 1) ?= ('small', 'all_except_small')      (* if the left operand is
Boolean, it is assumed to be integer 0 or 1 *)

R <-- A ?= (2, B ?= (3, 4))
```

The result R of this expression is calculated as follows:

If A = 1, R has the value 2 (regardless of the value of B)

If A is not equal to 1, R has the value 3 if B = 1 and 4 if B is not equal to 1.

We therefore have an equivalent of two nested if then else expressions.

5.8. Scope of a variable

As in any programming language, variables in Figaro have a scope.

Depending on its particular nature (SYSTEM_OBJECT variable, object field, tied variable), each variable has a different scope and a different evaluation priority (i.e. the order in which the value of the variable is sought in each of these contexts: global, object or local to a complex operator).

- The scope of a SYSTEM_OBJECT variable is **global**: all classes in the knowledge base can access it.
- A class field is associated with a class. Its scope is **local to the class**, i.e. it is unknown to and inaccessible by other classes except via an interface or interface chain of kind: left (top (right)).
- Finally, a tied variable is a variable defined in a complex operator such as FOR_ALL, THERE_EXISTS, THERE_EXIST AT_LEAST, SUM, PRODUCT, MINIMUM, MAXIMUM, FOR_ALL and GIVEN. These operators enable a variable to be defined that successively takes certain values in order to perform certain calculations. This variable is tied to the operator, i.e. its scope is **internal to the operator**. Outside it, the variable is unknown.

Now that the notion of scope is clearer, we must define the order in which Figaro seeks the value of a variable. A variable name can be overridden. That is, a tied variable can have the same name as a field of a class (though this is not recommended!). In such cases, priority is given to the variable that has the most reduced scope. Hence Figaro will first of all (thanks to the name of the variable) seek the value of the variable in the context of tied variables. Then if this fails, it will pass to the context of the class fields (local context).

Example:

```
CLASS valve
ATTRIBUTE
       x DOMAIN BOOLEAN DEFAULT TRUE;
INTERACTION
       IF FOR_ALL x A component WE_HAVE property(x)
       THEN x <-- TRUE ;
```

The variable *x* from the rule condition is related to the instruction ANY. In the expression property (*x*), the *x* found is that of FOR_ALL. In the THEN slot, what is involved is the field of the object.

Remark: So as to make it easier to read the knowledge base, it is advisable not to declare variables that have the same name as a class.

6. Description of a Figaro class

As we saw in chapter 2, a knowledge base is a set of definitions of classes. In this chapter we shall discuss in detail how to describe a class in Figaro.

6.1. General structure of a class definition

6.1.1. Declaration of a class

Syntax:

```
CLASS class_name[EXTENDS enumeration_of_classes];
```

The description of a class starts with the keyword **CLASS** followed by the name (the identifier) of the class.

```
CLASS motorised_pump;
```

The inheritance is specified by a slot EXTENDS which may be multi-valued (multiple inheritance), and in which are enumerated the superclasses of the class being described.

```
CLASS motorised_pump
    EXTENDS component pump ;
```

Remark: In contrast with classes, an object can inherit from just one class.

6.1.2. General structure of the description of a class

The characteristics associated with a CLASS are split into two major groups:

• Value fields represent the **static** characteristics of a CLASS. These are interfaces, constants and state variables (attributes, effects, failures).

• The rules of operation (rules of occurrence and interaction) describe the **dynamic** behaviour of a CLASS, i.e.:

- The events that can occur for an object in a class (change of state, fault, repair, etc.);

- The conditions under which these events occur;

- The consequences of these events on the system.

The figure below shows how the characteristics associated with a CLASS are organised.

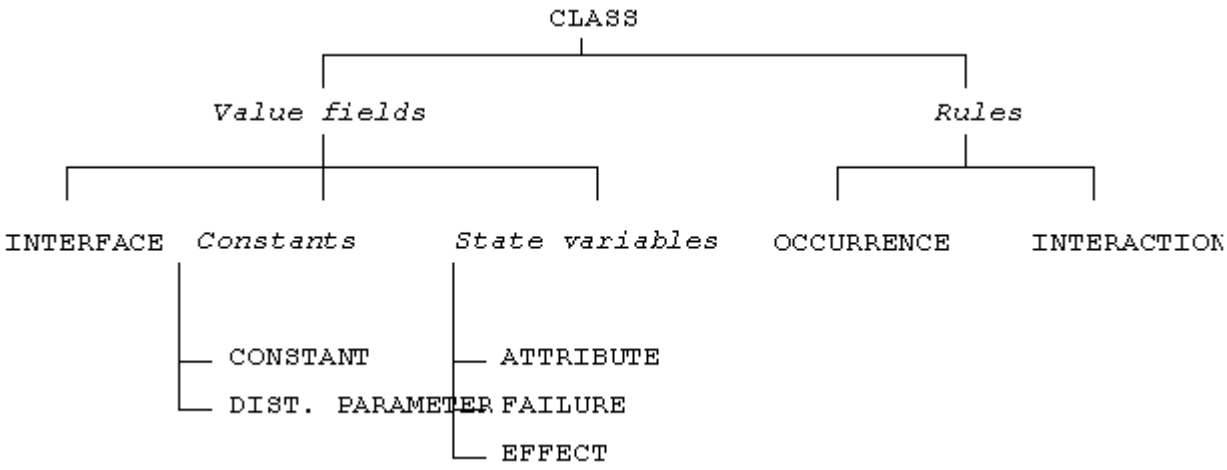

Figure 2. Characteristics of a CLASS

6.2. Value fields

There are three categories of value field for a class:

* **Constant fields** represent object characteristics that do not change over the life of the system (e.g. the length of a connecting cable). These fields are single-valued.

* **Interface fields** are fields containing the names of objects. The values of these fields do not change over the life of the system (relationships between the objects in a system are defined at the time the system is described and cannot change). These fields may be multi-valued.

* **State variables** characterise the states of the objects, and are liable to change over the life of the system. These fields are single-valued.

In Figaro, every field belongs to a category of field identified by a keyword (CONSTANT and DIST_PARAMETER for constant fields, INTERFACE for interface fields, ATTRIBUTE, FAILURE and EFFECT for state variables). Each field is described by its name and then by descriptive slots. The general structure of a field's description can be summarised as follows:

Syntax:

```
<category1_of_field>
       field1 <slot_name>  <list_of_values>
                   ...
                   <slot_name>  <list_of_values>;

       field2 <slot_name>  <list_of_values>
                   ...
                   <slot_name>  <list_of_values>;
       ...
<category2_of_field>       ...
```

Examples of field declarations:

```
ATTRIBUTE
       operation DOMAIN 'stop' 'start_request'
                        'start_refused' 'running'
         DEFAULT 'stop' ;
CONSTANT
       weight DOMAIN INTEGER;
```

Before describing each possible category of field in more detail, we need to set out certain ideas that are common to all field categories.

6.2.1. Preliminary observations

6.2.1.1. Initial state of a Figaro model

When writing a knowledge base, the user describes generic classes that may be re-used for different studies. To investigate a particular system, these classes are used to describe the system. A complete description of a system consists in creating the system objects and assigning values to each field that is characteristic of the class of an object. These values are:

- either an initial value for the model state variables,
- or a value for the model constants.

Example:

```
CLASS valve;
CONSTANT
    position DOMAIN 'open' 'closed' ;

(* facts base *)
OBJECT v1 IS_A valve;
 CONSTANT
    position = 'open;
```

The set of values attributed to the object fields when the system is being described constitutes the incomplete initial state of the model. The complete initial state of the model will be calculated by applying the interaction rules to this incomplete state.

Remark: the attribution of an initial value to each state variable of the model is not mandatory. Before applying the interaction rules, calculation of the initial state of a model attributes the default value declared in the knowledge base to each non-initialised variable.

6.2.1.2. Default value

It is sometimes useful to indicate a **default value** in the knowledge base for generic fields characterising classes. This reduces the burdensome phase of describing a system. The default values attributed to the fields of a class will be automatically attributed to the fields of each object created for this class. These values can, of course, be modified by the user.

A default value represents:

- either an initial default value for the "state variables" fields,
- or a default value for the "constant" fields.

For a given field of a class, a default value is created by using the DEFAULT slot. **Attributing a default value to a class field is never obligatory** (interaction rules can be provided to calculate the value of the variable in all situations)**.**

The following example highlights this idea by defining a class *valve* with attribute *position* and default value 'open'. Two objects in this class will then be created: one using the default value, the other redefining it.

Example:

```
CLASS valve;
ATTRIBUTE
    position DOMAIN 'open' 'closed' DEFAULT 'open';

(* facts base*)
OBJECT v1 IS_A valve;
(* the initial position value of v1 is 'open' *)

OBJECT v2 IS_A valve;
```

```
ATTRIBUTE
    position = 'closed' ;
(* the initial position value of v1 is 'closed' *)
```

Remark: For a given object field, an initial value is created by using the slot "=". This slot cannot be used for the characteristics of classes. Conversely, the slot DEFAULT is not permitted for the characteristics of objects.

The default value used with Figaro's inheritance mechanism allows an element of a class to be defined more precisely at its sub-class level. In our case we can, for example, define in a class the domain of a state variable and give in the sub-classes this variable's default value.

The following example splits electrical components into two families: sources of power (power lines, diesels, etc.) and passive elements (through which current passes such as circuit breakers and busbars). By default, we say that a source is fed, while a passive element is not.

Example:

```
CLASS electrical_component;
ATTRIBUTE
  fed DOMAIN 'yes' 'no'  ;

CLASS source_of_power EXTENDS electrical_component;
ATTRIBUTE
  fed DEFAULT 'yes';

CLASS passive_element EXTENDS electrical_component;
ATTRIBUTE
  fed DEFAULT 'no';
```

6.2.1.3. The notion of editing

In principle, when the user describes a system using a knowledge base, each time an object is created he can edit the set of fields characteristic of the object's class as well as their default values which he can modify.

However, the Figaro language allows the knowledge base designer to:

- restrict the set of fields that can be seen and edited by the user responsible for describing a system (when writing a knowledge base it is sometimes useful to define some variables that are necessary for the model specification, but which may be of no use in understanding it),

- put constraints on the editing of some fields (make the initialisation of certain fields mandatory, prevent the default values of certain fields from being modified, etc.)

The (optional) EDITION slot provides the editing characteristics of a class field in the knowledge base. Editing options are as follows. They may be combined in the same EDITION slot if they are linked by commas (list of values).

```
VISIBLE          NOT VISIBLE
MODIFIABLE       NOT MODIFIABLE
MANADATORY       NOT MANDATORY
```

*Remark: An option can be negated by adding the keyword NOT to it. Example: Negation of the option VISIBLE: **NOT** VISIBLE.*

Below are the details of options (options in **bold** are the default options of an arbitrary field):

*Option **VISIBLE***

The field will be visible by the user when editing an object of the class.

Option NOT VISIBLE

The field will be not be visible by the user when editing an object of the class.

*Option **MODIFIABLE***

The default value attributed to the field in the knowledge base can be modified by the user during the description of a particular system. See the important information about interfaces given below for the NOT MODIFIABLE option.

Option NOT MODIFIABLE

The default value attributed to the field in the knowledge base cannot be modified by the user during the description of a particular system.

This option has a particular significance as far as **interfaces** are concerned. It is impossible to give values for these in a knowledge base. If an interface is associated with the NOT MODIFIABLE option, this means that the user has no means of modifying this interface other than connecting objects by links in the KB3 software tool. **For the robustness of the applications of a knowledge base, it is important to specify this option for interfaces that are automatically filled in by drawing links.**

*Option **MANDATORY***

The user MUST provide an initial value for the field (this option is not relevant for constant fields whose completion is mandatory).

Option NOT MANDATORY

The initial value of the variable is not mandatory (it may, for example, be obtained by applying the interaction rules for calculating the complete initial state).

Remark: The options are applied in the following order of priority: 1. Visible (not visible), 2. Modifiable (not modifiable), 3. Mandatory (not mandatory). Hence a field declared not visible cannot be modified since it has been made inaccessible to the user (not visible).

Examples:

```
ATTRIBUTE
    field1 DOMAIN INTEGER
           DEFAULT 12;
            EDITION NOT VISIBLE;
    (* this field cannot be modified during the editing of an object in the class
*)

    field2 DOMAIN REAL
            EDITION VISIBLE, MODIFIABLE, MANDATORY;
    (* this field should be filled in during the editing of an object in the
class *)
```

Remark: Only classes can define the fields with an EDITION slot. In no cases is this slot recognised in a facts base, and it must not therefore be used in objects.

6.2.1.4. Observation on calculating the initial state of a Figaro model

Before any use of a Figaro model to generate a reliability model (Fault tree, state graph, etc.), the complete initial state of the model must be calculated.

For each essential state variable (not reinitialised at each application of the interaction rules) in the knowledge base, the knowledge base designer should therefore define the initialisation method adopted:

- attribution of a default initial value in the knowledge base (slot DEFAULT), modifiable or otherwise (slot EDITION) by the user,

- obligatory attribution (mention EDITION MANDATORY) of an initial value during the description of a system,

- calculation of the initial value via the interaction rules (this option is not recommended for an essential variable).

6.2.2. Interfaces

Syntax:

```
INTERFACE interface_name
    KIND class_name
    [CARDINAL integer_number
    | CARDINAL integer_number TO (integer_number | INFINITY) ]
```

As we saw in chapter 2, interface fields represent the relationships between objects in the classes. For a given class, the fields introduced by the keyword INTERFACE show the classes with which the class described will have some relationship.
- An interface field is identified by the interface name (name of the relationship)
- The slot KIND provides the name of the class that is related by this interface to the class described (it is, of course, mandatory). Use of the slot KIND is not permitted in objects.

```
CLASS electrical_panel;

CLASS pump;
    INTERFACE supply KIND electrical_panel;
```

- The slot CARDINAL specifies the constraints on the cardinality of the relationship defined by the interface field. In particular, this slot specifies whether or not the relationship is mandatory.

* Slot CARDINAL undefined

  If the slot CARDINAL is undefined (*this slot is not obligatory*), that means that an arbitrary object of the class described may be interfaced with several or no objects of the interfaced class (unlimited in number). In such a case, the relationship is not mandatory.

* **CARDINAL** integer_number

  If the slot CARDINAL is an integer n, that means that any object of the class described **MUST** be interfaced with n objects of the interfaced class (the cardinality of the set of interfaced objects is fixed at n). If n=1, we say that the interface is single-valued.

* **CARDINAL** integer_number **TO** (integer_number | **INFINITY**)

  If the slot CARDINAL is an interval (p, q), that means that any object in the class described will be related to between p and q objects of the interfaced class (the cardinal of the set of interfaced objects is included in a broad sense in the defined interval). Two particular cases may arise:

  . If p=0, this means that the relationship is not mandatory,

  . If q = INFINITY, this means that any object of the class described may be related to several objects of the interfaced class, (without number limitation).

Example:

```
CLASS electrical_valve;
INTERFACE     upstream KIND component;
              supply_220 KIND busbar CARDINAL 1;
              command_path KIND command CARDINAL 1 TO 2;
```

Remark: Use of the slot CARDINAL is not permitted in a facts base, and hence in objects.

6.2.3. Constants

Constants are fields whose value does not vary over the life of the system. By virtue of their nature, these fields can never be modified while the system is running.

There are two types of constant values: the distribution parameters used by the probability distributions (field DIST_PARAMETER), and the constants themselves (field CONSTANT).

6.2.3.1. CONSTANT

Syntax:

```
CONSTANT      constant_name
        [ LABEL       "description" ]
        [ DOMAIN      (INTEGER | REAL | BOOLEAN |
                      enumeration_of_symbols)
        [ DEFAULT (value | expression) ]
        [ EDITION      option_edition ]
```

Constants are characteristics of the objects in a class that do not change over the life of a system. In other words, events occurring in the system have no consequential effect on (i.e. do not alter) these characteristics. Constants are used mainly to represent the fixed characteristics of a system (e.g. length of a cable, impedance of a line, etc.).

Example:

```
CLASS cable;
CONSTANT
    length        DOMAIN REAL
                  LABEL "cable length"
                  DEFAULT 25
                  EDITION MANDATORY;
    impedance     DOMAIN REAL ;
```

Several slots enable the type and value of a constant to be defined precisely. The description of each of these is given below:

Slot LABEL (optional)

Allows a more complete description of the constant to be filled between double quotes. This makes the knowledge base and the system description easier to read. This slot is permitted for classes and objects.

*Slot DOMAIN (**mandatory**)*

Gives **the set of possible values** for the constant.

Figaro has three predefined types:

- BOOLEAN: contains only the values TRUE and FALSE.
- INTEGER: contains all the integer values.
- REAL: contains all the real values.

It is also possible to define one's own domain of values in the form of an enumeration. These values must be constants (it is not possible to use variables in an enumeration). Each value is a *symbol*, i.e. a string of characters between quotes. The declaration of an enumerated domain specifies the values that can be given to the field without introducing inconsistency into the model.

Example:

```
CLASS transformer;
CONSTANT
    primary_voltage     DOMAIN '400kV' '220kv'
                        DEFAULT '400kV'
                        EDITION MANDATORY;
```

Slot DEFAULT (optional)

This slot accepts a precise value belonging to the definition domain or a constant expression, i.e. in which only constant fields are used (the set of possible values for the expression must be included in the constant's definition domain).

An observation on the assignment of a default value in the form of an expression:

When using a Figaro model, the calculation of the initial state will flag up an error if it finds a constant in the model whose value is undefined (all constants must be calculated when the initial state is calculated) Hence, when in a knowledge base we define a default value for a constant in the form of an expression, we need to ensure that the use of this base makes it mandatory to input a value for constant fields that are part of the expression (e.g. using the option MANDATORY for the slot EDITION). If a field in the expression has no value, it will be impossible to calculate the expression.

Slot EDITION (optional)

(See § 6.2.1.3)

6.2.3.2. DIST_PARAMETER

Syntax:

```
DIST_PARAMETER       dist_parameter_name
      [ LABEL      "description" ]
      [ DEFAULT (value | expression) ]
      [ EDITION     option_edition]
```

As we saw previously, the occurrence of an event on an object depends on a probability distribution that is expressed as a function of a number of parameters.

Example: The occurrence of an operational fault in a pump is governed by an exponential distribution with parameter *lambda_pump*.

The field DIST_PARAMETER of a class gathers all the parameters that might be used to express the probability distributions of events that objects in the class might experience.

Example:

```
CLASS pump;
DIST_PARAMETER
    lambda_pump;
```

Distribution parameters are constants that will **always be real**; this means that we do not have to declare their domain of definition.

The slots associated with the declaration of a DIST_PARAMETER field are:
Slot LABEL (optional)

(see CONSTANT)

Slot DEFAULT (optional)

This slot accepts a real value or constant expression, i.e. in which only constant fields are used (the set of possible values for the expression must be included in the set of real numbers).

See CONSTANT for a comment on the assignment of a default value in the form of an expression.

Example:

```
CLASS   pump;
DIST_PARAMETER
        lambda   DEFAULT 1.e-8;
        lambda10 DEFAULT lambda * 10 ;
```

Slot EDITION (optional)

(See § 6.2.1.3)

6.2.4. State variables

With regard to the definition of a class of components, we know that, contrary to those that are fixed and will be declared as constants, certain characteristics can change over the course of time depending on the events that the system has experienced (e.g. the open or closed position of a circuit-breaker, the existence or otherwise of a failure).

Hence all the characteristics liable to change are described in the **state variables**. These characteristics are represented internally by variables, **always single-valued**, and the state of the system at any given moment will be defined by the values of these variables.

6.2.4.1. The notion of a reinitialised variable

To make it easier to model a class of systems in Figaro, it is often useful to recognise two types of variable:

- **essential** variables for the state of a system (non-reinitialised variables),

- variables for which the value, **for a particular state of the system**, is **deduced** from the values of other variables in this state by applying the interaction rules (reinitialised variables).

6.2.4.1.1.Essential variables

These are variables whose values change simply due to the occurrence of events.

For example, the state variable "faulty" can be associated with the class *pump*; this is initially FALSE but becomes TRUE if one of the two events "operational fault" or "startup fault" occurs.

6.2.4.1.2.Reinitialised variables

These are variables whose values will be reinitialised to some given value (reinitialisation value) before each application of the interaction rules. In other words, for a given system state, the values of these variables will always be **deduced** from the reinitialisation value and from the values of other variables in the model.

Take the example of a mesh network consisting of sources, links and nodes. This class of system is governed by the following rules:

- A source is always connected (to a source)

- An "interruption" event can occur for each object of class link.

- A node is connected (to a source) if there is a nearby connected node such that the link connecting the two nodes is not faulty.

In modelling such a class of systems, two state variables are of particular interest. The Boolean variable "connected" associated with objects of class "node". The Boolean variable "faulty" associated with objects of class "link".

The variable "faulty" changes for a link depending on the occurrence of the event "interruption" on this link. On the other hand, for a given node the variable "connected" must be recalculated after each event occurrence depending on the new states of the nearby nodes and links in the network (the variable depends solely on the

network connections). This variable is therefore deduced via application of the interaction rules. Hence it will be declared in Figaro as a reinitialised variable, the reinitialisation value being FALSE.

From the point of view of using Figaro models, the notion of reinitialised state variable is of significant interest. Since the value of such a variable is deduced from the values of other variables in the model, it is unnecessary to store its value when we wish to store a system state in memory. Section 9.4 explains that there are deeper reasons underlying the importance of having a maximum number of deduced variables in a Figaro model.

A variable's reinitialisation value **must** be a constant.

6.2.4.2. Categories of state variables in Figaro

There are three forms of state variable in Figaro: ATTRIBUTES, FAILURES AND EFFECTS.

Attributes combine all the state variables that are neither failures nor effects. We can define simple attributes (non-reinitialised) or reinitialised attributes (slot REINITIALISATION).

Failures are non-reinitialised Boolean state variables used to detect the existence or otherwise of a fault in an object. These state variables are associated with the events declared in the occurrence rules and with the reliability models that will be used to generate fault trees.

Effects are Boolean state variables reinitialised to FALSE. They are created to simplify the declaration of such variables since they are very frequently employed, especially in knowledge bases intended to be used in generating fault trees.

6.2.4.3. ATTRIBUTE

Syntax:

```
ATTRIBUTE     attribute_name
        [ LABEL       "description" ]
        [ DOMAIN (INTEGER | REAL | BOOLEAN | enumeration_of_symbols) ]
        [ DEFAULT (value | expression) ]
        [ REINITIALISATION (value | expression) ]
        [ EDITION        option_edition ];
```

Attributes are state variables that are characteristic of the class. Each attribute is associated with a definition domain and may be associated with a default initial value.

Example:

```
CLASS circuit-breaker;
ATTRIBUTE
    position DOMAIN 'open' 'closed'
             DEFAULT 'closed' ;
```

Several slots can be associated with each attribute.
*Slot DOMAIN **(mandatory)***

Enables the set of possible values to be defined for the attribute. This slot is identical to that for CONSTANTs

Example:

```
CLASS valve EXTENDS thermohydraulic_component;
ATTRIBUTE
    flowrate DOMAIN REAL;
    pressure DOMAIN 'low' 'medium' 'high';
```

Slot LABEL (optional)

(see CONSTANT)

Slot DEFAULT (optional)

This slot accepts a precise value belonging to the attribute's definition domain or an expression (the set of possible values for the expression must be included in the attribute's definition domain).

Observation on the assignment of a default initial value in the form of an expression:

In a knowledge base, a default initial value for an attribute is defined in the form of an expression; care should be taken that the use of this knowledge base makes it mandatory to input a value for the fields involved in the expression (using the option MANDATORY for the slot EDITION, for example). If a field in the expression has no initial value, the processing will be unable to evaluate the expression.

Slot REINITIALISATION (optional)

This slot allows an attribute to be declared as a reinitialised variable and gives it a reinitialisation value (see "Reinitialised variables" section in this chapter). This slot is permitted only for variables declared as attributes in the knowledge base.

The reinitialisation value MUST be a constant.

Slot EDITION (optional)

(see CONSTANT)

6.2.4.4. FAILURE

Syntax:

```
FAILURE
 failure_name
    [ LABEL      "description " ]
    [ DEFAULT (TRUE| FALSE) ];
    [ EDITION edition_option ]
    [ RELIABILITY_DATA
       [IF constant Boolean expression]
       [GROUP name(s) of group(s)]
       [DEFAULT_MODEL name_of_model]
       [reliability_model]
       …
       [reliability_model]
    ]   (end of optional section RELIABILITY_DATA)

    ; (a single ";" terminates the declaration of each failure)
```

When the system dependability analysis is being carried out, we are mainly interested in the consequences of any component malfunction on the operation of the system. For this reason many events modelled in a class of objects are events representative of the failures that can occur among objects in this class.

Example: objects of class pump can experience the event "operational fault".

To describe the consequences of a "failure event" on the system it is often helpful to associate Boolean state variables with the classes of objects, these being representative of the existence or otherwise of a failure.

The Figaro language is able to gather these particular variables under the field FAILURE.

Example:

```
CLASS pump;

FAILURE
    operational_fault ;
```

Remark: "Failure event", which will be modelled in the occurrence rules, must not be confused with

"failure variable", which memorises whether this event has occurred or not.

State variables in the category FAILURE are always Boolean variables (TRUE if the failure has occurred, FALSE otherwise). Their initial value is FALSE if not specified, as in the example above.

The modelling of failures is greatly enhanced when we wish to use the Figaro knowledge bases to capitalise on different models of small stochastic processes that might be associated with basic events in the fault trees generated (see [3] for a mathematical definition of fault trees of this kind). It is for this reason that the optional section RELIABILITY_DATA was added to the language to describe a failure.
This section includes:

- A condition (optional) which depends only on constants (e.g. considering the fault in which a valve refuses to open, we put initial_position = 'closed');
- A set of rule groups (see §7.4) in which the reliability models (to be discussed below) must be considered;
- A set of reliability models and their parameters, plus mention of a default model: all these models will be in the fault tree generated, but only one will be "active", namely the default model.

A typical example of a FAILURE declaration with two reliability models:

```
FAILURE oper_fault
    LABEL "operational fault"
    RELIABILITY_DATA
           IF initial_state = 'running'
           GROUP group_FT
           DEFAULT_MODEL MODEL_GLM
           MODEL_GLM
                  GAMMA gamma
                  LAMBDA lambda
                  MU mu
           MODEL_GLTM
                  GAMMA gamma
                  LAMBDA lambda
                  TM mission_time;
```

There are two ways in which the reliability models associated with a FAILURE can be used in a Figaro model:

- In forward chaining they are automatically replaced by a set of occurrence rules written in such a way as to produce the theoretical behaviour described in Appendix 2.
- In backward chaining, when the algorithm described in § 2.8.3 has to explain a fact of type failure_name = TRUE, it creates a basic event in the fault tree that is being constructed, and assigns to it the models associated with the FAILURE 'failure_name'.

The set of all possible models is given in Appendix 2. This list of models available in Figaro was obtained by reviewing the models usable in various fault tree calculating tools.

Here, by way of example, is the behaviour specified by the most widely used reliability model, the so-called MODEL_GLM. This model represents a repairable component that can fail at t=0 or during operation.

Its parameters are: **G**AMMA (failure rate on power-up), **L**AMBDA (failure rate during operation), **M**U (repair rate).

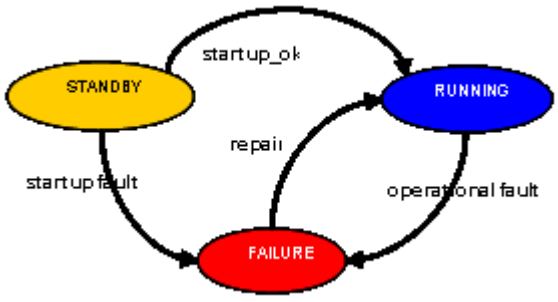

Figure 3. Graph of states of model MODEL_GLM

6.2.4.5. EFFECT

Syntax:

```
EFFECT effect_name
       [ LABEL       "description" ]
       [ EDITION     [ [NOT]VISIBLE] ];
```

Effects **are equivalent to Boolean variables reinitialised to FALSE**, but given their very frequent use (especially in models intended for constructing fault trees), it is convenient to be able to distinguish them since this greatly reduces the burden of their declaration (in general it is sufficient to declare the name of the effect). An effect can be declared NOT VISIBLE if its value is deemed to be of no interest to the final user of the knowledge base.

If a variable is reinitialised to FALSE, this means that, if no interaction rule allows the value of this variable to be **deduced**, then it is FALSE. Hence an effect corresponds to the behaviour of a component (generally of a functional nature, such as an obstruction, leak, etc.) observed following the realisation of a set of causes. It disappears as soon as the causes disappear: we say that *an effect is non-remanent.* Effects are similar to the notion of intermediate event in a fault tree.

6.3. Rules

As we saw in chapter 2, class rules allow the dynamic behaviour of the objects in a class to be defined. The rules describe the events that can be experienced by the various classes being modelled and their consequences on the system. There are two types of rule in Figaro.

Occurrence rules allow the occurrence of events that an object can experience to be modelled. These are of the form:

IF <condition>
MAY_OCCUR <event>
INDUCING <actions>

* Condition expresses the condition that the state of the system must satisfy for an event to be activated on an object in a class. This condition may depend both on **essential** and **deduced** variables.

* Actions describe the consequences of the event on the **essential** state variables in the class.

Interaction rules model the propagation of state changes brought about by the triggering of an event. These are of the form:

IF <condition>

THEN <actions>

OTHERWISE <actions>

In these rules the condition (IF) allows (THEN) or does not allow (OTHERWISE) a set of actions to be set in motion that affect the **essential** and **deduced** model variables.

Using a Figaro model involves applying the rules of the knowledge base to the base of facts describing the system to be examined. The rules of a Figaro model can be used in forward chaining mode (e.g. to generate the graph of system states) or in reverse chaining mode (e.g. to generate fault trees).

6.3.1. Conditions

Conditions in Figaro rules are complex (or otherwise) Boolean expressions. These correspond to tests on the state of the system that will be assessed by testing (in forward or backward chaining modes) whether the rule is applicable. If the condition of an object's rule is assessed as TRUE for the values of the variables in the current stat of the system, then the rule can be activated and the actions applied.

The syntax of usable expressions for defining rule conditions have already been completely defined. Recall simply that, for a given class we can use, in the expression associated with the condition of a rule:

- class fields,
- fields of interfaced objects by defining a set of objects by an interface and using the operators THERE_EXISTS, THERE_EXIST_AT_LEAST, ANY or GIVEN,
- fields of objects of a particular class in the model by defining the set of objects by the name of the class and using the operators THERE_EXISTS, THERE_EXIST_AT_LEAST, ANY or GIVEN,
- the SYSTEM_OBJECT fields (global variables).

The following example describes an interaction rule for a backup pump that is activated (*state* goes to *'request_start'*) only if the backed-up pump has failed.

Example:

```
CLASS pump;
FAILURE
    startup_fault;
    running_fault;

CLASS backup_pump EXTENDS pump;
ATTRIBUTE
    state DOMAIN 'stop' 'request_start' 'running'
          DEFAULT 'stop';
INTERFACE
    backed-up_pump KIND pump CARDINAL 1;
INTERACTION
    IF FAILURE( backed-up_pump ) AND RUNNING
    AND state = 'stop'
    THEN
          state <--  'request_start' ;
```

6.3.2. Actions

Actions are operations that enable the model state variables to be modified, thereby changing the state of the system. An action can therefore uniquely modify attributes, failures or effects. For a given class, an action can modify the **state variables**:

- of the class,
- of interfaced objects by defining a set of objects by an interface and using the operators FOR_ALL or GIVEN,
- of objects of a particular class in the model by defining the set of objects by the name of the class and using the operators FOR_ALL or GIVEN,
- of SYSTEM_OBJECT fields (global variables).

A rule can give rise to several **elementary actions** (list of actions). In this case, they must be concatenated using commas. There are three kinds of elementary action.

6.3.2.1. Assigning a value to a variable

A new value can be assigned to an attribute, failure or effect using the operator **<--**.

```
<variable> <-- <new value>
```

The new value must, of course, belong to the domain of possible values specified for the variable. In order to make certain knowledge bases more concise and readable, the management of Boolean variables (in particular FAILURE and EFFECT) is simplified. Hence **for a Boolean variable**:

```
<variable> <-- TRUE     is equivalent to          <variable>
<variable> <-- FALSE    is equivalent to          NOT <variable>
```

These two notations are strictly equivalent.

Examples:

```
...THEN position OF x <-- 'open' ,
        value OF y <-- value OF y + weight OF y ;
    (* the rule executes two elementary actions *)

...THEN obstruction;
    (* obstruction is a Boolean attribute, a failure or an effect *)

...THEN NOT fed;
    (* fed is a Boolean attribute, a failure or an effect *)
```

6.3.2.2. Action on a set of objects

The operator FOR_ALL allows the attributes of a set of objects to be successively modified. This acts like the complex operators THERE_EXISTS, THERE_EXISTS AT_LEAST, FOR_ALL, SUM and PRODUCT.

```
FOR_ALL x A <set of objects>
```

Syntax:

```
FOR_ALL variable
          A | AN        set_name
          [ OF_CLASS    class ]
          [ VERIFYING   condition ]
          DO action
```

FOR_ALL defines a variable related to the operator that will successively take on all the values of the set. The set is either an interface name (multi-valued) or a class name. As for all complex operators, it is possible to specify optional constraints (OF_CLASS and VERIFYING) to carry out the action (slot DO) uniquely on a subset of objects.

This operator's mechanism is as follows:

- If the set is non-empty, the action specified by the slot DO is carried out on each object in the set.

- If the set is empty, no action is carried out.

Examples:

```
FOR_ALL x A <set of objects>
    DO <assignment>

THEN FOR_ALL x A command
    DO position OF  x <-- 'open' ;

(* Nesting of FOR_ALL operators *)
THEN FOR_ALL x AN upstream
      DO ( FOR_ALL y A downstream OF x
                          VERIFYING MYSELF <> y
              DO value OF y <-- value OF x ) ;
```

Remark: The scope of the linked variable is reduced to the operator FOR_ALL. It is therefore accessible only in the action specified by DO. To combine several actions under the same slot DO, brackets, rather than commas, must be used. The comma has a higher priority than the keyword DO. The following examples should clarify this aspect:

Example:

```
THEN FOR_ALL x A upstream
          DO state OF x <-- 'stop', position OF x <-- 'open' ;
```

This example generates an error since Figaro interprets this phrase as two successive actions after the keyword THEN and not after the keyword DO:

```
(* This is how Figaro reads the phrase above *)
THEN FOR_ALL x A upstream
          DO state OF x <-- 'stop',
     position OF x <-- 'open' ;  (* x is unknown!!! *)
```

We ought to have written:

```
THEN FOR_ALL x A upstream
          DO ( state OF x <-- 'stop', position OF x <-- 'open') ;
    (* x is known: the brackets are necessary *)
```

6.3.2.3. Solution of a system of equations

Syntax:

```
... THEN SOLVE_SYSTEM (name_of_system) , ...
```

The action SOLVE_SYSTEM specifies the system of equations to be solved (see § 7.6). The solution mechanism is as follows:

If the rule can be activated, Figaro solves the system of equations.

- The values of the unknowns are then calculated

- Since these unknowns represent attributes, Figaro assigns them their new values.

The values of the attributes/unknowns have therefore been modified. The state of the system is altered.

The main difficulty encountered when using systems of equations is that, depending on the configuration of the system when the solution process starts, it can happen that no solution is possible or that there are infinitely many possible values. In an electrical system, for example, from the instant a part of the system is isolated from all sources of power, it becomes impossible to calculate its voltage.

To avoid this pitfall, the equations must be conditioned using the IF slot (see § 7.6) so as to make any problematic equations vanish. In the example of the electrical system, the condition to be satisfied for an equation vis à vis the voltages is that the component concerned must be properly connected to a source of power (which can easily be done via the propagation of the electrical flow, starting from sources, by the interaction rules).

6.3.3. Occurrence rules

The occurrence rules of a class model the occurrence of possible events and their consequences on that class (and sometimes on classes accessible via interfaces, as in the knowledge base example in § 5.3.7, where the firing of a transition affects the places upstream and downstream of the transition).
The occurrence of an event is governed by some particular probability distribution. There are several possible types of event depending on the probability distribution associated with the event:

- Events associated with an instantaneous distribution: these events occur instantaneously as soon as their condition of occurrence has been satisfied. The associated distribution parameter is a dimensionless rate of occurrence.

- Events whose occurrence is shifted in time (with respect to the instant at which the conditions of occurrence were satisfied). Among these, two cases in particular are frequently encountered:

    - events associated with an exponential distribution: these events can occur at the end of some random time. The distribution parameter is a rate of occurrence having the dimension of time.

    - events associated with a constant time distribution: these events are certain to occur at the end of a defined time T if the conditions of occurrence for the event are maintained throughout the period T.

An occurrence rule may be associated with a single event if it is shifted in time, but with one or more events if they are instantaneous.

Instantaneous occurrence rules

When a system is in a given state, it can happen that there might be a choice between several incompatible changes. Such is the case when, for example, a component with power applied can start or refuse to start. The system can choose between these two changes, but one of them **must necessarily** occur. Hence instantaneous occurrence rules allow the occurrence of events to be modelled.

- mutually incompatible (exclusive),
- of which one at least must occur instantaneously

The syntax of an instantaneous occurrence rule will therefore have the form:

```
IF <condition>
MAY_OCCUR        <event>
                          INDUCING <actions>
                 OR_ELSE   <event>
                          INDUCING <actions>

                 OR_ELSE ...
```

Each event is associated with an instantaneous distribution. **The sum of the distribution parameters of the events in the rule MUST add up to 1** (one of the events MUST happen).

*Remark: for an event triggered by the **last** slot OR_ELSE, it is not mandatory to provide the distribution parameter associated with the event. If this is not provided, Figaro will automatically add an instantaneous distribution parameter whose value is:*

1 - ∑(distribution parameters of other events triggered by the rule)

Order of application of occurrence rules in forward chaining mode.
In chapters 0 and 3 we examined the principle of using the rules of a Figaro model in forward chaining mode. We saw that, if the system is in a state in which no occurrence rule with instantaneous transitions applies, the software finds the applicable occurrence rules, chooses one, and then applies it to simulate an event.

In a system state, if the program finds any applicable instantaneous occurrence rules, this means that one (and only one) event for each rule must be activated instantaneously. Thus, the program does not choose **one** event but a **combination** of events formed by an event from each applicable rule.
The software then successively applies each event in the combination by triggering the actions brought about by the events. It is important to note that the order of application of the combination must not matter. Everything then happens as if the events were applied in parallel. Hence the knowledge base designer should be careful to ensure that those of the system's instantaneous occurrence rules that are simultaneously applicable do not involve incompatible facts. For further detail, see inconsistency type **Inc2** in § 9.2.

The FAILURE variables and events

Qualitatively speaking, we can classify the events that a system might experience into four categories:

- **Transitions** are events that describe the normal behaviour of components. Examples: normal start of a pump, day-night transition, etc.
- **Faults** are events that describe the faulty behaviour of components. Examples: refuse to start of a pump, unexpected opening of a circuit-breaker, etc.
- **Unavailability** represents a particular type ofon demand failure. Most often these are due to events that have occurred in the system's past and which are revealed when the components are put into function. (Example: a normally open valve was left closed by mistake of an operator). Events of type UNAVAILABILITY will be treated exactly the same as on demand failures. In other words, UNAVAILABILITY information is purely qualitative as compared with a simple on demand failure.

- **Repairs** represent changes of state due to maintenance work on a system. Example: repair to a pump following a fault at start up or during the pump's operation.

Hence to describe an event in an occurrence rule we might write:

MAY_OCCUR **TRANSITION** <event>
or
MAY_OCCUR **FAULT** <event>
or
MAY_OCCUR **UNAVAILABILITY** <event>
or
MAY_OCCUR **REPAIR** <event>

There are two advantages to this classification: first of all, it enables the output from a calculation code to be better presented (e.g. by showing faults in red and repairs in green in a sequence display). Due to the association with the FAILURE variables, it then allows shortcuts in writing.

- An event of class FAULT or UNAVAILABILITY may, or may not, be associated with a FAILURE variable.

  * If the event is associated with a FAILURE variable, i.e. the identifier of a FAILURE variable is the same as the event identifier, then the rule describing the occurrence of the fault or unavailability can be written in a more concise way. This rule is implicitly conditioned by the fact that the associated failure has the value FALSE; furthermore, the triggering of this event causes the associated FAILURE variable to be set to FALSE;

  * On the other hand, if the event is not associated with a FAILURE variable, using the keywords FAULT or UNAVAILABILITY will simply allow qualitative information about the event to be provided.

- Similarly, an event of class REPAIR may, or may not, be associated with one *or more* FAILURE variables.

  * If the event is associated with one or more FAILURE variables, i.e. if the event REPAIR is described explicitly as repairing one or more FAILURE variables, then the rule describing the occurrence of the repair can be written more concisely. This rule is implicitly conditioned by the fact that at least one of the associated failures is TRUE; in addition, triggering this event causes *all* the associated FAILURE variables to be set to FALSE;

  * On the other hand, if the event is not associated with a FAILURE variable, using the keyword REPAIR will simply allow qualitative information about the event to be provided.

During instantiation in Figaro 0, all the implicit conditions and actions mentioned above are made explicit since FAILURE is transformed into a simple ATTRIBUTE.

6.3.3.1. Syntax of an occurrence rule

```
OCCURRENCE [name_of_rule]
    [ GROUP      group_list]
    [ IF         condition]

    MAY_OCCUR
    TRANSITION   transition_name
                       [ LABEL      "description" ]
                       DIST         dist_and_parameters
                       [ INDUCING   actions ]

    | FAULT fault_name | failure_name
                       [ LABEL      "description" ]
                       DIST         dist_and_parameters
                       [ INDUCING   actions ]

    | REPAIR     repair_name
                       [ LABEL      "description" ]
                       DIST         dist_and_parameters
```

```
                    [ REPAIRS     list_of_failures ]
                    [ INDUCING    actions ]

| UNAVAILABILITY    unavailability_name | failure_name
                    [ LABEL       "description" ]
                    DIST          dist_and_parameters
                    [ INDUCING    actions ]

[ OR_ELSE     TRANSITION ... | FAULT ... |
              REPAIR ... | UNAVAILABILITY ... ]
[ OR_ELSE     ...]  ;
```

Remark: The syntax for the occurrence rules for objects is similar to that for the occurrence rules for classes, with one exception: the slot REPAIR is not permitted for an object.

The occurrence rules defined for a class are introduced by the keyword OCCURRENCE.

The syntax of an occurrence rule is as follows:

Name of the rule (optional)

Allows a rule to be assigned a name. This is useful for "commenting" the rules and in identifying them in debugging tools.

Example *fault_in_operation* is the name of the rule:

```
CLASS pump;
ATTRIBUTE
    state DOMAIN 'running' 'stop';
OCCURRENCE
    fault_in_operation
    IF state = 'running'
    MAY_OCCUR
          ..... ;
```

Slot GROUP (optional)

Specifies whether the rule belongs to one or more groups (see Advanced notions, § 7).

Slot IF (optional)

Describes the condition necessary for the occurrence rule to be activated. The implicit conditions explained above are added to this explicit condition.

*Slot MAY_OCCUR (**mandatory**)*

Describes which events can be triggered when the rule condition is satisfied for the state of the system. Each event is identified by its name and qualified by the keyword that describes the nature of the event (TRANSITION, FAULT, REPAIR, UNAVAILABILITY).

Example:

```
CLASS pump;
ATTRIBUTE
    state DOMAIN 'running' 'stop';
OCCURRENCE
    IF state = 'running'
    MAY_OCCUR
          FAULT fault_in_operation
          ..... ;
```

In the case of an instantaneous rule containing several events:

- The first one is introduced by the keyword MAY_OCCUR

- The following events are introduced by the keyword OR_ELSE

Example:

```
CLASS pump;
FAILURE power-up_fault ;
ATTRIBUTE
    state DOMAIN 'running' 'stop' 'power-up';
OCCURRENCE
    IF state(x) = 'power-up'
    MAY_OCCUR
          FAULT power-up_fault
                INDUCING state <-- 'stop'
                DIST INS(gamma)
    OR_ELSE
          TRANSITION start
                INDUCING state <-- 'running'
```

Each rule event can be precisely described by the following slots:

Slot LABEL (optional)

Inserts plain text (in double quotes) that is descriptive of the event. This label slot is optional (but recommended to obtain easily readable results for the investigation).

Slot REPAIR (optional)

This is present only if the type of event is REPAIR (see Repairs section). It declares the list of failures cleared up following a repair. The slot must be followed by the names of the FAILURE variables separated by spaces.

*Slot DIST **(mandatory**)*

Specifies the probability distribution associated with the event. The distribution must be declared using a syntax of the form:

name_of_distribution **(** numeric_expression, numeric_expression… **)**

This slot is mandatory *except* for the last event associated with an instantaneous rule (last slot OR_ELSE in the rule). If this is not provided, Figaro will automatically add an instantaneous distribution parameter whose value is 1 - the sum of the distribution parameters of the other events contained in the rule.

The names of the most commonly used distributions are EXP or EXPONENTIAL (the exponential distribution), INS or INSTANTANEOUS (the instantaneous distribution), C_T or CONSTANT_TIME (the Dirac distribution). Each of these distributions has just one parameter whose dimension is the reciprocal of time for EXP (transition rate) and time for C_T. The parameter for the INS distribution is a probability, and is therefore dimensionless.

There is an entire catalogue of other distributions: these can be found in the document describing the Figaro syntax [5] (see appendix). With the exception of the INS distribution (for which that makes no sense) and the EXP distribution (which is memoryless, by definition) they all have a so-called "memory" variant. The name of a distribution with memory is the name of the basic distribution followed by the suffix _M. The usefulness of these memory distributions was discussed in § 3.4.

The numeric expression that attributes a value to a parameter of the distribution associated with the event may be mainly:

- A constant expression involving DIST_PARAMETER fields and CONSTANT fields (this is the most frequent):

Example:

```
CLASS pump;
ATTRIBUTE
       state DOMAIN 'running' 'stop';
DIST_PARAMETER
       lambda_pump;
OCCURRENCE
       IF state = 'running'
       MAY_OCCUR
              FAULT fault_in_operation
              EXP DIST (lambda_pump)
              ..... ;
```

- An expression involving *variable* attributes. This specific case is not a problem for instantaneous distributions (we take the value of the parameter at the instant we need to choose between the alternatives), nor for memoryless exponential distributions. For all the other distributions, the significance of a parameter change that occurs while the event remains valid (the conditions of its occurrence are satisfied) might be a matter for discussion. This is particularly apparent for the C_T distribution, for which two "extreme" interpretations are possible: either we ignore any change in the value of the parameter while the transition is valid, or we recalculate a new date of occurrence at each change of parameter. Depending on the particular case, we may require one or other interpretation; it is for this reason that the user is given the choice in calculation tools options. Just one option is proposed for the other distributions: this is to recalculate the date of occurrence at each parameter change. Recall that § 3.4 explains in greater depth the semantic options that we might have taken for this type of transition.

In summary, variable parameters for distributions should be used only with the greatest of care!

Slot INDUCING (optional)

Shows the immediate consequences of the event, i.e. the **actions** to be executed.

We now give a few examples, highlighting the categories of events and their connections with failures.

6.3.3.2. Example: normal transitions

Normal transitions are events that describe the normal behaviour of the system being studied.

Example:

```
CLASS alternating_day_night ;
DIST_PARAMETER
    dn DEFAULT 12;
    nd DEFAULT 24 - dn;
ATTRIBUTE
    value DOMAIN'day' 'night';
OCCURRENCE
    IF value = 'day'
    MAY_OCCUR
    TRANSITION pdn
           LABEL "passing from day to night"
           DIST CONSTANT_TIME ( dn )
           INDUCING value <-- 'night' ;

    IF value = 'night'
    MAY_OCCUR
```

```
TRANSITION pnd
       LABEL "passing from night to day"
       DIST CONSTANT_TIME ( nd )
       INDUCING value <-- 'day' ;
```

This class defines an object that will model the alternating day and night (useful, for example, if certain events can only occur at night, given the system operation). The identifiers pdn and pnd provide purely qualitative information that can be used in calculation outputs to denote these events.

6.3.3.3. Faults

Two types of fault can be defined: on demand faults and operational faults.

On demand faults can occur instantaneously (instantaneous distribution) when an object is required to carry out one of the functions for which it was designed (stimulus). This object can then react in two ways:

- either it fails to behave as requested: we say that there has been an on demand fault,

- or it does what it is supposed to do: the stimulus is successful and will be described by a normal transition.

Operational faults are faults that occur while the system is running, without necessarily knowing what caused them (time delayed event, typically with exponential distribution).

- If the fault event is not associated with a FAILURE variable, the keyword FAULT provides qualitative information that will not be used in the rule interpretation.

- If the fault event is associated with a FAILURE variable (giving the name of a failure variable as the name of a fault event), the following process applies:

1. By definition, a fault can only occur if the component is not already faulty.

   It is therefore unnecessary to incorporate the condition <failure variable> = FALSE with the event's condition of occurrence. This is implicit and will be taken into account automatically by the software.

2. Similarly, the occurrence of a fault event always causes the associated failure variable to be set to TRUE.

   It is therefore unnecessary to describe this action explicitly in the rules.

Example of an on demand fault:

Take the example of a filter component. When the component is put into operation, two alternatives may happen:

* The filter may be blocked, causing the component to fail. The probability of this event is determined by an instantaneous distribution of parameter g_bo.

* The filter is not blocked and the stimulus is successful. The probability of this event is determined by an instantaneous distribution of parameter (1-g_bo).

To model this behaviour, the following occurrence rule is written:

```
CLASS filter EXTENDS thermohydraulic_component;
ATTRIBUTE stimulus DOMAIN BOOLEAN DEFAULT TRUE;
FAILURE blockage;
DIST_PARAMETER  g_bo ;
OCCURRENCE
    IF stimulus
    MAY_OCCUR
    FAULT  blockage
           LABEL "blockage on stimulus"
           DIST INSTANTANEOUS ( g_bo )
           INDUCING stimulus <-- FALSE
```

```
      OR_ELSE
      TRANSITION non_bo
            INDUCING stimulus <-- FALSE;
```

This rule is interpreted by Figaro tools as if we had written (in bold: the elements that were implicit in the first version of the rule):

```
CLASS filter EXTENDS thermohydraulic_component;
   ATTRIBUTE stimulus DOMAIN BOOLEAN DEFAULT TRUE;
   FAILURE blockage;
   DIST_PARAMETER  g_bo ;
   OCCURRENCE
       IF stimulus AND NOT blockage
       MAY_OCCUR
       FAULT  blockage
             LABEL "blockage on stimulus"
             DIST INSTANTANEOUS ( g_bo )
             INDUCING     stimulus <-- FALSE,
                          blockage <-- TRUE
       OR_ELSE
       TRANSITION non_bo
             DIST INSTANTANEOUS ( 1-g_bo )
             INDUCING stimulus <-- FALSE;
```

Example of operational fault:

Take the example of a circuit-breaker component which may experience the event 'unintentional opening'.

```
   CLASS circuit-breaker EXTENDS electrical_component;
   ATTRIBUTE position DOMAIN 'open' 'closed'
                     DEFAULT 'closed' ;
   FAILURE uo;
   OCCURRENCE
   unintentional_opening
   IF  position = 'closed'
   MAY_OCCUR
       FAULT uo
       LABEL "unintentional opening"
       DIST EXPONENTIAL ( l_uo );
```

This rule is interpreted by Figaro tools as if we had written:

```
   CLASS circuit-breaker EXTENDS electrical_component;
   ATTRIBUTE position DOMAIN 'open' 'closed'
                     DEFAULT 'closed' ;
   FAILURE uo;
   OCCURRENCE
   unintentional_opening
   IF  position = 'closed' AND NOT uo
   MAY_OCCUR
   FAULT uo
       LABEL "unintentional opening"
       DIST EXPONENTIAL ( l_uo )
       INDUCING uo <-- TRUE ;
```

6.3.3.4. Unavailabilities

Unavailabilities represent events that have occurred in the system's past and which appear when the components are first powered up. Unavailabilities are therefore a particular class of on demand fault and are always associated with an **instantaneous distribution**.

Unavailabilities will be treated exactly the same as on demand faults (see previous section). In other words, if a component in a system has to be stimulated several times, and if certain faults can only occur on **first stimulus**, the knowledge base designer should model differently the occurrence conditions of unavailability events and the occurrence conditions of on demand fault events.

6.3.3.5. Repairs

Repairs represent changes of state due to system maintenance. These are transitions whose effect is to eliminate an object's failures. To this end they are associated with a slot indicating the failures eliminated by the repair.

Example:

```
CLASS circuit-breaker EXTENDS electrical_component;
ATTRIBUTE position DOMAIN 'open' 'closed'
                  DEFAULT 'open' ;
FAILURE uo;
DIST_PARAMETER  m_uo;

OCCURRENCE
MAY_OCCUR
REPAIR rep_uo
    LABEL "repair of unintentional opening"
    REPAIRS uo
    DIST EXPONENTIAL ( m_uo );
```

The keyword REPAIRS is followed by the list of failures eliminated by the repair in question.

If the REPAIR event is associated with a set of FAILURE variables (by the slot REPAIRS), the following process applies:

1. By definition, a failure can only be repaired if the failure has actually taken place.

   Hence it is unnecessary to incorporate the condition <failure variable> = TRUE. This is implicit and will be taken into account automatically by the processing.

If several failures have been associated with the same repair, this repair can only take place if at least one of the failures in question has occurred.

2. Similarly, the occurrence of a repair event always means that the associated FAILURE variable(s) is(are) set to FALSE.

   It is therefore unnecessary to describe this action explicitly in the rules.

Remark: Events of type REPAIR are treated in a particular way by the fault tree generator. If a leaf of the tree corresponds to a FAILURE associated with a REPAIR event, then the basic event leaf will be declared as a basic event of class "GLM" and the distribution parameter of the REPAIR event will be associated with this basic event. However, it is more "proper" (i.e. makes the model easier to read) to explicitly associate a GLM failure model with the failure in a group of rules dedicated to generating fault trees, and to use the occurrence rules only for simulation in forward chaining mode (see § 7.4 covering rule groups).

6.3.4. Interaction rules

Syntax of an interaction rule:

```
INTERACTION
       [name_of_rule]
       [ GROUP       group_list ]
       [ STEP        step_names ]
       [ GIVEN       variable      A set_name
                                   [ OF_CLASS    class ]
                                   [ VERIFYING condition ]]
       [ IF          condition]
       THEN   actions
       [ELSE   actions]
```

Remark: The syntax of the interaction rules for objects is similar to that of the interaction rules for classes.

Interaction rules allow the consequences of an event to propagate throughout the system. In other words, interaction rules enable the **deterministic** relationships between one object and the other objects in the system to be expressed. These relationships can be expressed mainly in two ways:

- Either by altering the state of an object depending on the states of the other objects.

```
CLASS pump;

CLASS backup_pump EXTENDS pump;
   ATTRIBUTE
          start_request DOMAIN BOOLEAN DEFAULT FALSE;
   INTERFACE backed_up KIND pump;
   INTERACTION
          IF THERE_EXISTS x A backed_up SUCH_THAT FAILURE (x)
          THEN start_request ;
```

- Or by altering the state of the other objects depending on the state of an object.

```
CLASS pump ;
   INTERFACE backup KIND backup_pump;
   INTERACTION
          IF FAILURE
          THEN FOR_ALL  x A backup DO start_request (x);
```

The interaction rules defined for a class are introduced by the keyword INTERACTION.

We have the following slots to describe an interaction rule:

Name of the rule (optional)

Allows a rule to be assigned a name. This is useful for "commenting" the rules and in identifying them in debugging tools

Slot GROUP (optional)

Specifies whether the rule belongs to one or more groups (see § 7.4).

Slot STEP (optional)

Specifies that the rule belongs to one or more steps (see § 7.5).

Slot IF (optional)

Describes the condition necessary for the interaction rule to be activated. If no condition is associated with the rule (the rule must be activated regardless of the state of the system), the IF slot is omitted.

*Slot THEN (**mandatory**)*

Specifies the actions that are triggered when the condition of the rule is TRUE.

Slot ELSE (optional)

Specifies the actions that are triggered when the condition of the rule is FALSE.

Example:

```
CLASS component;
ATTRIBUTE
      fed DOMAIN BOOLEAN DEFAULT FALSE ;
INTERFACE
      upstream KIND component;
INTERACTION
      IF WORKING AND THERE_EXISTS x AN upstream SUCH_THAT fed (x)
      THEN fed <-- TRUE
      ELSE fed <-- FALSE;
```

Slot GIVEN (optional)

As we saw in chapter 2, when the rules of the knowledge base are associated with the facts base, a rule defined in a class will be instantiated for each object of that class in the order 0 model. The program examines whether the rule is applicable, and instantiates it if required.

The slot GIVEN defines an interaction rule (**condition and actions**) for a **set of objects defined by a tied variable (interfaced set of objects or a set of objects of a particular class). In this case, the rule will be examined ("instantiated") for all object pairs defined by:**

(Object O1 of the class that has the rule, Object O2 belonging to the set defined by the slot GIVEN)

If the set defined by the slot GIVEN is empty, the rule will not be instantiated.

Take the example of a pump that has several interfaced backup pumps. We wish to express the fact that, if the pump should fail, then all the backup pumps that are currently stopped must be started up. In other words, we require each pair (pump & backup pump) in the system to be examined. The slot GIVEN allows us to write **a single rule** to model this process:

```
CLASS pump;
INTERFACE
      backup_pump KIND pump;
ATTRIBUTE
      state DOMAIN 'running' 'stop' 'start_demand';
FAILURE
      faulty;
INTERACTION
      GIVEN x A backup_pump
      IF faulty AND state(x) = 'stop'
      THEN state(x) <-- 'start_demand';
```

If a system contains a normal pump P1 backed up by pumps P2 and P3, this rule will be instantiated twice:

- for the pair (P1, P2)

```
IF faulty(P1) AND state(P2) = 'stop'
THEN state(P2) <-- 'start_demand';
```

- for the pair (P1, P3)

```
IF faulty(P1) AND state(P3) = 'stop'
THEN state(P3) <-- 'start_demand';
```

The scope of the defined tied variable extends over the whole interaction rule. Hence a variable defined by a GIVEN slot can be used in the other slots of the rule (slots IF, THEN, ELSE).

The set of objects defined by the slot GIVEN is either the name of a (multivalued) interface field or an expression "AN OBJECT OF_CLASS class_name". Just as for complex operators, it is possible to specify optional constraints (OF_CLASS and VERIFYING) so as to treat just a subset of objects.

Example:

```
GIVEN x A <set of objects>
          VERIFYING < restriction conditions for the set >
          OF_CLASS < class name >

GIVEN x AN upstream
       OF_CLASS valve
       VERIFYING flowrate OF x > 2500
IF ...
```

It is possible to define several variables in the same slot GIVEN and to impose certain constraints between these variables.

```
GIVEN x A <set of objects>
  AND y A <set of objects>
  VERIFYING <restriction conditions for the set>

GIVEN x AN upstream
  AND y AN upstream
  VERIFYING x <> y
IF ...
```

Another (equivalent) syntax is possible: the use of several GIVEN slots separated by commas. The rule that precedes may also be written:

```
GIVEN x AN upstream,
GIVEN y AN upstream
  VERIFYING x <> y
IF ...
```

In such a case, Figaro instantiates *card(upstream)*card(upstream)* times the interaction rule, where *card(upstream)* represents the number of objects in the set *upstream*.

Remark: Given that the scope of the declared variables in GIVEN extends over the entire rule, if other variables of the same name are used in other slots by complex operators (FOR_ALL, THERE_EXISTS, THERE_EXISTS AT_LEAST, SUM, PRODUCT, FOR_ALL), we recall that variables with the smallest scope (hence that of the complex operators) will first be taken into account. The following example shows a case where two different tied variables have the same name. Though it may be clear to Figaro, from the point of view of easy readability by humans, it is best to adopt different names for variables.

Example :

```
CLASS pump;
INTERFACE
       upstream KIND valve;
INTERACTION
       GIVEN x AN upstream
       IF THERE_EXISTS x AN upstream VERIFYING flowrate OF x > 2500
              (* this is the x in THERE_EXISTS *)
       THEN flowrate OF x <-- 3000 ;
              (* this is the x in GIVEN *)
```

To finish off, the example below is typical of the use of GIVEN, combined with the object identifier MYSELF. Imagine that in a knowledge base we have the classes component and repair_team. To avoid input errors, the user is requested to provide just the interface cpt (a list of the components being managed) for each repair team. Hence for the class component, there is no direct access via an interface to the repair team managing the component. Nevertheless, we wish to express the fact that during the repair process, the component will "consume" a repair technician in the team that manages it. See below for how this is written in Figaro.

```
CLASS component;
  ATTRIBUTE state DOMAIN 'running' 'awaiting_rep' 'being_repaired'
                DEFAULT  'running';

  INTERACTION
    GIVEN x AN OBJECT OF_CLASS repair_team VERIFYING MYSELF INCLUDED_IN cpt(x)
    IF state = 'awaiting_rep' AND nb_available_rep(x) >0
    THEN state <-- 'being_repaired',
         nb_available_rep(x) <-- nb_available_rep(x) - 1;

CLASS repair_team;
  ATTRIBUTE nb_available_rep DOMAIN INTEGER DEFAULT 1 ;
  INTERFACE cpt KIND component;
```

We could also have written:

```
CLASS component;
  ATTRIBUTE state DOMAIN 'running' 'awaiting_rep' 'being_repaired'
                DEFAULT  'running';

CLASS repair_team;
  ATTRIBUTE nb_available_rep DOMAIN INTEGER DEFAULT 1 ;
  INTERFACE cpt KIND component;
  INTERACTION
    GIVEN x A cpt
    IF state(x) = 'awaiting_rep' AND nb_available_rep >0
    THEN state(x) <-- 'being_repaired',
    nb_available_rep(x) <-- nb_available_rep - 1;
```

This second version is simpler, but on the other hand it would appear more logical to have, in the class component, the rule that triggers the start of the repair rather than in the class repair_team.

7. Advanced notions

In this section we shall study those particular features of Figaro where the approach is a little more complex and which are less used when writing a knowledge base for the first time.

7.1. Precompiling

Before any knowledge base processing, Figaro tools apply a standard precompiler which recognises keywords starting with "#". The following sections clarify the various uses of precompilation in the Figaro context:

- Inclusion of files using #include
- Definition of macros using #define

- Definition of compilation variants (#define, #ifdef, #ifndef, #else, #endif, #undef)

7.1.1. Inclusion of files

The directive:

```
#include "filename"
```

searches for the file *filename* in the main Figaro directory (*filename* could be a relative path with respect to this directory) and inserts the contents of *filename* in place of the directive. Splitting a knowledge base into several files, for example, allows to separate the definition of default reliability data (done by definition of sub-classes containing only these data) from the (more stable) remainder of the knowledge base.

7.1.2. Definition of macros

It is possible to define "macros" introduced by the keyword #define. This saves a lot of time when writing knowledge bases containing repetitive structures such as the declaration of numerous faults. A macro can call other macros, as in the example below where we start by defining the concatenation of two arguments before using it in the declaration of a failure.

Example:

```
(* macro for concatenating two character strings *)
#define stick2(start,end) start/**/end


(* macro defining a fault in operation, using the macro stick2 *)
(* Note that there must not be any space separating the parameters names *)
#define OPER_FAULT(failure_name,comment,param,value)                          \
 CONSTANT param DOMAIN REAL EDITION NOT VISIBLE ROLE DESIGN                   \
      DEFAULT value;                                                         \
 ATTRIBUTE stick2(__inhibit__,failure_name) DOMAIN BOOLEAN EDITION VISIBLE,  \
      MODIFIABLE DEFAULT FALSE;                                              \
 FAILURE failure_name LABEL comment EDITION VISIBLE, MODIFIABLE              \
     DEFAULT FALSE;                                                         \
 RELIABILITY_DATA                                                           \
   GROUP Group_Fault_tree                                                   \
   IF ((visu OF GLOBAL_OBJ = 'unknown') AND (NOT                            \
    stick2(__inhibit__,failure_name)))                                      \
    DEFAULT_MODEL MODEL_GLM                                                 \
    MODEL_GLM GAMMA 0 LAMBDA param MU 0 MODEL_FROZEN;
```

The characters “\” at the end of the lines indicate that the macro goes on to the next line. These characters must be followed **immediately** by the end-of-line character.

Example of a call to the previous macro:

```
OPER_FAULT(FL,"fluid loss in %OBJECT",l_fl,1e-4)
```

Before any processing using the knowledge base, the preprocessor will be called and will "unwrap" the line that calls the macro, replacing it with the macro's text, which has itself been modified by the replacement of the string

- *failure_name* by FL,
- *comment* by "fluid loss in %OBJECT"
- *param* by l_fl
- *value* by 1e-4

This is a recursive process, since one macro can contain a call to another macro.

Apart from providing shortcuts for repetitive structures, macros can simulate the definition of simple functions.

Example

```
#define area(radius)\
3.1416*radius**2
```

7.1.3. Definition of variants in a knowledge base

Finally, using precompilation constants it is possible to define variants of a knowledge base. In the example that follows we have chosen to make the attribute `calculation` visible or not depending on the value of the precompilation constant `__DEBUG__`.

Example (note that all "#" must be in the first column. No indentation is allowed):

```
#ifdef   DEBUG
#define __VISUALISATION__ VISIBLE
#else
#define __VISUALISATION__ NOT VISIBLE
#endif


CLASS t;
ATTRIBUTE calculation EDITION __VISUALISATION__;
```

It is normally when we load the knowledge base into KB3 that we have to choose what value to give the precompilation constants, since this will alter the behaviour of the models used with this knowledge base. KB3 reads these values in the graphical interface configuration file, whose extension is .bdc. However, we can also define and delete precompilation constants directly in the knowledge base file using the instructions:

```
#define  __DEBUG__
#undef  __DEBUG__
```

7.2. Global elements

In modelling a class of systems we sometimes need to define rules or variables attached to the knowledge base itself, and hence to the complete system that the user will enter; and not to some particular component of that system.
To allow the definition of such elements, Figaro is able to declare particular objects using SYSTEM_OBJECT in a knowledge base. These objects exist in any system described with this knowledge base, and their variables are accessible from any other object in the model (no interface declaration is needed). The SYSTEM_OBJECTs are the only ones that can be declared in a knowledge base.

In the following example, the variable `nb_fail(fault_counter)` is directly accessible from any class of the knowledge base.

```
CLASS Fault_counter;
ATTRIBUTE
    nb_fail DOMAIN INTEGER DEFAULT 0;
INTERACTION
    THEN nb_fail <-- SUM FOR_ALL x AN OBJECT OF_CLASS component
                     OF_TERMS (1 * FAILURE(x));

SYSTEM_OBJECT fault_counter IS_A Fault_counter;
```

7.3. Heritage and overriding

The rules governing the overriding and heritage mechanisms implemented in Figaro are fully explained in reference /2/. This manual will outline the principles of these mechanisms and associated prohibitions.

The overriding mechanism is implemented using the two following principles:

- Overriding (redefinition) of a characteristic (or of a part only) is done on its name.

- "The last to speak is right": a class can locally redefine the characteristics of the class(es) inherited. Overriding can also be carried out among inherited classes. In this case, the order of appearance of father classes in the slot EXTENDS determines the order in which the characteristics and their slots are taken into account.

The main prohibitions associated with this mechanism are as follows:

- It is not permitted to modify the nature (constant, attribute, effect, failure) in the class of a field defined in a father.

```
CLASS pump;
ATTRIBUTE
    a DOMAIN BOOLEAN;

CLASS backup_pump
    EXTENDS pump ;
CONSTANT
    a DOMAIN BOOLEAN; (*PROHIBITED*)
```

The field *a* initially defined as an attribute is redefined as a constant.

- Overriding the domain of a field is not permitted:

```
CLASS pump;
ATTRIBUTE
    a DOMAIN BOOLEAN;

CLASS backup_pump
    EXTENDS pump ;
ATTRIBUTE
    a DOMAIN REAL; (*PROHIBITED*)
```

The field *a* initially defined as Boolean is redefined as real.

On the other hand, overriding an enumerated domain is permitted (extending, modifying, restricting it, etc.).

Overriding an enumerated domain involves the following restrictions:

1. The domain must stay as an enumerated domain.
2. Overriding an enumerated domain is permitted provided at least one element of the new domain is common to both domains.

- Overriding an interface is permitted only if the class used for the overriding is a sub-class of the overridden class.

```
CLASS hydraulic_component;
```

```
INTERFACE
    backup KIND hydraulic_component;

CLASS pump EXTENDS hydraulic_component;
INTERFACE
    backup KIND pump;
```

- Overriding an interaction rule by an occurrence rule (and conversely) is not permitted.
- Overriding the characteristics of a class in a particular object that is an instantiation of this class is not permitted.

7.4. Rule groups

In principle, when using a Figaro model all the interaction and occurrence rules belonging to objects in the system being studied are analysed. This basic principle can be refined by introducing the idea of **rule groups**.

It is possible in a knowledge base to define particular **groups** of rules based on the model's set of rules. When running a Figaro model, this possibility allows a selective analysis of the model by restricting the set of rules to be analysed to one or more rule groups.

For example, take the case of a knowledge base intended for use in automatically generating fault trees for thermohydraulic systems. Cold redundancy between components in this knowledge base is not modelled since, in order to investigate the system via a fault tree, we make the approximation that all system redundancies are hot (active).

We then want to use this same knowledge base to construct a "state graph" model to validate this approximation. To this end we shall enhance the initial knowledge base with rules modelling the active redundancies among components and will create two groups of rules, namely "standby" and "active" groups.

- All rules modelling the system without taking account of cold redundancies will be attributed to the "active" group.

- All rules modelling normal/backup reconfigurations will be attributed to the "standby" group.

If the knowledge base is to be used to generate a state graph model, only the rules in the "standby" group will be analysed.

Rule groups in the knowledge base are declared from the start in the section GROUP_NAMES (**the information listing the group identifiers used in the base is a piece of information that relates to the whole of the base and not to a particular class**).

```
GROUP_NAMES
    standby;
    active;
```

For an occurrence or interaction rule it is then possible to specify that the rule belongs to one or more groups (enumeration of groups) by using the slot GROUP

```
INTERACTION
GROUP standby active
IF THERE_EXISTS x AN upstream SUCH_THAT linked OF x
THEN linked;
```

It is not mandatory to attribute a rule to a particular group. The following conventions have been adopted in using a Figaro model:

- If no group name is specified, the entire set of rules in the model is processed.

- The user can request the processing to be based on a union of groups from a particular list.

  E.g: analysis of *standby* UNION *active* groups.

- Rules that do not belong to any group can be exploited specifically by requesting an analysis of the group identified by NO_NAME.

  E.g: analysis of *active* UNION NO_NAME groups.

It is very important to note that the organisation of rules in a knowledge base into rule groups is an important step in the design of a knowledge base. In particular, the role of each group of rules (including the group NO_NAME) should be explicitly stated for users of the knowledge base.

It should be noted that splitting the Figaro model into groups is only possible for rules. The definition of value fields for the model should therefore be appropriate, regardless of the rule groups being analysed. It can happen that, given the rules selected, some value fields are completely unused under certain types of processing; this is not too much of a problem, however, since the processing performance is unaffected.

Groups that have been defined in the knowledge base can be exploited in the facts base associated with that knowledge base. The redefinition of a GROUP in a facts base is not permitted and gives rise to an error message: duplication of group names. All data defined in the knowledge base are accessible in the facts base. They do not therefore have to be redefined.

7.5. Steps

As we saw in chapter 2, the order in which the interaction rules of a Figaro model are applied is absolutely not defined. In other words, the rule bases must be written in such a way that, whatever the order of application, the result obtained is the same.

However, it can happen that we may need to introduce an order when applying the interaction rules. The Figaro language therefore allows **the interaction rules to be combined into blocks of rules** called STEPS and **to order these steps** so that the rules can be examined in a precise order.

Take the example of a class of thermohydraulic systems for which we wish to model the propagation of fluid. This class of systems contains objects of class *component* that may, or may not, be sources of fluid. To model the propagation of fluid, two rules have to be modelled:

- *source* objects are always fed,
- objects of class *component* are fed if there is a fed component in the *upstream* interface.

To minimise the number of rules to be applied, it helps to apply the first rule first. It is for this reason that two steps are introduced into the knowledge base: an *initialisation* step and a *propagation* step.

```
STEPS_ORDER
    initialisation;
    propagation;

CLASS component;
INTERFACE
    upstream KIND component;
EFFECT
    fed DOMAIN BOOLEAN;
CONSTANT
    source DOMAIN BOOLEAN;

INTERACTION
    init
    STEP initialisation
    IF source = TRUE
    THEN fed;

    prop
    STEP propagation
    IF THERE_EXISTS x AN upstream SUCH_THAT fed (x)
    THEN fed;
```

7.5.1. Declaration of steps

A step can have one or more declared identifiers in the form of a list of identifiers (e.g. the identifiers "init" and "initialisation" - separated by a comma - identify the same step).

It is also possible to associate a condition with a step. A condition associated with a Figaro step is a Boolean expression that may, or may not, be complex. This condition will be evaluated when the rules associated with the step are evaluated:

- If the condition is not satisfied, no rule in the step is evaluated (this equates to an appreciable optimisation since the rule conditions are not evaluated).
- If the condition is satisfied, the conditions of the step rules are evaluated. This is the normal process of rule activation.

Remark: A step without a condition is assumed to be a step whose condition always returns TRUE. The step rules are therefore always evaluated.

This special feature has been added to optimise activation of the rules in Figaro processing. In some sense the step condition corresponds to a factorisation of the common part of the rule conditions that give rise to this step.

Step names and their order of execution are declared at the beginning of the knowledge base in the section STEPS_ORDER (the information listing the group identifiers used in the base is a piece of information that relates to the whole of the base and not to a particular class).

The steps are ordered according to their order of definition in the knowledge base. If the keyword STEPS_ORDER appears several times in the knowledge base, we assume that the steps are defined in the order in which the knowledge base is read.

Example 1: Declaration of steps without conditions.

```
STEPS_ORDER
    initialisation, init;
    propagation;
```

In this example the order number of the initialisation step is 1, and the order number of the propagation step is 2.

Example 2: Declaration of steps with conditions.

```
STEPS_ORDER
    initialisation CONDITION NOT initialisation_done(global_parameters);
    propagation;

CLASS parameters;
ATTRIBUTE initialisation_done DOMAIN BOOLEAN DEFAULT FALSE ;

SYSTEM_OBJECT global_parameters IS_A parameters;
```

Example 3

```
STEPS_ORDER                                    STEPS_ORDER
step1;                                         step1;
step2;                  is equivalent to       step2;
                                               step3;
STEPS_ORDER
step3;
```

Definition of new steps is permitted in the facts base. In this case, the steps declared in the facts base are concatenated following the steps in the knowledge base. They are therefore always higher in order than the steps in the knowledge base.

Duplication in the facts base of a step already defined in the knowledge base is not permitted and gives rise to an error message indicating duplication of step name. All data defined in the knowledge base are accessible in the facts base. They do not therefore have to be redefined.

7.5.2. Combination of rules by steps

When steps are declared, each model rule **MUST** be associated with **at least one step**. This association is made using the slot STEP in the interaction rules.
Example:

```
INTERACTION
    init
    STEP initialisation
    IF source = TRUE
    THEN fed <-- TRUE;
```

However, for reasons of expediency, an automatic system within the Figaro tools assumes that the rules for which the membership has not been specified are contained in the so-called 'default_step'.
This default_step can be located anywhere within the order of steps; it is sufficient to declare it as an explicitly defined step.
An interaction rule can be associated with more than one step: all this means is that, at each step declared in the slot STEP of this rule, the inference mechanism takes it into account.

7.5.3. Application of interaction rules in forward chaining mode.

The introduction of steps in a Figaro model modifies the way the application of the interaction rules in forward chaining mode is processed. In this case, the rules are processed step by step in the order specified by STEPS_ORDER, bypassing those steps for which the condition has not been satisfied. Bypassing step k equates to assuming, in the formal definition of Figaro 0 semantics in section 3.2, that the function I_k is the identity function.

It is important to note that organising a base of interaction rules into steps allows the rule base to be structured and model running time optimised.

7.5.4. Application of interaction rules in backward chaining mode.

To date, steps are not used when running a Figaro model in backward chaining mode.

7.6. Equations

Certain problems such as the calculation of current and voltage in an electrical network are hard to express in a rule-based formalism. In the best-case scenario we arrive at an iterative solution via many applications of the rules, which end up by making attributes converge towards the values we wish to calculate.
In such a case, it is far more elegant to consider (just for the time of the equations resolution) that a number of attributes are the unknowns of a set of equations.

7.6.1. Systems of equations

If a knowledge base requires one or more systems of equations to be written, each system must be identified by a name. The names of the systems of equations used in the knowledge base are declared at the beginning of the knowledge base in the section SYSTEM_NAMES (the information listing the equation systems identifiers used in the base relates to the whole of the base and not to a particular class).

```
SYSTEM_NAMES
    current; voltage
```

7.6.2. Equations

Syntax:

```
EQUATION      equation_name
      [ KIND LINEAR ]
      EQUATION_SYSTEM     equation_system_name
      [ IF          expression ]
      FORMULA        [ expression = expression ];
```

The equations in a system are described in the classes of the knowledge base (an equation belongs to a class).

An equation can belong only to a unique system of equations.

Equations will be defined by the class field EQUATION.

Slot details:

*Name of the equation (**mandatory**)*

Assigns a name to the equation. This is useful for "commenting" the equations and for identifying them in debugging tools.

*Slot EQUATION_SYSTEM **(mandatory**)*

Specifies the system to which the equation belongs.

*Slot KIND (**optional**)*

Must be followed by a keyword that will point towards a solution method appropriate to the problem. For the time being, only the LINEAR kind is envisaged, for which there are many efficient algorithms.

*Slot IF **(optional**)*

IF is an optional slot that expresses a condition that must be satisfied by the state of the model so that the formula may be taken into account in the system of equations. For example, electrical equations are indeterminate for components that are not connected to a source of power. Hence the equations that relate to a component must be conditioned by the fact that it is connected to a source; this is easily done through the normal interaction rules.

*Slot FORMULA (**mandatory**)*

This slot must be followed by the equation itself, in which the unknowns are preceded by a question mark '**?**'. These unknowns MUST be declared attributes. The value of these attributes will be calculated when the equation system is solved mathematically. The values of the attributes will be modified.

To simplify the processing, linear equations must be written in a "standard" form in which both sides of the "=" sign are in the form a1* ?x1 + a2* ?x2 +.... an* ?xn, where the ai are numerical expressions involving no equation unknowns.

Example:

```
CLASS  node;
ATTRIBUTE
       a DOMAIN REAL;
       b DOMAIN REAL;
       c DOMAIN REAL;
       x DOMAIN REAL;
       y DOMAIN REAL;
       isolated DOMAIN BOOLEAN;
EQUATION
       equ1
       KIND LINEAR
       EQUATION_SYSTEM current
       IF NOT isolated (* The formula only makes sense in this case *)
       FORMULA  a*?x + b*?y = c;
              (* ?x and ?y are the unknowns in the formula: they will be
                 calculated when the equation system is solved.
                 x and y are attributes *)
```

Recall that the solution of a system of equations starts at the required time via an interaction rule.

Example: start the solution for the "current" system.

```
INTERACTION
    r1
    STEP solve
```

```
THEN SOLVE_SYSTEM(current);
```

8. Figaro model documentation

Apart from the normal comments between "(*" and "*)", we can add formalised comments to a model, indicated by keywords.

Outside of classes, we can write a description for a knowledge base preceded by the keyword KB_DESCRIPTION. There must be *no more than one* description per knowledge base. This can be any character string, which may, if required, extend over several lines.

A description can be written within a class, preceded by the keyword DESCRIPTION. There can be *no more than one* description per class. This can be any character string, which may, if required, extend over several lines.

Finally, all fields declared in INTERFACE, CONSTANT, ATTRIBUTE, FAILURE, and EFFECT may be provided with a (unique) slot LABEL. The rules of inheritance and override apply to these fields, including their labels.

During an instantiation at order 0, the descriptions of classes and the knowledge base disappear, while the labels of the fields mentioned above are preserved. Within these labels, the character string %OBJECT is replaced by the name of the object to which the label belongs.

Here is an example of a knowledge base using all the possible types of formalised documentation:

```
KB_DESCRIPTION "Knowledge base with comments";

CLASS pump;
DESCRIPTION
"Generic pump, to be specialised
into sub-classes"; (* a description can be written over several lines *)

  CONSTANT flowrate DOMAIN REAL
                    DEFAULT 10
                    LABEL "Rated flowrate of the pump %OBJECT";
  FAILURE leak LABEL "Internal or external leak" ;

CLASS electric_pump EXTENDS pump;
DESCRIPTION "Pump driven by electricity";
```

9. Model consistency

Taking account of the complexity of some of the knowledge bases in KB3 (the most complex is the TOPASE tool dedicated to the reliability analysis of VHT stations [8], and comprises 27,000 lines of Figaro code), it would appear increasingly necessary to ensure that these do indeed do what the author intended, or at least to design a tool that could **warn the designer that his knowledge base, independently of the objectives it might pursue, has undesirable properties: we shall call these "inconsistencies"**. These undesirable properties may be of several types: contradictions, unbounded numerical values, circular definitions (A defined in terms of B and vice versa) that are impossible to resolve, division by zero, underflow or overflow in real number calculations, etc.

The objective of this chapter is to describe the principles behind the methods either to ensure the consistency of the knowledge bases **by construction** or to detect the existence of inconsistencies. Demonstrating inconsistency is impossible in the general case, i.e. for a knowledge base which has not been constructed in accordance with the recommended methods. This is hardly surprising, given that the Figaro language combines all the possibilities of propositional logic, calculation with real numbers and automatic systems.

Should it not be possible to prove the consistency of a knowledge base, it is sometimes possible to treat system models case by case. **This means that we can envisage the use of imperfect knowledge bases,** checking for each system studied that the models constructed do not "fall through the cracks" in the base. A typical example is that of knowledge bases that only guarantee model consistency provided the topology of the systems being studied is not looped.

This chapter is organised as follows: we start by exploring the relationships between consistency of order 1 and consistency of order 0. The relationships established allow us to limit the work involved in demonstrating consistency to order 0. Then, making use of the formal definition of the semantics of the order 0 Figaro language, we construct a typology of the inconsistencies that a Figaro 0 model might contain. Then we provide a set of rules for constructing a knowledge base that will avoid the identified inconsistencies. For the trickiest of these rules, namely the organisation of the model "steps", we provide the principles of a tool that will help the user perform this task. To finish, we examine the case of feasible demonstrations of consistency to order 0 only, i.e. case by case for models of systems input by the user.

9.1. The relationship between order 1 and order 0 inconsistency

Demonstrating (as far as possible) that a knowledge base is free of inconsistencies amounts to:

Whatever the system (while satisfying the constraints described in the knowledge base) entered by the user, the Figaro order 0 model generated from this system and knowledge base will contain no inconsistency.

Inconsistencies may appear either during the instantiation phase (transforming an order 1 model into an order 0 model), or during the phase running the model (when it is used to simulate the system being modelled). Potential instantiation problems may be detected **exhaustively** by the syntax controls (described below) carried out **on the knowledge base**.

9.1.1. Syntactical checks

A knowledge base is a collection of **classes** of objects described in the Figaro language of **order 1** (i.e. describing generic knowledge, in particular using operators adept at handling **sets**), that the knowledge base user can assemble in infinitely many ways within the system models.

Nevertheless, this freedom is constrained by the information given in the interface slots KIND and CARDINAL declared in the knowledge base (e.g. upstream of an electrical component there can only be (between zero and N) electrical components. This information means that KB3's syntactical checking of the knowledge base ensures that no problem in attempting to access a non-existent variable (e.g. the pressure of an electrical component) will ever arise when instantiating a model. This syntactical checking process is very complex since it has to take account of all the rules of inheritance between classes; its usefulness is verified on a daily basis by knowledge base designers.

Before running any model entered by the user, KB3 checks that all the constraints defined by the knowledge base designer have been met.

This checking process may produce error messages such as: "not enough/too many objects in interface X of object Y".

Satisfying the classes of object interfaced with a given object is ensured by the input interface, which is intolerant of errors in this domain.

Prior to any reliability processing, these checks ensure that the operation of instantiation of order 0 for some particular system will pass without problem.

This is not sufficient, however: this check can in no way protect against the kinds of inconsistency described in the first section of this chapter.

9.1.2. Consistency of behaviour

The precise definition of what we mean by behavioural consistency requires from the outset a formal definition of the model semantics specifying this behaviour.

Making use of the definition of the semantics of the Figaro language (see chapter 3), we shall show where inconsistencies can arise. It will then be possible to set out the rules for constructing a knowledge base that will ensure the consistency of any order 0 Figaro model produced using this base.

9.2. Typology of possible inconsistencies

The formalism of the operation of the "Figaro 0 automaton" (we shall reuse the notation of chapter 3) makes it possible to define what we mean by model inconsistency. We also define desirable properties.

The order 0 Figaro language can be considered as a sort of synthesis of existing concepts in artificial intelligence languages using production rules, and in stochastic Petri nets. It is therefore natural to draw inspiration from the work done in these two domains in order to develop methods for checking consistency in the Figaro language.

For Petri nets, the properties that can potentially be established from a net are: the ability to return to the initial state from any state, the bounded character of markings, the liveliness of transitions (a given transition T is "live" if from any state that can be reached from the initial state, it is always possible to find a sequence of transition crossovers, including the crossing of T); the lack of an absorbing state, the existence of invariants (linear combinations of place markings that are constant, regardless of changes in the net) [10],[11].

Except for the existence of invariants, these properties of Petri nets are very easy to transpose into the Figaro context. They involve the following properties:

– P1: the finite nature of the state space,

– P2: the liveliness of transitions (with the same definition as for Petri nets),

– P3: the non-existence of absorbing states,

– P4: the ability to return to the initial state from any arbitrary state.

We have already noted that an absence of numerical values in V is sufficient to ensure that the set of states (included in Ξ) actually attainable by the model is **finite** (the fact that some numerical variables have been **deduced** does not necessarily invalidate this property).

The last two properties are clearly to be sought only if we want to model a repairable system, e.g. with a view to calculating its asymptotic availability.

In artificial intelligence, three criteria are generally adopted to validate a rule base, whatever its domain:

- logical consistency: we cannot have one thing and its opposite simultaneously,
- completeness: a solution to the problem exists regardless of the initial data,
- pertinence: the knowledge base conforms with physical reality (this point is not covered here).

The transposition of the idea of consistency and completeness to a Figaro 0 model requires a certain amount of interpretation. We can encounter the following undesirable types of behaviour in the operation of the automatic process:

- Inc1: Impossible to calculate $I(V_0, Y_0)$, or $I(X), X$ being the state obtained via application of one or more groups of transitions from a previous state. There is also the case for which calculation is possible, but where the result depends on the order of the rules. These problems of completeness and consistency in a rule base were studied at EDF in 1985. The rules given in this article for safely constructing a knowledge base are deduced from the theorems proved in [5].

- Inc2: Inconsistency related to transitions applicable **at the same instant** from a given *instantaneous state X*. The "physical" meaning of this situation is that several actions of zero duration (but with random result) are initiated in parallel, and must therefore produce their effects at exactly the same time. Typically, this might involve simultaneous requests to start several components. Because of the idea of groups of transitions, there is no point in exploring all possible orders of initiating transitions, but that generally presupposes that the groups of transitions are mutually independent. Independence means two things here:

- the application of a transition does not undermine the condition for initiating **another** transition (on the other hand, it is to be expected that it makes its own condition false),
- regardless of the order of the transitions, for a given combination of transitions, each taken in a group, we get the same result for t1° t2°… tk(X).

For highly specific reasons in certain models, it is possible that one or other of these conditions may not be satisfied; that must, however, be the user's deliberate choice. Hence any tool using the Figaro language must alert the user that, due to a potential lapse in concentration, one of these conditions may not be satisfied.

The idea of a transition group is a great help in modelling since it allows the following kind of situation to be treated in a very elegant way: in a system having independent components that have to start at the same instant, there is no point in exploring the n! sequences that lead to the same result (i.e. a given combination of successes and failures among the 2^n existing possibilities). On the contrary, the Figaro 0 automatic system can directly produce these 2^n outputs resulting from the initial instantaneous state.

- Inc3: A more global undesirable behaviour can be added to these local inconsistencies, namely the infinite chaining of a succession of instantaneous transitions; this is possible since this series of transitions reverts to a state from which it can start again. At this present time, as far as we know there is no method for predicting this kind of behaviour.

It is useful to complete the remarks above by the observation that it would be **absurd** to attempt to demonstrate consistency properties that amalgamate the rules of occurrence and interaction. For a repairable system it is obviously desirable that there are rules that set a failure to TRUE, and that once done, other rules allow the same failure to be set to FALSE. Put simply, these operations cannot all be carried out in zero time, which would lead to a situation of the type described in the paragraph above.

We shall now give a number of methods of writing knowledge bases that avoid the inconsistencies Inc1 to Inc2 **by construction**, and/or ensure the properties P1 to P4 mentioned above.

9.3. Graph of dependencies between variables

Certain methods to be outlined draw on the simple concept of a graph of dependencies between variables. Since this idea is important and has useful applications quite apart from consistency checking, we shall devote a complete section to it in this document.

In a Figaro 0 model, we say that variable v1 "directly influences variable v2" in one of the following two situations:
- there exists an assignment instruction (<-) where v2 appears in the left part and v1 on the right (e.g. v2 <- v1),
- there exists an interaction or occurrence rule in which v1 appears as the premise and v2 in the left part of the conclusion.

The best way to represent a set of relationships that exert an influence between variables is to use a directed graph. In such a graph the existence of a path between two nodes associated with variables represents an indirect influence.

If we consider the variables from a probabilistic point of view, these variables can be identified with continuous-time stochastic processes; we can therefore speak of stochastic dependence between these processes. We can show that stochastic independence between two process variables is guaranteed provided there is no path between these two variables in the graph we have just defined.

This property is extremely useful since it allows the global model to be broken down into sub-models that can be solved independently. From a qualitative point of view (e.g. model checking), it means that the construction of a global graph of states can be replaced by a series of much smaller independent graphs; from a quantitative point of view, it simplifies the calculation of the probability of a state variable's being in a given state by considering only the sub-model containing the set of variables that influence the variable of interest.

Hence, if we wish to calculate the probabilities of a state variable's being in a given state or of being in different states for a **group** of variables, we can start by **restricting the model to the set of variables that influence this group of variables.** The sub-graph that contains only the variables that directly or indirectly influence the group of variables being considered defines the minimum sub-model to be solved in order to achieve this objective.

In practice, extracting such a sub-model involves eliminating from the global model any expression containing a reference to a variable that has to vanish. Hence the rules:

IF v1 OR v2 THEN v3 ELSE v4;

IF (v4 AND v5) OR NOT v3 THEN v6;

become: IF NOT (v1 OR v2) THEN v4; in the case where we are only interested in v4.

Furthermore, it is of interest to distinguish another kind of subsystem: **strongly connected components (SCC) of the graph**, since these constitute sub-models that are indivisible in terms of solution. All the variables of an SCC are interdependent.

We then construct a "supergraph", with each node corresponding to an SCC. This supergraph is necessarily circuit-free (if it contained one, there would be an SCC bigger than those identified). A solution technique starting from supergraph sources and progressing to the SCC of interest may be used later.

9.4. Safe methods for writing a knowledge base

9.4.1. A pyramidal graph of dependencies

A very safe method for constructing a knowledge base is to give a pyramidal structure to the graph of dependencies between variables, creating a "supergraph" whose root SCCs contain all the essential variables. The ideal is to have the smallest possible number of variables in each root SCC.

The example below of a knowledge base describing a telecommunications network is typical.

```
(* KB for modelling telecoms networks *)
CLASS component;
FAILURE          unavailability;
DIST_PARAMETER   gamma_unavail DEFAULT 0.00005 ;
EFFECT           linked LABEL "%OBJECT connected to a source" ;
ATTRIBUTE        unavail_tested DOMAIN BOOLEAN
                    DEFAULT FALSE ;
OCCURRENCE
  IF NOT unavail_tested
  MAY_OCCUR
       FAULT unavailability
       DIST INS ( gamma_unavail )
       INDUCING unavail_tested <-- TRUE
  OR_ELSE TRANSITION Ok
       INDUCING unavail_tested <-- TRUE;
(*-----------------------------------*)
CLASS node  EXTENDS component;
CONSTANT
  function DOMAIN 'source' 'target' 'intermediate'
           DEFAULT 'intermediate';
INTERACTION
  IF WORKING AND function = 'source' THEN linked;
(*-----------------------------------*)
CLASS uni_dir_link EXTENDS component;
INTERFACE
  origin KIND node CARDINAL 1;
  end KIND node CARDINAL 1;
INTERACTION
  IF WORKING AND linked OF origin THEN linked OF end;
(*-----------------------------------*)
CLASS bi_dir_link EXTENDS component;
INTERFACE
  extremity KIND node CARDINAL 2;
INTERACTION
  IF WORKING
   AND ( THERE_EXISTS x AN extremity SUCH_THAT linked OF x )
   AND ( FOR_ALL x AN extremity WE_HAVE  WORKING OF x )
  THEN FOR_ALL y AN extremity DO linked OF y;
```

Shown below is a three-node network (to simplify we assume vertices without faults) followed by the graph of dependencies between variables for the system; in this graph the variables that form part of the same SCC are included in a rectangle:

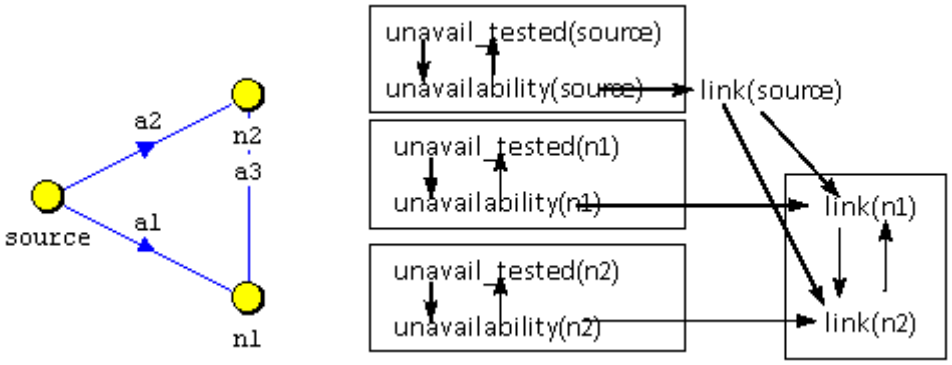

Figure 4. A network and corresponding graph of dependencies

The majority of bases developed to date generate pyramidal dependency graphs. This is so in particular for bases whose objective is to produce fault trees. Most of the essential variables are failures, and failures of different objects are mutually independent.

A simple and natural way of obtaining a pyramidal graph is to create dependencies only between essential variables that are internal to the classes (hence to objects after instantiation in some particular system) and not to create any dependency link in the sense of deduced variables towards essential variables.

By giving this kind of architecture to a knowledge base, we are able to demonstrate that there are no inconsistencies of classes Inc2 and Inc3, as well as properties P1 to P4 (provided there is no inconsistency of class Inc1) using very local reasoning that draws on several occurrence rules belonging to the same class.

These proofs can be carried out in the knowledge base, and hence **for any system modelled using this base**.

9.4.2. Reasoning through monotonic inference

The objective of the constructional rule that we give below is to avoid inconsistencies of type Inc1.

In reference [9], Hery and Laleuf proved a theorem that gave a sufficient condition for commutative convergence of an inference run like that of a step of the interaction rules in Figaro 0. An immediate consequence of this theorem is the following **corollary**:

IF
- all the actions of the rules imply that Boolean variables initialised as FALSE before the inference (EFFECTs in general) are set to TRUE,
- all the conditions pertinent to this set of variables test whether they are TRUE,

Then commutative convergence of the inference is ensured.

Though employed in the majority of knowledge bases whose objective is to produce fault trees, this very simple framework is said to be that of **"monotonic inference"** This name comes from the fact that when a Boolean quantity has been set to TRUE, no rule will allow this conclusion to be reversed. The second condition of the theorem ensures that no Boolean quantity, before stabilisation of those on which it depends, may be unintentionally set to TRUE,

The great advantage of this theorem is that it is extremely easy to check these assumptions (even without tools) for the knowledge base, as we can see in the example given above modelling telecommunications networks. It is therefore possible to prove **for the knowledge base that, regardless of the assembly of classes by a user of this base, no Inc1 inconsistency will be created.** This very strong property has been used in all the operational knowledge bases used at EDF for studies of safety at nuclear power plants.

Monotonic inference means that it is possible to model **without risk of inconsistency and in a very concise and natural way flow propagation in looped systems**, as we can see in the telecoms network example above. In the great majority of classical formalisms (fault trees, Boolean logic, Petri nets, Model Checking formalisms, etc.), this modelling problem can either lead to impossible situations or to very lengthy and barely readable expressions, even though we constantly come across it in actual systems.

9.4.3. Making good use of the steps

To avoid Inc1 inconsistencies in situations where monotonic inference provides an excessively restrictive framework, we can use steps.

The commutative convergence theorem [9], somewhat more general than the restricted application we used it for in the previous section, shows the importance of having the maximum number of so-called "non-receiver" state variables for an inference, i.e. that are not modified by any action of the rules. An effective way of reducing the number of receiver variables is, of course, to reduce the number of rules to be taken into consideration. This is precisely what breaking up into steps allows. Once the inference corresponding to a given step has finished, the variables that were receiver for that step may (possibly) become non-receiver variables for the following steps.

Here is a very simple example of steps. Suppose that we need to write a rule: "IF NOT effect1 THEN effect2" Because of the negation in the condition, we are out of the context of monotonic inference, **unless** we put the rules acting on effect1 into a step preceding the step into which we put the rule above.

9.5. Automatic sequencing of interaction rules

When writing interaction rules, it may prove difficult to organise these rules systematically into steps to ensure the proper operation of the whole model. In general, the knowledge base designer can easily establish local sequences of rules or groups of rules, but it is harder to manage the entire set of rules and even harder to add a new rule into an existing knowledge base.

An automatic sequencing tool has been developed to reduce these difficulties. This tool presupposes that a rule can only be used if the premises on which it is based are stable. In this context, the order in which the rules are executed can be deduced directly from the graph of dependencies between rules, a concept analogous to that of the graph of dependencies between variables, which we shall introduce later.

9.5.1. Definition of interdependencies between rules

We define a rule R2 as being dependent on a rule R1 if there exists a variable V, modified by R1, that is involved in the premises or in the terms for calculating R2.

This definition presents no difficulty in a Figaro 0 model, but it is far more advantageous to be able to reason at the knowledge base level. The definition has therefore been adapted to knowledge bases by taking account of the possibility of inheritance between classes.

Hence we assume the elements of different classes to be distinct, even if they are constructed through simple father-type inheritance: the variable V of class T is assumed different from the variable V of one of the sons of T. We then define a rule R2 to depend on a rule R1 via the intermediary of the variable V of class T in the following case:

- the variable V of a father-class T1 is modified by R1,
- the variable V of a father-class T2 is involved in the premises or in the terms for calculating R2.

Classes T, T1 and T2 can be combined.

9.5.2. Organisation of the rules

Once the first interdependencies have been constructed, the dependencies must be completed in such a way as to construct a partial-order relationship; i.e. a transitive antisymmetric relationship.

Transitiveness is obtained by adding the necessary dependencies: if R3 depends on R2, which depends on R1, then a dependency between R1 and R3 is added.

If the added interdependence relationships ensure that the set is transitive, then antisymmetry must be ensured: if R1 depends on R2 and R2 depends on R1 the R1 = R2. It is clear that we cannot simply fuse the rules together; antisymmetry is then obtained by combining the rules within the steps. The order relationship is no longer expressed between rules but between steps: a step S1 depends on another step S2 if one of the rules of S1 depends on a rule in S2. This grouping of the rules continues until the relationship between steps is antisymmetric.

The grouping of the rules violating the antisymmetry constraint within a rule is explained by the fact that these rules depend on one another in a circular sense. It is therefore necessary to combine these rules and execute

them together until the result converges. In the extreme case of a set of rules that are entirely interdependent, the algorithm leads to the creation of a single step containing all the rules.

Once the partial order relationship between steps has been established, it may be used directly to order the execution of the steps; we start with the independent steps and then execute the steps that depend only on those already accounted for, and so on until the steps have been exhausted.

9.5.3. Using the rule sequencing tool

The sequencing tool can be used either when a new knowledge base is being created or when modifying an existing base.

For a new knowledge base, the algorithm defines groups of interdependent rules and hence checks the pertinence of the groups with respect to the physical model.

The algorithm can also be used to complete a partial definition of the rule order. The designer expresses certain mandatory sequences such as the fact that the group of rules G1 must be carried out before group G2, and the algorithm completes the order by indicating whether the predefined sequences can be satisfied (where group G1 depends on group G2).

Finally, if the designer has carried out a division into steps, it is possible to subdivide each step in such a way as to obtain an optimal order of execution of the rules to accelerate the inferences.

When modifying a knowledge base, the principle is to add the new rules outside the steps defined in the knowledge base and to entrust the algorithm with the task of placing these rules correctly in the new steps or in the existing steps. The algorithm warns when the new rules introduce circular dependencies between existing steps.

9.6. Detecting inconsistencies in a Figaro 0 model

Despite all the elements we have provided up to now, there are, of course, cases where nothing can be proven about the knowledge base, since the rules of safe construction are too restrictive to allow certain types of system to be modelled. It is still possible to perform checks on a particular Figaro model of order 0.

Consider, for example, the following knowledge base for determining which nodes are linked with sources in a network of given topology (NB this is completely deterministic - the automatic system can take just one state: the complete initial state calculated from the incomplete initial state chosen by the user):

```
CLASS component;
ATTRIBUTE linked DOMAIN BOOLEAN DEFAULT FALSE;

CLASS node  EXTENDS component;
INTERFACE upstream KIND; component;
INTERACTION
   IF THERE_EXISTS x AN upstream SUCH_THAT linked OF x
    THEN linked
   ELSE NOT linked;

CLASS source  EXTENDS component;
  INTERACTION
    THEN linked;
```

The state variable "linked" is an ATTRIBUTE and not an EFFECT. It is therefore not reinitialised before each pass of the interaction rules. In a closed-loop topology containing two nodes n1 and n2, such that n1 has n2 in its upstream interface and vice-versa, the rules contained within the two nodes, after instantiation, can be summarised as: linked(n1) ← linked(n2) and linked(n2) ← linked(n1). So we see that depending on the initial values chosen by the user (we assume that he can make a mistake) for linked (n1) and linked(n2), two states are stable through application of the interaction rules, namely that for which n1 and n2 are linked, and that for which they are not. Actually, only the second state has any physical meaning, given that the system has no source.

Despite this defect (which can be corrected simply by declaring "linked" as an EFFECT), this knowledge base can be used in topologies without loop. In such a topology, the graph of dependencies between the different "linked" attributes of the system is acyclic. The inference will gradually stabilise the values of the various "linked" attributes, starting from the sources or root nodes in the topology of the system, for which the "linked" attribute is set to TRUE for sources and FALSE for root nodes, regardless of the initial state declared by the user.

More generally, it often happens that when using the equivalence rules (IF... THEN... ELSE) we have to be content with using the knowledge base only for systems whose topology is not looped.

It is therefore preferable by far to use monotonic inference whenever possible, although this can potentially lead to knowledge bases that are slightly less readable since the many and varied conditions that allow a variable to be set to TRUE may be dispersed among different rules.

Another example of detecting possible inconsistency only in Figaro 0 and even while the model is being executed is that of detecting inconsistencies of type Inc2**.** This kind of conflict is easily illustrated by a Petri net translated into Figaro language. Examination of the model would detect what are referred to as **structural** conflicts in the net, due to the fact that a place P is upstream of two instantaneous transitions t1 and t2 (see [11]). However, there may be many such conflicts in a model without its necessarily being a problem. It is when the model is executed that some of these structural conflicts are transformed into **effective** conflicts (see [11]). Such is the case in the example above, when drawing t1 consumes the tokens that would allow t2 to be validated, and vice versa. The Figseq tool detects these conflicts, warns the user of their existence and displays the sequence of events that led the system from the initial state to one in which the conflict is effective.

9.7. Conclusion to model consistency

We have shown that the Figaro modelling language is able to address legitimate concerns about the consistency of models for increasingly complex systems.

In particular, using simple design rules for knowledge bases in KB3, it is possible to ensure **that all models constructed by users of these knowledge bases will be consistent** (including those with looped topologies).

These principles have been implemented notably in the knowledge bases used at EDF **for probabilistic safety studies of nuclear power plants.** More generally, **the vast majority of the reliability or availability studies carried out using the KB3 tool were done using knowledge bases that satisfied these principles.**

When it is impossible to apply them in full, as is the case for the highly complex TOPASE base (27,000 lines of order 1 Figaro), it is at least possible to make use of tools for analysing dependencies between rules, and which help to organise these into subsets that are easier to handle.

Finally, certain checks can be made on the models of particular systems if it is not feasible to do so on a knowledge base.

10. References

[1] Un moyen d'unifier diverses modélisations pour les études probabilistes : le langage Figaro

Bouissou M., Bouhadana H., Bannelier M.

EDF internal report HT- 53/90/42A

[2] Conception d'un ensemble d'outils pour la représentation, l'analyse et l'exploitation de la connaissance en sûreté de fonctionnement : l'atelier Figaro.

Bouhadana H.

Doctoral thesis - June 92

[3] Gestion de la complexité dans les études quantitatives de sûreté de fonctionnement de systèmes

Bouissou M.

Lavoisier, éditions TEC&DOC, October 2008

[4] Définition du langage Figaro0

Bouissou M., Dia M., Ladeuil V.

EDF internal report HT - 53/93/040

[5] Syntaxe du langage de modélisation stochastique Figaro

Bouissou M., Buffoni L., Houdebine J.C.

June 2016

[6] Figaro WORKSHOP – spécifications du sous-système Figaro1

Humbert S., Ladeuil V., Leroy F.

EDF internal report HT - 53/95/016A

[7] Réflexions sur la génération d'arbres de défaillance dans l'atelier Figaro

Bannelier M., Villatte N.

EDF internal report HT - 53/93/22A

[8] M. Bulot, I. Renault, Reliability studies for high voltage substations using a knowledge base: TOPASE project concepts and applications. EDF internal report 96NR00101, ISSN 1161-0581, 1996.

[9] J.F. Hery, J.C. Laleuf, Cohérence d'une base de connaissances : la convergence commutative en langage L.R.C. EDF internal report HT 14/22/85, February 1985.

[10] R. David, H. Alla, Du Grafcet aux Réseaux de Petri. 2nd edition, Hermes, 1992.

[11] J. L. Peterson, Petri Net Theory and the Modeling of Systems. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[12] Dutuit Y., Signoret J.P., Thomas P., Prise en compte des transitions dynamiques au sein des réseaux de Petri stochastiques. 20th Lambda-mu Conference, St Malo, 2016.

11. APPENDIX 1: Evolutionary changes in the Figaro language

This appendix presents all the changes that the language has undergone since 1995, the year in which the previous official version of the reference manual came out.

These changes are due in particular to the increase in the number of available programs; thus the two parts that follow relate to what was added for the Monte Carlo simulation and for the generation of fault trees.

11.1. Additions for the Monte Carlo simulation

During a Monte Carlo simulation it is possible to access the date (of the simulated history) on which the event occurred for each simulated event. Thus the keyword CURRENT_DATE was introduced; this behaves as a global variable that can be consulted in the rules. For the same reason it was possible to introduce the functions STATE_TIME() and INTEGRAL(), whose arguments are Boolean expressions.

Moreover, since each random variable is sampled in each history, we were able to add the function RAND (with no argument) which generates a random real number between 0 and 1 as well as functions providing random numbers drawn according to different probability distributions defined by their parameters (their names begin with RAND_). The main use of these distributions is the definition of random times before faults or repairs; however, they can also be used in rules.

Finally, the function ALREADY_REALIZED(), whose argument is a Boolean expression, allows a reliability calculation to be made without being obliged to make the failure states absorbing by declaring them as target states when configuring the YAMS tool.

11.2. Additions for generating fault trees

Reliability models associated with failures have been added so as to simply specify in a knowledge base one or more models (with a default model) that will be found in the basic events of the fault trees generated.

Before this addition, the tree generator had in some way to recognise from the occurrence rules whether a given failure was on start-up or in operation, whether it was repairable or not, etc. and it appeared rather hard to go beyond these few simple characteristics. The modeller now has at his disposal an entire catalogue of models that can be associated with failures.

This catalogue (see §6.2.4.4) was compiled from the models available in several fault tree processing tools, and could be extended if new possibilities were to appear in these tools.

11.1. Replacement of the pseudo-class GLOBAL by system objects

The existence of the pseudo-class GLOBAL was a "singularity" as far as the Figaro object-oriented philosophy was concerned. It met a need: that of providing access to "global" variables from all objects. This pseudo-class has now disappeared, but on the other hand it is possible to declare SYSTEM_OBJECTs in a knowledge base (there may be several). These objects exist in any system described with this knowledge base, and their variables are accessible from any other object in the model.

12. APPENDIX 2: reliability models that can be associated with a FAILURE

These reliability models are intended to be assigned to the basic events (the leaves) of a fault tree generated by a Figaro model. They represent small independent stochastic processes. The leaf of the fault tree is assumed to have the value TRUE at some given instant if the corresponding process is in a state of failure at that instant.

Each model is given with its parameters (which are all real), and if necessary a small state graph representing the behaviour it is modelling. The initial state is in yellow, and the failure states are in red.

MODEL_GLM

This model represents a repairable component that can fail at t=0 or during operation. This is by far the most used.

Parameters: **G**AMMA (failure rate on power-up), **L**AMBDA (failure rate during operation), **M**U (repair rate).

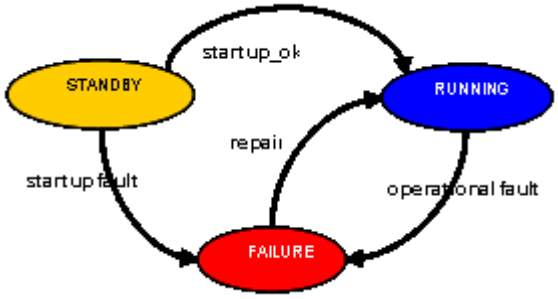

Figure 5. Graph of states of model MODEL_GLM

MODEL_GLTM

This model represents a non-repairable component that can fail at t=0 or during operation, over a time duration limited to TM.

Parameters: **G**AMMA (failure rate on start-up), **L**AMBDA (failure rate during operation), **TM** (mission time).

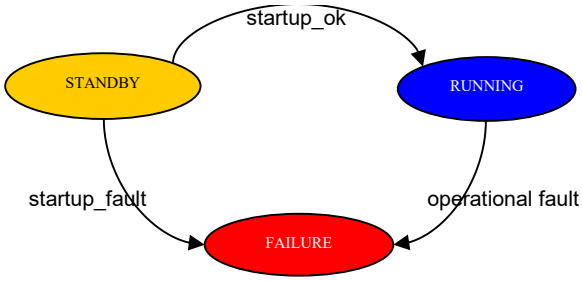

Figure 6. Graph of states of model MODEL_GLTM

MODEL_G

This model represents a non-repairable component that can fail at t=0. Depending on whether it has started or not, it remains running indefinitely (or respectively, failed).

Parameter: GAMMA (failure rate on startup).

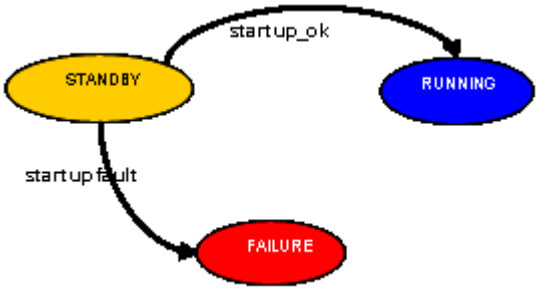

Figure 7. Graph of states of model MODEL_G

MODEL_FROZEN

This is not really a model: its purpose is to specify that the failure has a constant value, and hence keeps its initial value (which can be chosen by the knowledge base user) all the time.

Parameters: none

MODEL_F

This model, which has a highly specific use, represents a component associated with a constant fault **frequency**. This kind of model is used in probabilistic safety studies for nuclear power plants to represent initiator incidents.

Parameter: FREQUENCY (mathematically, this is the same as a failure rate).

MODEL_WB

This model represents a non-repairable component whose reliability is given by a **Weibull** distribution.

Parameters: ALPHA (scale parameter), BETA (shape parameter), T0 (offset).

The reliability of the component is given as a function of these three parameters by the formula:

$$\mathrm{R(t)} = \exp\left[-\left(\frac{t-T0}{ALPHA}\right)^{BETA}\right]$$

MODEL_RT

This model represents a repairable component that can fail at t=0 or during operation. The repair is not initiated as soon as the component enters the fail state: it is necessary to wait until a **random test**, which has a constant rate of occurrence, detects the fault. The advantage of modelling the test as random is to make the model Markovian.

Parameters: GAMMA (failure rate on start-up), LAMBDA (failure rate during operation), MU (repair rate), LAMBDA_TEST (rate of occurrence of tests).

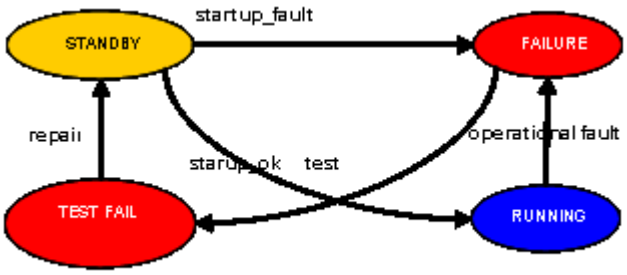

Figure 8. Graph of states of model MODEL_RT

MODEL_PET

This model represents a repairable component that can fail at t=0 or during operation. The repair is not initiated as soon as the component enters the fail state: it is necessary to wait until a **periodic test**, which takes place at fixed time intervals, detects the fault. This very complete model (so complete that it is virtually impossible to determine all its parameters!) takes account of the fact that the test is not infallible, and that it can even cause the failure of a component in good condition, etc.

Parameters: GAMMA (failure rate on start-up), LAMBDA (failure rate in operation), MU (rate of repair), T_INIT_TEST (instant of first test), T_INTER_TEST (time separating the starts of two tests), T_TEST (duration of a test), GAMMA_NO_DETECTION (probability that a test does not detect that a component has failed), GAMMA_TEST (probability that a test causes the failure of a component that was working properly), GAMMA_RECONFIG (probability of poor reconfiguration at the end of a repair).

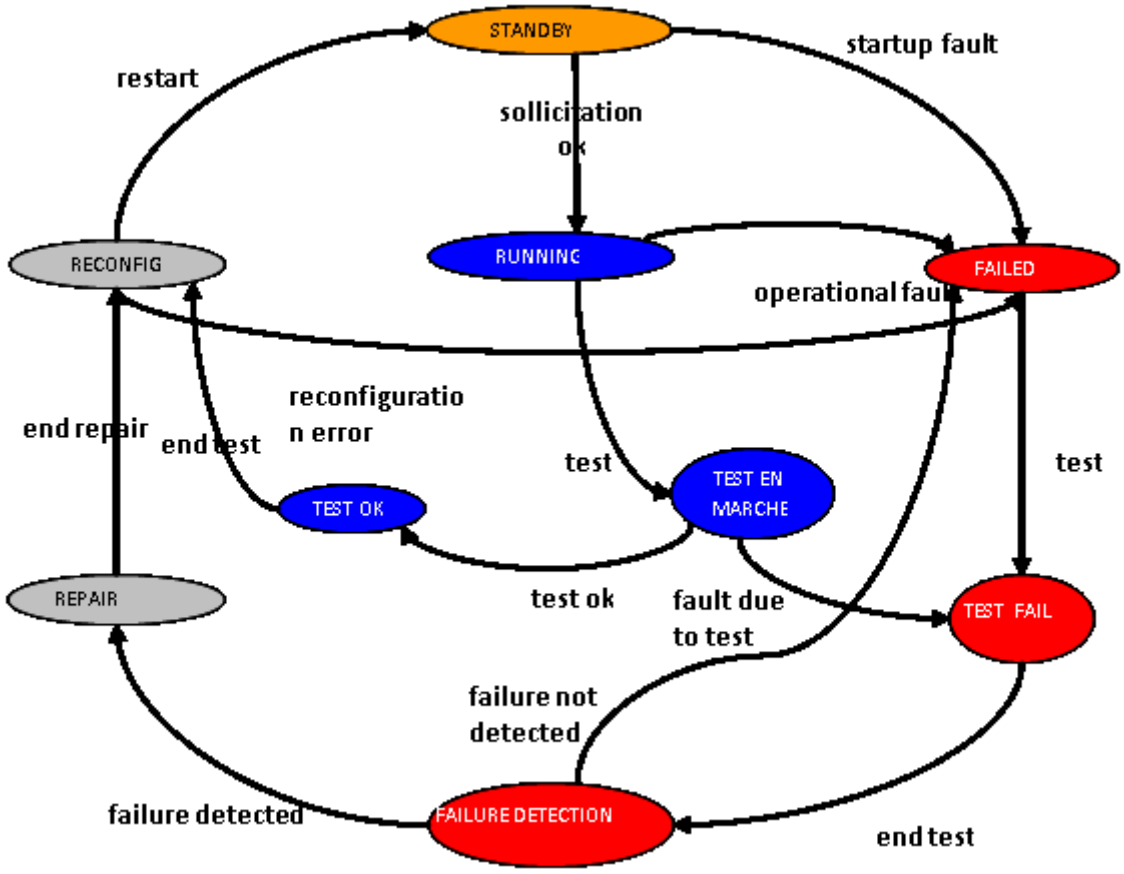

Figure 9. Graph of states of model MODEL_PET

13. APPENDIX 3: INDEX

K

L

M

N

O

P

R

S

T