INCONSISTENCY DETECTION IN KB3 MODELS

Marc Bouissou
EDF R&D
1, avenue du Général De Gaulle
92141 Clamart Cedex FRANCE
Marc.Bouissou@edf.fr
++33 1 4765 5507

Jean-Christophe Houdebine
Ariste
25-27, avenue de la Division Leclerc
92160 Antony FRANCE
Ariste@wanadoo.fr
++33 1 4674 5410

Summary

KB3 is a tool based on the Figaro object-oriented modelling language. Thanks to the definition of knowledge bases written in this language, KB3 enables a quick graphical input of system models, from which the tool is able to automatically generate reliability models such as fault trees and Markov graphs. Since the Figaro language allows the construction of models showing a very complex dynamic behaviour, it is important to formally define the semantics of this language, and to address the inconsistency problems that may be encountered in the construction of knowledge bases. In this article, we give a formal definition of the order-0 Figaro language, a simple Figaro sub-language which is the basis of all models processing. We give a typology of inconsistencies that can appear in a model, and show how some of them are properly detected by syntactical checks, and others can be eliminated through the use of some simple rules for building a knowledge base.

INTRODUCTION

EDF R&D developed KB3 for automatic generation of reliability models such as fault trees and Markov graphs. KB3 uses the Figaro modelling language [1], [2] created especially to help capitalise on knowledge on categories of systems with common operational characteristics and failure modes. Using knowledge bases written in this language, it is possible to very quickly enter a graphic description of a system, and to have KB3 automatically generate conventional reliability models.

Certain KB3 knowledge bases are extremely complex: the most complex, that of the Topase tool dedicated to the study of VHV substations [4], has 27,000 lines of Figaro language. Therefore, it is felt to be increasingly necessary to ensure that they do indeed what the author intended, or to at least design a tool that will **warn the designer that independently of its intended objectives, his knowledge base has undesirable properties that we refer to as «inconsistencies»**. These undesirable properties can be of several kinds: contradictions, unbounded numerical values, looped definitions (A defined in terms of B, and vice versa) that cannot be solved, divisions by zero, underflow or overflow in the calculation of real numbers, etc.

The objective of this article is to describe the principles of methods for either ensuring the consistency of knowledge bases as they are built, or for detecting any inconsistency they might contain. As a rule, i.e. for knowledge bases not built in the manner recommended here, it is impossible to demonstrate consistency. This is hardly surprising given that the Figaro language combines all the possibilities of the logic of proposals, of calculation with real numbers, and of automata.

When it is not possible to prove the consistency of a knowledge base (kb), it may still be possible to process system models case by case. **This makes it possible to use imperfect knowledge bases,** by checking that the models built for each system do not involve the defects of the base. A typical example is that of knowledge bases that ensure the consistency of models only if the topology of the systems studied is not looped.

The Figaro language has two levels: "order 1" and "order 0". The order-0 language is a very simple sub-language of the order-1 language and can only be used for the description of specific systems. It is the order-1 language that is used in knowledge bases, for it serves to write generic models of components which will be instantiated on the system architectures entered by the user.
Article [3], published in the Proceedings of the same Conference as this paper, presents the syntax of the Figaro language (with both levels) and some examples of models.

The present article is broken down as follows: it starts by exploring the relationships between order-1 and order-0 consistency. The relationships determined serve to limit demonstration work at order 0. There follows a formal definition of the order-0 Figaro language which is used to draw up a typology of the kinds of inconsistency an order-0 Figaro model might contain. A set of rules for building a knowledge base so as to avoid the identified kinds of inconsistency is then given. For the most difficult of these rules —the arrangement of «steps» in the model— we outline a tool that helps the user do just this. Towards the end, the paper looks at cases of demonstrations of consistency which can be carried out only for the order-0 language, i.e. case by case on system models entered by the user.

RELATIONSHIP BETWEEN ORDER-1 AND ORDER-0 INCONSISTENCY

Demonstrating (in so far as it is actually possible) the absence of inconsistency in a knowledge base involves showing that:
irrespective of the system (complying with the constraints described in the kb) entered by the user, the model in order-0 Figaro language generated with the system and the knowledge base will contain no inconsistencies.

Inconsistency can be found either at the instantiation phase (transformation of the order-1 model into an order-0 model) or in the phase when the model is run (when it is used to simulate the system modelled). Any instantiation problems can be detected **exhaustively** by the syntactical checks described below which are carried out on the **knowledge base**.

Syntactical checks

A knowledge base is a collection of **classes** of objects described in **order-1** Figaro language (i.e. serving to describe generic knowledge, particularly by means of operators suitable for manipulation of **sets**) that the kb user can assemble in an infinite number of ways within system models.
This freedom is nevertheless restricted by the data given in the KIND and CARDINALITY facets of the interfaces declared in the knowledge base (e.g. upline of an electrical component there can be only more electrical components, in numbers ranging from 0 to N).
With this information, the syntactical knowledge-base check carried out by KB3 ensures that no problem of the type «access to non-existent variable requested» (e.g. the pressure of an

electrical component) can be detected during the instantiation of a model.
This syntactical check is very complex for it must take account of all the inter-class inheritance rules, and it is of great help for knowledge-base designers.

Before any model entered by a user can be put to use, KB3 checks that all the constraints defined by the knowledge-base designer are complied with.
This check can generate error messages of the type «not enough (too many) objects in interface X of object Y".
Compliance with the classes of objects allowed in a given interface of a given object is checked by the entry interface; this interface will allow no errors of this kind.

These checks ensure that the procedure of order-0 instantiation for a given system, which is systematically carried out prior to any reliability processing, will be trouble-free.
But that is not enough in itself, of course: this check provides no protection whatsoever against the types of inconsistency described in the introduction to this communication.

Behavioural consistency

To precisely define what is meant by 'behavioural consistency', a formal definition of the semantics of the model specifying behaviour must first be obtained.
The order-1 Figaro language is complex, and it is therefore difficult to give a formal definition, for it would be unreasonably lengthy and quite unintelligible. On the other hand, it is possible to give a simple formal definition of order-0 Figaro language.
Consequently, it is on the basis of such a simple description that we will show where inconsistency can appear. It will then be possible to state the rules for knowledge-base construction that will ensure that any model in order-0 Figaro developed from that knowledge base is fully coherent.

SEMANTICS OF A FIGARO 0 MODEL

Foreword: in order to keep this presentation simple, the description will not include the possibility given by Figaro to define Dirac-type probabilistic distributions that can be used to model deterministic triggering of events at the end of determined periods. Nor will we go into details of the temporal behaviour of the automaton: we will restrict ourselves to defining the states to which the automaton can evolve from a given state, and whether the change in state is instantaneous or occurs after a certain time. These simplifications mean it will not be necessary to define a schedule, a relatively complex notion that would only make things more difficult to understand.
The distinction between instantaneous and non-instantaneous or «tangible» states is a **necessity** that gives the language sufficient modelling power: these two types of state are not processed in the same way, as will be seen in the paragraph defining how the automaton can evolve from a given state.

Consider *L*, a set of probability distributions belonging to one of the following two categories: continuous distributions covering $[0,+\infty]$, discrete distributions associating probabilities with a finite number of modalities.
A model in Figaro 0 language is a tuple *(Ξ, O, T, I, σ, Y_0, V_0)* made up 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 instant *t*. Each state variable x_i can be one of several types: Boolean, integer, real number, or a so-called «enumerated» variable which can take values in a finite set E_i. *X* is in fact the concatenation *V*, *Y* of two vectors, *V* and *Y*. *V* covers the so-called **«essential»** variables and Y covers the so-called **«deduced»** variables.
V_0 is the initial value of *V*, and Y_0 is the so-called «re-initialisation» value of *Y*. ***Y* is a function of *V* and of Y_0 defined using function *I* described below**.
T is the set of **transition groups** of the model. A **transition** is a mapping of Ξ into Ξ, which, for any state vector, provides a correspondence with another state vector, generally obtained by modifying a small number of essential variables. A transition group is a couple consisting of a set of so-called «bound» transitions and one of the probability distributions of *L*. The transition group contains a single transition if its distribution is *not* of the discrete type. If the distribution *is* of the discrete type, the transition group contains as many transitions as the distribution has modalities. A transition represents a local change in the state of a component in the system modelled, e.g. occurrence of a failure mode. *A transition group associated with a discrete distribution models all the possible outcomes of a non-deterministic process considered to be instantaneous;* for example, the various possible results of throwing dice, or the choice between component startup and failure to start up. Through improper use of language, we do not distinguish the notions of «transition group» and «transition» for groups containing a single transition. The notion of transition group is an original feature of the Figaro language not found in all the formal means of describing non-deterministic automata, starting with Petri nets. Its importance will be highlighted below, when we describe how the automaton can evolve from a state for which *several groups of instantaneous transitions* can be applied.

O is a mapping of Ξ into $\mathcal{P}(T)$. In practice, *O* is defined by the set of so-called **«occurrence» rules**. These rules are used to associate a set (possibly an empty set) of transition groups to any state *X* of the model. The transitions belonging to the groups of *O(X)* are said to be **enabled** in state *X*.

I is a mapping of Ξ into Ξ , which for any state vector *X* **for which *I* is defined** gives a corresponding state vector. Function *I* is in practice defined by the **so-called «interaction» rules** of the Figaro model (and possibly a **system of linear equations**), and this function **may depend on the order, σ, of the rules**.

Function *I* is defined as the composite function of a finite set of functions of Ξ into Ξ, designated $I_0, I_1,…I_P$. In other words, $I(X) = I_p(I_{p-1}(…I_0(X)…))$.
The interaction rules of the model are grouped together in so-called «**steps**» corresponding to the different functions $I_1,…I_P$. As for I_0, it is a special re-initialisation function of the deduced variables: $I_0(V, Y) = V, Y_0$

Inference performed in a step
A **deterministic automaton** that operates in the same way as a simple inference engine defines the function corresponding to each step. {R_k} is the set of rules for a given step (to simplify notation, we will leave out the step subscript). Rule R_k is a mapping of Ξ into Ξ , which, for a given state vector *X* gives the corresponding state vector $R_k(X)$.
Each rule in practice consists of a **condition** calculated by a Boolean function of state X, which triggers it, and of **actions**, which are most commonly instructions regarding assignment of state variables with constant values or values calculated from X.
There is also another type of possible action, for the moment used only in the Topase knowledge base: solving a system of linear equations, which can be used to calculate a new value for X.
The inference engine applies rules (R_k) in cycles *arranged in order* σ until the same value is obtained for X at the end of two successive cycles of the rules.

Applying a rule consists in assessing its condition, and if its condition is true, carrying out the corresponding action(s). In particular, if the condition is not true, vector X is not modified.
This type of operation of interaction rules provides for substantial flexibility in modelling, particularly for anything involving propagation of flows in a system. In a preliminary version of the Topase tool, it even made it possible to calculate currents and voltages at any point in an electrical circuit, using Ohm laws, before the same calculation could be carried out to a higher performance level by solving a system of linear equations.

Once 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 are equivalent to the operation of the following **non-deterministic** automaton:

Initialisation:

The user defines an initial **incomplete** state with the data of V_0 (just one constraint: compliance with the domain of V). Then, $X_0 = I(V_0, Y_0)$, the complete initial state of the model, is calculated. It should be noted that this way of defining the initial state of the system makes it possible to solve a certain number of tricky problems for a complex model, such as the risk of defining an incoherent state (e.g. with two electrical components connected in series, where there is no current in one but current in the other), or the impossibility for the user to define the initial state (this would be the case, for example, if he had to enter the currents and voltages at every point in an electrical circuit). Thus, it is easy for the user to define the initial state, and the results are reliable: there are few variables to initialise, and their meaning is clear and easy to determine (like the position of a valve, the number of components started up at the initial instant, etc.).

Evolution from a given state *X*:

O(X) is the set of transition groups that are enabled in state *X*.

- If this set is empty, state *X* is absorbing.
- If it is reduced to a group containing a single transition *t*, the only state the automaton can evolve to from state *X* is *I(t(X))*.
- If it contains several transition groups, there are two possible cases (the following breakdown constitutes a grouping into subsets of all the possible situations):
  - *O(X)* contains **only** transitions of continuous distributions. In this case, state *X* is said to be «tangible» and the next state for the automaton will be *I(t(X)), t* being any one of the transitions of *O(X)* (non-determinism).
  - *O(X)* contains groups $G_1, G_2.., G_n$ of discrete-law transitions. State *X* is then said to be "instantaneous", and the transitions of continuous distributions are **overlooked**. In this case, the next state for the automaton will be $I(t_1(t_2 \dots (t_k (X))\dots))$, where t_i represents any choice (non-determinism) of transition in group G_i: $t_i \in G_i$. The order of application of the transitions (in other words, the choice of subscripts for the groups) is not specified. This can produce consistency problems which are discussed below.

<u>TYPOLOGY OF POSSIBLE INCONSISTENCIES</u>

Once this formalisation of the operation of the "Figaro 0 automaton" has been accepted, it is possible to define what is meant by inconsistency of a model. Desirable properties are also defined.

The order-0 Figaro language can be considered as a sort of synthesis of the existing concepts in production-rule-based artificial-intelligence languages and in stochastic Petri nets. It is therefore natural to take inspiration from the work done in these two fields to develop methods for consistency checking in the Figaro language.

For Petri nets, the properties that can <u>possibly</u> be established from a net are: the possibility of returning to the initial state from any other state, the bounded character of marking, the liveness of transitions (a given transition T is said to be live if, from any state that can be reached from the initial state, it is still possible to find a transition firing sequence including transition T), the absence of an absorbing state, the existence of invariants (linear combinations of the markings of places that are constant, irrespective of the evolution of the network) [6], [7].

These properties of Petri nets are very easily transposed under Figaro, except those that refer to invariants. They become the following properties:

- P1: finite character of the state space,
- P2: liveness of transitions (with the same definition as for Petri nets),
- P3: non-existence of absorbing states,
- P4: possibility of returning to the initial state from any other state.

It is important to note that the absence of numerical values in *V* guarantees that the set of states that can really be reached by the model, which is part of Ξ, is **finite** (the fact that there are numerical **deduced** variables does not compromise this property).

The last two properties are obviously to be sought only when one wants to model a repairable system, for example in order to calculate its asymptotic availability.

In artificial intelligence, three criteria are generally used to validate a rule base, irrespective of the domain of the base:

- Consistency in the logical sense: you cannot have something and its opposite at the same time,
- Completeness: there is a solution to any problem, irrespective of the initial data,
- Pertinence: the base conforms to physical reality (this point is beyond the scope of this communication).

Transposition of the notions of consistency and completeness to a Figaro 0 model requires a little interpretation. In fact, the following undesirable behaviours can be encountered in the operation of a Figaro 0 defined automaton:

- Inc1: Impossibility of calculating $I(V_0, Y_0)$, or even *I(X), X* being the state reached by applying one or more transitions to some previous state. There is also the case where calculation is possible but its result depends on the order of the rules. These problems of completeness and consistency of a rule base were studied by Electricité de France (EDF) in 1985. The rules for reliable construction of a kb given in this paper are deduced from the theorems demonstrated in [5].

- Inc2: Inconsistency linked to transitions groups enabled **at the same time** in a given *instantaneous* state *X*. The «physical» meaning of this situation is the fact that several actions of no duration (but with random results) are triggered in parallel and must therefore produce their effects at exactly the same time. Typically, this can concern requests for simultaneous startup of several components. Due to the notion of transition groups, there is no need to explore all possible orders of application of transitions, but it generally presupposes that the transition groups are all independent of each other. Independence here means two things:

- application of a transition of a given group does not call into question the conditions triggering **other** transition groups (on the other hand, it is normal for it to make its own condition false),
- irrespective of the order of application of transitions, for a given combination of transitions taken each from a different group, the same result is obtained for $t_1(t_2 \dots (t_k (X))\dots)$.

One or another of these conditions may possibly not be met, for special reasons in certain models, but it has to be a deliberate choice by the user. This is why any tool using the Figaro language must inform the user of any non-compliance with one of these conditions in order to warn him against a possible error due to inadvertence.

The transition group notion is a great help in modelling because it allows to handle very neatly the following kind of situation: in a system having n independent components that are required to start up at the same instant, it is useless to explore 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 automaton is able to generate directly these 2^n outcomes of the initial instantaneous state. The example of knowledge base given in [3] is an application of this ability of the Figaro language.

- Inc3: To these local inconsistencies can be added a more global kind of undesirable behaviour, namely infinite linking of a series of instantaneous transitions, which is possible if a series of transitions leads back to a state from which the series can be triggered again. For the moment (to our knowledge), there is no method for detecting this kind of behaviour before running the model.

To complement the above comments, it is probably worth remarking that it would be **absurd** to try to demonstrate properties of consistency combining interaction and occurrence rules: for a repairable system, it is obviously wholly desirable for there to be rules that assign TRUE to a failure and, as soon as that is done, other rules that will assign FALSE to the same failure. It is simply a matter of ensuring that these operations cannot all take place in no time, which would result in a situation of the type described in the preceding paragraph.

We are now going to detail a number of methods for writing knowledge bases which, **right at the construction step**, make it possible to avoid inconsistencies Inc1 to Inc3 and/or to ensure that properties P1 to P4 are obtained.

GRAPH OF DEPENDENCIES BETWEEN VARIABLES

Some of the methods that will be referred to are based on the simple concept of the graph of dependencies between variables. Since this notion is important and has interesting applications beyond the field of consistency checks, this article devotes some time to it.
In a Figaro 0 model, it is said that variable v1 "directly affects variable v2" in the following two situations:

- When there is an assignment instruction (<-) where v2 appears on the left and v1 on the right (v2 <- v1, for instance),
- When there is an interaction or occurrence rule in which v1 is involved in the premise and v2 appears in the left-hand part of an assignment in the conclusion.

The best way to represent a set of dependency relations between variables is to use an oriented graph: in a graph, the existence of a path between two nodes associated to variables represents an indirect influence.
When variables are considered from a probabilistic point of view, they are identified to continuous-time stochastic processes; one can then talk of stochastic dependence between those processes. It can be demonstrated (but it is beyond the scope of this article) that stochastic independence between two process variables is guaranteed from the moment there is no path between the two variables in the graph we have just defined.

This property is extremely useful, for it makes it possible to break the global model down into sub-models which can be solved independently. From a *qualitative* point of view (for model-checking, for instance), it means that construction of a global graph of model states can be replaced by a series of much smaller independent graphs. From a *quantitative* point of view it simplifies calculation of the probability of being in a given state for a state variable by considering only the sub-model containing the variables that influence the variable concerned.

Consequently, if one wants to calculate the probabilities of being in a given state for a state variable, or of being in certain states for a **group** of variables, one can start by **restricting the model to the variables that influence this group of variables**. The sub-graph containing only the variables that directly or indirectly influence the group of variables concerned can be used to define the minimum sub-model to be solved for that calculation.
In practice, extracting such a sub-model amounts to eliminating from the global model any expression containing a reference to a variable that will disappear. Thus, the rules:
IF v1 OR v2 THEN v3 ELSE v4;
IF (v4 AND v5) OR NOT v3 THEN v6;

will become: IF NOT(v1 OR v2) THEN v4 whenever v4 alone is of interest.

In addition, it is interesting to distinguish another type of subsystem: **closely connected components of the graph** (ccc), for they are sub-models that cannot be divided for the purposes of resolution, for all the variables of a ccc are interdependent.
From this can be built a "supergraph" whose nodes each correspond to a ccc. The supergraph necessarily contains no circuit (if it had one, there would be a bigger ccc than those identified). Subsequently a resolution technique that starts with the sources of the supergraph and works towards the ccc of interest could be used.

RELIABLE METHODS OF WRITING KNOWLEDGE BASES

Have a pyramidal dependency graph

A very reliable method of building a kb is to give the graph of dependencies between variables a pyramid structure, with a «supergraph» whose ccc with no incoming arc contain all the essential variables. Ideally, there should be the smallest possible number of variables in each root ccc.

The example of a knowledge base that can be used to describe a telecommunication network given in [3] is typical. Here we have a simple three-node network (to simplify things further, it is assumed the edges have no faults) followed by the graph of dependencies between variables for this system; in the graph, the variables that are part of the same ccc are boxed in:

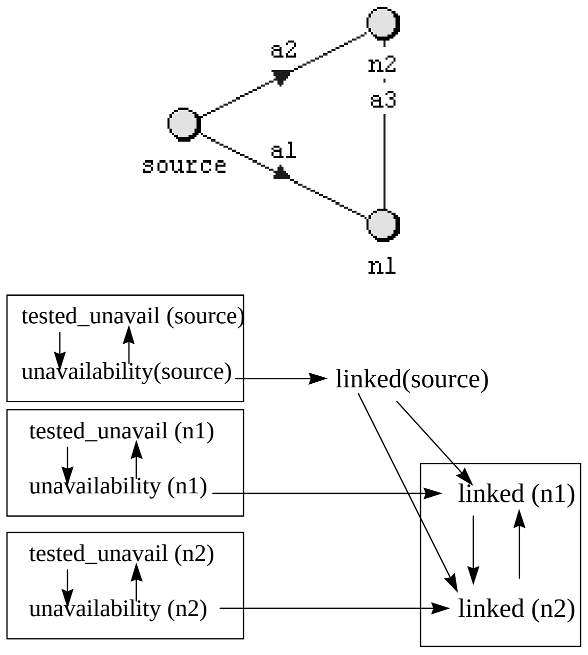

Figure 1: a network and corresponding dependency graph

Most of the knowledge bases developed to date generate pyramidal dependency graphs. This is especially the case of knowledge bases whose purpose is to produce fault trees. Most of the essential variables in them are failures, and failures of different objects are independent of each other.

A simple and natural way of getting a pyramidal graph is to create dependencies between essential variables only within classes (i.e. within objects after instantiation on a particular system) and to create no dependency link from deduced variables to essential variables.

By giving a knowledge base this kind of architecture, it is possible to demonstrate the absence of inconsistencies of types Inc2 and Inc3, and the presence of properties P1 to P4 (subject to there being no type Inc1 inconsistency) by means of very local reasoning involving just a few occurrence rules of the same class. This can be demonstrated at the level of the knowledge base, and consequently **for any system modelled with the knowledge base**.

Reason by monotonic inference

The purpose of the construction rule to be described next is to avoid inconsistencies of type Inc1.
In reference [5], Hery and Laleuf demonstrated a theorem giving a sufficient condition for commutative convergence of an inference carried out like the inference of a step of interaction rules in Figaro 0 language. An immediate consequence of this theorem is the following **corollary**:

If

- **all** the actions of rules assign the value TRUE to Boolean variables that were initialised to FALSE before inference (generally these are EFFECTs), and
- **all** the rule premises using these Boolean variables are tests of the type "a_variable = TRUE",

the commutative convergence of the inference is guaranteed.

This framework is very simple, and is used in most knowledge bases aimed at producing fault trees. It is referred to as the «**monotonic inference**», a name derived from the fact that when a Boolean variable has been set to TRUE, no rule can change this conclusion. The second condition of the theorem is there to ensure that no Boolean variable is set to TRUE without due consideration, before stabilisation of those it depends on.
The enormous advantage of this theorem is that it is extremely easy to check (even without tools) its assumptions at the level of the knowledge base, as can be seen in the example of modelling of a telecommunication network given in [3]. It is therefore possible to prove **at the knowledge base level that irrespective of the assembly of classes performed by a user of the knowledge base, no type Inc1 inconsistency can be created**. This very interesting property has been used in all the operational knowledge bases used by EDF for safety analysis of nuclear power plants.

Thanks to monotonic inference it is possible to **model propagation of flows in looped systems very concisely, naturally, and without any risk of inconsistency**, as can be seen in the example developed in [3]. In the majority of conventional formalisms (fault trees, Boolean logic, Petri nets, model-checking formalisms, etc.), however, this modelling problem results either in impossibility or in very cumbersome and illegible expressions, whereas it is constantly encountered in the study of real systems.

Use steps to best advantage

To avoid type Inc1 inconsistencies in situations where monotonic inference is too restrictive a framework, steps can be used.
The theorem on commutative convergence in [5], which is more general than the restricted application given in the preceding paragraph, shows the importance of having as many so-called «non-receptive» state variables as possible at inference level. These are state variables which are modified by absolutely no action of the rules.
An efficient means of reducing the number of receptive variables is of course to reduce the number of rules to be taken into account. This is precisely what is achieved by breaking things down into steps. Once the inference corresponding to a given step is completed, the variables which were receptive for that step can (possibly) be non-receptive for the following steps.
Here is a very simple example of how steps are applied: suppose that you need to write a rule of the type «IF NOT effect1 THEN effect2". Because of the negation in the condition, this is beyond the scope of monotonic inference, **unless** you put the rules acting on effect1 in a step prior to the step in which the above rule is applied.

AUTOMATIC ORDERING OF INTERACTION RULES

When writing interaction rules, it may be difficult to systematically implement a breakdown of these rules into steps that ensure the whole model will work smoothly.
In general, the knowledge-base designer can easily set up local sequences of rules or groups of rules, but it is more difficult to manage all the rules, and even more difficult to add a new rule to an existing knowledge base.

To reduce these difficulties, an automatic sequencing tool has been developed. It assumes that a rule should be used only when its premises have been stabilised.

In this framework, the execution order of the rules can be deduced directly from the graph of dependencies between rules (presented below), a concept similar to that of the graph of dependencies between variables.

Definition of interdependencies between rules

It is stated that a rule R2 depends on a rule R1 if there is a variable V, modified by R1, which affects the premises or terms of calculation of R2.
In the case of a Figaro 0 model, this definition involves no difficulty, but the number of rules and variables considered can be very high, and, given the fact that optimisation has to be recalculated for each modification of the model (addition or elimination of an object), the calculation is often too complicated to be performed again in real time.
The definition has therefore been adapted to the knowledge bases, taking account of the possibilities of inheritance between different classes.
Elements of different classes are considered to be distinct, even if they are simply built by parent inheritance: variable V of class T is considered to be different to variable V of a child of T. It is then stated that the dependence of a rule R2 on a rule R1 is governed by the variable V of class T in the following case:

- variable V of class T1, a parent of T, is modified by R1,
- variable V of class T2, a parent of T, is involved in the premises or terms of calculation of R2.

Classes T, T1, and T2 can be distinct, or coincide.

Ordering of rules

Once the first interdependencies have been built, the dependencies must be complemented in order to build a partial order relationship, i.e. a transitive and antisymmetrical relationship.
Transitivity is obtained by adding the necessary dependencies: if R3 depends on R2 which depends on R1, a dependency between R1 and R3 is added.
When the added interdependence relationships ensure the transitivity of the whole, its antisymmetry must be ascertained: if R1 depends on R2 and R2 depends on R1, then R1 = R2. It is obvious that rules cannot simply be merged; antisymmetry is achieved by grouping rules in steps. The order relationship is no longer expressed between rules but between steps: a step E1 depends on another step E2 if one of the rules of E1 depends on a rule of E2. Rules are grouped until the relationship between steps is antisymmetrical.
Grouping of rules violating the constraint of antisymmetry in a step can be explained by the fact that these rules depend on each other. It is therefore necessary to group them together and to run them together until the result converges. In the extreme case of a set of entirely interdependent rules, the algorithm results in creation of a single step containing all the rules.
Once the relationship of partial order between steps has been established, it can be used directly to determine the order of execution of steps; independent steps are run first, followed by the steps depending only on those already taken into account, and so on until all the steps have been run.

Using the rule-sequencing tool

The sequencing tool can be used either when creating a new knowledge base or when modifying an existing one.
For a new knowledge base, the algorithm defines the groups of interdependent rules and thus enables the kb designer to check the pertinence of those 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 obligatory sequences such as the fact that rule group G1 has to be applied before group G2, and the algorithm completes the order by signalling if the predefined sequences cannot be complied with (if group G1 depends on group G2, for instance).
When the designer has already broken operations down into steps, it is also possible to subdivide each step to get the optimum order of execution of rules and to accelerate the inference process.
The principle of modifying a knowledge base involves adding new rules outside the steps defined in the knowledge base and letting the algorithm place the new rules correctly in new steps or existing steps. The algorithm gives a warning when the new rules introduce looped dependencies between the existing steps.

DETECTION OF INCONSISTENCIES IN A FIGARO 0 MODEL

Despite everything outlined above, there are of course cases where nothing can be demonstrated at the level of the knowledge base since the reliable construction rules are too restrictive to allow modelling of certain types of systems. It is still possible to run checks on a given order-0 Figaro model.
Take, for example, the following knowledge base which determines which nodes are linked to sources in a network of given topology (NB: this knowledge base is completely deterministic — the automaton can take only 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  SORT_OF component ;
INTERFACE upline KIND component;
INTERACTION
   IF THERE_EXISTS x AN upline
         SUCH_THAT linked OF x THEN  linked
         ELSE NOT linked;

CLASS source SORT_OF component ;
  INTERACTION
   THEN linked ;
```

The state variable "linked" is an ATTRIBUTE, not an EFFECT. It is therefore not reinitialised every time the interaction rules are run. In a looped topology containing two nodes n1 and n2, where n1 has n2 in its upline interface, and vice versa, after instantiation the rules contained in the two nodes are the two assignments linked (n1) ← linked(n2) and linked(n2) ← linked(n1).
Thus, it can be seen that depending on the initial values chosen by the user (it is assumed he can be wrong) for linked(n1) and linked(n2), two states are stable after application of the interaction rules: that where n1 and n2 are linked, and that where they are not. In fact, only the second case has a physical significance, given the absence of a source in the system.

Despite this defect (which could be corrected by simply declaring "linked" as an EFFECT), this knowledge base can be used in unlooped topologies; in an unlooped topology, the graph of dependencies between the different "linked" attributes of the system is acyclic. The inference will gradually stabilise the values of different «linked» attributes, working from the sources or root nodes of the system topology and setting the «linked» attribute to TRUE for the sources and FALSE for the root nodes, irrespective of the initial state declared by the user.

More generally, it often happens that when using "equivalence" rules (IF ... THEN ... ELSE) one has to restrict the use of the knowledge base to systems whose topology is not looped.

It is therefore far preferable to use monotonic inference whenever possible, despite the fact that it can make for slightly less legible knowledge bases, since with it the different conditions that mean a variable must be set to TRUE can be spread across different rules.

Another example of inconsistency detection that is only possible at the level of Figaro 0, and even then only while the model is running, is detection of type Inc2 inconsistencies.

This type of inconsistency can be easily illustrated by a Petri net translated into Figaro. Examination of the model would detect what are called **structural conflicts** in the network due to the fact that a place P is an input place of two instantaneous transitions t1 and t2 (cf. [6] or [7]).
But there can be many of these in a model without it being a problem. It is only when the model is run that some of these structural conflicts become **effective conflicts** (cf. [6] or [7]). This is the case in the example given above, when firing of t1 removes the tokens used to enable t2, and/or vice versa.

The Figseq tool, which is used to analyse dynamic Figaro models, detects these conflicts, warns the user that they are there, and displays the sequence of events that brought the system from its initial state to that in which the conflict is effective.

CONCLUSION

We have shown that the Figaro modelling language on which the KB3 tool for automation of dependability studies is based answers some legitimate questions regarding the consistency of systems models of increasing complexity.

In particular, by complying with simple rules for building the knowledge bases used with KB3, it can be ascertained that **all the models built by the users of these bases will be consistent** (including those with looped topologies).

In particular, these principles have been put into practice in the knowledge bases used by EDF **for probabilistic safety analysis of nuclear power plants**. More generally, **most** of the reliability and availability studies that have been carried out with KB3 have involved knowledge bases applying these principles.

When it is impossible to apply them wholly, as is the case for the very complex Topase knowledge base (27,000 lines of order-1 Figaro language), it is at least possible to use tools for analysing dependencies between rules which help organise rules as more easily controllable sub-sets.

And some checks can be carried out on models of particular systems even if they cannot be carried out at the level of an entire knowledge base.

REFERENCES

[1] M. BOUISSOU, N. VILLATTE, H. BOUHADANA, M. BANNELIER
"Knowledge Modeling and Reliability Processing: Presentation of the Figaro Language and Associated Tools", SAFECOMP’91, Trondheim, October 1991.

[2] M. PILLIERE, P.MOUTTAPA, N. VILLATTE, I. RENAULT
"KB3 Computer Program for Automatic Generation of Fault Trees"
RAMS’99, Washington (U.S.), January 1999.

[3] S. MUFFAT, M. BOUISSOU, S. HUMBERT, N. VILLATTE
"KB3 tool: Feedback on Knowledge Bases"
ESREL 2002, March 2002.

[4] 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.

[5] J.-F. HERY, J.-C. LALEUF
"Cohérence d' unebase de connaissances: la convergence commutative en langage L.R.C."
EDF internal report HT 14/22/85, February 1985.

[6] R. DAVID, H. ALLA
"Du Grafcet aux Réseaux de Petri"
2nd edition, Hermes, 1992.

[7] J. L. PETERSON
"Petri Net Theory and the Modelling of Systems"
Prentice-Hall, Englewood Cliffs, New Jersey, 1981.