Dysfunctional Analysis and Simulation

Implementation Guide

BPA Dysfunctional Analysis and Simulation (SD9)
BPA Delivery 7 for V5R19 (V5.7)

DASSAULT SYSTEMES

Version 1.4

Modification tracking

Version	Who	Date	Added Modification
Version 1	RUG	20/04/06	Creation
Version 1.1	RUG	15/02/07	Modification of the treatment process considering the new generator functionalities
Version 1.2	PTG	20/08/08	Page layout and design
Version 1.3	RUG	27/08/08	Screenshots and document update
Version 1.4	RUG	26/05/09	Document update

Table of Content

Acronyms

DM	*Degraded Mode*
FE	*Feared Event*
FMEA	*Failure Modes and Effects Analysis*
FT	*Fault Tree*
PHI	*Preliminary Hazard Identification*
UE	*Unexpected Event*
V&V	*Verification & Validation*

Introduction

To meet always increasing safety requirements in industry, design and safety assessment methods are developed in order to fit the complexity of new complex systems. They incorporate heterogeneous components, perform a large number of functions, and interact with operators through advanced interfaces. As a consequence, it is becoming harder to manage all the aspects of safety assessment and to maintain the safety levels required by societal needs.

The unavoidable increase in the complexity of today's systems leads to an increasing difficulty in the comprehension of their functionality and in managing all the aspects for the safety analysis purposes. This implies that there must be a suitable increase in the capability of the safety engineers to maintain safety levels.

The goals of the BPA SD9 consist in the improvement and integration of safety activities of these systems.

The introduction of BPA SD9, for supporting the design and safety process, gives the designer a way to perform the validation/verification-analysis; at the same time it provides the safety engineer with a potentially new means for performing his work more effectively during the design task.

Values proposition

From a process point of view, this BPA highlights:

- How to integrate safety assessment in the design process
- How to construct an AltaRica model from functional and safety specifications in DAS environment,
- How to utilize this model to help designer obtaining an optimal architecture from safety aspects and to validate safety requirements.

Indeed, several approaches are available to generate safety analysis by automatic processes and to validate safety protections:

- **fault tree generation**: fault tree models are generated in ARALIA format so as to do qualitative and quantitative assessments for each identified feared state; this functionality is well fitted to systems having a static behaviour.
- **sequence generation**: for system having dynamic features, it is necessary to use a suitable method which can handle with strongly time-dependant behaviour like the sequence generator; it gives the whole scenarios composed with ordered events (failures, reparations, reconfigurations,…) leading to an unexpected state.
- **FMEA**: it is also possible to generate automatically a FMEA of the system from an AltaRica model.
- **Step-by_step simulation**: this process is useful to validate the functional behaviour and the safety protections as software/hardware reconfigurations, degraded modes, circuit breakers, fuses… The validation can be done graphically with variation of icon and connection colors, and all variable changes can be supervised by mean of a global panel.

In addition to these processes, a fundamental advantage to work in DAS environment during design is the capability to compare several architectures built from an initial library composed of reusable components and equipments.
Especially for critical system as X-by-Wire technologies, the designer has to implement efficient mechanisms to tolerate failures during the operational life. The automatic generation of safety assessments will spare him wasting time to construct manually a fault tree for each variant of the system and to introduce errors in fault trees construction.

Finally, the formal feature of AltaRica language, the ability to simulate manually the system and the syntaxical/coherence checkers are convenient to verify the well-built model.

Safety assessment during the design process

The safety process is therefore parallel to the design process and ranges along the full length of the system development life cycle, continuously interacting with it.

A fundamental point in enhancing the safety process is to have a "common language" to exchange information between the design and safety engineers starting from the system requirements definition phase.
SD9 constitutes a suitable environment to support designers in the product development cycle:

- *Functional specification*
- *Physical architecture modeling*
- *Safety assessment*
- *Verification & Validation processes*
- *Help to choice relevant fault tolerant mechanisms*
- *…*

A main advantage of SD9 environment is its ability to develop a system model which is safety oriented. The language supporting the modelization is suitable for representing dysfunctional features: failure modes, common cause failure, fault propagation, failure rates, behavior in presence of failures …

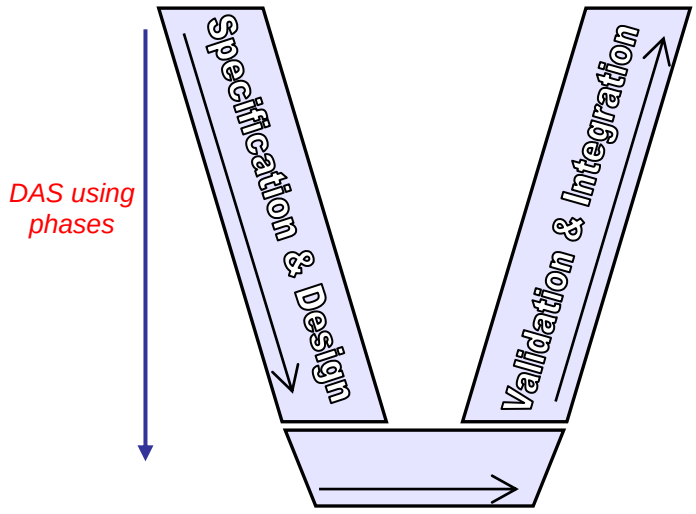

Figure 1: Development Cycle

System modeling

The system modeling is based on the assembling of elementary components and equipments (set of components and/or other equipments) from a user library. The creation and the update of components, equipments and system architectures allow capitalizing information on the following aspects:

- *the nominal functioning behavior for each component,*
- *the failure modes of components (stemming from a FMEA analysis) and their failure rates,*
- *the component behavior when failures occur*
- *the architecture and the I/O interface of sub-systems (equipments),*
- *variants of the system architecture,*
- *the unexpected states of the system by including a monitoring component (observer).*

During the design process, functional or dysfunctional knowledge of a component can change. The user has just to edit it and to make an adjustment on it.

The library is composed of different tabs relating to:

- *components*
- *equipments*
- *projects/systems*
- *types*
- *operators*
- *graphical shapes*

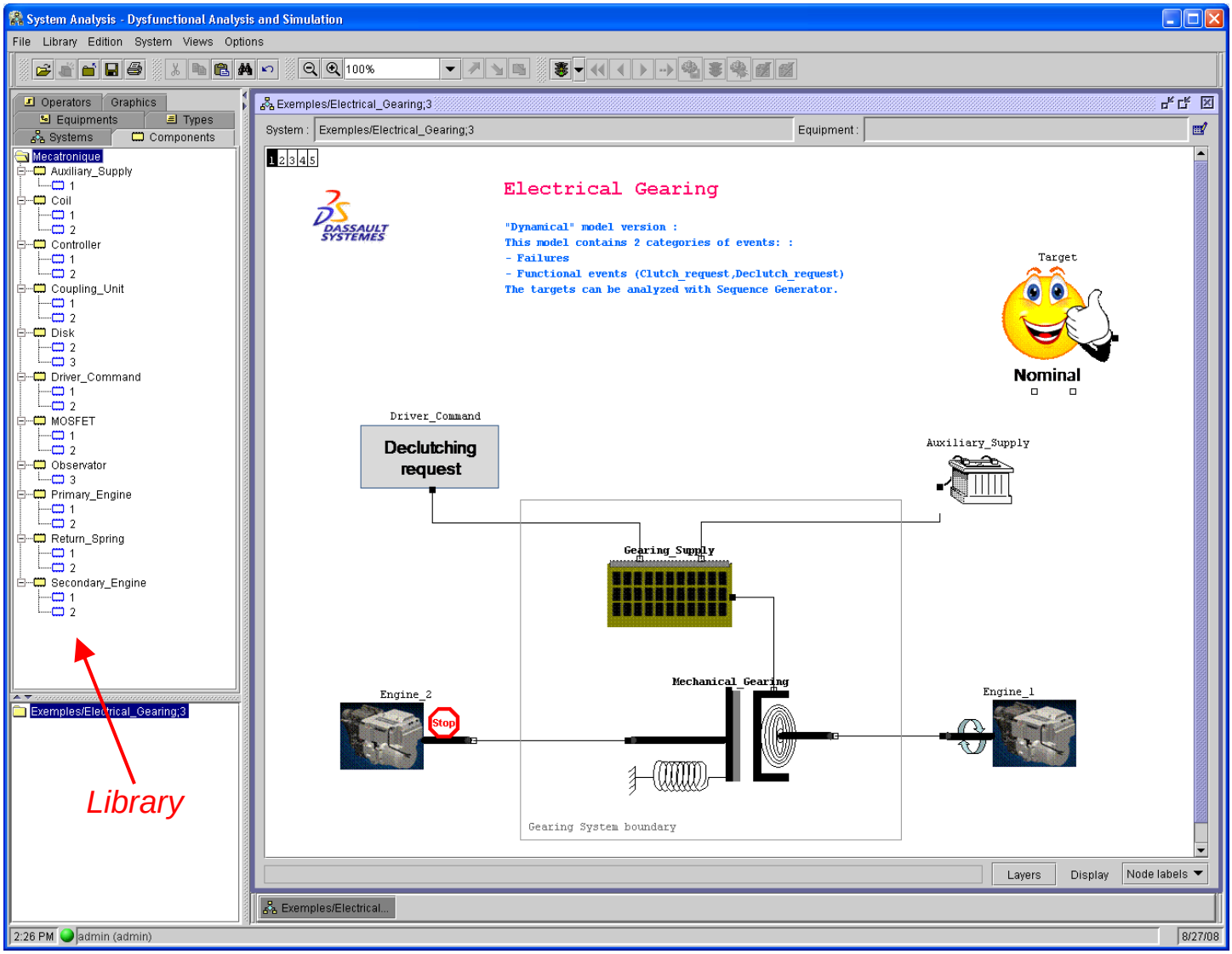

Figure 2: The library tabs

Creation of a component

In SD9 environment, a component is characterized by:

- *state variable(s),*
- *input/output flow variables and their type,*
- *event(s) (functional events or failures),*
- *a behavior.*

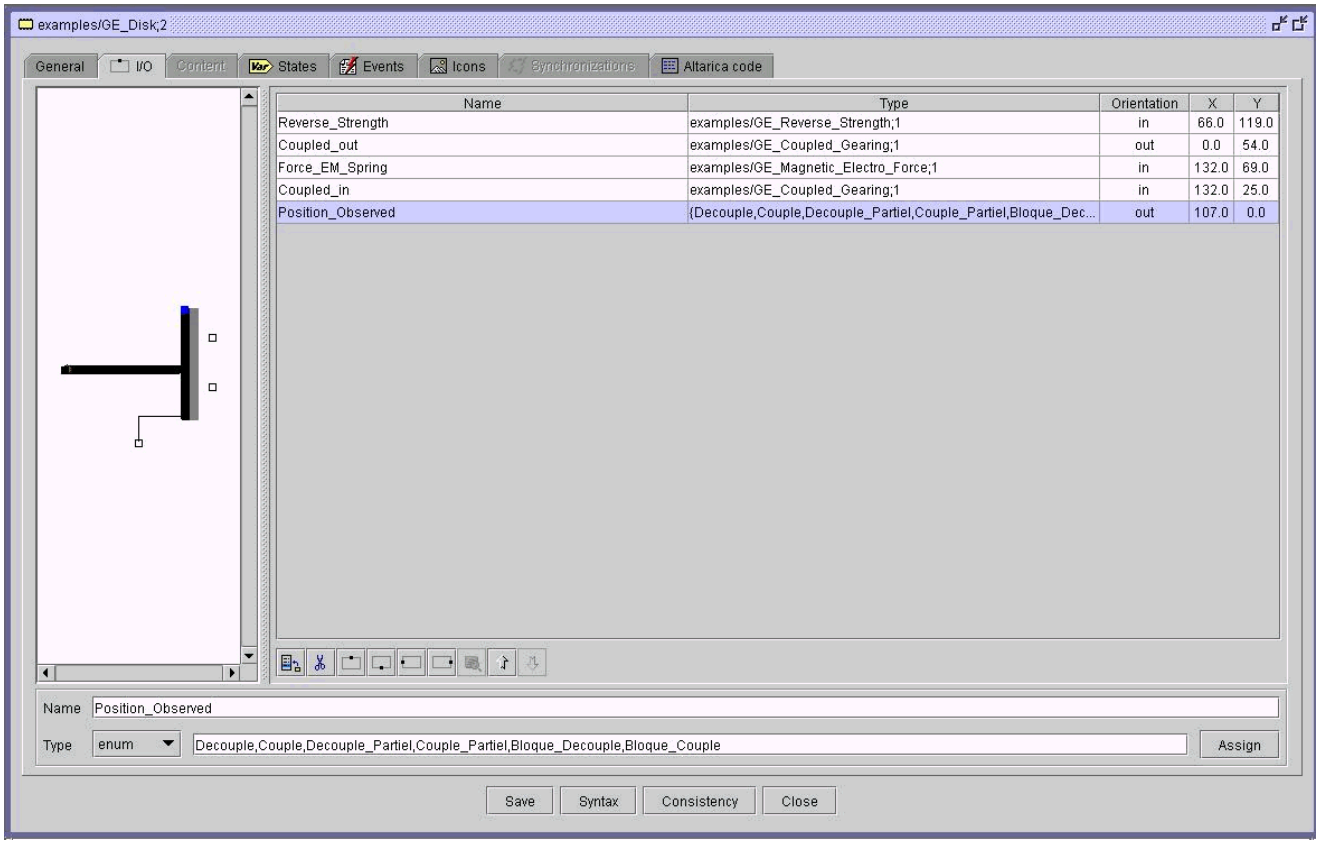

Figure 3: Edition window of a component

The AltaRica language is used to specify the behavior. This is a formal language based on states/transitions formalism. A behavior is specified by mean of transitions and assertions.
The modeling principle in AltaRica is graphically sketched on the following figure. The states are symbolized by circle containing assertions. The transitions are represented by oriented arcs connecting the states.

The assertions specify the value of output flows. These values are depending on the current states of the component and possibly on the value of input flows.

The transitions describe how the component can change state. They are composed of a guard (a condition on the current state and possibly the value of the input flows), an event, and the final state.

Thus, a transition is fired if the condition (a Boolean) is true, and if the event occurs.

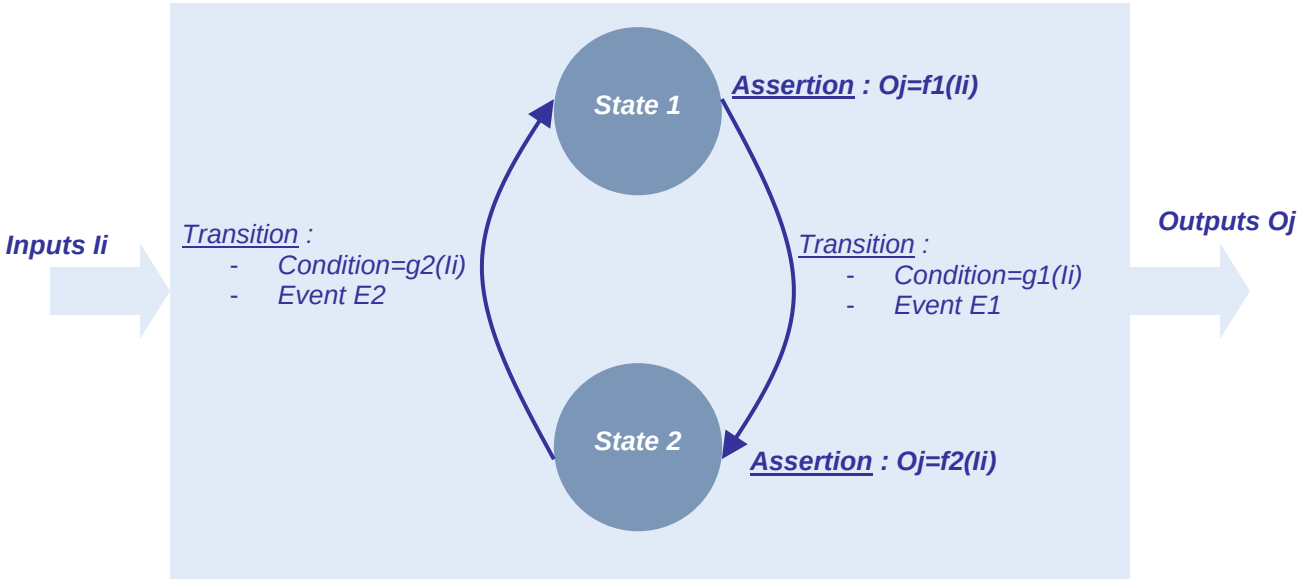

Figure 4: A graphical schema of a component

Let's describe more precisely the component features:

State variables of the component:

Typically, the state variables allow characterizing the internal state of a component from a functional and/or dysfunctional point of view. For example: Nominal/Degraded Mode/Failed or ON/OFF…

The functional specifications and a FMEA analysis of a component can help the designer to define the most relevant type and the number of these variables.

Note: it is possible to create a user type associated to a state variable so as to reuse it from the library.

Input/Output flow variables

A component is linked to others by mean of an Input/Output interface.
The components to be linked must have the same type definition to their input/output flows.

When a failure occurs on a component, its dysfunctional state is spread over the other components by mean of its output flow.

The nature and the number of input/output flow variables are determined from functional specifications (component interface).

Mostly, input/output flows of a component correspond to physical data. Instead of modeling the set of possible values, we advocate to represent simply the "state" of information propagated by the flows.
For example, let's consider the value of an angle measured by a sensor. The value is contained in intervalle [0; 360].

For safety analyze purpose, it is not useful to handle with the real type of the variables. It is more convenient to represent an abstraction of the types: the state of variables. For example **Nominal, Erroneous, Drifting** *are possible values for a sensor measure…*
This method allows reducing the complexity of models by avoiding a fine discretization of variables.

Events

Events are used in AltaRica code to represent principally failures. They can also be used to integrate functional events in components. For example, a request sent by a calculator to an actuator may be modeled by an event.
These objects are necessary to add a functional behavior to a safety-oriented model.
As a consequence, when the event-sequence research is performed, the scenarios are composed with failures and possibly functional events.

It is possible to associate a probability law to each event. These laws are exploited for a quantitative assessment purpose in FT tools.

AltaRica Code

When a component is edited, a tab allows adding an AltaRica code to a component.
This formal states/transitions formalism is suitable for modeling complex systems.
Behavior modeling principles are not limited to a Boolean description like OK/KO or TRUE/FALSE, but many user types can be defined: degraded modes, software or hardware reconfigurations, detection mechanisms, transitional states, latent faults…

Moreover fault tolerance mechanisms implemented in software technology can easily be modeled by using immediate transitions to defined automatic and immediate switching to a safe state. These special transitions use an event having a Dirac(0) law.

Example: the following AltaRica code represents a transition of a component: if the component is in the nominal state (State=Nominal) and the input flow value is KO (Sensor_Value=KO), then the component state instantaneously becomes Degraded (State:=Degraded). no_event is a immediate event having a Dirac(0) law.

```
State=Nominal & Sensor_Value=KO |- no_event ->
                State:=Degraded;
```

Thus, reconfiguration principles can easily be described by immediate transitions.

However, it is important to use them with care to avoid unstable behavior of the system, particularly when infinity of transitions can be successively fired (unstable loop in a system).

Creation of equipment

From a hierarchical point of view, an equipment corresponds to a sub-system.
An equipment is composed with other equipments and/or elementary components.

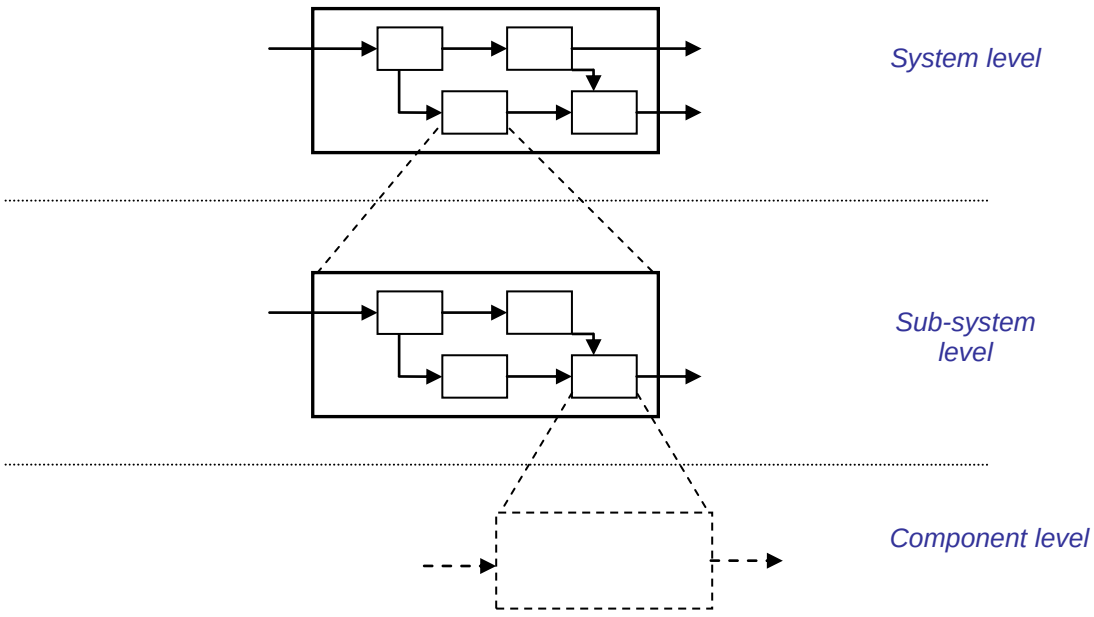

Figure 5: Hierarchical decomposition of a system

A hierarchical decomposition of a system into several sub-systems is a casual manner to handle with complex systems during the design process.
From a functional point of view, a sub-system corresponds to a set of interacting elements which performs one of the functions defined in the specifications.

For example, an equipment can be created to bring components together, particularly when they cooperate to perform a common function. The next figure illustrates an equipment specifying an actuator supply:

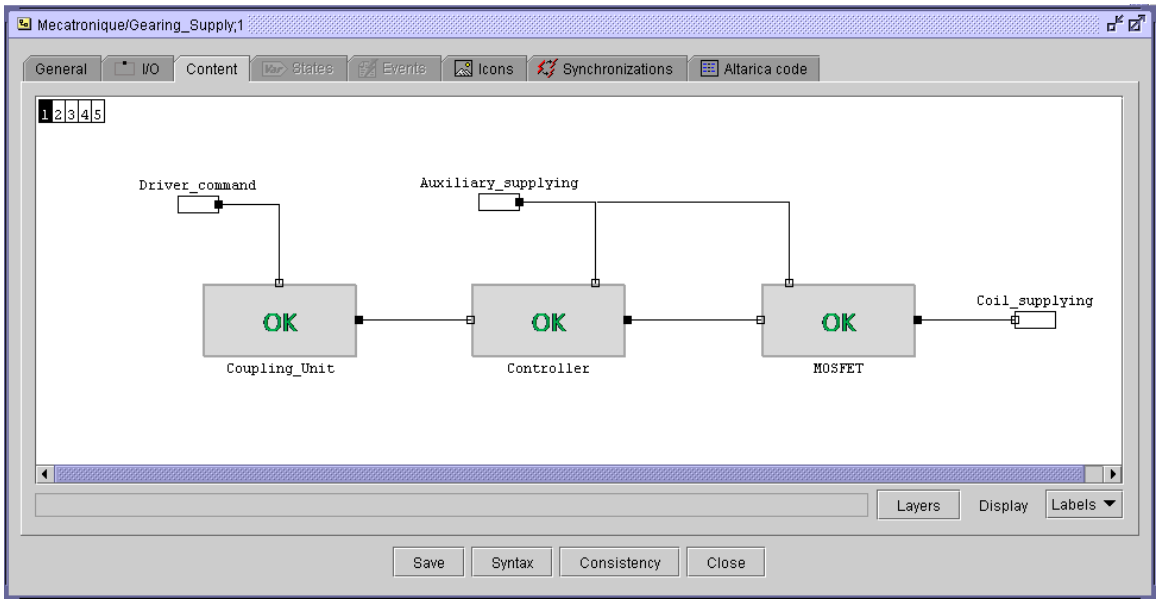

Figure 6: An example of equipment content

Mostly, we recommend designers to use equipments so as to improve readability and understanding of the graphical representation of a system. Moreover, the creation of equipments in library allows gaining time when they will be reused in other architectures.

Creation of a system

The last step of the system modeling consists in linking components/equipments to create one or several suitable architectures having the same functional/dysfunctional requirements.
Indeed in DAS tool, it is possible to create several variants of a same system. This functionality is useful in specification/design process when the hardware architecture is not yet definitive. Thus, safety assessments (fault tree and/or sequences generation) will help designer to choose the most reliable architecture from a qualitative or quantitative aspect.

Let's highlight that the creation of several architecture variants is very fast after having created the components and equipments in library. Each architecture variant can use the same elements in library, but with a different structure or including different safety mechanisms (number of fuses, circuit breakers…).

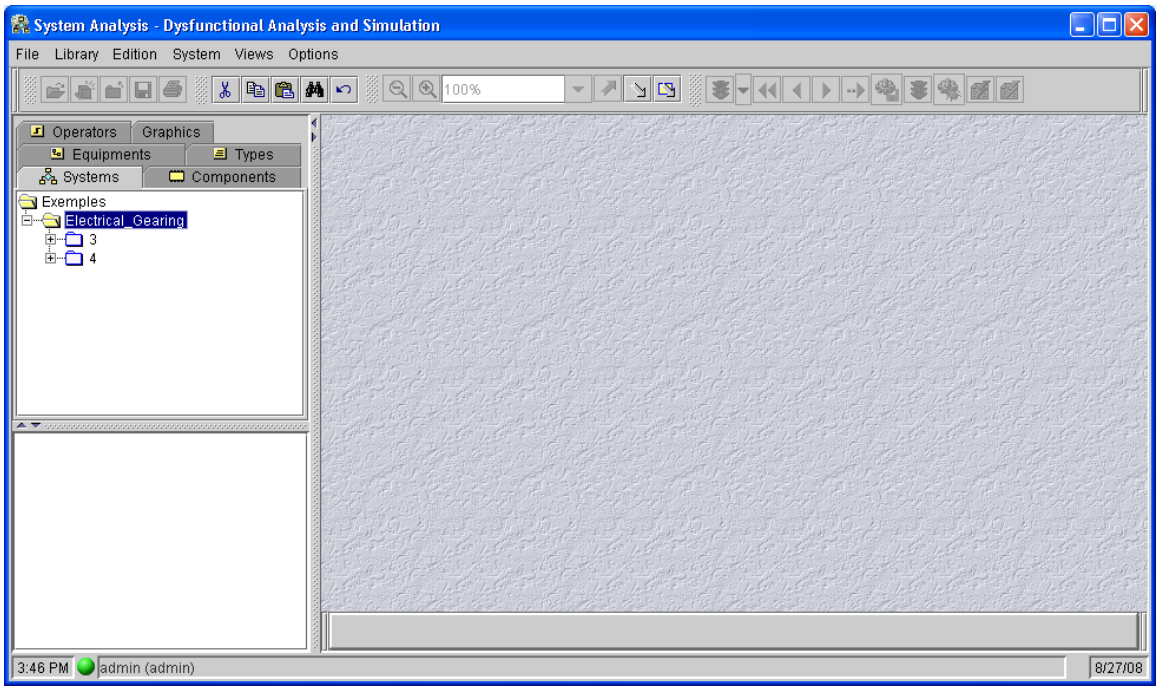

Figure 7: Some variants of a system

We advocate creating components as generic as possible to make the reuse easier.

The system level is the highest level in the hierarchical decomposition. Two methodologies are possible to design architecture:

- *The top-down approach (modeling by refinement)*
- *The bottom-up approach (modeling by assembling elementary components in equipments)*

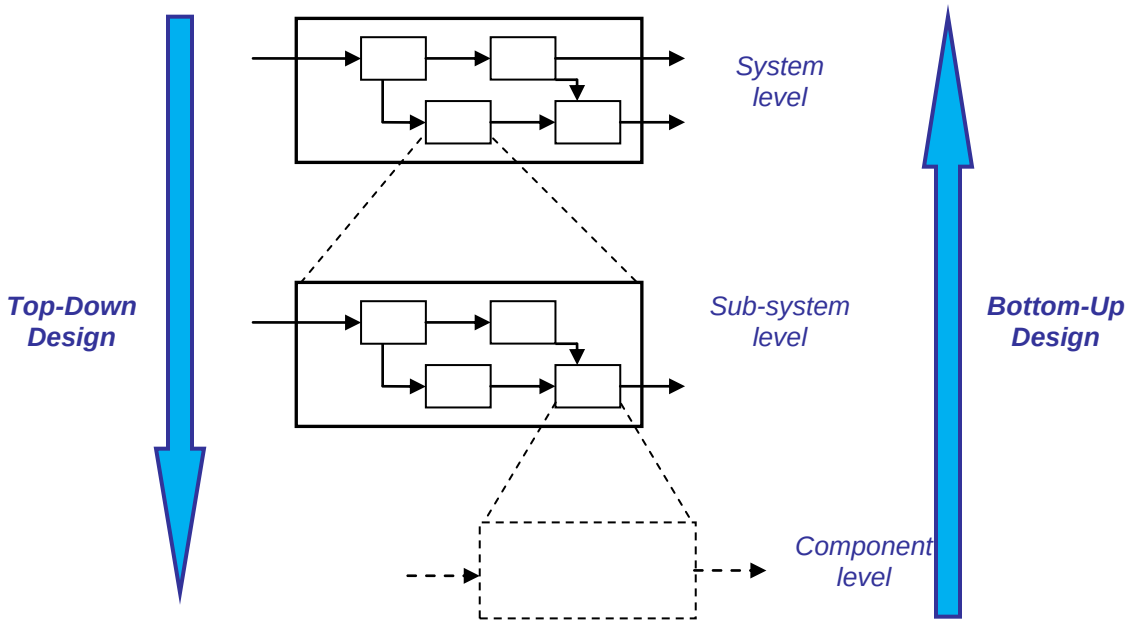

Figure 8:Two design and modeling approaches

Finally, in some cases, it can be necessary to add external components which will interact with the system:

- *to include the environment interaction (which can impact on the dysfunctional behavior of the system)*
- *to include human impact (for example a command request sent to an actuator…)*
- *to visualize and monitor graphically the system behavior (for example by adding an Observer).*
- *to connect system with other systems which will stimulate it by sending some functional requests*
- *to represent several possible configurations or modes (for example initialization, braking, acceleration, standing phases…)*
- *…*

The following figure is an example of a system connected with "external" components:

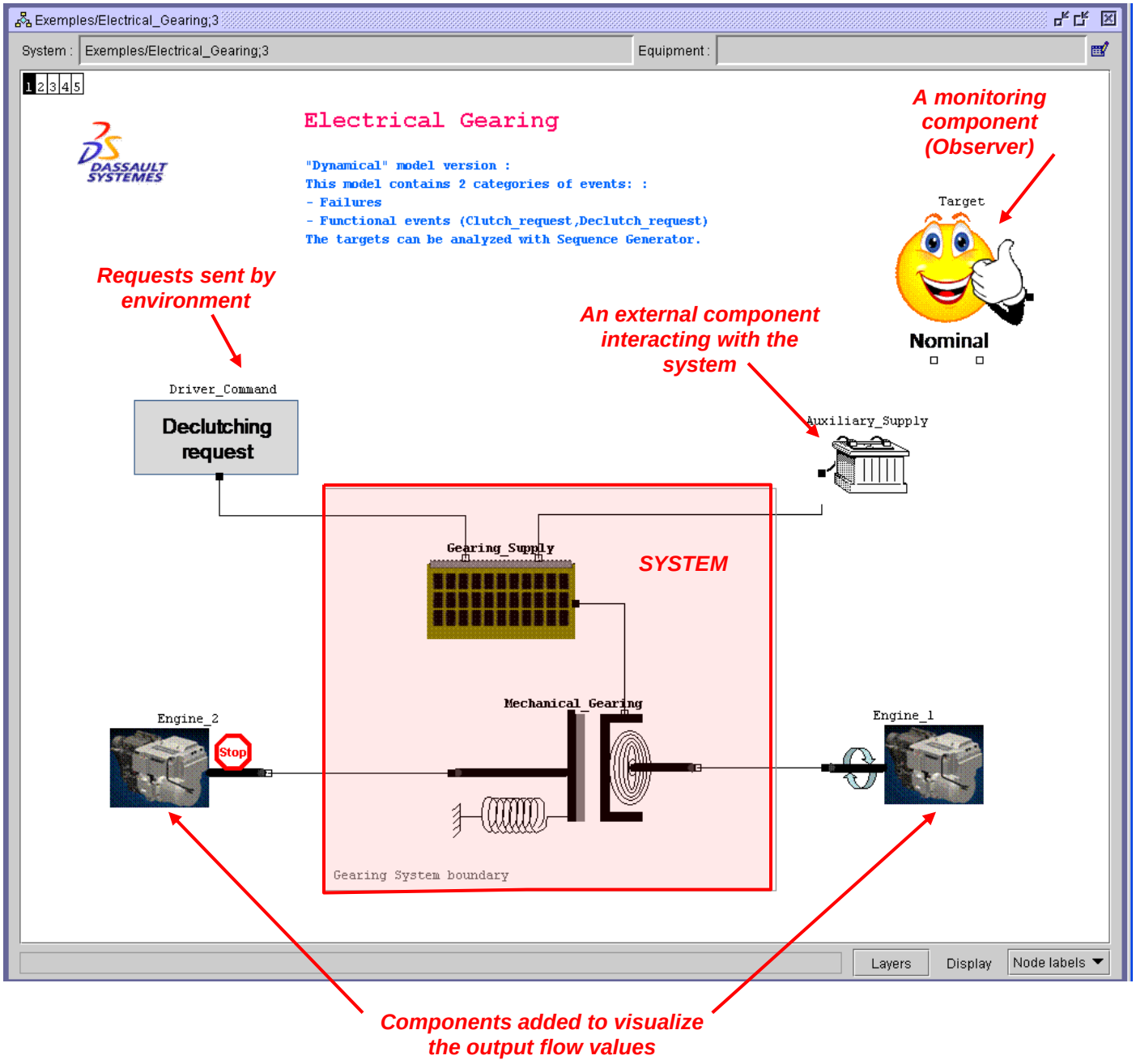

Figure 9: A system in interaction with external components

Defining an observer

After having created the system, we may include the characterization of known Unexpected Events in the modeling.

These UE are identified during the system specifications from a Preliminary Hazard Identification. They will be used to define the targets of the safety processing (Top event of Fault Trees and Target of Sequence generations).

We propose to create a specific component allowing monitoring specific states of the whole system. The goals are:

- *to monitor the system state during the simulation (for V&V purpose),*
- *to monitor the system state during the fault injection,*
- *to specify the target for a safety processing (the top event of a fault tree, or the target of sequences).*

The behavior of such a component is formalized by an automaton which evolves in accordance with some variable of test points.

In simple cases, the system state can be deducted from internal states of some elementary components.

Sometimes, the system must be considered as a "Black Box", particularly when this one interferes with environment. The state is deducted by comparing input flow values of the system with the resulting output flow values.
For each request value sent to the system input, specific flow values of the outputs are expected. Wrong values mean that the system behavior is incorrect:

- *either because the nominal behavior don't satisfy functional requirements*
- *or because a failure occurs.*

From a modeling point of view, it is possible to hide the graphical links between the Observer component and the test points by creating "system assertions". This improves the graphical readability of the model.

Verification & Validation of a model

DAS tool is an environment that supports the design of complex systems.
The introduction of this tool, gives the designer a way to perform the validation/verification-analysis; at the same time it provides the safety engineer with a potentially new means for performing his work more effectively.

Several methods are available to carry out V&V tasks:

- *syntax control procedure*
- *consistency control procedure*
- *step-by-step simulation*
- *event sequence generation*
- *model-checking (Plug-in)*

Syntax control procedure

This processing allows detecting errors in AltaRica modeling and displaying the causes (typing errors, bad assertion declaration, using keywords…).

Consistency control procedure

AltaRica is a formal language; processing as consistency control allows eliminating the residual ambiguities of the modeling as conflict transition, indeterminist behaviors…

Some examples of anomalies detected during the consistency control:

- *an output variable is never valuated in a component (problem of incompleteness),*
- *an output variable is never valuated in some conditions (problem of incompleteness),*
- *2 different values are affected to a component output or state in identical conditions (problem of coherence)*
- *A component never reaches a stable state,*
- *…*

Before launching a simulation or a safety processing, a consistency control must be performed.

Consistency control is available also:

- *At component level*
- *At equipment level*
- *At system level*

Step-by-step simulation

Simulation can be performed to validate some requirements of the system. For example, a sequence of events can be replayed to validate a nominal behavior, the efficiency of fault tolerant mechanisms, the reachability of degraded modes…

Simulation can also be used to inject failure modes so as to assess the effects of some failure modes in the system.

Moreover, simulation is supported by a graphical animation which allows the graphical monitoring of the system behavior (components, equipments and links)

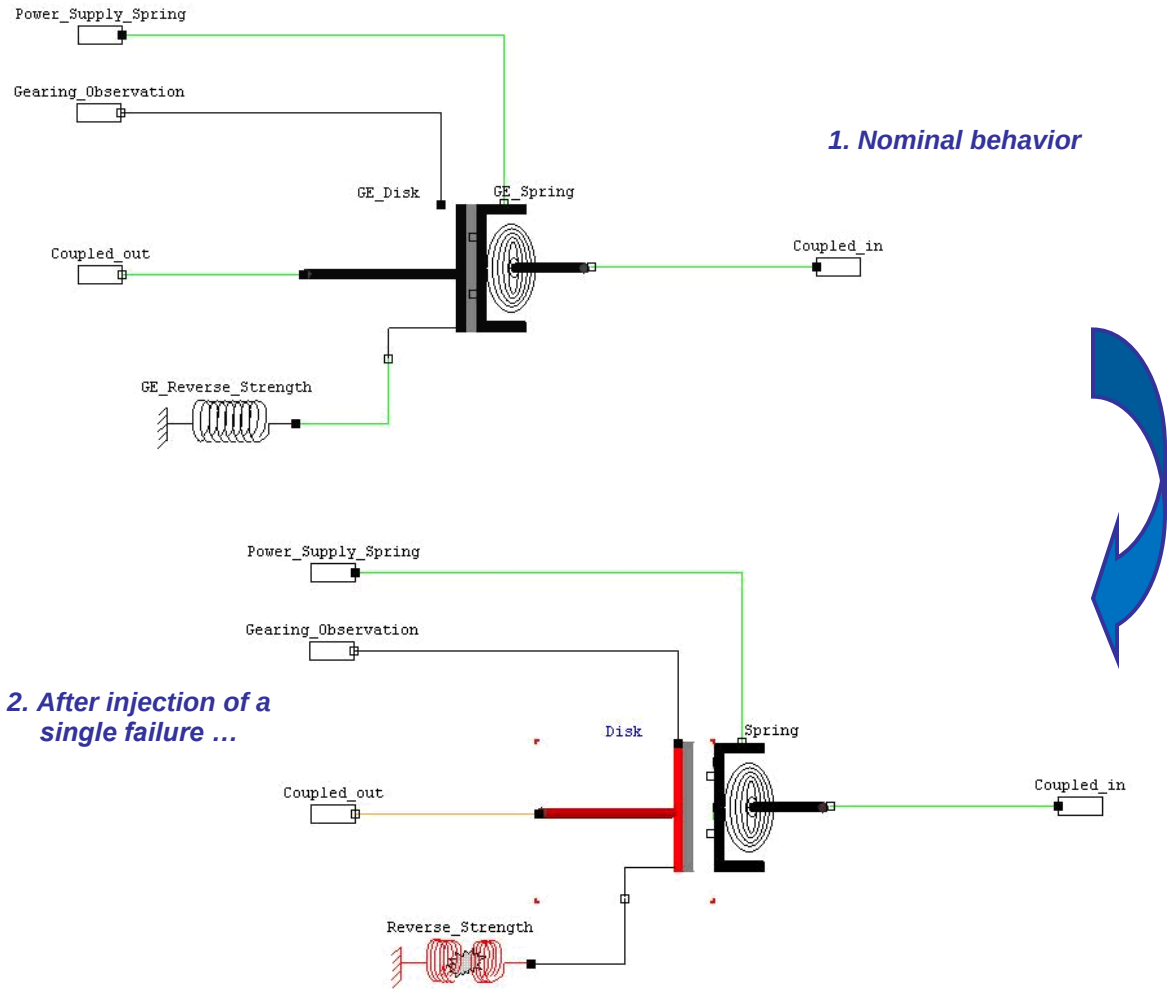

Figure 10: Example of a graphical animation during a simulation

Debug window

During the simulation, user can display a debug window. This window can be spited into several parts/tabs to display information about variables, transitions, history…
With tool can help designer debugging the model and can be used in addition to the graphical mode.

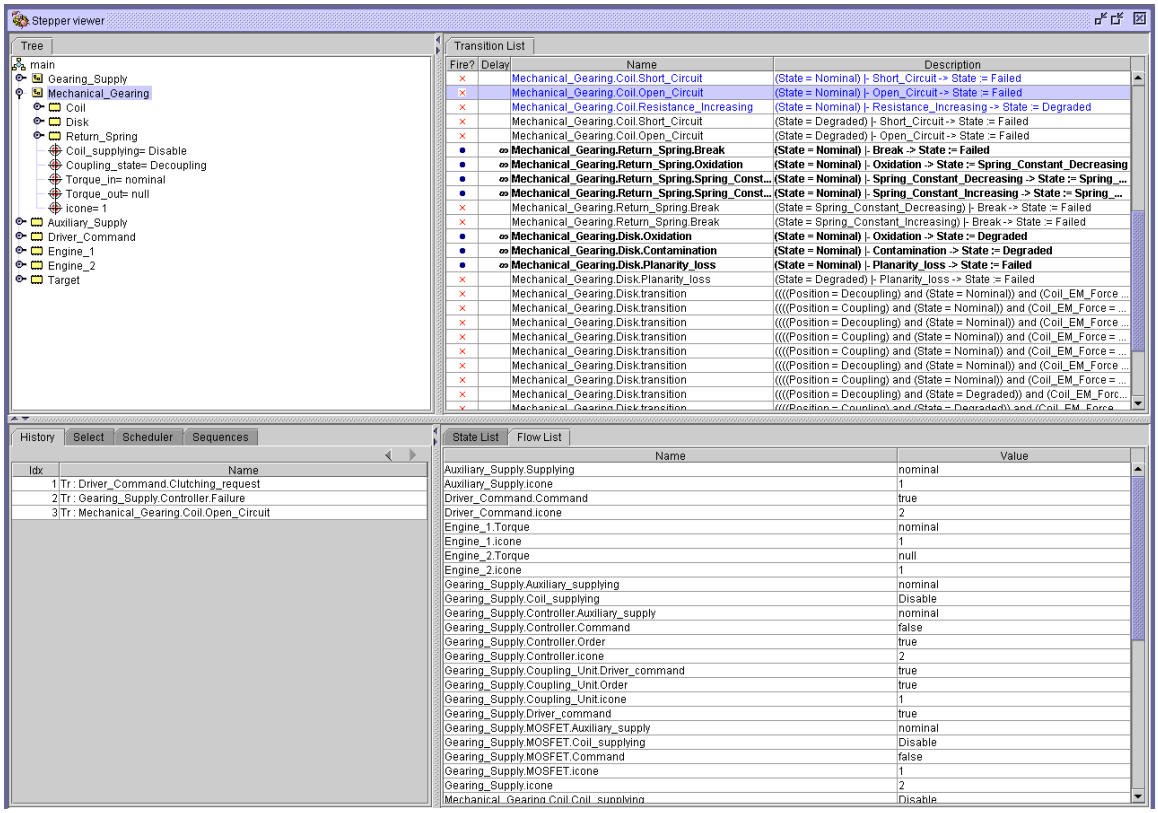

Figure 11: Some tabs of the Debug Window

This window is very convenient to monitor some parts of the model.

Some available tabs:

- *State List*
- *Flow List*
- *Transition List (user can select a transition to be fired from with list)*
- *History (list of transitions/events fired)*
- *Tree view (flat view of the system decomposition)*
- *Sequence view (to replay a list of loaded sequences)*
- *…*

Generation of fault tree

The classical Fault Tree approach provides a graphical and logical framework for analyzing the reliability of a system with respect to its safety critical main functions.
In general, a fault tree provides a conceptually simple modeling framework to represent the system level interaction between component reliabilities. Moreover the traditional fault tree is a static representation of component failure interaction that uses the traditional Boolean logic gates (like AND, OR and not) but it is not able to represent the sequential behavior of failure events. Therefore, the fault tree graphical and logical representation is usually powerful enough for systems / sub-systems including only hardware components or characterized by very simple control laws.

SD9 allows generating automatically a Fault Tree from a model. A necessary requirement to get a relevant fault tree is that the behavior of the system remains ***static****.*

Although a fault tree provides a static representation of failure sequences leading a system to a feared event, this is really convenient in design process to have a graphical abstraction of a dysfunctional behavior. Moreover, it is easy to obtain qualitative and quantitative results from a fault tree (minimal cuts, probability of the UE…).

In SD9, the fault trees are generated in ARALIA format (.dag). Then, they can be edited and processed with many existing tools (for example Aralia Fault Tree Analyzer).

It is important to notice that minor design changes can have an extensive impact on the related safety analysis. A main advantage of using SD9 is that fault tree models are generated automatically. This aspect is really convenient in a design process, when functional and hardware architectures are continuously changing.

Example of a fault tree generation:

Let's consider a static model:

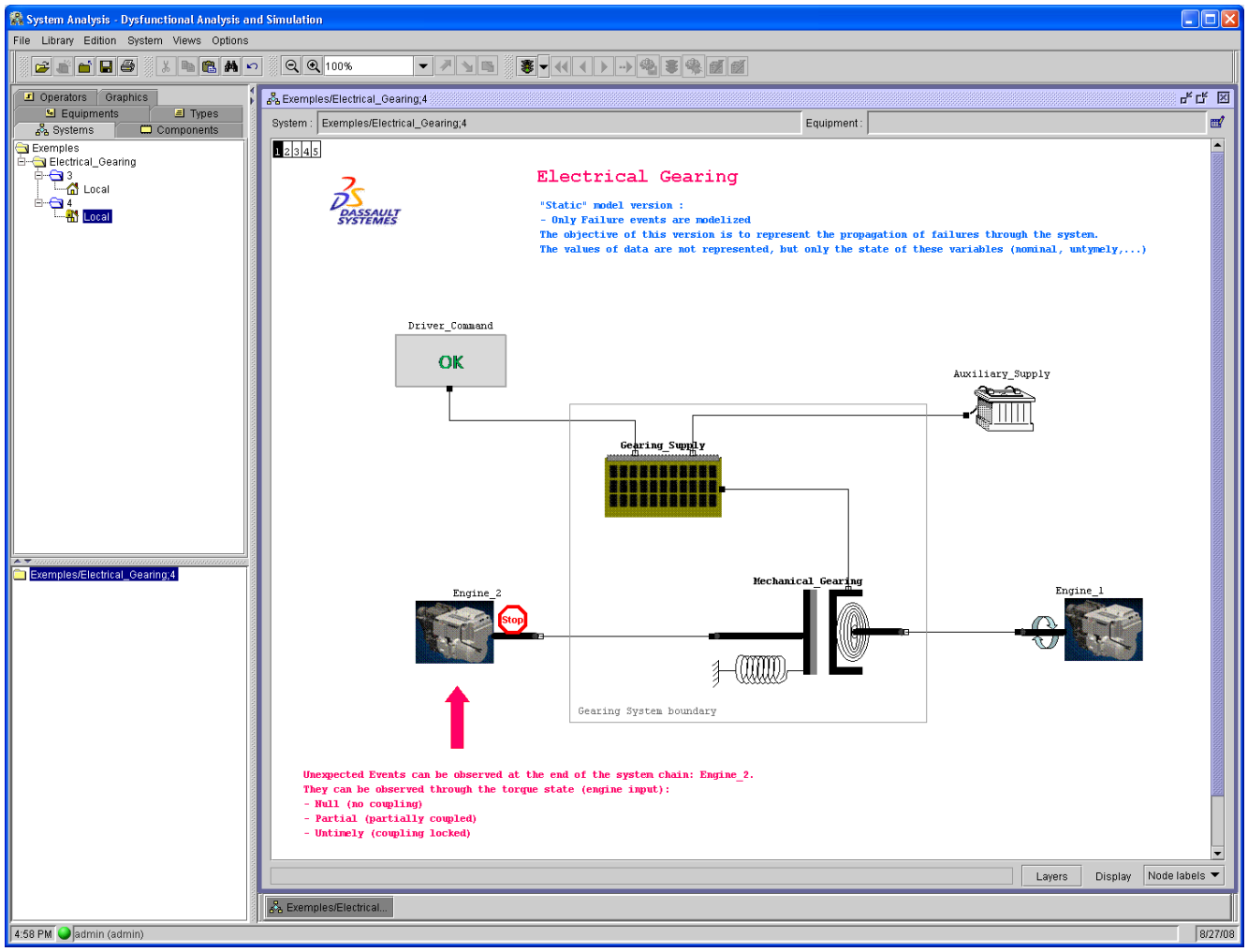

Figure 13: A static system

The goal of the safety analysis is to generate a FT from this model.

The designer has just to select the suitable Plug-in in the menu System: ***Fault Tree generation (ABC).***

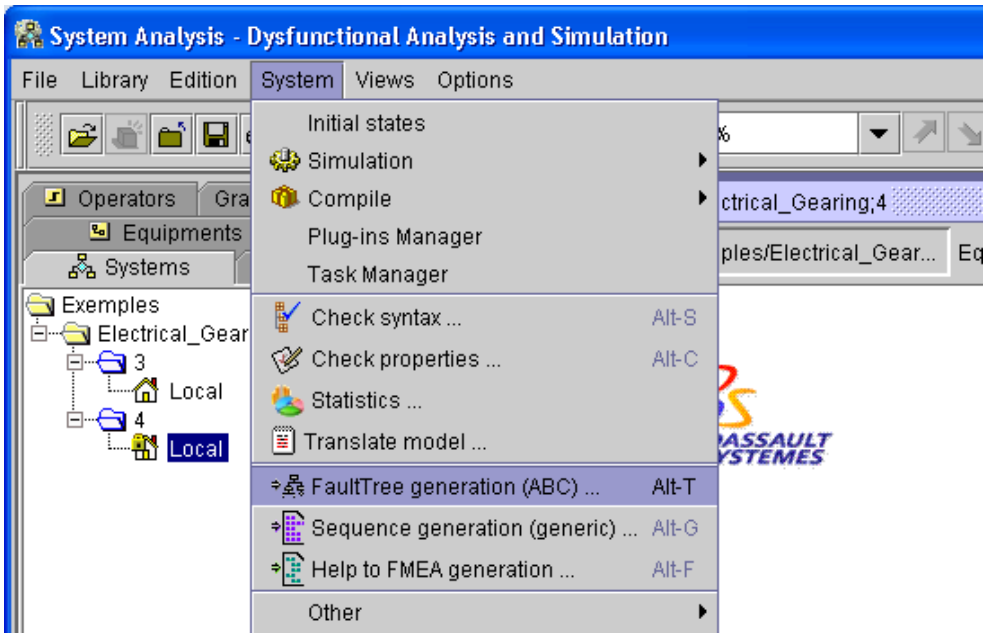

Figure 14: Fault Tree Plug-in

Then, user has just to select the target (UE), the FT name and the algorithm.

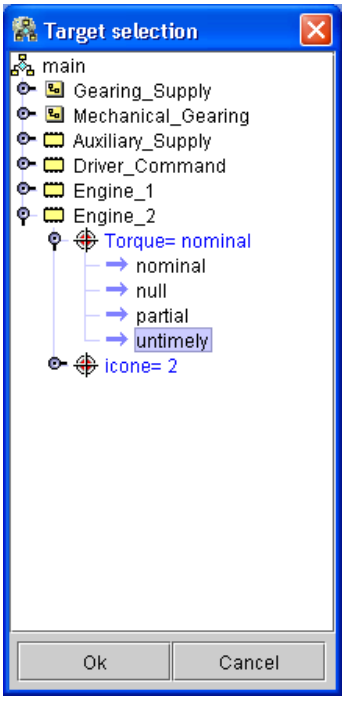

Figure 15: Target selection

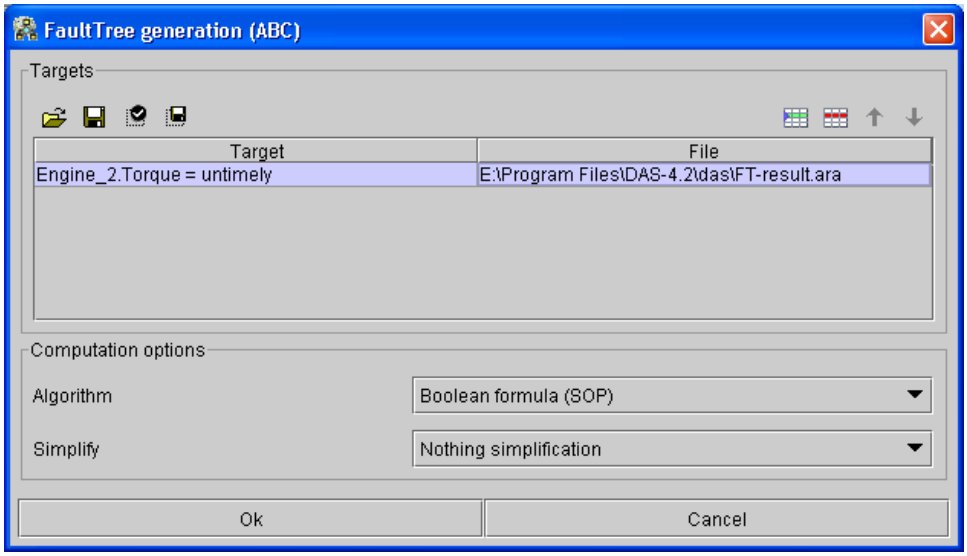

Figure 16: Parameters of the Faut Tree generator

Identification of critical paths

Fault tree is not a suitable formalism/technique to represent and analyze the dynamic behavior of complex systems.

Contrary to systems including only hardware components, the reliability of embedded computer systems can be difficult to be analyzed by using the traditional static fault tree representation for several reasons. First, hardware and software are integrated and must be considered as a single model. Second, fault tolerance techniques as automatic recovery on detected errors raise the possibility that these automatic mechanisms may themselves fail.

The proposed solution to handle with dynamic behavior is the Sequence Generator.
The principle of this method consists in identifying all ordered combinations of failures/events leading the system in a specified state (degraded or unexpected state…).

Mostly, sequence generator is used:

- *to investigate influence of "intermittent failures",*
- *when Fault tolerant mechanisms are modeled (dynamic reconfigurations, degraded modes, diagnostic, fault monitoring…),*
- *functional behavior interacts with dysfunctional behavior,*
- *…*

*The Sequence Generation method can be launched in SD9 by selecting the plug-in from the menu '**System**':*

- *Select Menu **System ➔ Sequences generation (generic)***

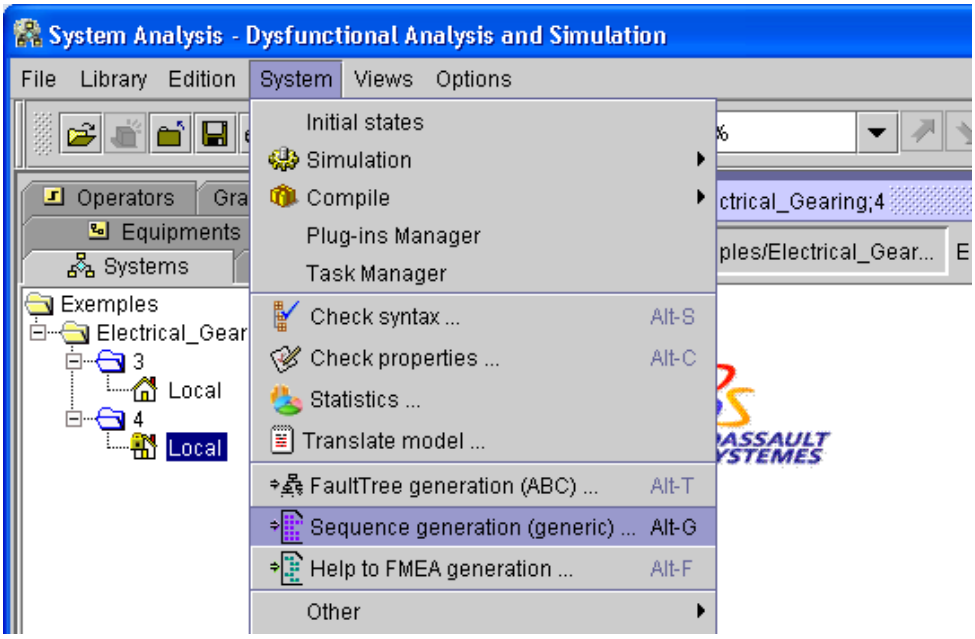

Figure 17: The Sequence Generation Plug-in

Then, as the fault tree generator, user has just to select the "target" and the different parameters in the following window:

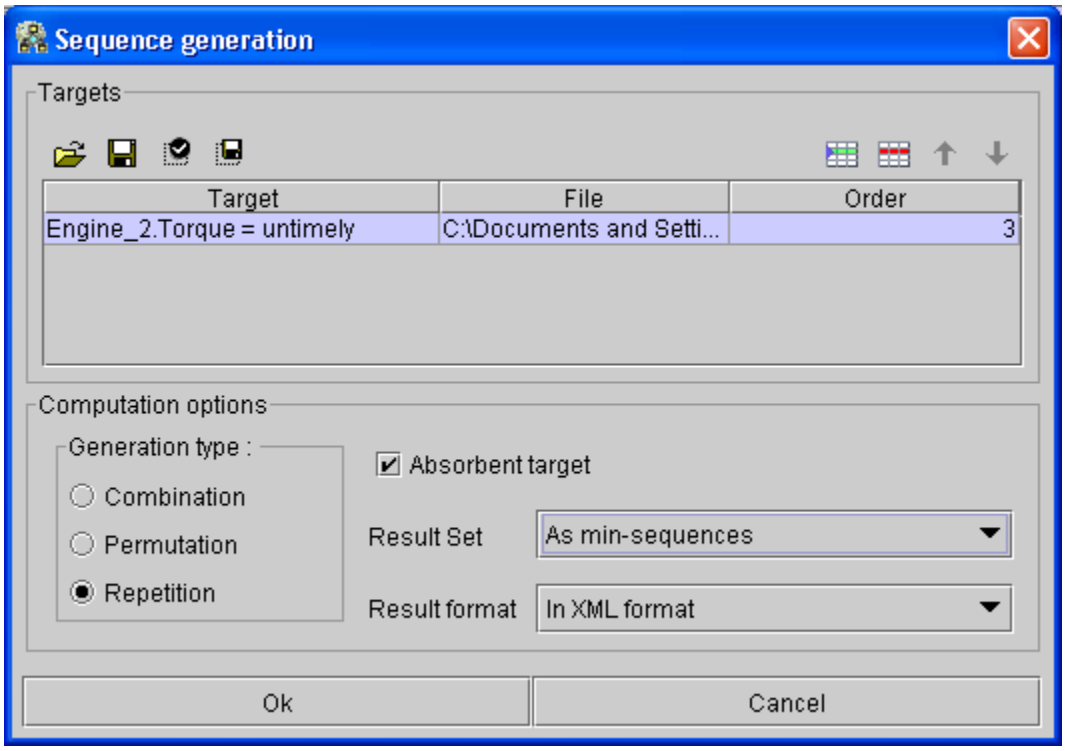

Figure 18: Sequence Generator Window

Several parameters are available:

- *The generation type :*
  - *Combination*
  - *Permutation*
  - *Repetition*
- *The Order*
- *Absorptive top event*
- *The results format :*
  - *Aralia format*
  - *Min-cuts format*
  - *XML format*
- *The result Set :*
  - *As min-sequences*
  - *As sequences (not minimized)*
  - *As min-cuts*

Option 'Generation type'

The option choice depends on the system structure and modeling. Three different values are available:

Combination *is suitable for static systems, where events are not ordered. All sequences are not simulated during the sequence generation, because some combinations are considered as equivalent. For example, if the sequence {A, B} is simulated, then the sequence {B,A} will be not, because this option suppose that they are equivalent.*
The user may select this option only for static systems: the behavior is not dependant of the event firing ordering.

Permutation *is used when the events are ordered. This option is suitable for dynamic systems for which the behavior can be different according to the firing ordering. Contrary to the previous example, both {A,B} and {B,A} are simulated. But only one may lead the system to the target state...*

Repetition *must be selected when an event can be fired several times. This option is often used for functional events; the following figure displays an example of an automaton representing a counter:*

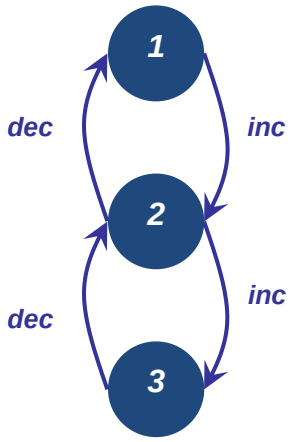

Figure 19: Example of a counter

On this example, to reach the state 3 from state 1, it is necessary to fire the same event "inc" two times. If the option "Repetition" is not selected, the component will never reach this state.

Note*: the processing duration is dependent of this option.*

Parameter 'Order'

The ***Order*** *(1 to 9) corresponds to the maximal length of event-sequences.*

Note*: the processing duration is dependent of this option*

Box 'Absorptive top event'

*The box "**Absorptive top event**" means that no more events are fired when the target state is reached. This option is not suitable for system having a reconfiguration delay (the system briefly transits in a no-safe state), or when reparation policies are implemented in the model. Indeed, for a system having a reconfiguration mechanism, the system can reach an unexpected state during a transitional period. Thus, functional events must be fired to recover the system. If the option is checked, the reconfiguration mechanism can't be tested because the transitional feared state is considered as absorptive.*

***Note**: the processing duration is dependent of this option*

Option 'Result Set'

***As sequences (not minimized):** all sequences leading to the target are displayed. They are no post-processing on sequence list. Thus, a lot of non-minimal sequences are included in the result file. In general, this choice is selected to check if the model is correct. Each sequence can be re-simulated with the step-by-step simulator, because the event ordering is kept.*

***As min-sequences**: A post-treatment is performed on the sequence list. All sequences which are non-minimal are removed. A sequence is non-minimal if it contains a smallest sequence of the list. For example, if the following sequences are identified by the generator:*
{A, B}
{A, C, D, B}
{A, B, C}
{E, A, C}
Only sequences {A, B} and {E, A, C} remain.

Note: Each minimal sequence can be re-simulated with the step-by-step simulator, because the event ordering is kept.

***Min-Cuts:** allows displaying a summarized view of the whole sequences. A sequence which contains a smaller one is removed from the list. This option must be handled with care, because the resulting sequences are no more ordered. Repetitive events are removed.*

Note: For time-dependant system, a minimal cut may the system not reach the target, particularly if the event ordering or the event repetition is (are) lost.

Note 2: for a static system, min-cuts and minimal sequences are equivalent.

Option 'Result Format'

This option allows to specify the file format of the sequences

***Aralia format**: the sequences are saved in a file with .dag extension (Aralia fault tree format). This option is useful to build a fault tree from a dynamical system for example for qualitative or quantitative processing purposes.*

Note: a fault tree has to contain only failures. In some cases, a system can include both failures and functional events. User must remove all functional events from the sequence file.

***MCS format**: this selection allows to save the sequences in Min-Cuts format. However the sequences are not converted into minimal-cut sets.*

***XML format**: this selection allows to save the sequences in a XML format. This option is very useful to generate a generic format which can be manipulated to convert the sequence file into a custom format (tabulated file, RTF, PDF, XLS…). Moreover a lot of information are stored in the result file: event information (laws, attribute), sequences options,…*

An example of using SD9 in a design/safety process

Let's illustrate on a simple example how SD9 can assist a designer to improve the design process including safety aspects. The aim of this study consists in consolidating system architecture by adding fault tolerance mechanisms.

The system is an electrical clutch. An electric control allows the coupling and the decoupling of the clutch. We suppose having an initial physical architecture of the system.
This system must satisfy the following safety requirement:
No single failure of the supplying control system has to lead the clutch in the unexpected state "locked in the coupled position".

Step 1:

*The first step consists in modeling the system and the component related to the unexpected state (the **Target** component):*

The following figure shows the complete modeling:

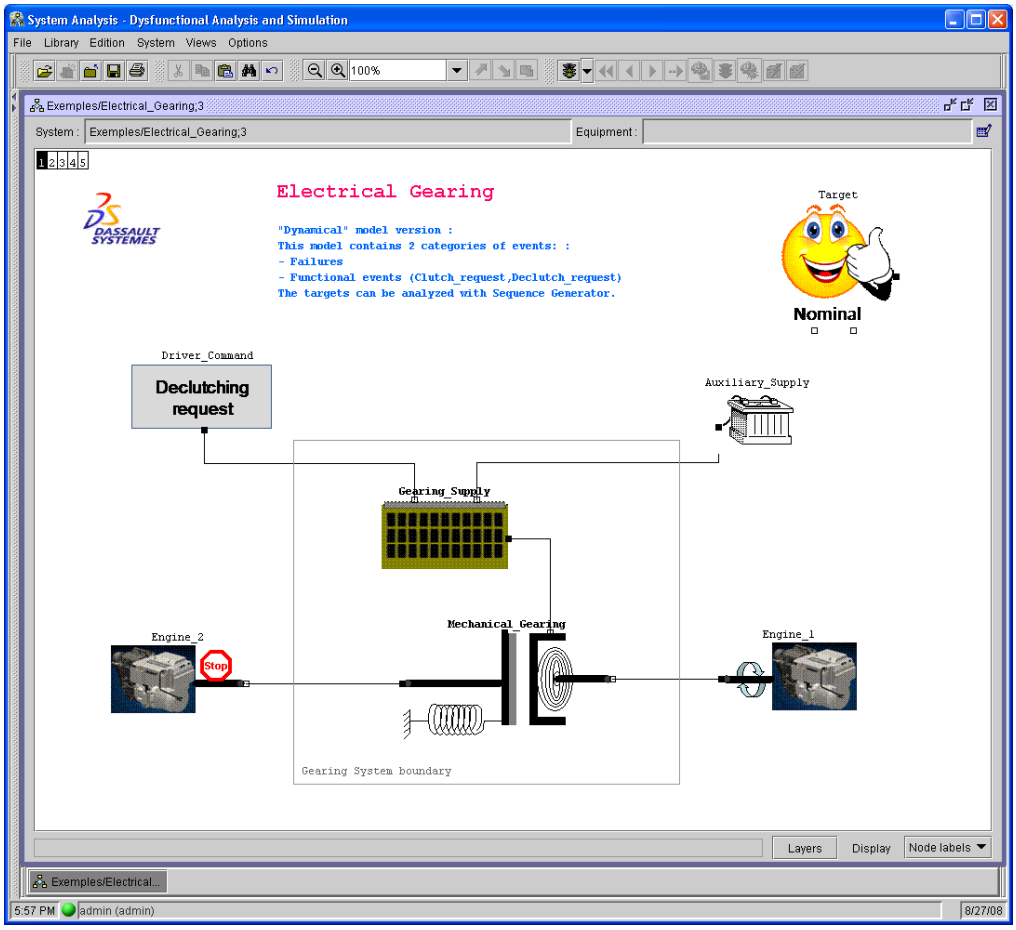

Figure 20: System modeling

The Gearing supplying is modeled with an equipment.

Step 2:

Now, a qualitative system assessment can be processed to verify if the requirement is fulfilled.
This system is featured by a dynamic behavior, thus, the fault tree generator cannot be applied. The sequence generator must be chosen.

The following options are selected:

- *repetition (because of the presence of functional events which can occur several times during a simulation),*
- *the order is set to 2 (sequences can include failures AND functional events),*
- *The unexpected state is considered as Absorptive,*
- *Results are displayed as minimal sequences. .*

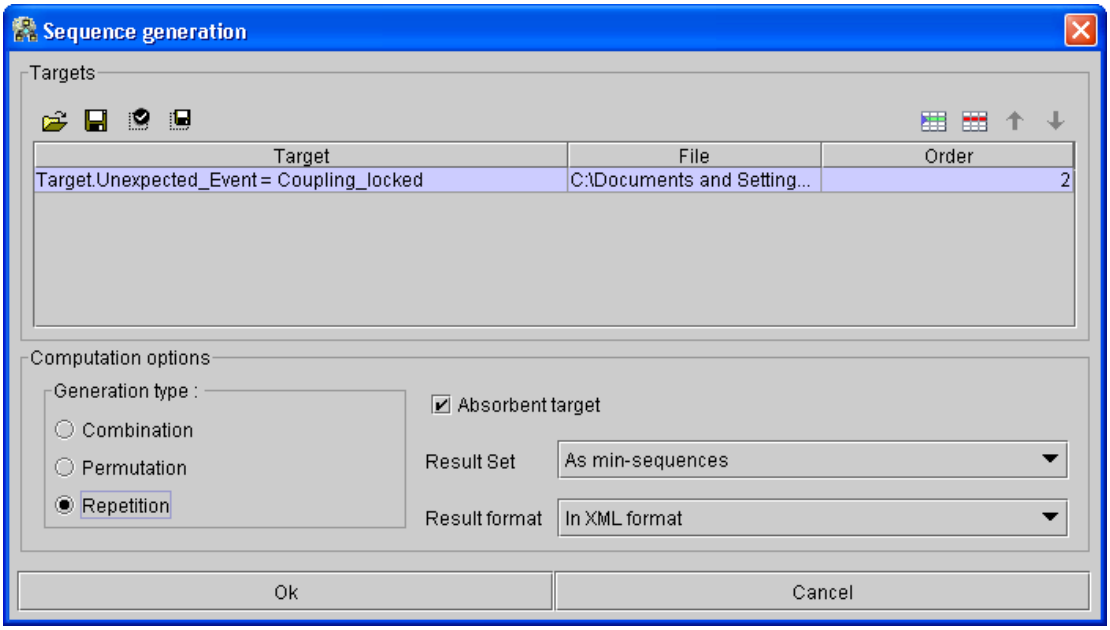

Figure 21: Options of the Sequence generation

A file containing the sequences is created:

```
<?xml version='1.0'?>
<seqgen>
	<define>
		<target name="Target.Unexpected_Event" value="Coupling_locked">
			<param name="filter" value=""/>
			<param name="resultset" value="minseqs"/>
			<param name="finder" value="repetition"/>
			<param name="repetition.order" value="2"/>
			<param name="locker" value="true"/>
			<param name="format" value="XML"/>
		</target>
	</define>
	<abstract>

/* Order of products :
	1	2
 */
	</abstract>
	<result>
		<seq><tr id="7" evt="Gearing_Supply.MOSFET.Close_Locking"/></seq>
		<seq><tr id="11" evt="Gearing_Supply.Coupling_Unit.Close_Locking"/></seq>
	</result>

</seqgen>
```

Figure 22: Sequences leading to the target

Two sequences composed of a single supplying failure lead the system in the identified feared state. Thus, the requirement is not satisfied.

Step 3:

To fulfill the safety requirement, it is necessary to strengthen the system, for example by adding a monitoring component in the supplying equipment:

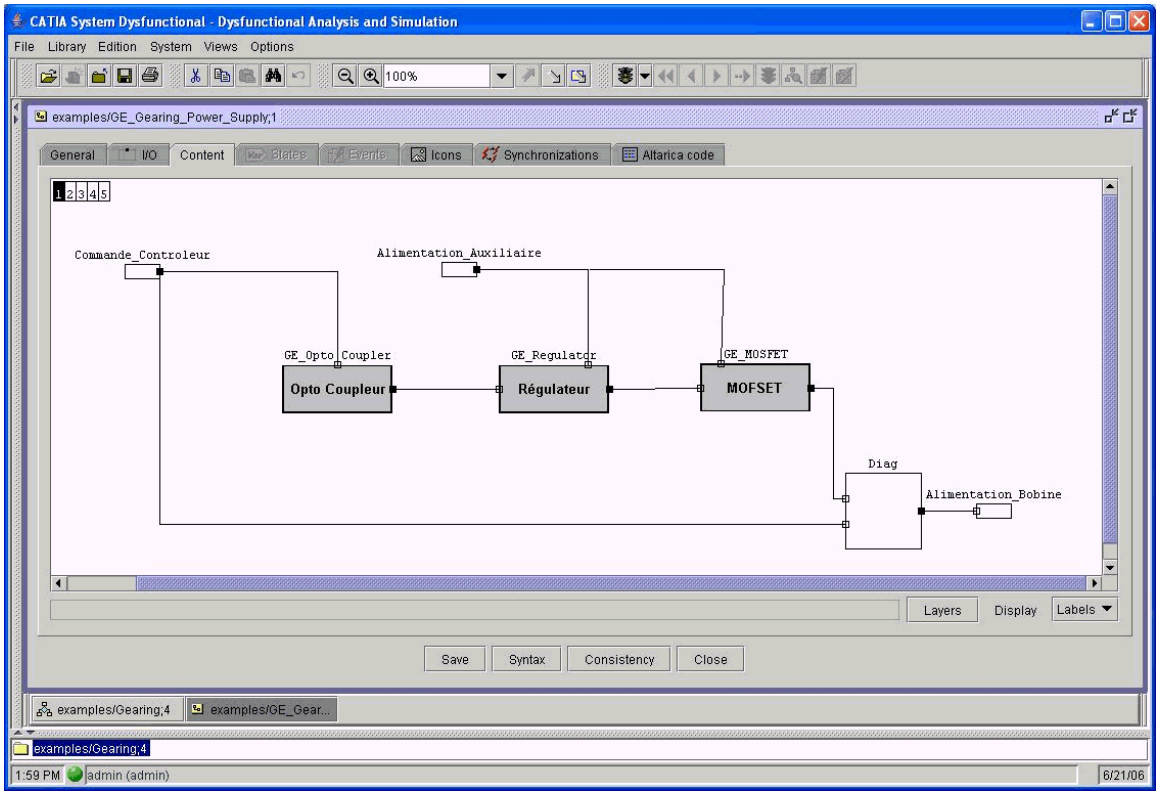

Figure 23: A new variant of the supplying

A new variant of the equipment and system are created, now user can compare the both architectures on a safety efficiency point of view.

In the updated equipment, a component called "Diag" is added. Its role consists in verifying the coherence between the command sent to the supplying, and the power provided to the clutch.

Step 4:

The sequence generator is launched on the new architecture variant.

The file contains no sequence with single failure. The "Diag" component has improved the safety aspect of the system and the requirement is now fulfilled.

It is possible to translate the sequences into a fault tree model by using the suitable output format. This processing is not presented here.

Modeling Determinist aspects

A model can contain three categories of events:

- ***Instantaneous events**: they are characterized by the Dirac law with a parameter (delay) equal to zero. Their priority is highest. If the condition of a transition is valid, this one is instantaneously triggered before any timed transitions. Instantaneous transitions don't require interaction with the user.*
- ***Deterministic events**: they are characterized by the Dirac law with a delay different from zero. Often, they are used to modelize functional events.*
- ***All other events**: i.e. events having a law different to Dirac, or without law. They are considered as stochastic events.*

By adding a deterministic delay on some events (transitions), it is possible to introduce a firing order according to the global behavior of the system. This is useful to make a transition firerable before another one.

Note: A scheduler is implemented in the simulation engine to take into account deterministic delays.

Example:

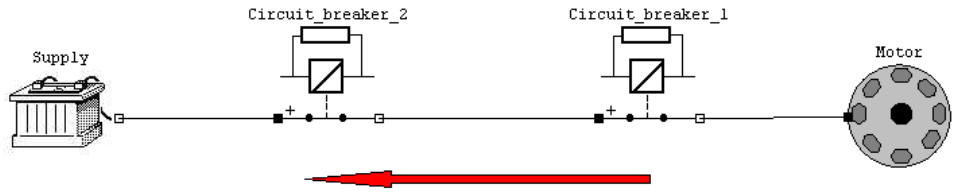

Figure 24: an example

This system is composed of one motor which can fail, one supply which can be struck down by a short circuit, and two circuit breakers protecting the supply from a short-circuit.
When the motor failed a short circuit is propagated to the supply. One of the both circuit breakers must protect the system.
Here, it can be useful to specify which circuit-breaker opens first:

- *To specify the short circuit propagation direction (first circuit breaker 1)*
- *To modelize sensitivity of the circuit-breaker (which can depends on the technology)*
- *To introduce a delay of circuit breaker opening…*

Thus, the time aspect may be introduced in the modeling.

Dirac laws

To take into account the timed delay in the modeling, it is necessary to activate the option in preferences:

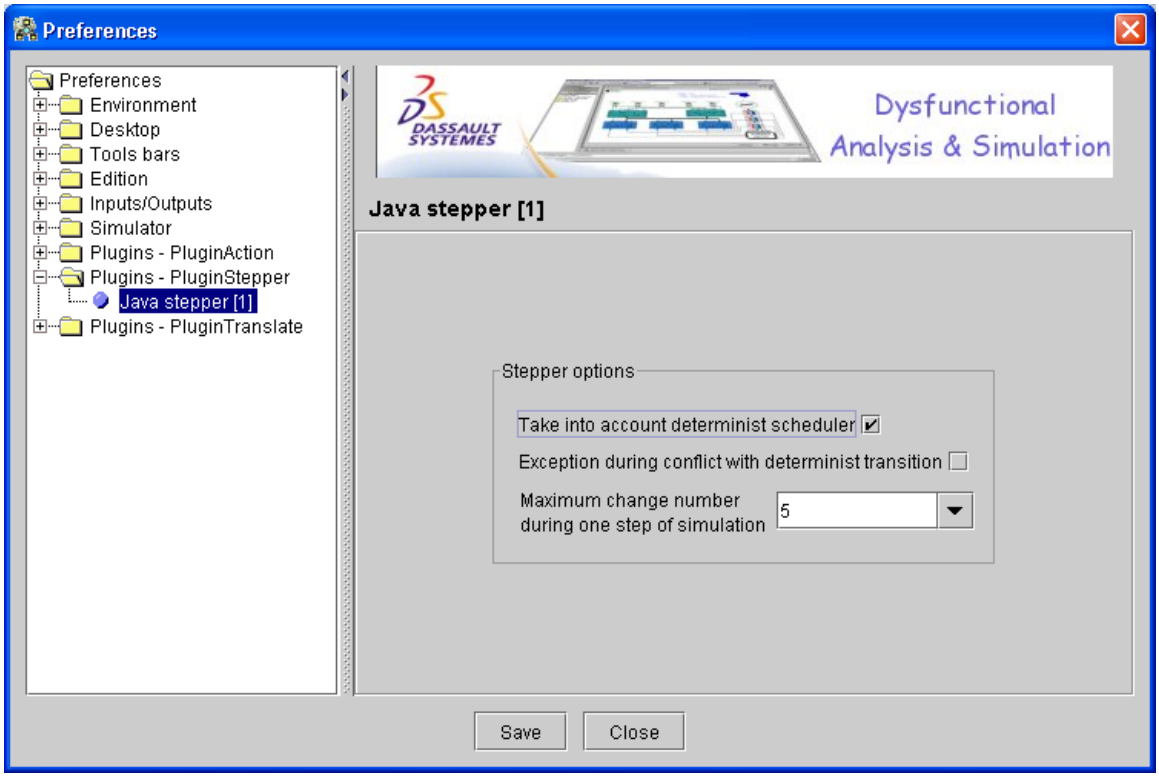

Figure 25: Option to activate the determinist scheduler

The option 'Take into account determinist scheduler' must be enabled.

Then, to introduce a delay to a transition, the law DIRAC is associated to the events. The syntax is:

Dirac(delay)

Thus, in the previous example, time constraints are added to the events 'protection' of the components Circuits_breaker_1 & Circuits_breaker_2:

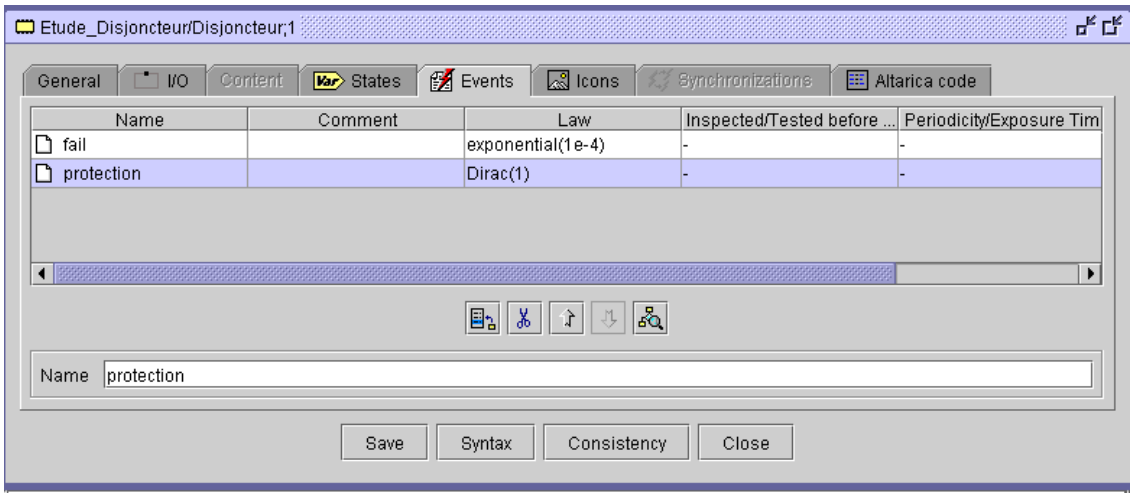

Figure 26: Specifying a Dirac law

We associate the law Dirac(1) to the event 'protection' of the component Circuit_breaker_1 and Dirac(2) to the event 'protection' of the component Circuit_breaker_2.

Now, if we launch the simulation and inject a failure 'Fail' in the motor, a short-circuit is propagated though the system. Then, we choose the event 'protection' to be fired :

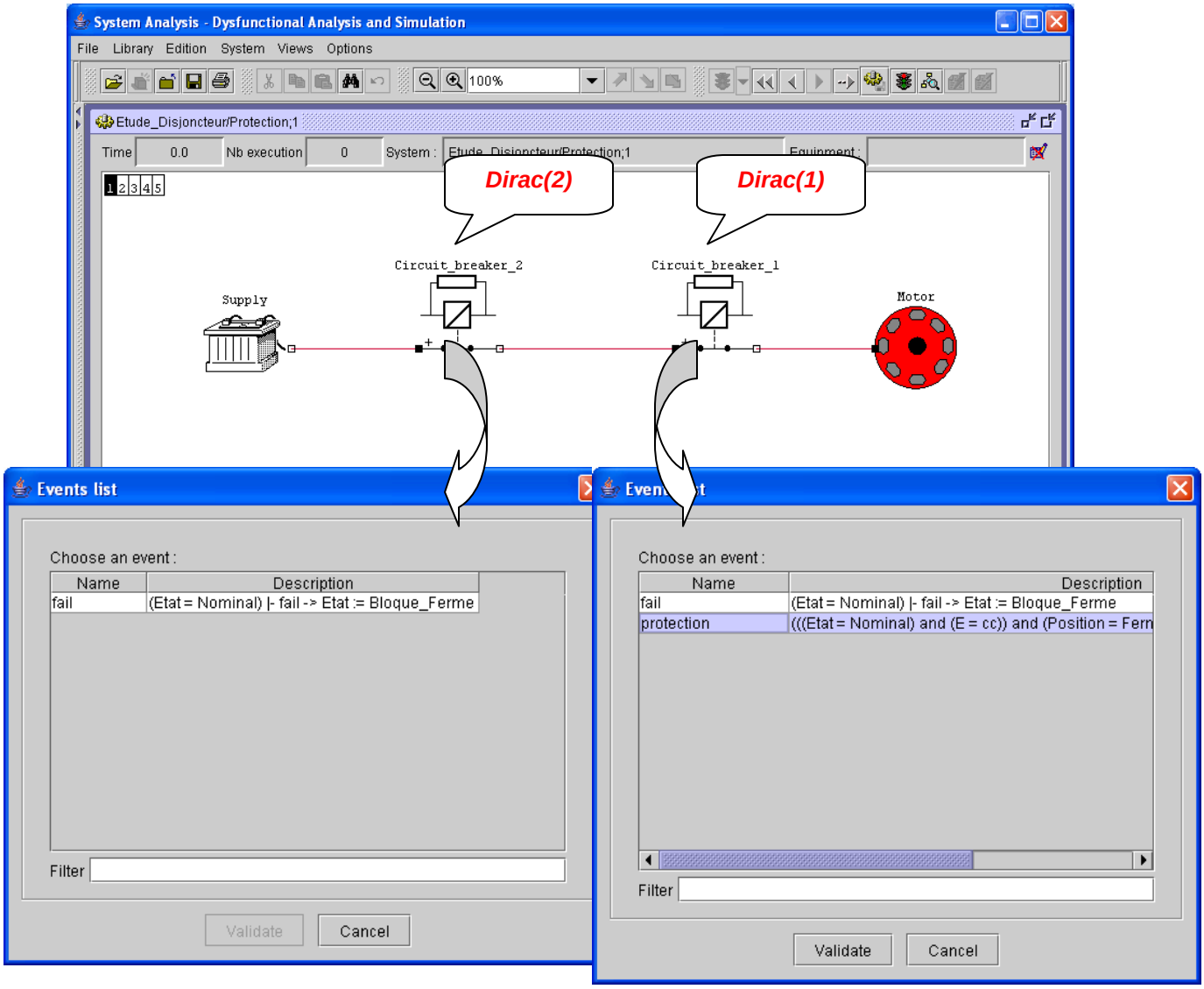

Figure 27: Simulation of the system

Let's remark that the event 'protection' of the component Circuit_breaker_2 is inhibited because its delay is higher than the first component.

Conflict of transitions

Sometimes, several transitions are in conflict:

- *when immediate transitions (with Dirac(0)) can be fired simultaneously,*
- *when deterministic transitions (Dirac with a delay different from zero) can be triggered in same time according to the scheduler.*

It is important to notice that the global behavior depends on the firing order of transitions. If a conflict occurs, the user must be also advised. To activate this warning, open the Preference windows (figure 24), and check that the option 'Exception during conflict with determinist transitions' is enabled.
Now, if a conflict occurs during a processing - for example in the previous example by choosing a same delay for the both events 'protection' - the following message is displayed when the conflict is detected:

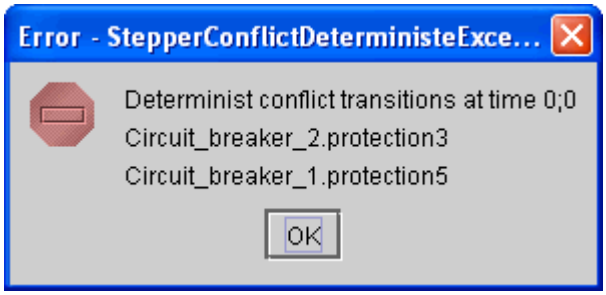

Figure 28: Detection of a conflict of transitions

Specifying priorities on events

It is possible to take of ambiguity when transitions are in conflict:

- *several immediate transitions (with Dirac(0) events)*
- *several deterministic transitions which can be fired simultaneously.*

*To do this, a priority may be associated to the events which are in conflict. These priorities are defined in external clauses by mean of the keyword **priority**.*

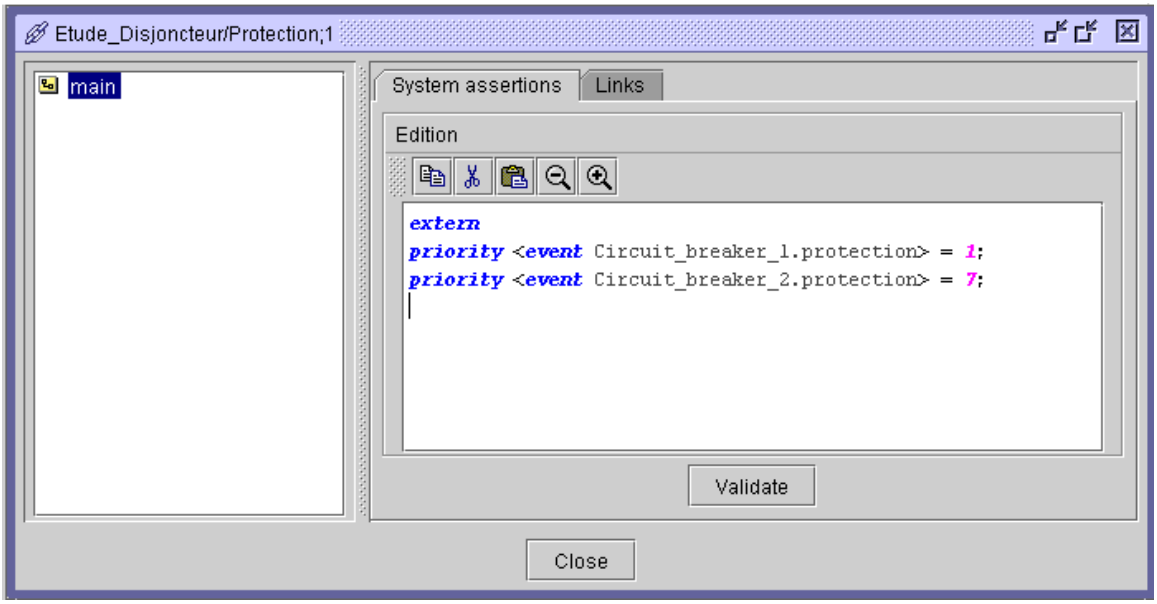

Figure 29: Event priority defining

The priority can be defined:

- *in the AltaRica code of the equipment which contain the components in conflict,*
- *in the system assertions*

When a conflict occurs, the event having the highest priority is fired first.

The syntax to define a priority is :

***Priority <event** component_name.event_name**> =** priority_value;*