ARBRES DE DEFAILLANCE DYNAMIQUES : UNE BIBLIOTHEQUE POUR LA NOUVELLE GENERATION D'ALTARICA !

DYNAMIC FAULT TREES: A LIBRARY FOR ALTARICA NEXT GENERATION!

PERROT Benoît, PROSVIRNOVA Tatiana, RAUZY Antoine, SAHUT D'IZARN Jean-Philippe
Dassault Systèmes
10, rue Marcel Dassault
78140 Vélizy Villacoublay
Tel.: (+33)1 61 62 84 69
Benoit.Perrot@3ds.com

Résumé

Dans cette communication, nous montrons comment traduire les arbres de défaillance dynamiques en la nouvelle génération d'AltaRica (ce nouveau langage de modélisation pour la sûreté de fonctionnement, dérivé du langage AltaRica est décrit en détails par ailleurs). Pour cela, nous analysons les concepts sous-jacents aux arbres de défaillance dynamique, et nous donnons une sémantique des arbres de défaillance dynamiques en termes de systèmes de transitions gardées. Cette représentation nous permet d'étendre les arbres de défaillance dynamiques au delà du cadre strictement markovien auxquels ils sont aujourd'hui restreints, et d'appliquer l'arsenal algorithmique développé pour la nouvelle génération d'AltaRica.

Summary

The purpose of this communication is to translate dynamic fault trees in AltaRica next generation (this new modeling language for safety analysis is described in detail in another communication). To do this, we will analyze the basic concepts of dynamic fault trees, and we will describe a semantic for dynamic fault trees in a guarded transition system. This representation will enable us to extend dynamic fault trees outside of the strict markovian perimeter where they were bound until now. We will also apply all the algorithmic capacities developed for the next generation of AltaRica.

Introduction

Les arbres de défaillance dynamiques sont des arbres de défaillance classiques enrichis de portes extra-logiques permettant de modéliser des phénomènes dépendants de l'ordre des événements, tels que les redondances froides ou la survenue de pannes. Jusqu'ici, ce formalisme est resté essentiellement théorique dans la mesure où d'une part la sémantique des portes dynamiques n'a pas été complètement clarifiée et d'autre part les outils algorithmiques manquent encore pour traiter des modèles de taille industrielle.

Le formalisme des systèmes de transitions gardées (GTS, *Guarded Transition System*) introduit par A. Rauzy en [3], se veut une généralisation des réseaux de Petri, des diagramme-blocs de fiabilité (DBF), et du modèle du parallélisme Arnold-Nivat. Ils reprennent le concept de circulation de valeurs des DBF, les concepts de composition et de synchronisation du modèle Arnold-Nivat, sans introduire de cout de modélisation supplémentaire. L'introduction d'un calcul de valeurs par point fixe permet en outre de modéliser simplement des réseaux bouclés. Les algorithmes de traitement des réseaux de Petri ou des machines à état finies se transposent facilement dans le monde des GTS – là encore sans modification notable de complexité ; l'algorithme de compilation vers arbre de défaillance décrit en [5] est immédiatement applicable. Plus simplement, la réalisation d'un simulateur stochastique de GTS ne présente aucune difficulté théorique. Afin de rendre plus aisée la description de GTS, nous avons défini le langage formel AltaRica nouvelle génération (dérivé d'AltaRica), qui permet d'écrire simplement des GTS complexes au moyen de construction de haut niveau ; la nouvelle génération d'AltaRica est décrite dans une autre communication du Lambda/Mu 2010 [1].

L'objectif de cette communication est de montrer qu'il est possible de représenter un arbre de défaillance dynamique par un GTS, et de le construire au moyen d'une bibliothèque écrite en langage AltaRica nouvelle génération. Le bénéfice de cette traduction est multiple : premièrement, elle permet de trier ce qui est réellement utile ou en tous cas différentiant dans les concepts mis en œuvre dans les arbres de défaillance dynamiques ; deuxièmement, elle permet d'étendre ces derniers au delà du cadre strictement markovien auquel se restreignent par la force des choses leurs utilisateurs ; troisièmement, elle permet d'appliquer aux arbres de défaillance dynamique l'arsenal algorithmique développé sur les GTS, notamment :

- Le simulateur graphique pas à pas, pour observer l'impact des événements sur un modèle,
- Le générateur de séquences, pour obtenir des séquences d'événements respectant différents critères de longueur, probabilité, criticité, ou autres,
- Le simulateur stochastique
- Le compilateur vers arbre de défaillance statique (sous certaines conditions)
- Le compilateur vers chaine de Markov (sous certaines conditions).

L'article est organisé comme suit. En première section, nous donnons un aperçu du langage AltaRica nouvelle génération. Dans une deuxième section, nous proposons une implémentation d'une bibliothèque d'arbres de défaillance statiques, que nous étendons dans la troisième section aux arbres de défaillance dynamiques. La quatrième section décrit une mise en œuvre de cette bibliothèque sur un exemple concret.

A propos d'AltaRica nouvelle génération

Le langage de modélisation AltaRica nouvelle génération [1] est le successeur direct d'AltaRica [2]. Il en hérite l'objectif premier : proposer aux ingénieurs fiabilistes un langage de haut niveau, leur permettant de modéliser des systèmes suivant leurs architectures logiques. Les outils de traitement associés au langage s'occuperont de réaliser les calculs fiabilistes, voire de revenir aux arbres de défaillance si désiré.

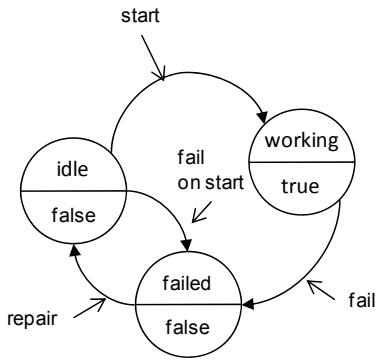
La nouvelle génération d'AltaRica s'appuie sur le formalisme des systèmes de transitions gardées grâce auquel les comportements opérationnels et de mise en défaillance d'un composant s'expriment en termes de variables, d'instructions et de transitions entre états déclenchées par des événements aléatoires. À ce pouvoir d'expression « élémentaire », viennent s'ajouter des constructions de haut niveau – reprises du langage Modelica [4] et empruntées aux langages de programmation généraux du type C++ ou Java, permettant la description de systèmes hiérarchiques par composition, la réalisation de bibliothèques de composants réutilisables. Les modèles fiabilistes peuvent désormais rester très proches de l'architecture logique du système étudié, ce qui permet leur partage et leur vérification par des non spécialistes.

Cette nouvelle itération d'AltaRica apporte des solutions aux difficultés rencontrées par les utilisateurs du dit langage. En affirmant son orientation objet, AltaRica nouvelle génération permet une plus grande réutilisabilité des modèles, et ouvre la voix à une meilleure cohabitation interdisciplinaire. L'introduction de la loi de choix sur les transitions, donne un cadre plus général et mieux défini pour la gestion des transitions activables simultanément en AltaRica nouvelle génération. Le calcul des assertions, désormais par point fixe, autorise la modélisation de réseaux bouclés.

De la même manière que son prédécesseur, l'intérêt de la nouvelle génération d'AltaRica ne tient pas tant dans son pouvoir d'expression que dans les outils de traitement associés. Un modèle décrit en ce langage est compilé en un système de transitions gardées « brute ». Ce GTS sert ensuite d'entrée au simulateur pas à pas (pour la mise au point du modèle), au générateur de séquences critiques d'événements, au simulateur stochastique. Sous certaines conditions, il peut être compilé en arbres de défaillance (idem) ou en chaîne de Markov. Cette approche permet une flexibilité accrue par rapport à la réalisation d'arbres par des experts, donc une plus grande réactivité aux évolutions du système considéré. Les études de sûreté de fonctionnement peuvent être alors initiées, prises en compte et entretenues dès les premières étapes de la conception.

Exemple

Un composant qui pourrait être démarré, défaillir et être réparé, peut se représenter ainsi (à gauche son graphe d'états, à droite son écriture en AltaRica nouvelle génération):




```
type Status = enumeration(idle, working, failed);

class StartableRepairable
  Status s(init = idle);
  Boolean output(reset = true);
transitions
  stop: s == working -> s := idle;

  start: s == idle -> s := working;
  fail_on_start: s == idle -> s := failed;

  fail: s == working -> s := failed;

  repair: s == failed -> s := idle;
assertions
  output := (s == working);
end;
```

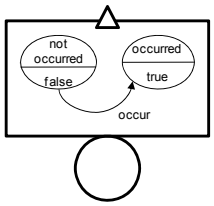
Chaque transition (« fail_on_start », « fail », etc.) peut être décorée par une loi de délai, exprimant la probabilité de son occurrence. La loi de délai prend la forme d'une fonction mathématique quelconque, posée sur le temps écoulé, sur les valeurs des variables du GTS, etc.

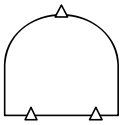
Représentation des Arbres de Défaillance Statiques en AltaRica nouvelle génération

Les arbres de défaillance sont constitués d'événements de base (représentant en général la survenue d'une panne) liés entre eux par des portes logiques (e.g. « et », « ou »). Ils permettent d'exprimer une défaillance particulière d'un système (dit « événement redouté ») comme une expression Booléenne de la défaillance de ses composants. En utilisant un moteur de calcul adéquat, il est alors possible de déterminer des ensembles de coupes minimales, d'estimer la probabilité d'occurrence de cet événement redouté, etc.

Bien que conçu pour s'abstraire notamment de ce formalisme, il est tout à fait possible et extrêmement simple de représenter un arbre de défaillance en AltaRica nouvelle génération. Il suffit de définir les quelques classes nécessaires pour décrire les événements de base et les portes logiques souhaitées (Figure 1), puis d'instancier ces dernières à volonté dans un arbre particulier (Figure 3).

	`class Event`
	`  Boolean output(reset = false);`
	`end;`

	`class BasicEvent`
	`  extends Event;`
	`  Boolean occured(init = false);`
	`transitions`
	`  occur:`
	`  not occured -> occured := true;`
	`assertions`
	`  output := occured;`
	`end;`

	`class AndGate`
	`  extends Event;`
	`  Boolean input_1(reset = false);`
	`  Boolean input_2(reset = false);`
	`assertions`
	`  output := input_1 and input_2;`
	`end;`

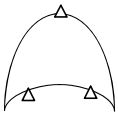
	`class OrGate`
	`  extends Event;`
	`  Boolean input_1(reset = false);`
	`  Boolean input_2(reset = false);`
	`assertions`
	`  output := input_1 or input_2;`
	`end;`

Figure 1 La bibliothèque « Arbre de Défaillance » en AltaRica nouvelle génération

Mise en application

Soit le système ci-dessous :

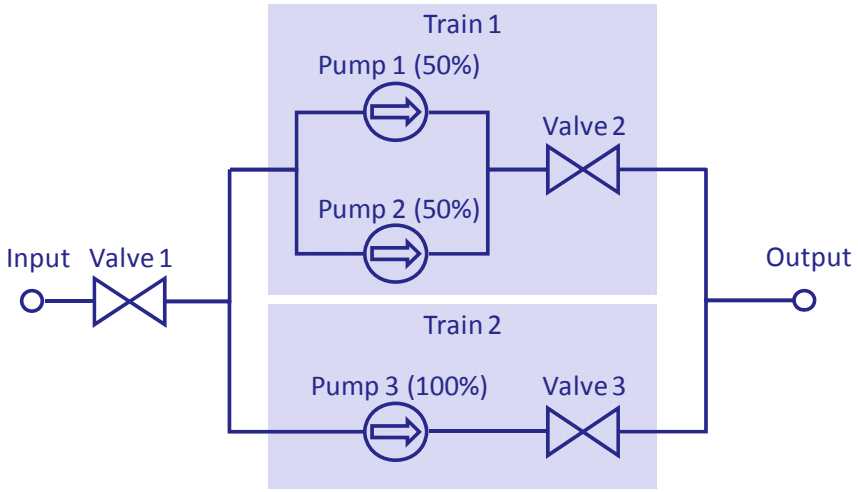


Figure 2 Exemple : un système hydraulique simple

Il est possible de dessiner un arbre de défaillance statique dans un atelier d'édition AltaRica nouvelle génération en mettant en œuvre la bibliothèque décrite dans cette section :

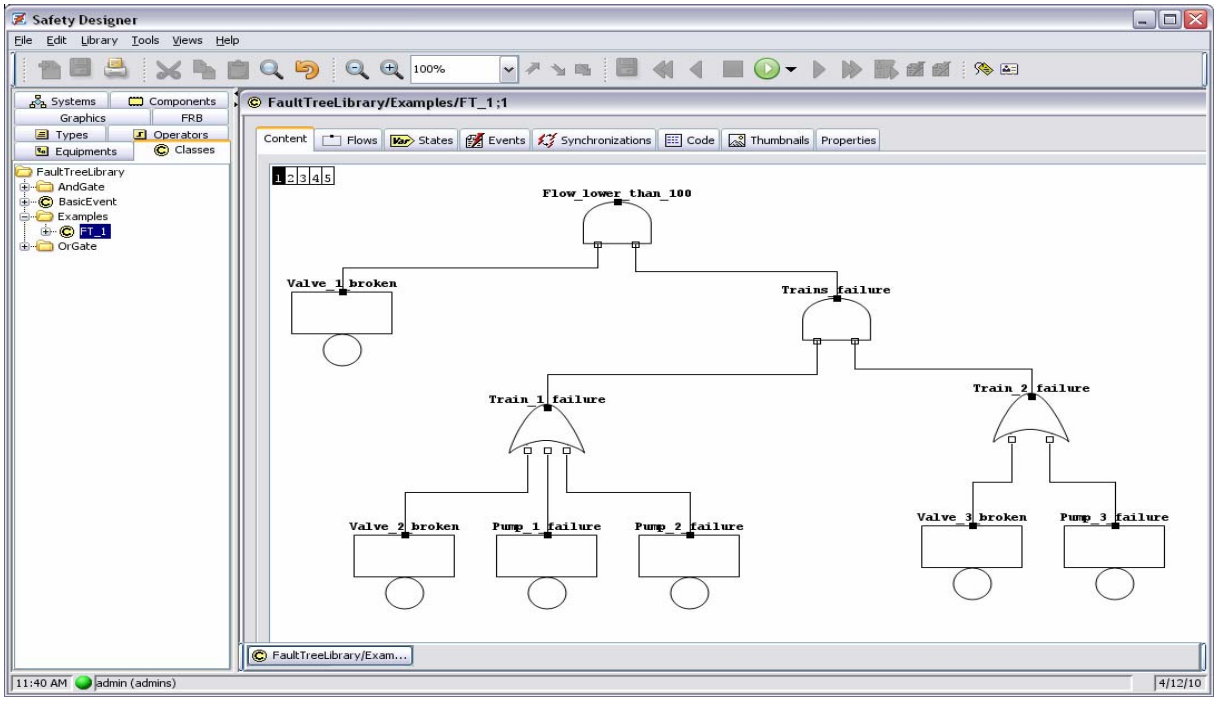


Figure 3 Exemple d'utilisation de la bibliothèque Arbres de Défaillance Statiques

Il est aussi possible de revenir à un arbre de défaillance « classique » grâce au compilateur vers arbre de défaillance, pour l'analyser dans un atelier d'édition et de calcul spécialisé, tel qu'Aralia Fault Tree Analyzer :

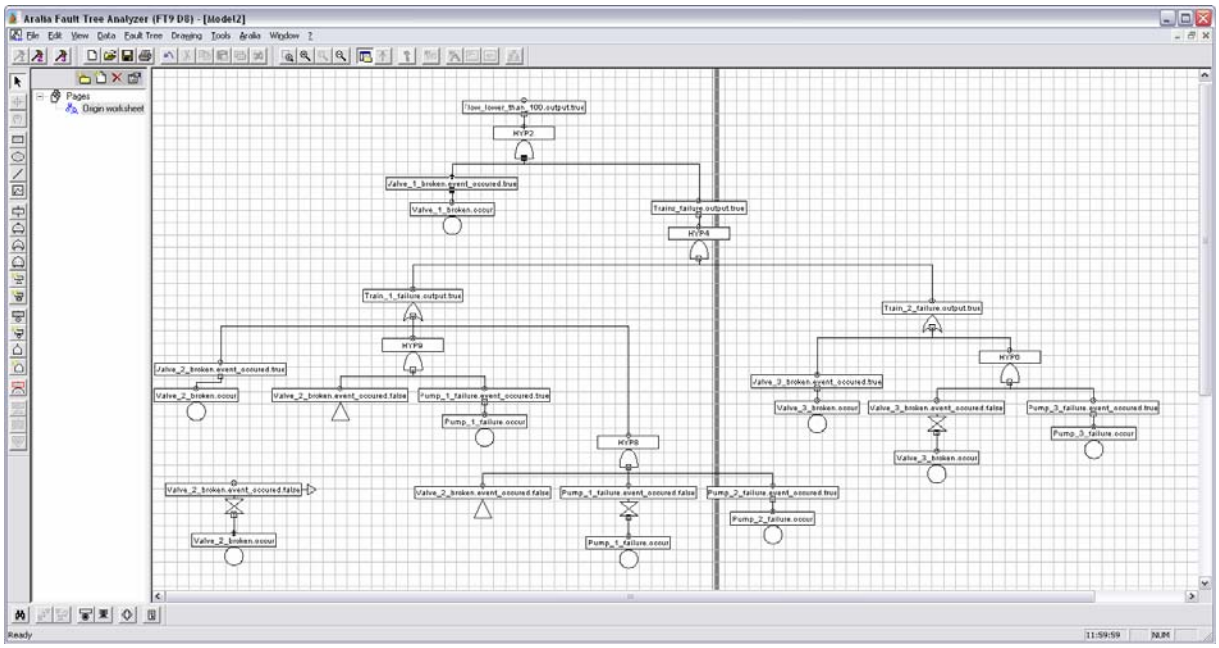


Figure 4 Arbre de défaillance « classique » obtenu par compilation

Le point clé est que n'importe quelle classe AltaRica nouvelle génération étendant la classe « Event », peut être utilisée dans un tel diagramme. C'est cette observation qui nous invite à construire les composants AltaRica nouvelle génération nécessaires pour représenter les arbres de défaillance dynamiques.

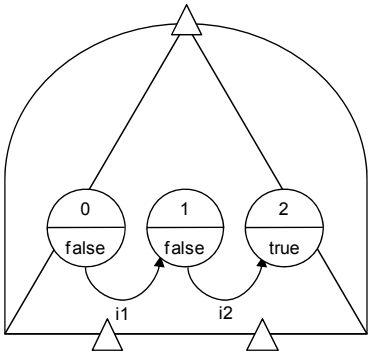
Représentation des Arbres de Défaillance Dynamiques en AltaRica nouvelle génération

La porte PAND

La porte PAND (*Priority And*) a pour vocation de capturer une séquence d'événements. Elle dispose d'un certain nombre d'entrées qui doivent s'activer dans un ordre précis [6]. Lorsque cet ordre n'est pas respecté, plusieurs comportements peuvent être considérés :

a) Ignorance du non respect de la séquence (l'entrée irrespectueuse devra réapparaitre au moment opportun pour activer la porte) ;
b) Redémarrage de la séquence (la mémoire des entrées qui sont déjà apparues, dans le bon ordre, est perdue – elles doivent à nouveau survenir, dans le bon ordre) ;
c) Désactivation définitive de la porte.

Ces différentes déclinaisons peuvent être représentées par les GTS suivants :



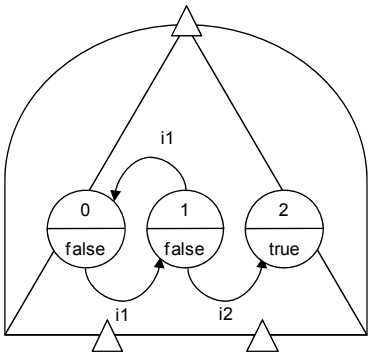

```
class PANDGate_ignore
  extends Event;
  Integer s(init = 0);

  Boolean i1(reset = false);
  Boolean i2(reset = false);

transitions
  to_1(delay = 0): s == 0 and i1 -> s := 1;
  to_2(delay = 0): s == 1 and i2 -> s := 2;

assertions
  output := (s == 2);
end;
```

Figure 5 Porte PAND (ignorance)

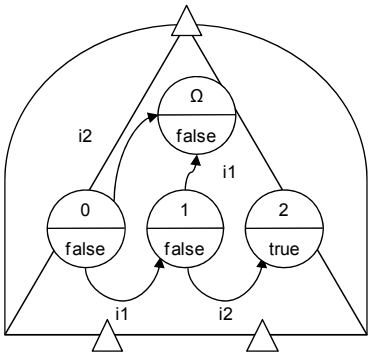



```
class PANDGate_reset
  extends PANDGate_ignore;

transitions
  reset_on_1(delay=0): i1 and s != 0 -> s := 0;

end;
```

Figure 6 Porte PAND (remise à zéro)




```
class PANDGate_strict
  extends PANDGate_ignore;

transitions
 stuck_on_1(delay=0): i1 and s != 0 -> s := -1;
 stuck_on_2(delay=0): i2 and s != 1 -> s := -1;

end;
```

Figure 7 Porte PAND (stricte)

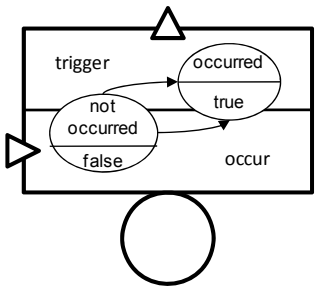
La porte FDEP

La porte FDEP (*Functional Dependency*) consiste en une entrée « gâchette » et un ensemble d'entrées de composants dits « dépendants ». La porte FDEP n'a pas de sortie. Lorsque la gâchette est activée, les composants attachés à la porte sont désactivés [6].

Cette définition décrit implicitement une nouvelle classe de portes événementielles de base, dont la sortie peut être activée de deux manières :

- lorsque l'événement de base associé survient,
- ou lorsque la gâchette est activée.

Cette nouvelle sorte d'événements de base peut être représentée comme suit :




```
class FDEP_BasicEvent
  extends BasicEvent;
  Boolean trigger_cmd(reset = false);

transitions

trigger(delay = 0):
  trigger_cmd and not occurred -> occurred := true;

assertions
  output := occured;
end;
```

Figure 8 Evénements de base attachés à une porte FDEP

L'équivalence entre les représentations est immédiate – les liens de dépendance sont simplement rendus apparents :

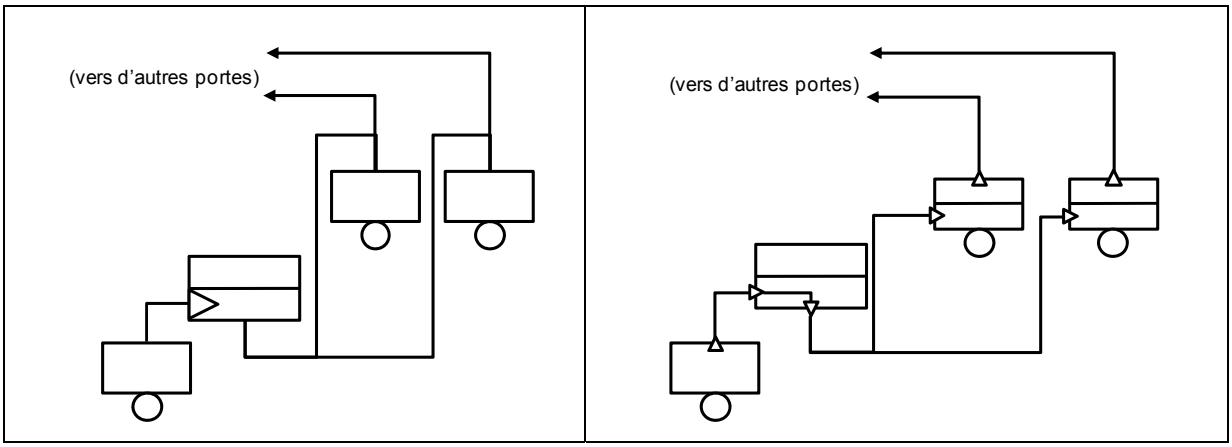
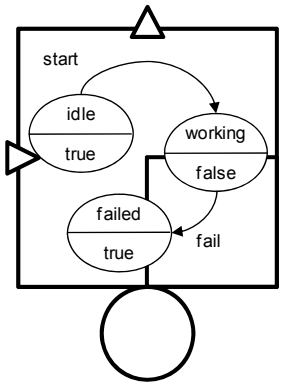


Figure 9 Apparition des liens de dépendance introduits par une porte FDEP

La porte SPARE

La porte SPARE possède une entrée pour un composant « principal » et des entrées pour les composants de secours. Il est dit que chaque entrée ne peut être liée qu'à une construction de type événement de base. Au démarrage, le composant principal est en fonctionnement et les composants de secours sont en attente. Lorsque le composant principal défaille, le premier de secours disponible est activé, et ainsi de suite. La porte SPARE défaille quand tous les composants qui lui sont attachés ont défaillis.

Ici encore, la définition fait implicitement intervenir des événements de base particuliers, correspondants à des composants démarrables sur demande. Ils peuvent être représentés comme suit :




```
type Status = enumeration(idle, working, failed);

class SPARE_BasicEvent
  extends Event;

  Status status(init = idle);

  Boolean startable(reset = false);
  Boolean start_cmd(reset = false);

transitions

start(delay = 0):
  start_cmd and status == idle -> status := working;

fail:
  status == working -> status := failed;

assertions
  output := (status == failed);
end;
```

Figure 10 Evénements de base attachés à une porte SPARE

La représentation d'une porte SPARE est alors immédiate :

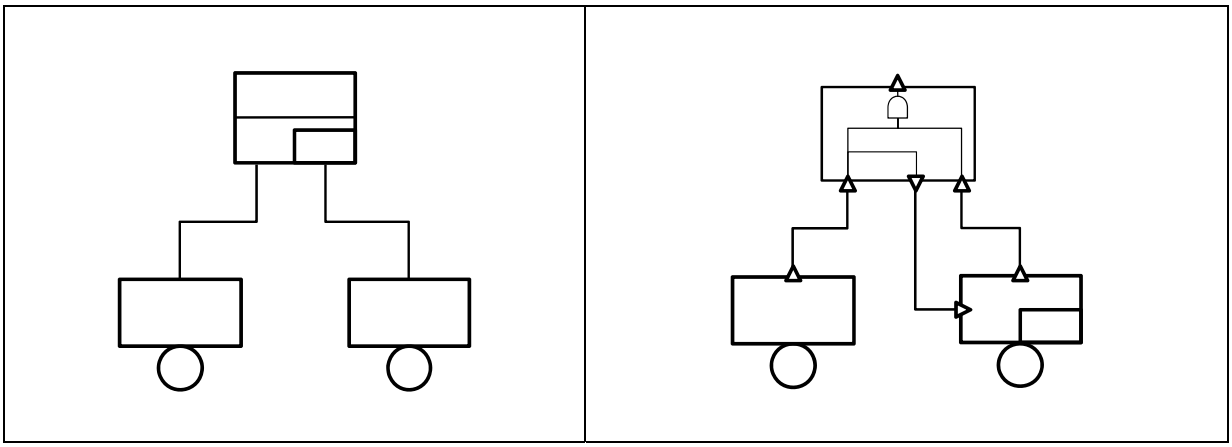


Figure 11 Apparition des liens de dépendance introduits par une porte SPARE

Mise en application

Description du système

Le système traité en [7] est repris ici (l'opérateur est ignoré). Le HECS (Figure 12) est un ordinateur hypothétique, constitué de deux processeurs A1 et A2 en redondance chaude et d'un processeur de secours A en redondance froide. Le HECS possède 5 puces mémoires (de M1 à M5). Ces puces mémoires sont connectées aux bus via deux unités d'interfaçage ; si une unité d'interfaçage défaille, les puces qui lui sont rattachées deviennent inaccessibles. La puce M3 est partagée entre les deux unités – les deux unités doivent défaillir pour que M3 ne soit plus utilisable. Le système intègre aussi une console et un logiciel d'exploitation.

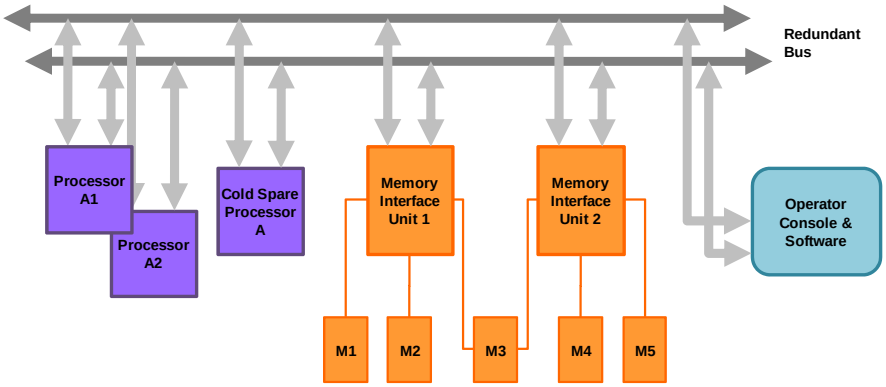
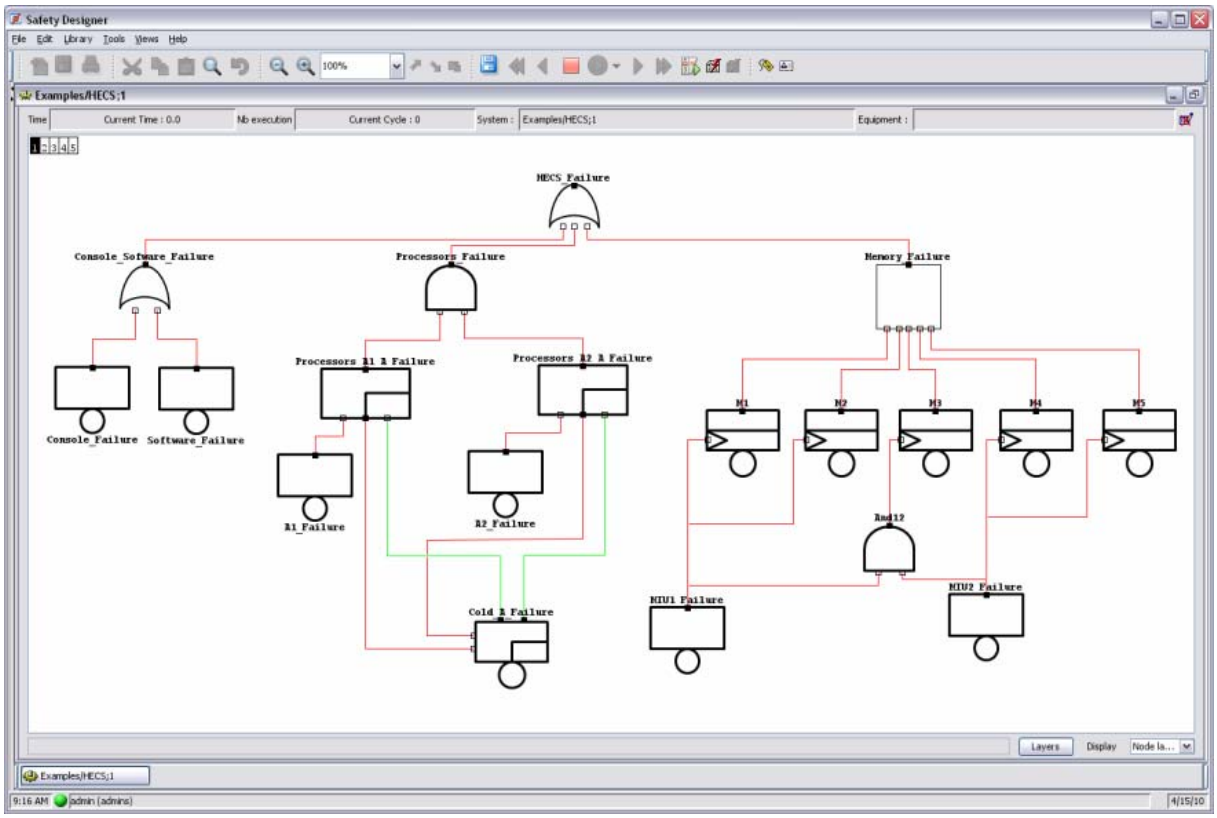


Figure 12 HECS : *Hypothetical Example Computer System*

L'exigence de fonctionnement du HECS est qu'au moins un des trois processeurs fonctionne, qu'au moins une des trois puces mémoires soit accessibles, qu'au moins un des bus soit opérationnel, et que la console et le logiciel d'exploitation fonctionnent.

Modélisation en arbre de défaillance dynamique via AltaRica nouvelle génération

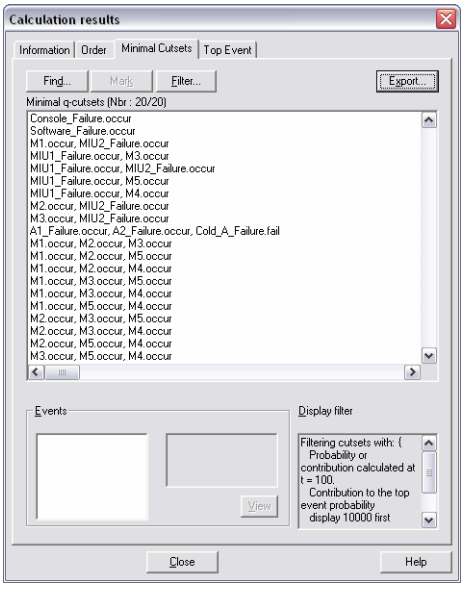
Ainsi que proposé dans l'article dont ce modèle est originaire, nous pouvons constituer un arbre de défaillance dynamique pour étudier la fiabilité du système. Nous le réalisons dans un atelier AltaRica nouvelle génération grâce à la bibliothèque décrite dans cet article (attention à la logique inversée des arbres de défaillance pour la lecture du diagramme : un fil rouge pour « faux » signifie que le composant est disponible, un fil vert pour « vrai » signifie que le composant est indisponible) :



Au démarrage, tous les composants fonctionnent, le processeur A est en attente (donc indisponible, d'où la couleur verte des fils sortants).

Etude du système

Grace aux outils de traitement GTS appliqués à ce modèle, il est notamment possible d'obtenir les ensembles des coupes minimales du système :



- Défaillance de la console
- Défaillance du logiciel d'exploitation
- Défaillance de MIU1 et de : M3 ou M4 ou M5
- Défaillance de MIU2 et de : M1 ou M2 ou M3
- Défaillance de MIU1 et MIU2
- Défaillance de : A1 et A2 et A
- Défaillance d'au moins 3 puces mémoires

Elles peuvent bien sûr être rejouées dans le simulateur graphique.

Il est à noter que cette représentation en GTS nous autorise également à utiliser n'importe quelle distribution de probabilité pour les événements de panne. Certains outils pourraient alors interdire le traitement (e.g. compilateur vers chaine de Markov), mais le simulateur stochastique permettra toujours d'étudier le modèle.

Conclusion

Nous avons construit une bibliothèque AltaRica nouvelle génération d'arbres de défaillance dynamiques, permettant d'exprimer de tels arbres en systèmes de transitions gardées, et même de les représenter graphiquement dans un atelier d'édition de modèles AltaRica nouvelle génération. Nous avons illustré son utilisation sur un exemple concret, montrant également l'utilisation de quelques-uns des outils de traitement des GTS.

Références

[1] B. Perrot, T. Prosvirnova, A. Rauzy, J. P. Sahut d'Izarn, 2010, Introduction au nouveau langage de modélisation pour la sûreté de fonctionnement : AltaRica nouvelle génération, actes du congrès Lambda Mu 17, La Rochelle, octobre 2010.

[2] A. Arnold, A. Griffault, G. Point, A. Rauzy, 2000, The AltaRica Language and its Semantics. Fundamenta Informaticae, 34:109–124.

[3] A. Rauzy, 2008, Guarded transition systems: A new states/events formalism for reliability studies, in Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability.

[4] 2010, Modelica® - A Unified Object-Oriented Language for Physical Systems Modeling – Language Specification, http://www.modelica.org/

[5] A. Rauzy, Mode Automata and their Compilation into Fault Trees, 2002, in Reliability Engineering & System Safety, Volume 78, Issue 1, Pages 1-12.

[6] H. Boudali, P.Crouzen and M. Stoelinga, 2007, A Compositional Semantics for Dynamic Fault Trees, in 5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07), October 22-25, 2007, Tokyo, Japan. pp. 441-456. Lecture Notes in Computer Science 4762. Springer. ISSN 0302-9743 ISBN 978-3-540-75595-1.

[7] R. Manian, J. B. Dugan, D. Coppit and K. J. Sullivan, 1998, Combining Various Solution Techniques for Dynamic Fault Tree Analysis of Computer Systems, in The 3rd IEEE International Symposium on High-Assurance Systems Engineering.