Vérication des EFFBDs : Model checking en Ingénierie Système

Seidner Charlotte

▶ **To cite this version:**

Seidner Charlotte. Vérication des EFFBDs : Model checking en Ingénierie Système. Modélisation et simulation. Université de Nantes, 2009. Français. NNT : . tel-00440677

HAL Id: tel-00440677

https://theses.hal.science/tel-00440677v1

Submitted on 11 Dec 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université de Nantes

École Doctorale STIM

« Sciences et Technologies de l'Information et de Mathématiques »

Année 2009

Thèse de Doctorat de l'Université de Nantes

Spécialité : Automatique et Informatique Appliquée

Présentée et soutenue publiquement par :

Charlotte Seidner

le 3 novembre 2009

à l'École Centrale de Nantes

Vérification des EFFBDs : Model checking en Ingénierie Système

Jury :

Président :	C. André	Professeur, Université Nice Sophia Antipolis (I3S)
Rapporteurs :	F. Kordon	Professeur, Université P. & M. Curie (LIP6)
	F. Vernadat	Professeur, INSA Toulouse (LAAS)
Examinateurs :	J.-P. Lerat	P.D.G. de Sodius
	É. Niel	Professeur, INSA Lyon (Ampère)
	O. (H.) Roux	Maître de Conférences HDR, IUT de Nantes (IRCCyN)
Invité :	J.-L. Wippler	Ingénieur système, C-S

Directeur de thèse : Olivier (H.) Roux
Laboratoire : Institut de Recherche en Communications et en Cybernétique de Nantes
Composante : IUT de Nantes

N° ED : 503-068

Remerciements

En premier lieu, je tiens à remercier très vivement Fabrice Kordon et François Vernadat qui ont accepté d'être les rapporteurs de cette thèse, malgré leur calendrier déjà bien chargé. Toute ma reconnaissance également à Charles André, Éric Niel et Jean-Luc Wippler qui ont bien voulu faire partie de mon jury.

C'est avec toute ma gratitude que je remercie Olivier Roux qui, tout au long de cette formidable aventure, a su être un directeur de thèse enthousiaste, toujours à l'écoute et prêt à me soutenir dans les moments de doutes. Autant de qualités qui s'appliquent également à Jean-Philippe Lerat : je le remercie vivement de sa confiance et de son chaleureux accueil au sein de Sodius.

J'en profite en outre pour remercier l'ANRT qui, par le biais de la Cifre, a permis le financement de mes travaux de recherche. Qui dit Cifre, dit bien sûr laboratoire de recherche *et* partenaire industriel. Deux mondes parallèles (mais pas si différents que ça, au fond), deux expériences enrichissantes et surtout deux fois plus de rencontres mémorables ! J'espère donc n'oublier personne ; toutes mes excuses par avance à ceux que j'aurais omis...

Côté IRCCyN, je remercie vivement Jean-François Lafay et Michel Malabre, ses directeurs successifs, ainsi qu'Yvon Trinquet, responsable de l'équipe *Systèmes Temps Réel* pour leur accueil. Un grand merci également aux « anciens » (qui me pardonneront, j'espère, ce terme affectueux !), Jean-Pierre, Pierre (et nos mémorables discussions de ferrovipathes amateurs de spadassins avunculaires), Jean-Luc, David, Richard et tous les autres ! Je remercie également les « moins anciens » : Florent, Mikaël, Sébastien et bien sûr Didier qui sait mieux que quiconque combien sa présence et son soutien m'ont été précieux pendant cette dernière année.

Du fond du cœur, j'adresse tous mes remerciements à Isabelle, « notre grande soeur à tous » (et bien sûr à Babas), pour la délicate amitié dont elle m'honore ! Un merci ému également à Thibault, pour son indéfectible et inaltérable amitié, malgré les kilomètres et les tempêtes, et à Paul-André, tour à tour désopilant et délicat, et dont le chaleureux accueil et le rire tonitruant resteront à jamais dans mon cœur ! Je n'oublie pas non plus Jordan et sa bonne humeur communicative ainsi que mes compagnons doctorants, sans ordre particulier : Émilie, Louis-Marie, Rola, Di, Pedro, Matthias, Loïc, Eddy et Jonathan.

Enfin, j'adresse une pensée reconnaissante à tous mes « co-bureaux » successifs, Morgan, Médésu, Dalia et Céline, qui ont généreusement fait semblant d'apprécier la décoration de notre bureau. Je remercie tout particulièrement Morgan (qui a d'ailleurs contribué à ladite décoration) car c'est par son entremise que j'ai pu rencontrer à la fois mon futur directeur de thèse *et* mon partenaire industriel.

Côté Sodius, je tiens à remercier Thomas pour son soutien bienveillant malgré mes nombreuses impertinences. Je remercie également Albin, Mickaël et Vincent dont l'aide et la présence me furent plus d'une fois très précieux ! J'adresse aussi une pensée reconnaissante à Laurent et Éric A. pour leurs conseils avisés d'anciens « thésards » ainsi qu'à Véronique et Jean-Yves pour leur diligente efficacité. Je remercie également Valéry, Yann (et son « siamois » Sébastien, qui me pardonnera, j'espère, cette nouvelle impertinence) et Anas. Grâce à eux, j'ai pu surmonter les difficultés de l'implémentation et de l'intégration de mon travail dans les projets Kimono et Omotesc (sans parler de l'apprentissage de la rigueur dans le développement et la gestion de projet !). Enfin, je remercie Philippe et Nathalie qui m'ont cornaquée lors de mes premiers pas à Sodius, sans oublier mes compères parisiens, Régis et Laurie !

Je remercie également Alain Faisandier, chef de projet pour Kimono et Omotesc, et Thérèse Renard, co-fondateurs de MAP système, pour les passionnantes discussions que nous avons pu avoir sur l'analyse fonctionnelle, la sûreté de fonctionnement et plus généralement sur l'Ingénierie Système.

Quelques mots, encore, pour remercier Camille, qui fut le premier à me suggérer de faire un mastère de recherche, ainsi que Yann et son impressionnante érudition. Je voudrais également saluer Fahima, Frédérique et Denis, de la Faculté de Pharmacie de l'Université de Montréal, dont l'enthousiasme m'a largement incitée à poursuivre mon parcours académique en mastère puis en doctorat.

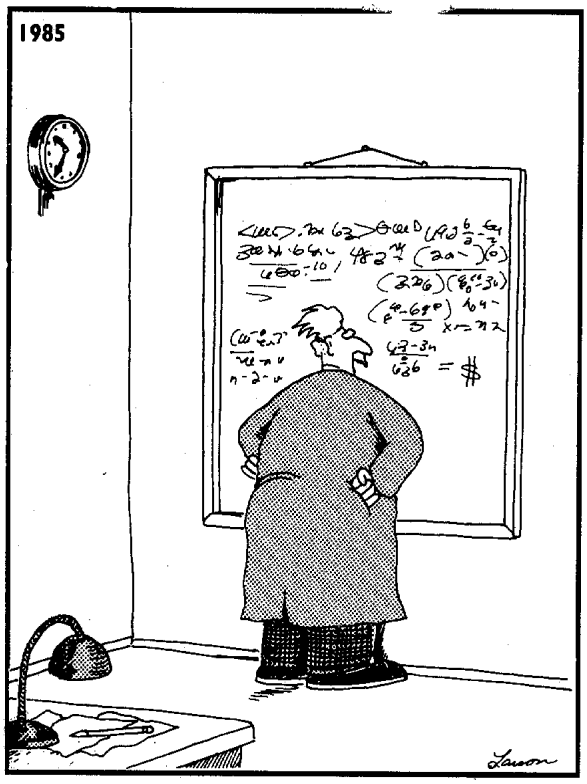
Plus généralement, je remercie bien sincèrement tous ceux qui m'ont encouragée, m'ont soutenue (et, il faut bien l'admettre, m'ont supportée) pendant toutes ces années. Mention spéciale à ma nombreuse famille éparpillée de par le monde, et tout particulièrement à Père (« cachant une grande bonté sous une apparence bourrue ») et Mère (« douce, timide et effacée ») qui a relu tous mes articles et mémoires, traquant inlassablement la moindre coquille !

Pour finir, et bien qu'ils n'en sauront jamais rien, j'aimerais remercier Boulet, Robert Charlebois, Jesse Sykes, DMST, GY!BE, Don et Seán de Constellation Records ainsi que mes pourvoyeurs en déviances vinyliques de Music Box (à Nantes) et l'Oblique (à Montréal).

À ceux qui sont partis

À ceux qui sont arrivés

À ceux qui sont là, tout simplement



Einstein discovers that time is actually money.

Gary Larson - The Far Side (1985)

MDA, SysML et UML sont des marques déposées de l'OMG.
CORE est une marque déposée de Vitech Corp.
MDWorkbench est une marque déposée de Sodius.

Table des matières

Table des figures

Sigles et Acronymes

AADL	*Architecture Analysis and Design Language*
AdD	Arbre de défaillances
AFIS	Association française d'Ingénierie Système
AMDEC	Analyse des modes de défaillance, de leurs effets et de leur criticité
CTL	*Computation Tree Logic*
DGA	Délégation Générale pour l'Armement
DoD	*Department of Defense*
FFBD	*Functional Flow Block Diagram*
FMECA	*Failure Modes, Effects and Criticality Analysis*
EFFBD	*Enhanced Functional Flow Block Diagram*
GMEC	*Generalized Mutual Exclusion Constraint*
GSPN	*Generalized Stochastic* Petri *Nets*
INCOSE	*International Council on Systems Engineering*
IS	Ingénierie Système
Kimono	Kit de modélisation des nouveaux outils
MDA	*Model Driven Architecture*
MDE	*Model Driven Engineering*
OMG	*Object Management Group*
Omotesc	Outils de modélisation pour la maîtrise de la testabilité des systèmes complexes
PEA	Programme d'études amont
PN	Passage à niveau
SdF	Sûreté de Fonctionnement
QBF	*Quantified Boolean Formula*
SE	*Systems Engineering*
STE	Sytème de transitions étiquetées
STT	Système de transitions temporisé
SysML	*Systems Modeling Language*
TA	*Timed Automaton*
TCTL	*Timed Computation Tree Logic*
TPN	*Time* Petri *Net*
UML	*Unified Modeling Language*
V&V	Vérification et Validation
XMI	*XML Metadata Interchange*
XML	*eXtensible Markup Language*

Chapitre 1

Introduction : Ingénierie Système et Vérifications Formelles

Résumé *Nous introduisons dans ce premier chapitre les concepts et les enjeux ainsi que les principaux processus de l'Ingénierie Système (IS). Nous verrons que chacun de ces processus concourt à la maîtrise de la sûreté de fonctionnement (*SdF*), qui joue une part prépondérante dans la conception et le développement des systèmes actuels. Pour garantir cette sûreté de fonctionnement, le recours à des méthodes formelles telles que le* model checking*, dont la puissance n'est plus à démontrer, semble une voie prometteuse.*

Cependant, l'utilisation et l'intégration de ces méthodes formelles ne se fait pas sans difficultés : leur complexité intrinsèque empêche ainsi de les employer directement dans un outil de conception, sous peine d'être pas ou mal appliquées par l'ingénieur système. Cette problématique générale de la vérification formelle en Ingénierie Système, au cœur de notre thèse Cifre*, nous a alors amenés à choisir un langage de modélisation adapté à la fois aux contraintes de l'IS et à celles du model checking. Il nous a ensuite fallu formaliser ces modèles de « haut-niveau » et les propriétés de* SdF*, ce qui nous a permis de définir leur transformation vers des modèles équivalents de « bas-niveau ». Ces étapes successives, détaillées dans les chapitres 3 à 7, nous ont alors permis de réaliser un outil de simulation et de vérification d'architectures fonctionnelles et dysfonctionnelles, déployé et utilisé industriellement.*

Sommaire

1.1 L'Ingénierie Système

Au cours des dernières décennies, le développement de grands projets d'ingénierie, toujours plus complexes, a mis en évidence la nécessité de disposer d'outils, de méthodes et de processus permettant d'en assurer la maîtrise, tout au long de leur cycle de vie. C'est ainsi que des organismes majeurs tels que la NASA ou le DoD *(Department of Defense)* américain ont cherché, dès les années 1960, à définir des méthodologies globales de conception, de production et d'exploitation adaptées à la complexité et à la criticité toujours croissantes de leurs programmes spatiaux et militaires.

Cette démarche, connue sous le nom *d'Ingénierie Système* (ou Ingénierie *des* Systèmes, IS et, en anglais, *Systems Engineering*, SE) a par la suite trouvé ses applications dans tous les secteurs de l'activité industrielle : aérospatiale et défense, mais aussi automobile, systèmes d'information, télécommunications, etc. Dans la suite de ce chapitre, nous présentons les définitions essentielles ainsi que les enjeux et les processus de l'Ingénierie Système. Cette présentation est volontairement très succinte : le lecteur est invité à consulter [41, 54] pour plus de détails.

1.1.1 Système et Ingénierie Système : définitions essentielles

Avant toute chose, il nous faut définir la notion de *système*. Plusieurs approches ont été proposées, offrant des points de vue complémentaires. Nous avons choisi de reprendre la définition donnée par J.P. Meinadier [54], qui présente une synthèse des ouvrages de référence [40, 82].

Définition 1.1 *Un* système *est un ensemble composite de personnels, de matériels et de logiciels organisés pour que leur interfonctionnement permette, dans un environnement donné, de remplir les missions pour lesquelles il a été conçu.*

Cette définition, passablement détaillée, appelle plusieurs commentaires :

- un système est *complexe* : il ne se réduit pas à la somme de ses constituants mais se définit également par leurs interactions ;
- un système est *hétérogène* : il manipule des constituants divers et fait donc appel à des concepts, des métiers et des standards différents ;
- un système est fait par et pour des *êtres humains* : son ingénierie doit donc mêler aux sciences de l'ingénieur des disciplines humaines telles que la psycho-ergonomie ou la gestion de projets.

Il convient d'ajouter qu'un système possède une *structure dynamique* : son bon fonctionnement dépend non seulement d'une bonne architecture organique mais également de la bonne synchronisation entre ses processus. En outre, un système évolue dans le temps, au cours d'un *cycle de vie* comprenant généralement les étapes (successives ou simultanées) suivantes :

- conceptualisation ;
- conception ;
- réalisation des constituants ;

- intégration ;
- exploitation et maintien en condition opérationnelle ;
- retrait de service.

Nous pouvons à présent définir de façon plus précise l'IS. Nous reprenons ci-dessous la définition donnée dans [54] et qui offre de nouveau une synthèse des ouvrages de référence.

Définition 1.2 L'Ingénierie Système *est un processus :*
- *collaboratif et interdisciplinaire de résolution de problèmes ;*
- *s'appuyant sur les connaissances, méthodes et techniques issues des sciences et de l'expérience ;*
- *mis en œuvre pour définir un système qui satisfasse un besoin identifié et qui soit acceptable par l'environnement*[1] *;*
- *défini sur tous les aspects du problème, dans toutes les phases de développement et de la vie du système.*

L'Ingénierie Système est ainsi une démarche méthodologique qui propose une approche globale (ou *holistique*) et pluridisciplinaire pour la conception et la mise en œuvre des systèmes complexes. Ajoutons qu'elle est promue au niveau international par l'INCOSE *(International Council on Systems Engineering*[2]*)* et au niveau national par son chapitre français, l'AFIS (Association française d'Ingénierie Système[3]).

1.1.2 Normes et processus en Ingénierie Système

Bien évidemment, les démarches d'IS dépendent en partie du problème à résoudre et du système à concevoir, mais il existe des invariants, définissant une approche générique. Ces processus génériques ont fait l'objet de normalisations successives et complémentaires :

- le standard militaire MIL-STD-499 (*Systems Engineering Management*, 1969), mis à jour dans le standard MIL-STD-499B (*Systems Engineering*, 1994), créés et déployés par le US DoD pour ses systèmes d'armes et de défense [82, 83] ;
- la norme IEEE 1220 (*Standard for application and Management of the Systems Engineering Process*, créée en 1995 et révisée en 2005) [40] ;
- la norme ANSI/EIA-632 (*Processes for Engineering a System*, 1999) [25] ;
- la norme internationale ISO 15288 (*Systems Engineering – System Life-Cycle Processes*, 2003) [43].

La norme de l'IEEE est pour une large part la transcription du standard MIL-STD-499 à l'ingénierie civile ; elle se focalise sur les processus techniques de conception et de réalisation ; c'est donc une vision « maître d'œuvre ». La norme de l'EIA la complète en décrivant la réalisation des produits jusqu'à leur mise en service ; elle inclut donc la vision « maître d'ouvrage ». Enfin, la norme ISO étend les processus techniques à tout le cycle de vie du

1. La notion *d'environnement* est à prendre au sens large : il comprend tout ce qui est susceptible d'interagir avec le système ou de contraindre son cycle de vie. Il inclut par exemple d'autres systèmes, des utilisateurs ou des concurrents, des normes, etc.
2. http://www.incose.org
3. http://afis.fr

système et inclut donc les processus d'exploitation, de maintien en condition opérationnelle et de retrait de service.

Comme nous le verrons par la suite, notre champ d'étude se restreint aux premières étapes du cycle de vie du système ; c'est pourquoi nous ne décrivons ici que les processus techniques amont, spécifiés par la norme IEEE 1220 et illustrés figure 1.1. Cette figure appelle plusieurs commentaires et compléments, donnés ci-dessous.

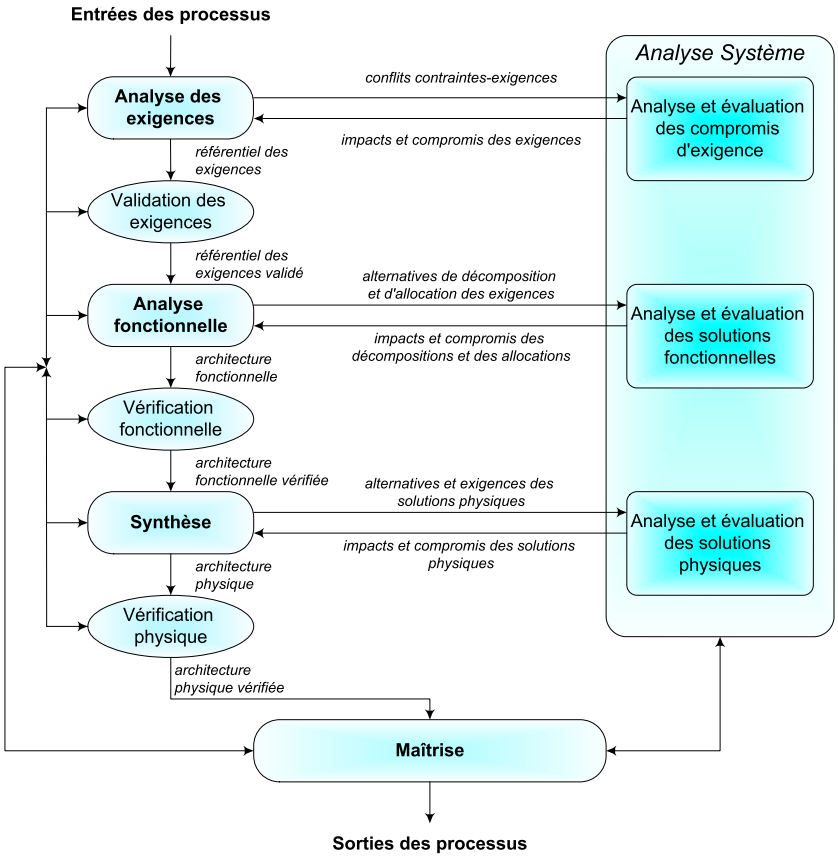


FIGURE 1.1 – Les processus de l'Ingénierie Système *(d'après [40])*

Entrées et sorties des processus Les entrées des processus sont les besoins et les exigences du client, les connaissances technologiques, les cadres normatifs, les produits des développements antérieurs, etc. Les sorties, quant à elles, englobent la définition organique et fonctionnelle du système, mais également la documentation des résultats d'analyse, des compromis, etc.

L'analyse système Ce processus, décomposé en trois sous-processus (colonne de droite), cherche à analyser au niveau système les problèmes tels que les conflits d'exigences ou les solutions alternatives puis à préparer les prises de décision.

La maîtrise Ce processus de maîtrise (ou *control*) s'intéresse tout particulièrement à la gestion technique de l'ensemble du processus mais également au suivi des résultats de conception, ce qui permet de suivre l'évolution du projet et de contrôler le bon déroulement des processus.

Processus centraux Parmi ces processus figurent *l'analyse des exigences*, *l'analyse fonctionnelle* et la *synthèse* (colonne de gauche). Ceux-ci sont rarement purement séquentiels : en effet, dans les systèmes technologiques (à l'inverse des systèmes d'information), les phases d'analyse et de synthèse sont généralement itératives, interdépendantes et génèrent des boucles de rétroactions ou même nécessitent l'ingénierie simultanée des exigences et de conception.

Processus de vérification et validation (V&V) Chacun des processus centraux comprend un sous-processus de *vérification* ou de *validation* s'appliquant sur le produit obtenu. Rappelons incidemment que la distinction entre ces deux techniques peut se résumer en ces termes : on *vérifie* que l'on a construit *un bon* système et l'on *valide* la construction *du bon* système. La vérification est ainsi une démarche interne, qui s'appuie sur des normes, des règlements, etc. En revanche, la validation est en général orientée vers l'extérieur et s'appuie sur des documents contractuels tels que les cahiers des charges. Nous verrons dans la section 1.3 que ces processus de V&V sont essentiels à la mise en œuvre de la Sûreté de Fonctionnement (ou SdF) du système.

1.1.3 Langages et outils de modélisation

L'identification des besoins, la construction d'architectures organiques et fonctionnelles, la spécification ou vérification d'un comportement, le partage d'informations entre les parties prenantes, ... sont autant d'activités d'IS qui nécessitent l'emploi de *modèles*. Ceux-ci permettent en effet d'appréhender et, dans une certaine mesure, de maîtriser la complexité des systèmes à réaliser. De l'identification des problèmes à la construction de solutions, les modèles employés sont généralement d'un niveau d'abstraction décroissant.

À l'évidence, le choix d'un modèle ou d'une famille de modèles dépend de son pouvoir d'expression et du système lui-même ; il dépend également du besoin courant : approche globale du système, définition d'interfaces pour la répartition du travail entre les métiers, simulation d'un comportement, etc. Quant au niveau de détails que doit atteindre le modèle, il répond à un précepte simple : la modélisation d'Ingénierie Système s'arrête là où commencent les génies correspondant aux différents métiers.

La définition ou le choix d'une méthodologie optimale de modélisation, si tant est qu'il en existe une, dépasse le cadre de notre réflexion. Nous nous bornons à présenter dans cette section quatre langages de modélisation qui figurent parmi les plus utilisés ; le lecteur qui voudrait approfondir le sujet est invité à consulter [52, 54, 71] ainsi que les sites Internet de l'INCOSE et de l'AFIS. À titre d'information, nous mentionnons en outre quelques outils logiciels largement employés en IS.

Les diagrammes de flux fonctionnels L'architecture fonctionnelle d'un système, et en particulier son comportement dynamique, peut être modélisée au moyen d'un diagramme

de type [E]FFBD (*[Enhanced] Functional Flow Block Diagram*). Également connus sous le terme de diagrammes de comportement *(behavior diagrams)*, ces diagrammes figurent parmi les plus utilisés en Ingénierie Système [41, 52].

Les diagrammes FFBDs représentent les fonctions exécutées par le système et l'ordre dans lequel l'exécution doit avoir lieu. Les diagrammes EFFBDs, quant à eux, représentent en outre les flux de données échangés entre les fonctions. Ces langages sont développés et utilisés depuis les années 1950 ; ils disposent d'un grand pouvoir expressif et se prêtent relativement facilement à une formalisation cohérente, ce qui sera détaillé dans le chapitre 4.

***The Unified Modeling Language* (UML)** Le langage UML est un langage de modélisation graphique qui trouve ses racines dans la modélisation « orientée objet » du génie logiciel. Le langage est défini depuis 1997 dans un standard validé par *l'Object Management Group* (OMG) [59].

Depuis 2005 et la version 2.0, UML propose treize types de diagrammes, qui se regroupent en diagrammes de structure (ou statiques), de comportement (dont les diagrammes d'activités) ou d'interactions. Le langage permet ainsi de décrire le système tout au long de sa conception, selon de nombreux points de vue, ce qui reste bien sûr impossible avec un langage tel que les EFFBDs. En outre, sa standardisation industrielle facilite la communication entre les différentes parties prenantes du projet. Enfin, l'emploi de *stéréotypes* permet d'adapter et d'étendre le langage (fortement orienté logiciel, bien qu'il s'en défende) à son propre domaine. Le profil MARTE *(Modeling and Analysis of Real Time and Embedded systems)* est ainsi développé depuis 2005 pour intégrer de façon explicite les concepts et mécanismes propres aux systèmes embarqués temps-réel [58].

Cependant, le langage souffre de nombreuses limites [17, 77, 78] ; nous en rappelons quelques unes ci-dessous :

- le nombre très élevé de diagrammes rend le langage difficile à maîtriser et pesant à utiliser ;
- la sémantique est parfois mal définie, voire incohérente (voir cependant la remarque ci-dessous) ;
- la plupart des outils logiciels implémentant UML sont incomplets et parfois incompatibles entre eux, certains points de la sémantique étant laissés à la libre interprétation des constructeurs.

C'est pour pallier certains de ces inconvénients qu'a été développé au cours de la dernière décennie le langage SysML, décrit au paragraphe suivant.

> **Remarque 1.1.** *De nombreux efforts de formalisation d'UML – généralement par le biais de profils proposant une vue restreinte du langage d'origine – ont été menés ces dernières années. Citons ainsi l'approche UML/PNO*[4] *où les contraintes temporelles du système sont modélisés au moyen de réseaux de* Petri *[61]. Le recours aux concepts et outils de la logique linéaire [34] permet alors d'évaluer formellement des performances telles que les temps d'exécution ou de réponse du système.*
>
> *Par ailleurs, les auteurs de [8, 9] ont proposé récemment une formalisation de la sémantique d'UML MARTE basée sur* CCSL*, un langage de contraintes d'horloges. Ce langage permet en*

4. Petri Net Object.

particulier de spécifier et de vérifier formellement le comportement d'un système communicant vis-à-vis de différents protocoles de communication [50].

***The Systems Modeling Language* (SysML)** Le langage de modélisation SysML est une création conjointe de l'INCOSE et de l'OMG ; le langage, spécifique au domaine de l'IS, se définit d'ailleurs comme étant basé sur un sous-ensemble d'UML [80]. Ainsi, SysML n'utilise que sept des diagrammes UML mais en ajoute deux. Ceux-ci permettent de représenter la gestion des besoins et exigences ainsi que les mécanismes d'analyses de performances quantitatives.

SysML est, depuis 2007 et la sortie de la version 1.0, une spécification officielle de l'OMG. Incidemment, SysML définit un stéréotype «EFFBD» ; il permet de rapprocher les diagrammes d'activités du formalisme EFFBD (tous deux représentent en effet le comportement dynamique et fonctionnel du système). Cependant, ce stéréotype fait encore partie d'une annexe non normative du standard.

Plusieurs outils logiciels implémentent SysML, au moins partiellement. Cependant, le langage manque de maturité et hérite de nombreuses imprécisions sémantiques d'UML.

SysML a cependant déjà fait l'objet de recherches visant à intégrer le langage dans une démarche de vérification formelle. Ainsi, les auteurs de [27] et [28] indiquent qu'il est possible de vérifier des exigences de sûreté [5] définies dans le modèle par des méthodes formelles telles que le model checking et la preuve de théorèmes (cf. section 1.2.1). En revanche, la démarche proposée est seulement semi-formalisée et ne résout pas le problème de l'utilisabilité des méthodes formelles dans un contexte d'IS, que nous évoquons plus bas.

***The Architecture Analysis and Design Language* (AADL)** Le langage de description d'architectures AADL, bien qu'apparu récemment, a déjà suscité beaucoup d'intérêt de la part de la communauté temps-réel mais également en Ingénierie Système [29]. Fortement ancré dans le monde des systèmes embarqués temps-réel et distribués, il permet de décrire de façon graphique et textuelle les couches matérielles et logicielles d'un système [79]. Il propose ainsi une vue organique plutôt que purement fonctionnelle, ce qui le place en marge de notre champ d'application.

> **Remarque 1.2.** *On peut cependant retrouver les notions de* fonction*, d'*item *et de* ressource *de l'analyse fonctionnelle au travers des* threads*, des* data *ou des* memories*, par exemple, mais plus généralement, le langage n'est pas naturellement adapté à la diversité des domaines et systèmes habituellement rencontrée en IS.*

Dès l'origine, un effort de formalisation a accompagné le développement du langage ; ainsi, le comportement de certains éléments est décrit au moyen d'automates hybrides [6]. De nombreuses analyses sont alors possibles : signalons ainsi les travaux d'A.E. Rugina [67], menés sur la transformation de certains types de modèles AADL vers les GSPNs (*Generalized Stochastic* Petri *Nets* [57]) afin de réaliser des études de fiabilité sur les architectures organiques. Sa démarche, quoique seulement semi-formalisée, est très proche de la nôtre ; il en va de même de sa problématique.

5. Exprimées avec la logique CTL (cf. section 6.1.1).
6. Certains auteurs émettent cependant quelques réserves sur le degré de formalisation d'AADL [8].

D'autres auteurs ont montré qu'il est possible d'analyser des modèles AADL *via* une transformation vers différentes classes de réseaux de PETRI [62] ou d'automates temporisés [6] (voire vers les systèmes de transitions temporisés [38]), selon une démarche très proche de la nôtre, afin de vérifier formellement des formules de logique temporelle (type LTL ou CTL, cf. section 6.1.1) traduisant la présence de *deadlocks*, le dépassement d'échéances, etc. [1, 13, 65].

Outils de modélisation L'outil de référence de gestion et d'analyse des exigences reste DOORS® [7], édité par TELELOGIC (détenu à présent par IBM). Le logiciel RHAPSODY® [8], développé par I-LOGIX (racheté par TELELOGIC puis IBM), implémente quant à lui les langages UML et SYSML ; il est ainsi dédié à la modélisation des architectures organiques et fonctionnelles. AADL est intégré à l'environnement de développement et d'analyse TOPCASED [9] (qui a par ailleurs fusionné avec l'environnement OSATE) ainsi qu'à la suite OCARINA [86].

Enfin, les ateliers logiciels CORE [10] (VITECH) et MDWORKBENCH [11] (SODIUS) proposent des plates-formes de conception intégrant à la fois la description des besoins et exigences ainsi que celle des architectures organiques et fonctionnelles.

1.2 Les processus de vérification en IS

1.2.1 Intérêts et méthodes

Nous l'avons vu, l'Ingénierie Système s'attache à concevoir des systèmes complexes, mobilisant des efforts financiers et humains généralement conséquents et faisant intervenir et interagir des composants, matériels ou logiciels, et des êtres vivants. Il est ainsi crucial non seulement d'assurer certaines performances, mais également de garantir l'absence de risques (qu'ils soient humains, environnementaux, financiers, ...) ou tout au moins de limiter leur impact. Les processus de vérification contribuent pour une large part à la maîtrise de ces risques, puisqu'ils cherchent à garantir que le système répond à des normes données ou à mettre en évidence les raisons qui empêchent le système de « bien fonctionner ».

De nombreuses méthodes et techniques ont été développées durant les dernières décennies pour assurer le bon fonctionnement [12] du système et ce, dès les premières étapes de sa conception. Parmi celles-ci, la réalisation de *simulations* ou de tests sur un modèle comportemental du système est souvent favorisée en IS. En effet, il s'agit d'une méthode simple et relativement peu coûteuse, consistant à générer (de façon automatique ou non) des cas de tests pertinents dont la validation conduira à celle du système. En revanche, même pour un système d'une taille « raisonnable », l'analyse des comportements possibles ne peut être exhaustive et l'on court le risque de ne pas identifier des situations potentiellement dangereuses pour le système et/ou son environnement. Cette méthode peut donc venir en appui d'une autre technique de vérification, mais ne peut garantir à elle seule la sûreté du système.

7. http://www.telelogic.com/products/doors
8. http://www.telelogic.com/products/rhapsody
9. http://www.topcased.org
10. http://www.vitechcorp.com/products
11. http://www.mdworkbench.com
12. Nous nous plaçons volontairement dans le domaine de l'architecture fonctionnelle : la vérification de l'architecture organique sera ainsi en dehors de notre champ de réflexion.

Les *méthodes formelles*, telles que la preuve (automatique ou assistée) de théorème ou le model checking, s'appuient quant à elles sur un cadre mathématique précis et non ambigu permettant de modéliser à la fois le système et les propriétés qu'il doit vérifier, tout en proposant des algorithmes de vérification de ces propriétés [13]. La démarche du model checking est ainsi la suivante [4, 22] :

- construction d'un modèle Σ du système S ;
- formalisation de la propriété P (donnée par exemple en langage naturel) dans une logique adaptée pour obtenir un modèle Φ de P ;
- utilisation d'un algorithme d'exploration partielle ou exhaustive de l'espace d'états [14] afin de vérifier si Σ satisfait Φ (ce que nous noterons $\Sigma \vDash \Phi$ dans la suite).

Remarque 1.3. *On classe généralement les propriétés comportementales en deux catégories principales [48] :*

- *sûreté* (safety) *: « Un comportement* mauvais *ne se produira jamais » ;*
- *vivacité* (liveness) *:« Un comportement* bon *finira toujours par se produire ».*

La propriété d'équité (fairness) *est un cas particulier de la vivacité : elle stipule qu'un comportement bon, sollicité un nombre infini de fois, pourra se produire un nombre infini de fois. Nous verrons dans le chapitre 6 plusieurs exemples de propriétés de sûreté et de vivacité adaptées au contexte de l'IS.*

Le model checking est un outil particulièrement puissant mais dont l'emploi reste généralement cantonné à la conception de systèmes particulièrement critiques (et généralement informatiques ou électroniques), pour lesquels le rapport entre la garantie de sûreté qu'il apporte et sa complexité de maîtrise et de mise en œuvre reste intéressant. C'est ainsi qu'est apparue notre principale problématique : comment concevoir un outil de vérification formelle d'architectures fonctionnelles qui soit immédiatement utilisable dans des démarches de conception d'Ingénierie Système ?

1.2.2 La vérification formelle des EFFBDs

Afin de répondre à cette problématique, nous avons cherché à développer un outil de vérification formelle, basée sur le principe du model checking et où les modèles Σ et Φ seraient exprimés selon un langage de « haut niveau », aisément manipulable par l'ingénieur système et, à ce titre, intégrable dans un projet de conception d'IS.

Démarche générale Pour mener à bien cette étude, nous aurions pu développer directement les algorithmes de model checking sur ces modèles de haut niveau. Cependant, il est rapidement apparu plus simple de faire appel à des transformations vers des langages de « bas niveau » tels que les réseaux de PETRI ou les automates temporisés, qui soient équivalents [15], puis de mener les vérifications sur ces modèles intermédiaires.

13. Il existe également des méthodes formelles de construction de tests, mais elles ne résolvent pas le problème de la non exhaustivité.

14. Quoique le model checking puisse être confronté au problème de l'explosion de l'espace d'état, un certain nombre de travaux de recherches ont été menés ces dernières décennies, avec succès, pour tenter de maîtriser cette limitation [23].

15. Nous préciserons bien sûr au cours de ce manuscrit le sens et la portée de ces équivalences.

Ce principe, illustré figure 1.2, permet de bénéficier des outils et des résultats obtenus sur ces modèles de bas niveau, tout en cachant la complexité de ces traitements à l'utilisateur. Nous donnons ci-dessous un aperçu des différents modèles et langages mis en œuvre dans la démarche.

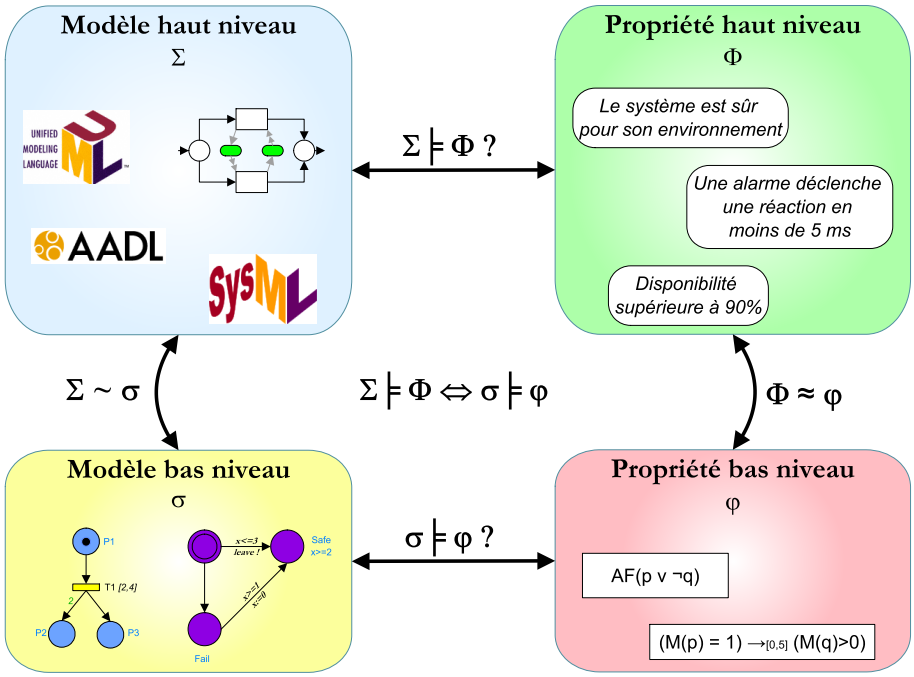


FIGURE 1.2 – Démarche générale

Choix des modèles de haut niveau Σ Au cours du séminaire bibliographique du master de recherche qui a précédé ce travail, nous avons montré que, parmi les langages usuels de conception fonctionnelle en IS, les diagrammes EFFBDs sont ceux qui se prêtent le mieux à des vérifications formelles [71]. Nous avons en effet montré que les langages UML et SysML, à vouloir embrasser trop de points de vue, souffrent d'une sémantique trop imprécise ; SysML est en outre trop peu mature et encore peu déployé industriellement pour convenir à la problématique. En ce qui concerne AADL, dont la sémantique paraît plus « solide », nous avons noté que le langage, organique plutôt que fonctionnel, souffre d'une faible maturité et d'un ancrage par trop poussé dans le domaine des systèmes embarqués temps-réel.

Nous avons également montré qu'aucune formalisation de la syntaxe ni de la sémantique des EFFBDs n'avait été proposée à ce jour [16] ; en revanche, leur implémentation dans des outils logiciels fournit une sémantique *de facto*.

16. E. Herzog a proposé une sémantique semi-formelle des FFBDs (qui sont strictement moins expressifs que les EFFBDs) et une traduction vers les réseaux de Petri, qui se révèle être une bisimulation atemporelle [39].

Expression des propriétés Φ En ce qui concerne le formalisme modélisant les propriétés comportementales, nous avons choisi d'adapter la logique TCTL *(Timed Computation Tree Logic* [5]) au contexte des EFFBDs. Cette logique, que nous avons nommée EFFBD-TCTL, permet d'exprimer des propriétés complexes telles que :

- « La fonction F ne s'exécute jamais en même temps que la fonction G » ;
- « La quantité courante de la ressource R est toujours comprise entre 2 et 10 unités » ;
- « Quelles que soient les circonstances, l'exécution de la fonction F déclenche toujours celle de la fonction G et ce, en moins de 3 unités de temps ».

Choix des modèles de bas niveau (σ et φ) et d'un model checker Quant aux modèles de bas-niveau, sur lesquels sont effectuées les vérifications proprement dites, nous avons retenu les réseaux de PETRI temporels ou *Time* PETRI *Nets* (TPN [55]). Nous détaillons les raisons de ce choix au détriment de formalismes tels que les automates temporisés dans le chapitre 5.

De façon similaire, nous avons choisi d'employer la logique TPN-TCTL [20] pour exprimer les propriétés φ. Nous avons ainsi pu faire appel, dans la construction de notre outil de vérification, au model checker du logiciel ROMÉO[17], proposant entre autres fonctionnalités la vérification au vol de formules TPN-TCTL sur des TPNs [32].

1.3 De la vérification à la Sûreté de Fonctionnement

La vérification (formelle) de propriétés de sûreté s'inscrit dans une démarche plus générale de maîtrise de la Sûreté de Fonctionnement[18] (SDF). Cette recherche de la maîtrise est en effet une activité transverse, et essentielle, de tout projet d'IS puisque tous les processus du cycle de vie y participent.

1.3.1 Attributs, entraves et méthodes de la SDF

Nous introduisons dans cette section les concepts élémentaires de la SDF. C'est à dessein que nous n'approfondirons pas ces notions ; nous renvoyons le lecteur aux ouvrages de références pour plus de détails [10, 49].

On définit souvent la SDF comme étant la confiance (justifiée) que l'utilisateur d'un système peut attribuer au service qu'il lui délivre [49]. Ce terme regroupe les attributs suivants :

- fiabilité (*reliability*) : le système remplit-il sans défaillance les fonctions requises ?
- maintenabilité (*maintainability*) : le système est-il réparable ou modifiable ?
- disponibilité (*availability*) : le système est-il toujours prêt à l'utilisation ?
- sûreté ou sécurité-innocuité[19] (*safety*) : le système peut-il générer des événements aux conséquences néfastes ou catastrophiques ?

17. http://romeo.rts-software.org/
18. La traduction anglaise est bien *dependability* et non *safety*, qui n'en représente qu'une part (cf. plus bas les attributs de la SDF).
19. À ne pas confondre avec la sécurité-immunité ou confidentialité (*security*), qui est également un attribut de la SDF mais que nous ne développerons pas ici.

La Sûreté de Fonctionnement est *entravée* par la présence de *défaillances* ou pannes *(failures)*. Celles-ci sont la manifestation *d'erreurs (errors)* qui sont elles-mêmes les activations de *défauts (faults)* présents dans le système mais qui pouvaient être jusque-là dormants. Dans la suite, nous utiliserons en général le terme générique de *faute* pour désigner défauts, erreurs et pannes.

> **Exemple 1.** *Pour reprendre un exemple classique en informatique, un* bug *de programmation (comme une division par zéro dans une fonction de calcul) est un* défaut. *Lorsque la fonction est appelée, elle génère une* erreur *qui peut se transformer en* défaillance *(du point de vue de la fonction appelante) si aucun dispositif de confinement ou de reprise, tel qu'une gestion des exceptions, n'a été prévu.*

Les moyens de la SdF sont généralement classés en quatre catégories :

- *prévention* des fautes, par l'emploi de méthodologies de conception adaptées et souvent spécifiques à chaque métier ;
- *tolérance* aux fautes, par l'introduction de redondances et de modes dégradés ;
- *élimination* des fautes, au cours des étapes de V&V ;
- *prévision* des fautes, en effectuant des analyses dysfonctionnelles (c'est-à-dire en modélisant les fautes affectant les fonctions durant leurs exécutions).

Ce dernier point correspond à notre seconde problématique, dérivée de la première : comment concevoir un outil de vérification formelle d'architectures dysfonctionnelles qui reste utilisable en Ingénierie Système[20] ?

1.3.2 Vérification des EFFBDs défaillants

Pour répondre à cette problématique, nous avons dû dans un premier temps compléter la sémantique des EFFBDs pour y inclure le comportement des éléments connaissant des fautes. En parallèle, nous avons pu étendre et compléter la logique EFFBD-TCTL pour prendre en compte les nouveaux éléments et exprimer des propriétés telles que :

- « La fonction F ne tombe jamais en panne en même temps que la fonction G » ;
- « La défaillance de la fonction F entraîne l'exécution de la fonction G en moins de 3 unités de temps, et ce quel que soit l'état du système ».

Nous avons également pu intégrer ces ajouts aux transformations $\Sigma \rightarrow \sigma$ et $\Phi \rightarrow \varphi$ afin de permettre *in fine* le model checking des architectures dysfonctionnelles de haut niveau.

1.4 Notre contribution

Notre thèse s'est déroulée dans le cadre d'un partenariat Cifre entre l'ANRT, le laboratoire de l'IRCCyN (et plus spécifiquement son équipe Systèmes Temps-Réel) et Sodius[21],

20. Certains outils usuels en IS, comme les arbres de défaillances et les AMDEC (documents d'Analyse des Modes de Défaillance, de leurs Effets et de leur Criticité) permettent déjà d'effectuer un certain nombre d'analyses; nous en discuterons dans le chapitre 7.

21. http://www.sodius.com

une jeune PME d'IS basée à Nantes et Paris et spécialisée dans l'ingénierie « dirigée par les modèles » (*Model-Driven Engineering*, MDE) ; elle développe le logiciel MDWORKBENCH, une plate-forme de conception et d'analyse des systèmes (basée sur ECLIPSE d'IBM).

À ce titre, nos travaux ont été, pour une large part, motivés et soutenus par la participation de SODIUS à deux PEAs (Programmes d'études amont) successifs de la DGA (Délégation Générale pour l'Armement) :

- le projet KIMONO[22] : entamé en 2004 et achevé en 2007, il avait pour but la création d'un atelier logiciel intégré de conception et d'analyse des systèmes d'armes complexes. En particulier, la nécessité d'y inclure un outil de vérification formelle des architectures fonctionnelles a fait émerger notre première problématique.
- le projet OMOTESC[23] : entamé en 2007 et devant s'achever en décembre 2009, il correspond en partie à une extension de KIMONO aux problématiques de la SDF (il en reprend d'ailleurs de nombreux éléments constitutifs). Notre rôle était de concevoir et d'intégrer un moteur de simulation et de vérification des architectures fonctionnelles et dysfonctionnelles.

Notre contribution se décline ainsi en éléments théoriques et pratiques[24].

1.4.1 Formalisation et traduction des modèles de haut niveau

Comme nous l'avons mentionné plus haut, notre première tâche fut de proposer une formalisation de la syntaxe et de la sémantique comportementale (ou dynamique) des EFFBDs ce qui, à notre connaissance, n'avait jamais été réalisé auparavant.

Cette première étape nous a alors permis de valider les motifs de traduction vers les TPNs, que nous avions définis au cours du master de recherche [70], en prouvant la correction de notre démarche.

À partir de ces résultats, nous avons également pu prouver un certain nombre de propriétés sur un sous-ensemble d'EFFBDs, dits « bornés ». Nous avons également énoncé des conditions suffisantes garantissant l'appartenance d'un modèle à l'ensemble des EFFBDs bornés.

En parallèle, nous avons défini la logique EFFBD-TCTL, sa syntaxe et sa sémantique, ainsi que les règles de transformation des formules EFFBD-TCTL en formules TPN-TCTL.

Enfin, après avoir défini précisément le comportement des fautes sur les éléments constitutifs des EFFBDs, nous avons étendu les formalisations des EFFBDs et d'EFFBD-TCTL pour prendre en compte les effets de ces fautes sur le comportement des modèles.

1.4.2 Développement et intégration d'outils de simulation et de vérification

À partir de ces résultats théoriques, nous avons implémenté dans l'atelier KIMONO les différentes transformations vers les TPNs et les formules TPN-TCTL sous forme d'un *service*. *Via* l'appel au model checker de ROMÉO, le service est alors capable de fournir les résultats

22. Kit de modélisation des nouveaux outils.
23. Outils de modélisation pour la maîtrise de la testabilité des systèmes complexes.
24. Pour des raisons contractuelles, c'est à dessein que nous ne donnerons que peu de détails sur les implémentations.

de vérification en termes d'éléments de haut niveau (fonctions et flux de données), cachant ainsi complètement la complexité de la méthode à l'utilisateur.

Dans un second temps, nous avons écrit à partir de la définition de la sémantique nouvellement formalisée des EFFBDs un moteur de simulation, intégré à l'atelier OMOTESC, prenant en entrée des modèles EFFBDs comportant, ou non, des fautes. Nous avons également pu étendre les capacités de vérification (*via* une transformation en TPN) aux architectures dysfonctionnelles.

> **Remarque 1.4.** *Le développement autour de l'interface graphique (édition et manipulation des modèles, affichage et sauvegarde des résultats sous forme de chronogrammes, ...) a été réalisé par l'équipe de développement de* SODIUS.
>
> *D'une manière générale, la plupart des choix de conception technologiques (écriture des [méta-] modèles ou des transformations selon un certain langage, choix des interfaces de saisie des propriétés-types offertes à la vérification, ...) ont été guidés par l'intégration de nos résultats à une plate-forme de développement et d'analyse complexe, dédiée à l'Ingénierie Système.*

1.4.3 Publications

Outre un rapport technique [75], notre travail a fait l'objet de plusieurs publications internationales (dont trois conférences et une revue avec comités de lecture) :

[72] Charlotte Seidner, Jean-Philippe Lerat et Olivier (H.) Roux
Usability of formal verification on EFFBD models: applying PETRI nets to Systems Engineering issues. In *17th International Symposium of the INCOSE*, juin 2007.

[73] Charlotte Seidner et Olivier (H.) Roux
Behavior diagrams model checking: formal methods applied to Systems Engineering and design. In *6th Annual Conference on Systems Engineering Research*, avril 2008.

[74] Charlotte Seidner, Jean-Philippe Lerat et Olivier (H.) Roux
Usability and usefulness of formal verification in a system design process. In *18th International Symposium of the INCOSE*, juin 2008.

[76] Charlotte Seidner et Olivier (H.) Roux
Formal methods for Systems Engineering behavior models. In *IEEE Transactions on Industrial Informatics*, novembre 2008.

1.5 Organisation du manuscrit

Ce chapitre d'introduction, volontairement détaillé, s'est attaché à présenter les contextes de l'Ingénierie Système et de la SDF puis la problématique d'une vérification formelle adaptée à ces contextes. Nous y avons également donné la démarche générale de notre travail, détaillée tout au long des six chapitres suivants.

Le chapitre 2 introduit la plupart des notations et définitions employées dans la suite du mémoire. Nous y présentons en particulier les *systèmes de transitions temporisés* (STT), le langage de très bas niveau qui permet de décrire la sémantique comportementale des différents modèles. Nous y définissons également différents types de *relations d'équivalence* qui nous permettrons de prouver la correction de notre démarche.

Le chapitre 3, quant à lui, présente de façon informelle la structure d'un diagramme EFFBD et introduit les éléments de bases que sont les structures de contrôle, les fonctions et les flux de données. Nous y introduisons l'exemple du passage à niveau, qui nous servira d'illustration tout au long de ce document.

Devant la richesse de ce langage, la présentation de la syntaxe et la sémantique formelles des EFFBDs fait l'objet d'un chapitre séparé (chapitre 4) ; en particulier, nous y détaillons les raisons nous ayant conduit à exprimer cette sémantique sous la forme d'un système de transitions temporisé. À notre connaissance, cette formalisation est la première démarche de ce type définie sur les EFFBDs. Le chapitre se conclut par plusieurs définitions et résultats préliminaires, dont la description des EFFBDs *bornés*.

Nous donnons la description formelle des TPNs en introduction du chapitre 5. Nous y donnons également les *motifs* de la traduction des EFFBDs en TPNs ainsi que la démonstration de l'équivalence entre les deux modèles. Enfin, nous énonçons plusieurs résultats complémentaires sur la traduction des EFFBDs bornés.

Le chapitre 6 est consacré à la définition des différentes logiques de type TCTL ; nous y présentons en particulier EFFBD-TCTL et sa traduction vers TPN-TCTL. Le chapitre se conclut sur la détermination de la complexité algorithmique du model checking de formules EFFBD-TCTL.

Alors que les chapitres 3 à 6 étaient plus particulièrement axés sur notre problématique principale de vérification formelle des architectures fonctionnelles, le chapitre 7 est consacré à la vérification des modèles défaillants. Nous y présentons ainsi la syntaxe et la sémantique des modèles EFFBDs avec fautes ainsi que les nouveaux motifs de traduction vers les TPNs et la définition d'EFFBD-TCTL étendue aux fautes.

Enfin, la conclusion de ce manuscrit (chapitre 8) donne un bilan des travaux réalisés ainsi que les principales perspectives qui s'en sont dégagées.

Remarque 1.5. *Nous avons choisi de reporter dans l'annexe A les aspects techniques de la méta-modélisation et de l'utilisation de la plate-forme* OMOTESC.

Chapitre 2

Notations et Définitions

Résumé *Avant de présenter de façon formelle les modèles EFFBDs, il nous est apparu nécessaire de rappeler dans ce deuxième chapitre un certain nombre de définitions classiques du temps-réel et des méthodes formelles.*

En particulier, nous y définissons non seulement les notations et conventions adoptées dans ce manuscrit mais également le formalisme des systèmes de transitions temporisés (STT) ainsi que les principales relations d'équivalences entre modèles formels, dont la bisimulation temporisée.

Par la suite, ces notions préliminaires nous permettrons en particulier de justifier l'équivalence entre les modèles EFFBDs et leur traduction en TPNs et, partant, la correction de notre démarche de model checking de ces modèles de haut niveau.

Sommaire

2.1 Notations et définitions générales

2.1.1 Ensembles et opérateurs usuels

Dans ce document, nous noterons les ensembles usuels de la façon suivante :

- $\emptyset$ est l'ensemble vide ;
- $\mathbb{B} = \{\texttt{vrai}; \texttt{faux}\}$ est l'ensemble des booléens ;
- $\mathbb{N}$ représente l'ensemble des entiers naturels, $\mathbb{N}_{>0}$ l'ensemble des entiers naturels strictement positifs et $\mathbb{Z}$ l'ensemble des entiers relatifs ;
- soient a et b deux entiers ($a \leq b$) ; $\mathbb{N}_{a..b}$ représente l'ensemble des entiers compris entre a et b (inclus) ;
- $\mathbb{Q}$, $\mathbb{Q}_{\geq 0}$ et $\mathbb{Q}_{>0}$ désignent respectivement l'ensemble des rationnels, des rationnels positifs ou nuls et des rationnels strictement positifs ;
- de façon similaire, $\mathbb{R}$, $\mathbb{R}_{\geq 0}$ et $\mathbb{R}_{>0}$ désignent respectivement l'ensemble des réels, des réels positifs ou nuls et des réels strictement positifs ;

Nous adoptons en outre les notations et conventions suivantes :

- soit E un ensemble fini ; son cardinal est noté $|E|$;
- soient A et X deux ensembles ; l'ensemble des applications de X dans A est noté A^X ; en outre, si X est un ensemble fini, tout élément de A^X peut se représenter par un vecteur de $A^{|X|}$;
- $\vec{0}$ est le *vecteur nul*, dont toutes les composantes sont égales à 0 ; la dimension de ce vecteur dépend du contexte ;
- les opérateurs binaires usuels $+, -, <, \leq, =, \geq$ et $>$, ainsi que les fonctions usuelles min et max sont étendus, élément par élément, aux vecteurs de A^n, avec $A = \mathbb{N}, \mathbb{Q}, \mathbb{R}$ et $n \in \mathbb{N}$;
- soit X un ensemble et P une proposition logique sur X ($P \in \mathbb{B}^X$) ; l'ensemble des éléments de X qui vérifient P est noté $\{x \in X / P(x)\}$;
- l'opérateur de différence (ou soustraction) d'ensembles est noté $\setminus$;
- l'opérateur de conjonction logique `et` est noté $\wedge$, la disjonction $\vee$, la négation $\neg$, l'implication $\Rightarrow$ et l'équivalence $\Leftrightarrow$;
- le symbole $\bowtie$ représente tout opérateur de l'ensemble $\{<; \leq; =; \neq; \geq; >\}$;
- le quantificateur $\exists!$ se lit « il existe un et un seul » ;

D'une manière générale, les ensembles (non ordonnés) sont notés entre accolades, les n-uplets entre parenthèses. De plus, les unions d'ensembles ne contiennent pas d'éléments dupliqués ; ainsi, $\{a\} \cup \{a\} = \{a\}$.

Enfin, sauf mention contraire, une accolade ouvrante (à laquelle ne correspond aucune accolade fermante) représente une conjonction de conditions ou de propositions logiques.

2.1.2 Valuations

Une *valuation* ν sur un ensemble de variables X est un élément de $(\mathbb{R}_{\geq 0})^X$. Dans le cas où les variables représentent des horloges, l'application qui, à un instant donné, associe à chacune des horloges le temps écoulé depuis son démarrage est ainsi une valuation.

Soient $\nu \in (\mathbb{R}_{\geq 0})^X$ une valuation et $\delta \in \mathbb{R}_{\geq 0}$ un réel positif. La valuation $\nu + \delta$ est définie pour tout $x \in X$ par $(\nu + \delta)(x) = \nu(x) + \delta$.

2.2 Systèmes de transitions

2.2.1 Systèmes de transitions étiquetées

Le comportement dynamique de modèles tels que les réseaux de PETRI peut être représenté comme une succession *d'états* reliés par des *transitions* étiquetées par des *actions*. Leur sémantique peut donc être décrite par un *système de transitions étiquetées* (STE, *Labelled Transition System*), dont nous rappelons la définition ci-dessous.

Définition 2.1 (Système de transitions étiquetées) *Un* système de transitions étiquetées *est un quadruplet* $\mathcal{S} = (Q, Q_0, \Sigma, \rightarrow)$ *où :*

- Q *est un ensemble* d'états [1] ;
- $Q_0 \subseteq Q$ *est l'ensemble des* états initiaux ;
- Σ *est l'alphabet contenant l'ensemble des* étiquettes de transitions *(ou* actions*)* ;
- $\rightarrow \subseteq Q \times \Sigma \times Q$ *est la* relation de transition.

La notation $q \xrightarrow{\sigma} q'$ signifie que le triplet (q, σ, q') appartient à $\rightarrow$.

> **Remarque 2.1.** *Il est également possible de définir un ensemble* $Q_f \subseteq Q$ *d'états* accepteurs *ou finaux. Lorsque cela n'est pas précisé, on supposera dans la suite que* $Q_f = Q$.

Définition 2.2 (Exécution) *Une exécution est une séquence (éventuellement infinie) de transitions, notée* $\langle q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \ldots \rangle$.

L'ensemble des exécutions possibles d'un STE $\mathcal{S}$ est noté $[\![\mathcal{S}]\!]$ [2].

La *trace d'une exécution* correspond à la séquence des actions effectuées au cours de cette exécution : nous en donnons la définition formelle ci-dessous.

Définition 2.3 (Trace) *La* trace *de l'exécution* $\langle q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \ldots \rangle$ *est la suite des étiquettes* σ_i *; on la note* $[\sigma_1 \sigma_2 \ldots]$.

Les traces sont donc les mots formés par concaténation des symboles de l'alphabet Σ. Réciproquement, un mot $w = [\sigma_1 \sigma_2 \ldots]$ est *accepté* par le STE $\mathcal{S}$ s'il existe une exécution de $\mathcal{S}$ à partir d'un état initial de Q_0 dont la trace est w, en supposant, comme indiqué dans la remarque ci-dessus, que tous les états sont accepteurs. Cette remarque nous conduit à la définition du langage accepté par un STE.

Définition 2.4 (Langage accepté) *Soit* $\mathcal{S}$ *un STE; le* langage accepté *par* $\mathcal{S}$*, noté* $\mathcal{L}(\mathcal{S})$*, est l'ensemble des mots acceptés par* $\mathcal{S}$.

1. Dans cette définition, Q contient les états *accessibles* par l'application de $\rightarrow$ depuis Q_0, mais il peut également contenir des états « inacessibles ».

2. Lorsque cela n'introduit pas d'ambiguïté, on notera $[\![\mathcal{S}]\!]$ l'ensemble des exécutions d'un modèle $\mathcal{S}$ dont la sémantique est définie par un STE $||\mathcal{S}||$.

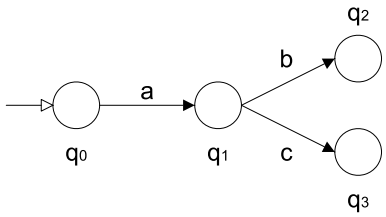


FIGURE 2.1 – Exemple de système de transitions étiquetées

Exemple 2. *Considérons le système de transitions $\mathcal{S}$ illustré figure 2.1. Il possède les caractéristiques suivantes :*

- $Q = \{q_0; q_1; q_2; q_3\}$;
- $Q_0 = \{q_0\}$;
- $\Sigma = \{a; b; c\}$;
- $\rightarrow = \{(q_0, a, q_1); (q_1, b, q_2); (q_1, c, q_3)\}$.

L'ensemble des exécutions possibles est $[\![\mathcal{S}]\!] = \{\langle q_0 \xrightarrow{a} q_1\rangle; \langle q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2\rangle; \langle q_0 \xrightarrow{a} q_1 \xrightarrow{c} q_3\rangle\}$ *et le système accepte le langage* $\mathcal{L}(\mathcal{S}) = \{[a]; [ab], [ac]\}$.

2.2.2 Systèmes de transitions temporisés

Les *systèmes de transitions temporisés* (STT, *Timed Transition Systems* [38]) forment une extension naturelle des STEs permettant de prendre en compte l'écoulement du temps dans la description de la sémantique comportementale de modèles tels que les EFFBDs ou les TPNs. Ils définissent en effet deux types de transitions :

- les *transitions d'action* représentent les évolutions discrètes du système (comme pour les STEs) ;
- les *transitions continues*, ou temporelles, modélisent l'écoulement du temps à partir d'un état donné.

Nous donnons la définition formelle d'un STT ci-dessous :

Définition 2.5 (Système de transitions temporisé) *Un* système de transitions temporisé *est un quadruplet* $\mathcal{S} = (Q, Q_0, \Sigma, \rightarrow)$ *où :*

- Q *est un ensemble* d'états ;
- $Q_0 \subseteq Q$ *est l'ensemble des* états initiaux ;
- Σ *est* l'alphabet des actions, *disjoint de* $\mathbb{R}_{\geq 0}$;
- $\rightarrow \subseteq Q \times \{\Sigma \times \mathbb{R}_{\geq 0}\} \times Q$ *est la* relation de transition *composée :*
  - *de la relation de transition discrète* $\xrightarrow{\sigma \in \Sigma} \subseteq Q \times \Sigma \times Q$;
  - *de la relation de transition continue* $\xrightarrow{\delta \in \mathbb{R}_{\geq 0}} \subseteq Q \times \mathbb{R}_{\geq 0} \times Q$;

Nous faisons les hypothèses usuelles suivantes sur les STT :

- attente nulle : $\forall q, q' \in Q,\ q \xrightarrow{0} q' \Leftrightarrow q = q'$;
- déterminisme temporel : $\forall q, q', q" \in Q, \forall \delta \in \mathbb{R}_{\geq 0}$, si $q \xrightarrow{\delta} q'$ et $q \xrightarrow{\delta} q"$ alors $q' = q"$;

- additivité temporelle : $\forall q, q', q" \in Q$, $\forall \delta, \delta' \in \mathbb{R}_{\geq 0}$, si $q \xrightarrow{\delta} q'$ et $q' \xrightarrow{\delta'} q"$ alors $q \xrightarrow{\delta+\delta'} q"$;
- continuité temporelle[3] : $\forall q, q" \in Q$, $\forall \delta, \delta' \in \mathbb{R}_{\geq 0}$, si $q \xrightarrow{\delta+\delta'} q"$ alors il existe $q' \in Q$ tel que $q \xrightarrow{\delta} q'$ et $q' \xrightarrow{\delta'} q"$;

La notion d'exécution s'étend au cas temporisé comme défini ci-dessous :

Définition 2.6 (Exécution temporisée) *Une exécution d'un système de transitions temporisé $\mathcal{S}$ est une séquence (éventuellement infinie) de transitions continues et discrètes, notée $\langle q_0 \xrightarrow{\delta_1} q'_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\delta_2} q'_1 \xrightarrow{\sigma_2} \dots \rangle$.*

L'ensemble des exécution possibles de $\mathcal{S}$ est noté $[\![\mathcal{S}]\!]$.

Il est en effet possible d'écrire toute exécution comme une alternance de transitions continues (de durée éventuellement nulle) et discrètes.

Les notions de traces et de langage s'étendent également au cas temporisé :

Définition 2.7 (Trace) *La* trace *de l'exécution $\langle q_0 \xrightarrow{\delta_1} q'_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\delta_2} q'_1 \xrightarrow{\sigma_2} \dots \rangle$ est la séquence des couples (δ_i, σ_i), notée $[(\delta_1, \sigma_1)(\delta_2, \sigma_2)\dots]$.*

Les traces sont également des *mots temporisés* ; on notera que les réels δ_i représentent des dates relatives et non absolues.

De même que dans le cas des STE, un mot $w = [(\delta_1, \sigma_1)(\delta_2, \sigma_2)\dots]$ est *accepté* par le STT $\mathcal{S}$ s'il existe une exécution de $\mathcal{S}$ à partir d'un état initial de Q_0 dont la trace est w. On définit alors le langage (temporisé) accepté par un STT par :

Définition 2.8 (Langage temporisé accepté) *Soit $\mathcal{S}$ un STT ; le* langage temporisé accepté *par $\mathcal{S}$, noté $\mathcal{L}_T(\mathcal{S})$, est l'ensemble des mots temporisés acceptés par $\mathcal{S}$.*

2.3 Relations d'équivalence entre systèmes de transitions

2.3.1 Égalité de langages

À l'évidence, la comparaison de deux systèmes de transitions nécessite la définition de *relations d'équivalence* entre les modèles. La comparaison des traces ou, plus généralement, des langages acceptés permet d'établir un premier niveau d'équivalence.

Définition 2.9 (Égalité de langages) *Soient $\mathcal{R} = (R, R_0, \Sigma, \rightarrow_{\mathcal{R}})$ et $\mathcal{S} = (S, S_0, \Sigma, \rightarrow_{\mathcal{S}})$ deux STTs ; ils sont équivalents en termes de langage temporisé accepté si et seulement si $\mathcal{L}_T(\mathcal{R}) = \mathcal{L}_T(\mathcal{S})$.*

> **Exemple 3.** *Les deux systèmes de transitions illustrés figure 2.2 acceptent le même langage temporisé $\mathcal{L}_T = \{(0, a); (0, a)(1.25, b); (0, a)(0, c)\}$.*

Cependant, la simple égalité des langages ne fournit pas d'analyse fine du comportement respectif des systèmes comparés. Ainsi, pour reprendre l'exemple de la figure 2.2, on note que

3. Nous nous plaçons en effet dans le cadre du *temps continu* (ou *dense*) et non en *temps discret*.

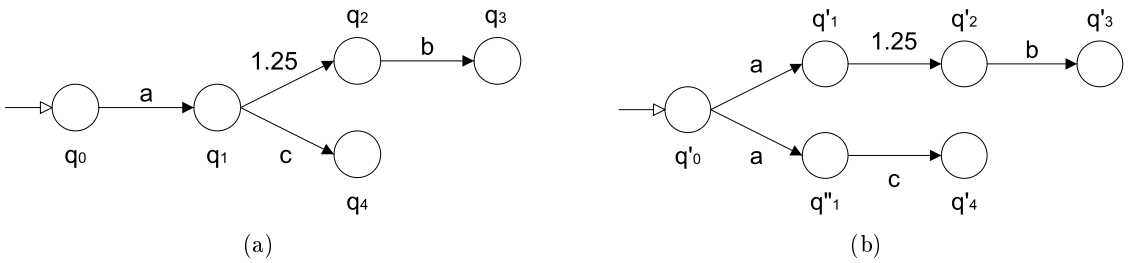


FIGURE 2.2 – Équivalence de langage temporisé

le « choix » entre l'écoulement d'une unité de temps et l'action c est effectué à partir de l'état q_1 dans le premier cas alors qu'il est réalisé dès l'état initial dans le second cas. Intuitivement, les comportements de ces deux systèmes ne sont donc pas « comparables ».

2.3.2 Simulation temporelle

C'est pour pallier cette déficience qu'a été développée la notion de *simulation* : cette relation, qui est plus forte que l'égalité de langages, permet de comparer plus finement le comportement d'un système avec un autre.

Définition 2.10 (Simulation) *Soient $\mathcal{R} = (R, R_0, \Sigma, \rightarrow_{\mathcal{R}})$ et $\mathcal{S} = (S, S_0, \Sigma, \rightarrow_{\mathcal{S}})$ deux STTs et $\triangleleft$ une relation binaire*[4] *sur $\mathcal{R}$ et $\mathcal{S}$. $\triangleleft$ est une* simulation temporelle *de $\mathcal{R}$ par $\mathcal{S}$ si et seulement si :*

- *pour tout $r_0 \in R_0$, il existe $s_0 \in S_0$ tel que $r_0 \triangleleft s_0$;*
- *pour tout $(r, s) \in R \times S$ tel que $r \triangleleft s$, s'il existe $r' \in R$ tel que $r \xrightarrow{\delta}_{\mathcal{R}} r'$ avec $\delta \in \mathbb{R}_{\geq 0}$, alors il existe $s' \in S$ tel que $s \xrightarrow{\delta}_{\mathcal{R}} s'$ et $r' \triangleleft s'$;*
- *pour tout $(r, s) \in R \times S$ tel que $r \triangleleft s$, s'il existe $r' \in R$ tel que $r \xrightarrow{\sigma}_{\mathcal{R}} r'$ avec $\sigma \in \Sigma$, alors il existe $s' \in S$ tel que $s \xrightarrow{\sigma}_{\mathcal{R}} s'$ et $r' \triangleleft s'$;*

On dira dans ce cas que $\mathcal{R}$ est *simulé (temporellement) par* $\mathcal{S}$, ce que l'on notera $\mathcal{R} \triangleleft \mathcal{S}$.

Exemple 4. *Reprenons l'exemple de la figure 2.2. De façon triviale, le système de gauche simule le système de droite mais la réciproque n'est pas vraie.*

2.3.3 Bisimulation temporelle

La relation de *bisimulation temporelle* découle naturellement de la notion de simulation. De façon informelle, elle stipule que toute action, discrète ou continue, effectuée par l'un des système en relation peut être simulée de façon équivalente par l'autre et réciproquement.

Définition 2.11 (Bisimulation temporelle) *Soient $\mathcal{R}$ et $\mathcal{S}$ deux STTs et $\sim$ une relation binaire sur $\mathcal{R}$ et $\mathcal{S}$. $\sim$ est une bisimulation temporelle si et seulement si $\sim$ est une relation de simulation de $\mathcal{R}$ par $\mathcal{S}$* et *$\sim^{-1}$, définie par l'équivalence $Y \sim^{-1} X \Leftrightarrow X \sim Y$, est une relation de simulation de $\mathcal{S}$ par $\mathcal{R}$.*

4. $(r, s) \in \triangleleft$ se note $r \triangleleft s$.

Dans ce cas, on dira des deux systèmes de transitions qu'ils sont *en bisimulation temporelle*, ce que l'on notera $\mathcal{R} \sim \mathcal{S}$.

Remarque 2.2. *À l'évidence, deux systèmes en bisimulation temporisée acceptent le même langage temporisé.*

Remarque 2.3. *La bisimulation ne doit pas être confondue avec la* cosimulation (two-way simulation)*, moins forte. Dans la cosimulation, $\mathcal{S}_1$ est simulé par $\mathcal{S}_2$* via *une relation $\triangleleft$ et $\mathcal{S}_2$ est simulé par $\mathcal{S}_1$* via *une relation $\triangleright$ mais $\triangleleft \neq \triangleright^{-1}$.*

Considérons par exemple les systèmes illustrés figure 2.3 : on peut vérifier que le premier système simule l'autre et réciproquement. Cependant, les états des deux systèmes ne sont pas en bijection : ces deux systèmes sont donc en cosimulation mais non en bisimulation.

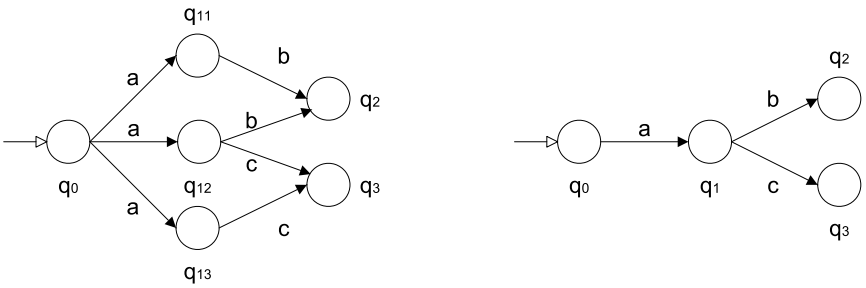


FIGURE 2.3 – Systèmes de transitions en cosimulation

Remarque 2.4. *On définit également une action dite* silencieuse *ou* non observable*, généralement notée ϵ. On parlera alors d'une ϵ-transition pour désigner une transition étiquetée par ϵ. L'alphabet étendu $\Sigma \cup \{\epsilon\}$ permet ainsi de modéliser des actions internes*[5].

Étant donné un STT $\mathcal{S}$ sur l'alphabet $\Sigma \cup \{\epsilon\}$, on peut en construire une ϵ-abstraction sous forme d'un STT $\mathcal{S}^\epsilon$ où seules figurent les actions de Σ. Nous ne donnons pas la définition de la construction ; le lecteur pourra se référer par exemple à [53] pour plus de détails.

La notion de bisimulation (temporisée ou non) entre deux systèmes $\mathcal{R}$ et $\mathcal{S}$ s'étend au cas où ceux-ci comportent des ϵ-transitions. De façon informelle, une relation de bisimulation (forte) entre les systèmes ϵ-abstraits $\mathcal{R}^\epsilon$ et $\mathcal{S}^\epsilon$ induit en effet une relation de bisimulation dite faible *entre $\mathcal{R}$ et $\mathcal{S}$. En outre, deux systèmes en bisimulation faible acceptent le même langage.*

Considérons par exemple les deux systèmes illustrés figures 2.4(a) et 2.4(b). Leur ϵ-abstraction est définie par un même STT, illustré figure 2.4(c) : ils sont donc en bisimulation faible. Ils acceptent en outre le même langage $\mathcal{L} = \{[a], [ab], [c]\}$.

Dans la suite, et sauf mention contraire, nous ne traiterons que des relations fortes et omettrons donc l'adjectif.

5. ϵ est en effet l'élément neutre pour la concaténation de langages : $\forall a \in \Sigma$, $\epsilon a = a\epsilon = a$.

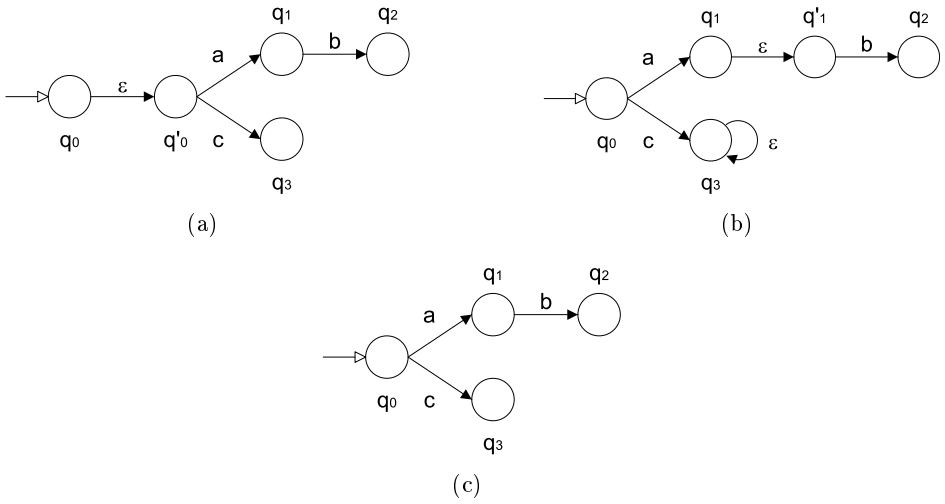


FIGURE 2.4 – Systèmes de transitions en bisimulation faible

Chapitre 3

Description des EFFBDs

Résumé *La conception d'un système d'ingénierie complexe nécessite des outils de représentation simples et interdisciplinaires mais suffisamment expressifs pour capturer correctement les scénarios d'enchaînements fonctionnels du système. Le formalisme EFFBDs (ou* Enhanced Function Flow Block Diagrams*), essentiellement graphique, a été développé dans ce but précis ; il est ainsi largement employé tout au long de la conception fonctionnelle de grands projets d'Ingénierie Système, dans des domaines aussi divers que la défense, l'aérospatiale, le ferroviaire ou même le biomédical, et ce, depuis plusieurs décennies.*

Ce chapitre propose une présentation informelle des EFFBDs s'articulant autour des différents éléments de modélisation offerts par le langage : structures de contrôle, fonctions et flux de données. Reflet de la richesse d'expressivité des EFFBDs, cette présentation est passablement détaillée. Nous verrons que leur syntaxe et leur sémantique formelles, établies dans le chapitre suivant, sont également très riches.

En guise de conclusion de ce chapitre, nous proposons deux exemples illustrant l'expressivité des modèles EFFBDs ; le second nous servira d'ailleurs de support illustratif tout au long de ce manuscrit.

Sommaire

3.1 Généralités

3.1.1 Aperçu historique

Lorsqu'il doit décrire les structures fonctionnelles et les flux de données d'un système, l'ingénieur chargé de sa conception a souvent recours à des représentations graphiques relativement simples, d'autant que le choix d'une représentation peut s'avérer déterminant, voire crucial pour communiquer et échanger efficacement des modèles avec les différentes parties prenantes, et ce tout au long du cycle de vie du système.

Dans ce but, de nombreuses représentations ont été développées depuis plus d'un demi-siècle ; nous ne traiterons ici que des *Enhanced Functional Flow Block Diagrams* (EFFBDs). Ces diagrammes figurent en effet parmi les plus utilisés en Ingénierie Système [47, 52, 60] et permettent de représenter le comportement de systèmes complexes, distribués, hiérarchiques, concurrents et communicants.

Les EFFBDs sont dérivés des diagrammes FFBDs *(Functional Flow Block Diagrams)*. Ceux-ci ont été conçus et développés en 1967 par J. Long [52], alors ingénieur au sein de TRW Inc., société dont les activités étaient pour une large part liées aux secteurs de la défense et de l'aérospatiale. Le but premier de ces diagrammes était de fournir une aide à la description du comportement de missiles balistiques, trop complexe pour être décrit rigoureusement (et lisiblement) sous forme textuelle. Ils ont été exploités dès 1968 par la NASA, afin de concevoir la séquence (temporelle) des activités nécessaires à l'accomplissement des missions de vol de ses programmes spatiaux [1]. Ils ont par la suite fait l'objet d'une norme, avec le standard militaire américain MIL-STD-499 [82].

Cependant, les FFBDs ne permettent pas de représenter les échanges de flux de données entre les fonctions : plusieurs modèles ont alors été développés vers la fin des années 1970 afin de combler ce manque. Les travaux de M. Alford ont donné lieu aux *Behavior Diagrams* ou diagrammes de comportement [2, 3], ceux de J. Long ont conduit aux EFFBDs [2].

> **Remarque 3.1.** *L'atelier logiciel CORE, développé depuis 1992 par* Vitech Corp. [3]*, constitue la principale implémentation des diagrammes FFBDs et EFFBDs. Il comprend un module de vérification dynamique de ces diagrammes,* COREsim *[88] qui permet en particulier d'exécuter une simulation du modèle en mode continu ou pas-à-pas. Les résultats sont donnés sous la forme de fichiers d'enregistrement* (logs) *et de chronogrammes* (timelines).

Nous donnons dans la suite de ce chapitre la présentation informelle de la structure et du comportement dynamique des EFFBDs. Elle a été compilée, pour une large part, de [88, 52]. Comme nous l'avons précisé dans la section 1.5, cette présentation est aussi détaillée que le formalisme est expressif ; c'est pourquoi elle fait l'objet d'un chapitre à part entière.

1. La conception et la description fonctionnelle du système de mesure du laboratoire médical et comportemental intégré, destiné à être embarqué à bord d'une station spatiale pour des vols de longue durée, a ainsi été entièrement réalisée au moyen de modèles FFBDs [16].
2. Ces deux types de diagrammes sont sensiblement identiques, la différence majeure étant le sens de lecture des diagrammes (de haut en bas pour les diagrammes de comportement et de gauche à droite pour les EFFBDs).
3. http://www.vitechcorp.com

3.1.2 Structure générale d'un EFFBD

Un EFFBD décrit dans un *scénario* les fonctions réalisées par le système et l'ordre dans lequel elles doivent être exécutées. Cet ordre est spécifié par :

- les paramètres dynamiques des fonctions (*i.e.* leur durée d'exécution) ;
- les *structures de contrôle* (ou *constructions*) qui transfèrent le *flux de contrôle* le long de leurs *branches* ;
- les *flux de données* (items et ressources).

Le formalisme propose un large choix de structures de contrôle, comme des branches parallèles, des boucles, des branches de sélection, etc. Une branche peut contenir un nombre quelconque (fini) de structures de contrôle (il est même possible d'avoir des branches vides, c'est-à-dire ne portant aucune structure). Dans la suite, on parlera indifféremment d'une branche et de l'ensemble des constructions qu'elle porte. Les sections 3.2 à 3.4 présentent de façon informelle la syntaxe et la sémantique de ces structures de contrôle.

Les flux de données, qui ne sont manipulés que par les fonctions, sont reliés à celles-ci par des flèches, éventuellement pondérées, dont le sens indique s'il s'agit d'une consommation ou d'une production. Ils peuvent être manipulés par un nombre quelconque de fonctions et chaque fonction peut manipuler un nombre quelconque de flux. La section 3.5 donne une description informelle de la syntaxe et de la sémantique des flux de données.

La figure 3.1 illustre un exemple d'EFFBD ; ce diagramme ne correspond pas à un système réel mais donne un aperçu des principaux éléments constitutifs d'un EFFBD.

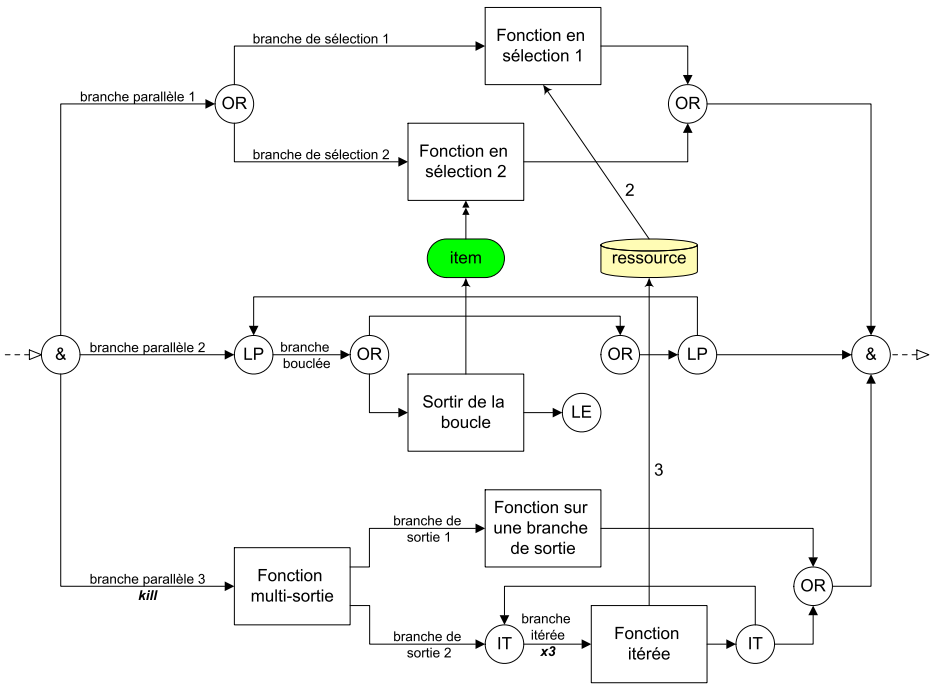


FIGURE 3.1 – Exemple de diagramme EFFBD *(adapté de [52])*

Le diagramme se lit de la gauche vers la droite (ce qui est renforcé par l'ajout de flèches entre les constructions, représentant les fragments de branches). Les flèches en traits discontinus situées de part et d'autre du diagramme n'ont d'autre but que de repérer ses points d'entrée et de sortie mais n'ont pas de représentation normalisée. Toute branche peut être annotée (l'annotation est placée au début de la branche) mais seules les annotations en gras et italique (ici, `kill` et `x3`) ont une valeur sémantique.

La plupart des structures de contrôle se composent de deux *nœuds* (représentés par des cercles) encadrant une ou plusieurs branches ; les fonctions sont figurées par des rectangles. Toutes les structures sont « emboîtées » : on sort de chaque structure dans l'ordre inverse d'entrée [4].

3.2 Structures à branches multiples

3.2.1 Structures parallèles et réplications

Une *structure parallèle* comprend deux nœuds AND (également notés `&` dans les diagrammes) et n branches parallèles ($n > 1$). Lorsque l'on entre dans la structure, c'est-à-dire lorsque le nœud AND ouvrant est activé, la première construction de chaque branche est activée. On sort de la structure (ce qui active la construction suivant le nœud AND fermant) dès que la dernière construction de chaque branche parallèle a fini son exécution : ceci peut induire des états de synchronisation entre les branches.

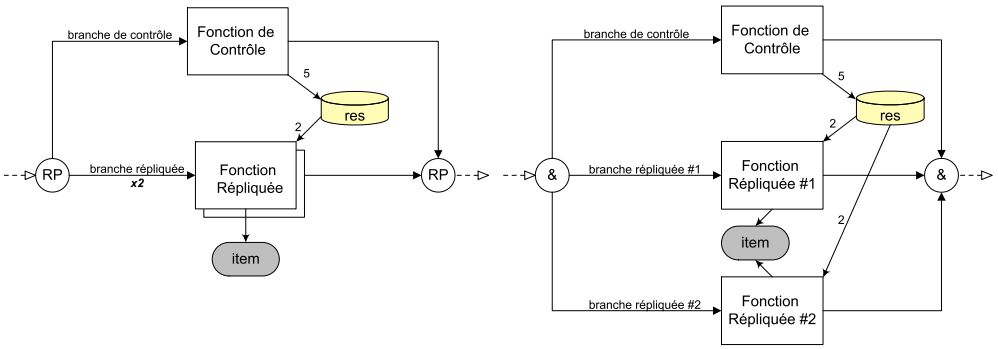
En outre, un nombre quelconque de branches peut porter le marqueur de terminaison forcée *kill*. Le marqueur modifie le comportement décrit précédemment comme suit : dès que la dernière construction d'une branche *kill* s'achève, on force toute la structure à se terminer. Toute construction contenue dans la structure et qui était alors active est « tuée » ; on poursuit l'exécution de la suite de la structure normalement. L'ensemble des branches peut être marqué : c'est le cas par exemple de l'un des motifs d'interruption par *time out*, décrit dans la section 3.6.1, p. 57.

Enfin, une *structure de réplication* se compose de deux nœuds RP, d'une branche de contrôle, d'une branche répliquée et du nombre n de réplications à effectuer ($n > 1$), comme illustré figure 3.2(a). Il est possible [5] (et c'est le choix qui sera fait dans la suite de ce document) de voir cette structure comme un raccourci d'écriture permettant de décrire des branches identiques et indépendantes, sans surcharger le diagramme avec des informations redondantes. La construction est ainsi similaire à une structure parallèle comprenant $n + 1$ branches, l'une (quelconque [6]) étant la branche de contrôle et les n autres les copies de la branche répliquée. La figure 3.2(b) illustre cette équivalence.

4. Ce comportement sera essentiel pour la définition d'un EFFBD bien formé, comme décrit dans la section 4.1.2, p. 67.

5. L'extension (non normative) de SysML pour les EFFBDs précise que la sémantique des réplications (de même que celle des ressources et des branches *kill*) n'est pas encore définie [80]. De même, CORE ne permet pas encore de simulation des réplications, leur sémantique n'y étant pas non plus définie.

6. Pour faciliter la lecture, il s'agit généralement de la branche supérieure.



(a) Structure de réplication (b) Structure parallèle équivalente

FIGURE 3.2 – Dépliage d'une structure de réplication

3.2.2 Structures de sélection

Une *structure de sélection* se compose de deux nœuds OR encadrant n branches de sélection ($n > 1$). Lorsque l'on entre dans la structure, l'une des branches est choisie (il s'agit donc d'un *ou exclusif*) et sa première construction est activée. On sort de la structure lorsque la dernière construction de la branche choisie a fini son exécution.

Le *processus de sélection*, c'est-à-dire l'ensemble des règles qui déterminent le choix de la branche à exécuter, prend plusieurs formes dans les différentes implémentations : ajout de probabilités sur les branches, lecture d'un script interne, appel explicite à l'utilisateur pendant la simulation... Cependant, le formalisme d'origine des EFFBDs ne définit aucun processus de sélection particulier. C'est pourquoi nous considérerons dans la suite que le choix s'effectue de manière non déterministe, quel que soit l'état courant du système.

3.3 Structures de répétition

3.3.1 Structures de boucle

Une *structure de boucle* se compose de deux nœuds LP et d'une branche bouclée. Lorsque le flux de contrôle atteint la structure, la première construction de la branche bouclée est activée. Une fois que la dernière construction de la branche a fini son exécution, le contrôle retourne au début de la boucle : le comportement ainsi défini est donc *infini*.

Par ailleurs, lorsque l'on atteint un nœud LE (pour *Loop Exit* ou sortie de boucle), on interrompt la boucle le contenant (toutes les structures contenues sont terminées) et le contrôle passe à la construction qui suit la boucle. Dans le cas de boucles imbriquées (ou plus exactement *emboîtées*), on choisit la boucle contenante la plus proche, comme illustré figure 3.3. Sur cette figure, les structures de boucle sont délimitées par les pointillés pour faciliter la lecture. Le nœud `LE_2` dépend de la boucle interne (2) alors que le nœud `LE_1` dépend de la boucle externe (1). On exécute donc `G` (respectivement `H`) après avoir traité le nœud `LE_2` (respectivement `LE_1`). On remarquera qu'un nœud LE n'est pas relié par une flèche à son nœud LP fermant de référence puisque cette destination est déterminée de façon unique par l'architecture du système.

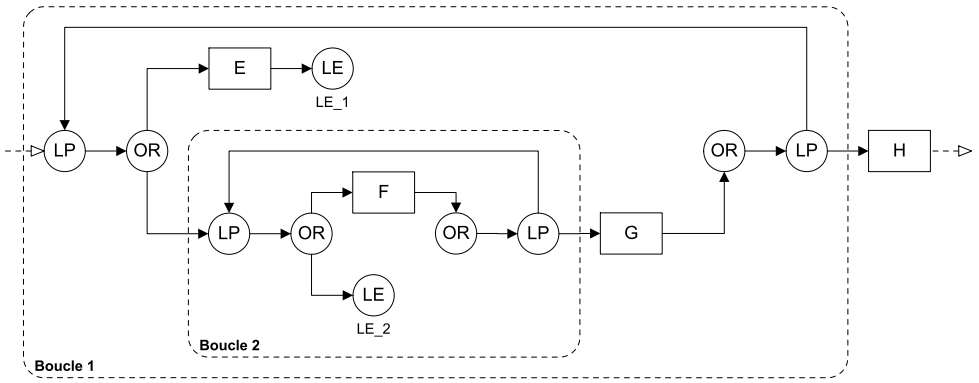


FIGURE 3.3 – Imbrication de structures de boucle

Certaines interprétations, dont SysML, proposent en outre d'associer à la structure une « condition de boucle », évaluée à chaque fois que l'on exécute la branche de boucle, à la manière d'un `tant que`*(condition)* `faire (...)`. Il est ainsi possible de sortir d'une boucle sans passer par un nœud LE. Cependant, ce mécanisme ne fait pas partie du formalisme natif des EFFBDs et ne sera pas considéré dans la suite.

3.3.2 Structures d'itération

Une *structure d'itération* se compose de deux nœuds IT encadrant une branche itérée, d'un compteur interne (qui indique le nombre de fois que l'on a débuté l'exécution de la branche) et d'un entier i_M représentant le nombre d'itérations à effectuer ($i_M > 1$). L'entrée dans la structure active la première construction de la branche itérée et initialise le compteur à 1. Lorsque la dernière construction de la branche a fini son exécution, deux comportements sont possibles :

- si la valeur du compteur est strictement inférieure à i_M, on active de nouveau la première construction de la branche et l'on incrémente le compteur d'une unité ;
- sinon, on sort de la structure en passant le contrôle à la construction qui la suit.

3.4 Modélisation des fonctions

3.4.1 Fonctions et structures fonctionnelles

Les *structures fonctionnelles* (ou *function constructs*), représentées par des rectangles, sont les conteneurs des *fonctions*, qui sont elles-même les éléments centraux de l'EFFBD. Il est possible qu'une fonction soit portée par plusieurs structures fonctionnelles, ce qui permet de décrire plusieurs instances d'une même fonction et donc généralement d'améliorer la lisibilité du modèle (au même titre que les réplications). Cependant, on supposera par la suite qu'à une fonction ne correspond qu'une structure fonctionnelle ; lorsque cela n'introduit aucune ambiguïté, on ne distinguera pas une fonction et sa structure fonctionnelle contenante.

Les EFFBDs permettent de décrire des système hiérarchisés : une fonction peut ainsi être *décomposée* dans un *sous-scénario*, ce qui est décrit dans la section 3.4.2. De plus, une fonction peut avoir un ou plusieurs *états de sortie* (ou plus simplement des *sorties*) ; dans ce dernier cas, à chaque état de sortie correspond une branche de sortie. Les n branches de sortie ($n > 1$) d'une fonction multi-sortie convergent sur un nœud OR fermant ; le choix de l'une ou l'autre des branches de sortie est soit interne à la fonction (selon un processus de sélection similaire aux structures de choix) lorsque celle-ci n'est pas décomposée, soit décrit dans le sous-scénario dans le cas contraire.

On considère dans la suite que la structure fonctionnelle d'une fonction multi-sortie contient non seulement la fonction mais également les branches de sortie et le nœud OR fermant tel qu'illustré figure 3.4 (les pointillés délimitent les deux structures fonctionnelles dans les cas simple et multi-sortie). On assimile ainsi la structure fonctionnelle à deux nœuds encadrant plusieurs branches, au même titre qu'une structure de sélection, par exemple [7].

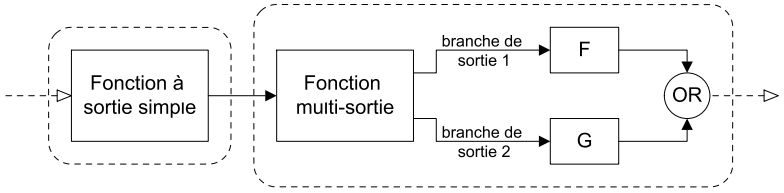


FIGURE 3.4 – Structures fonctionnelles de fonctions simple et multi-sortie

Une fonction peut démarrer son exécution si et seulement si elle a été activée par le flux de contrôle et, dans le cas d'une fonction consommant des flux d'entrée, si ces entrées sont disponibles. Selon l'implémentation, la durée d'exécution de la fonction peut être :

- un réel positif ou nul ;
- déterminée par une loi de probabilité telle que la durée (tirée à chaque exécution) reste positive ou nulle ;
- déterminée par un script...

Dans la suite de ce document, et en l'absence d'une définition précise dans le formalisme des EFFBDs, on considère que la durée d'exécution d'une fonction appartient à un intervalle réel positif à bornes entières de la forme $[a, b]$ avec $a \in \mathbb{N}$ et $b \in \mathbb{N} \cup \{\infty\}$.

Une fois que la durée d'exécution s'est écoulée, l'exécution est terminée : les (éventuels) flux de sortie sont produits et le flux de contrôle passe à la structure suivante (qui est la première structure de la branche de sortie choisie, dans le cas des fonctions multi-sortie). L'exécution d'une fonction ne peut être suspendue ; elle ne peut non plus être interrompue, sauf si elle est terminée par une structure « tueuse » (*kill*, LE, ...). Dans ce dernier cas, la fonction ne produit pas ses flux de sortie ni ne passe le contrôle à la structure suivante.

3.4.2 Fonctions décomposées et sous-scénarios

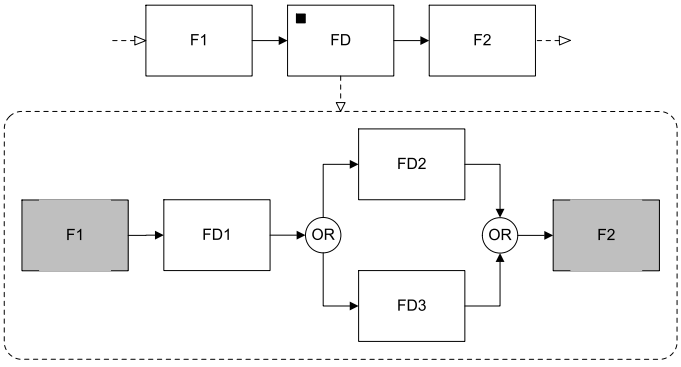
Une fonction décomposée se distingue graphiquement par un carré noir dans le coin supérieur gauche du rectangle. Une telle fonction ne manipule pas de flux et n'a pas de durée

7. La pertinence de ce choix apparaîtra lors de la définition d'un EFFBD bien formé.

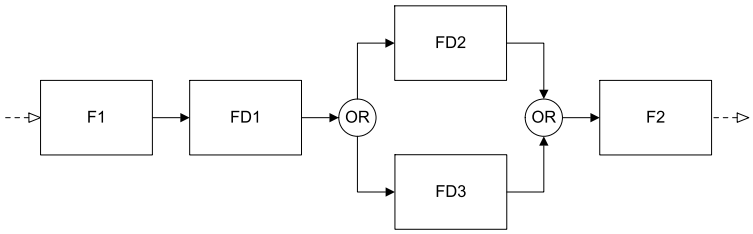
d'exécution propre : celle-ci est déterminée par le sous-scénario. Un modèle EFFBD peut comporter un nombre quelconque (fini) de niveaux de décomposition, mais les règles de conception usuelles préconisent de limiter la description à une demi-douzaine de niveaux [8].

Lorsque le flux de contrôle atteint une fonction décomposée, on active la première construction de la branche principale du sous-scénario ; la fonction parente est active tant qu'au moins un des éléments de sa décomposition est actif.

Comme précisé dans la section 3.4.1, les fonctions décomposées peuvent être à leur tour à sorties simple ou multiples : dans le premier cas, on peut toujours « déplier » le sous-scénario dans son scénario parent, comme illustré figure 3.5. Les rectangles discontinus (sur fond grisé) présents dans le sous-scénario indiquent les points de connexion du sous-scénario avec le niveau supérieur. Ils sont déduits de l'architecture du modèle et pourront ainsi être omis si cela ne gêne pas la compréhension. Lorsque l'on sort de la dernière construction du sous-scénario (dans l'exemple, la structure de choix), on désactive la fonction décomposée et l'on active la structure qui la suit dans le scénario parent (`F2` dans l'exemple). Ce cas trivial n'est donc pas considéré dans la suite de ce document.



(a) Fonction décomposée et sous-scénario associé



(b) Modèle déplié équivalent

FIGURE 3.5 – Dépliage d'une fonction décomposée à sortie simple

Dans le cas des fonctions décomposées à sorties multiples, c'est l'ajout de nœuds EXIT (figurés par des rectangles arrondis) dans le sous-scénario qui permet d'affecter correctement les branches de sortie de destination. À chaque nœud EXIT est associé un état de sortie de la fonction parente (et donc la branche de sortie correspondante) mentionné sous le nœud (figure 3.6). Bien que cela ne soit pas obligatoire, il est recommandé d'associer à chaque état

8. D'après [54], par exemple, « dans une hiérarchie, l'homme n'appréhende facilement que 7 items *[ici, des fonctions]* en largeur *[i.e. en parallèle]* et 4 niveaux en profondeur. »

de sortie un et un seul nœud EXIT : on suppose dans la suite que tous les systèmes respectent cette recommandation[9]. De même, la branche de sortie de référence peut être située non pas dans le niveau hiérarchique immédiatement supérieur, mais dans un niveau hiérarchique encore supérieur. Bien que cela n'introduise pas plus de complexité, on considérera cependant que la branche de sortie d'un nœud EXIT est toujours située dans le niveau hiérarchique immédiatement supérieur.

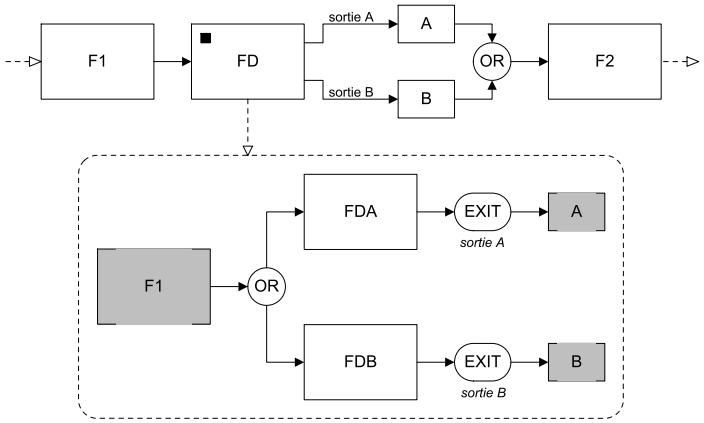


FIGURE 3.6 – Fonction décomposée multi-sortie

Lorsque le flux de contrôle atteint un nœud EXIT, on termine toute structure active dans le sous-scénario et le contrôle passe à la première construction de la branche de sortie de référence du nœud EXIT. Il est possible que le sous-scénario lié à une fonction multi-sortie (dite alors *décomposée hybride*) ne se termine pas par un nœud EXIT, comme illustré figure 3.7. Lorsque l'on arrive à la fin du sous-scénario, (ici, en terminant la fonction `FHC`), on choisit l'une des branches de sortie dont on active la première construction, comme si la fonction parente n'était pas décomposée[10]. Dans cet exemple, les séquences possibles d'exécution des fonctions sont donc : `{F1,FHA,A,F2}`, `{F1,FHC,A,F2}`, `{F1,FHC,B,F2}`, et `{F1,FHB,B,F2}`.

> **Remarque 3.2.** *Aucune règle du formalisme des EFFBDs n'interdit explicitement l'usage de* fonctions décomposées récursives *(au sens algorithmique du terme). Le modèle illustré figure 3.8 comporte ainsi une fonction récursive, FD. On remarquera que les points de connexion du sous-scénario ont été adaptés en conséquence. En revanche, ils ne font pas apparaître les contraintes selon lesquelles, par exemple, on ne peut sortir par la branche contenant F_2 que si l'on est au premier niveau de récursion. Une exécution de ce modèle est donc de la forme $F_1.(E_1)^n.E_2.(E_3)^n.F_2$ avec $n \in \mathbb{N}_{>0}$.*
>
> *La prise en compte de ces constructions dans l'écriture de la syntaxe et de la sémantique est cependant délicate : ainsi, comment définir des conditions d'arrêt limitant le nombre de niveaux de récursion ? De même, si l'architecture proposeée dans la figure 3.8 paraît « raisonnable », qu'en serait-il si l'on remplaçait la structure de sélection centrale par une structure parallèle ? Quel serait le sens d'une telle construction ?...*

9. Les résultats présentés dans ce document s'appliqueraient cependant directement au cas où plusieurs nœuds EXIT pointent vers la même branche de sortie.

10. C'est pourquoi on ajoute à la fin du sous-scénario le nœud OR ouvrant, sur fond grisé, qui pointe vers les deux branches de sortie de la fonction `FH`.

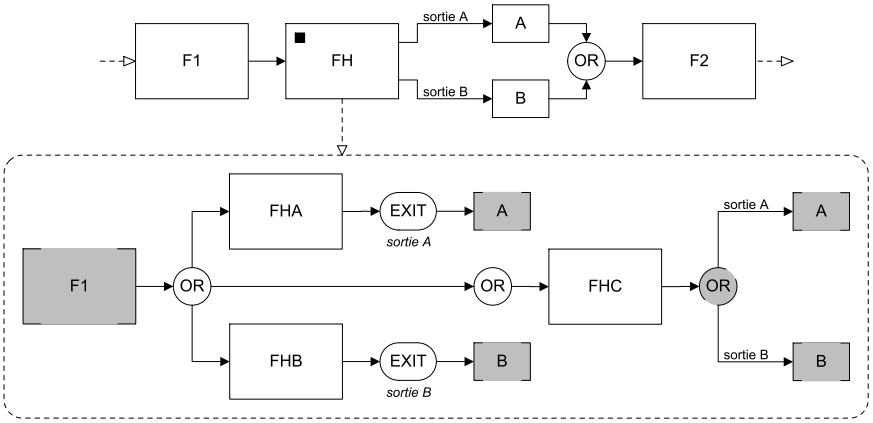


FIGURE 3.7 – Fonction décomposée hybride

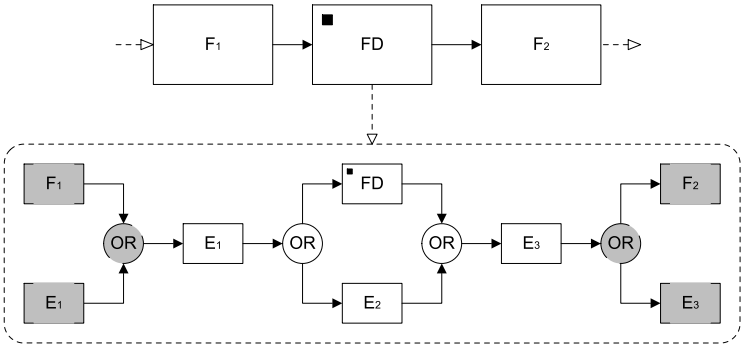


FIGURE 3.8 – Fonction récursive

Autant de questions qui dépassent le cadre de notre étude : nous supposerons que les modèles EFFBDs considérés dans la suite ne comportent pas de fonctions récursives[11]*. En tout état de cause, si l'emploi de récursion est largement répandu en conception de logiciel, par exemple, il est usuellement déconseillé de modéliser des fonctions récursives au niveau système.*

3.5 Modélisation des flux de données

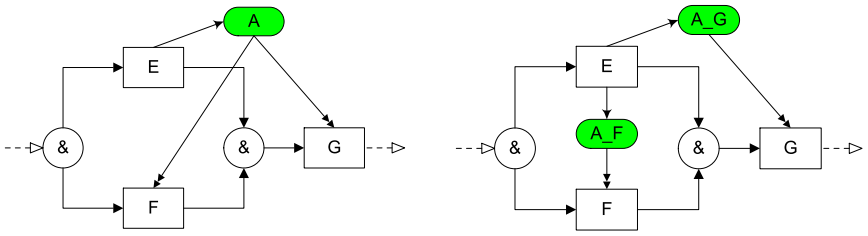
3.5.1 Items

Le formalisme originel des EFFBDs permet la modélisation des flux d'information au moyen des *items* (représentés par des rectangles arrondis sur fond gris ou vert). Il est important de noter que l'on ne modélise pas la valeur de l'information *(« la vitesse mesurée par le capteur vaut 3 $m.s^{-1}$ »)* mais le fait que l'information est disponible ou non *(« le capteur de vitesse a effectué une mesure »)*.

11. On remarquera d'ailleurs que l'atelier logiciel CORE ne prend pas en compte la récursivité : ainsi, la fonction *FD* contenue dans le sous-scénario est traitée comme une fonction non décomposée.

Remarque 3.3. *Une gestion plus fine des items, incluant des manipulations sur les valeurs, nécessiterait l'emploi d'outils spécifiques à chaque métier (comme Matlab© de* MATHWORKS*) : on sort alors du cadre interdisciplinaire caractéristique de l'Ingénierie Système.*

Un item est *émis par* (ou *sort de*) une (ou plusieurs) fonction(s) lorsque celle-ci achève son exécution. Plus précisément, les items sont *diffusés* (*broadcast*) à l'ensemble des fonctions qui le reçoivent : on produit en réalité autant d'exemplaires que de fonctions réceptrices, chaque exemplaire ne pouvant être consommé que par « sa » fonction. On considérera par la suite que l'on a « déplié » dans le modèle la diffusion des items, comme illustré figure 3.9. Dans cet exemple, on remplace l'item `A`, produit par la fonction `E` et consommé par les fonctions `F` et `G` par les deux items `A_F` et `A_G`. Chacun représente la même information mais l'item `A_F` (respectivement `A_G`) ne peut être consommé que par la fonction `F` (respectivement `G`). Ce fonctionnement s'étend bien sûr au cas où l'item est produit par plus d'une fonction.



(a) Diffusion d'un item à deux fonctions (b) Modèle déplié équivalent

FIGURE 3.9 – Dépliage d'items diffusés

Lors de la production, l'exemplaire nouvellement émis vient s'ajouter à l'ensemble des exemplaires (dépliés) déjà produits et qui n'ont pas encore été consommés. Une fois produit, l'exemplaire est indiscernable des autres : en particulier, on ne mémorise pas « l'âge » des exemplaires.

Remarque 3.4. *Il serait possible de définir une politique de remplacement (ou « d'écrasement ») au lieu d'un ajout, mais ce n'est pas le choix qui a été retenu dans le formalisme. Cependant, les résultats présentés dans ce document s'étendraient aisément à ce cas.*

La consommation d'un item s'effectue une fois que la fonction réceptrice est activée ; elle est synchrone : tous les items d'entrée de la fonction sont consommés en même temps. La consommation peut se faire selon deux modes :

- *consommation bloquante* (flèche double) : tant que l'item n'est pas disponible (i.e. l'ensemble des exemplaires produits pour cette fonction est vide), la fonction ne peut démarrer son exécution (l'item ajoute donc un contrôle supplémentaire) ;
- *consommation non bloquante* (flèche simple) : si l'item n'est pas disponible, mais que les autres items (bloquants) le sont, la fonction peut démarrer son exécution.

Un item peut être en consommation bloquante pour une fonction et non bloquante pour une autre : cette caractérisation n'est pas intrinsèque à l'item mais bien au couple (item, fonction réceptrice).

Lorsqu'aucune consommation d'un item n'est bloquante, celui-ci est dit de type *data store* (ou conteneur de données) et est représenté par un rectangle arrondi sur fond gris. Dans le cas contraire, c'est-à-dire si l'item déclenche au moins une fonction, on parle de *trigger* (représenté sur fond vert). Comme l'on ne s'intéresse ici qu'au comportement dynamique des fonctions, il est inutile de prendre en compte les relations de consommation non bloquante. On ne traitera donc dans la suite que des triggers « purs ».

La consommation consiste à retirer l'un des exemplaires d'item de l'ensemble existant : la consommation est donc destructive. On pourrait étendre les résultats présentés dans ce document au cas non destructif (on effectue alors une simple lecture du nombre d'exemplaires disponibles).

Remarque 3.5. *Toute manipulation d'item est synchrone par rapport aux débuts et fins d'exécution des fonctions.*

Le *niveau* d'un item est défini comme le nombre d'exemplaires disponibles à un instant donné. C'est donc le nombre de fois que l'information a été produite sans avoir été consommée.

Remarque 3.6. *Dans le cas où l'item a été déplié, il faut considérer le niveau de chaque « sous-item ». Cependant, pour des raisons de lisibilité, il est utile de ne donner qu'une seule valeur par item replié. Ainsi, dans le simulateur que nous avons implémenté (cf. annexe A), nous avons choisi pour valeur le* maximum *des niveaux des items dépliés d'un même item. Ce choix permet par exemple un dimensionnement immédiat des conteneurs d'exemplaires d'items.*

À titre d'illustration, la figure 3.10 représente sous forme de chronogramme l'évolution temporelle du système décrit figure 3.9. La durée d'exécution de chaque fonction est fixée à une unité de temps. L'exécution d'une fonction est représentée sur fond vert et l'attente d'un item sur fond jaune. Le sous-item `A_F` *est consommé immédiatement après sa production par la fonction* `G` *mais l'item* `A_G` *est mis en mémoire entre les dates* $t = 1$ *et* $t = 2$.

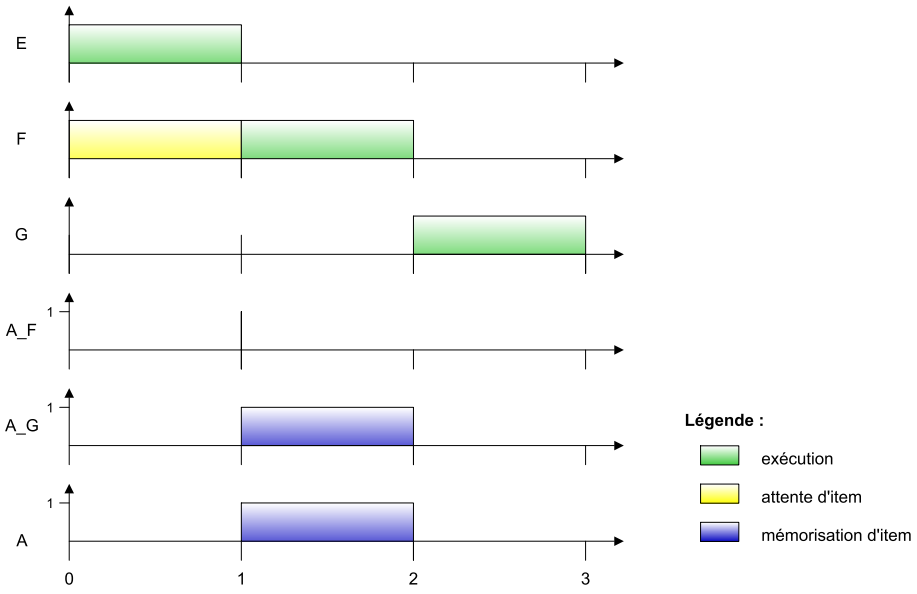


FIGURE 3.10 – Chronogramme d'exécution du modèle de la figure 3.9

CORE a adopté une convention différente et plus complexe : la valeur choisie (qui est la seule affichée) est le maximum sur les niveaux dans les états de durée nulle (ici à la date $t = 1$) et le minimum dans les autres cas. Le chronogramme correspondant est donné figure 3.11. On notera qu'il n'est pas possible de voir qu'un exemplaire de l'item `A` *(à savoir* `A_G`*) doit être gardé en mémoire entre les dates $t = 1$ et $t = 2$.*

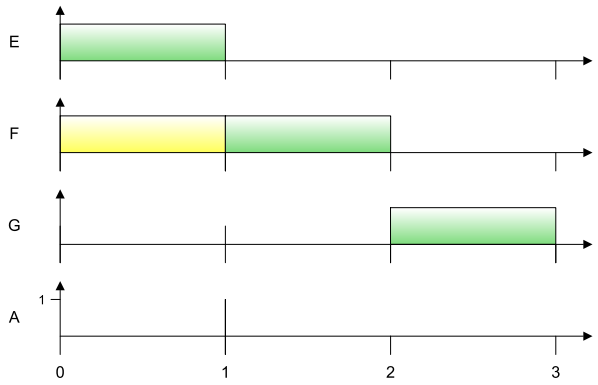


FIGURE 3.11 – Chronogramme d'exécution du modèle de la figure 3.9 obtenu par CORE

3.5.2 Ressources

La manipulation des items ne se fait que par des quantités unitaires et ne permet de représenter que des échanges d'informations ; pour modéliser des quantités différentes de l'unité ainsi que des données représentant par exemple de l'énergie, une charge processeur ou des déchets, les concepteurs de CORE ont développé et implémenté le concept de *ressource*. Ce concept n'est pas normalisé mais suffisamment cohérent (et intéressant en terme d'expressivité du modèle) pour être ajouté au formalisme.

À la différence des items, les ressources ne sont pas diffusées mais *partagées* entre les fonctions consommatrices. De plus, elles peuvent avoir une valeur initiale. Cette valeur ainsi que les quantités échangées peuvent être définies par :

- un réel positif [12] ;
- une loi de probabilité telle que la valeur reste positive ;
- un script...

Sauf mention contraire, on considérera par la suite que les valeurs initiales et les quantités sont toujours des entiers positifs (voire strictement positifs pour les quantités). Les quantités sont notées sur les flèches reliant ressources et fonctions (par convention, une quantité égale à l'unité peut être omise).

Le *niveau* d'une ressource est défini comme la quantité disponible à un instant donné. D'après ce qui précède, le niveau est toujours un entier positif ou nul.

Une ressource peut être *consommée* ou *produite* par une fonction ; elle peut également être *capturée*, ce qui revient à la consommer au début de l'exécution puis à la produire, dans la même quantité, à la fin de l'exécution.

12. Et même strictement positif pour les quantités.

Une fonction ne démarre son exécution que si toutes les ressources à consommer sont présentes au moins dans les quantités nécessaires (et si les items d'entrée sont disponibles). Toutes les ressources sont ensuite consommées en retirant les quantités aux niveaux courants, de façon synchrone avec les items, ce qui permet de démarrer l'exécution.

Remarque 3.7. *Toute manipulation de ressource est également synchrone par rapport aux débuts et fins d'exécution des fonctions.*

Remarque 3.8. *Il est également possible de* lire *une ressource, ce qui correspond à une consommation non destructive : la ressource doit être présente au moins dans la quantité définie pour que la fonction réceptrice s'exécute, mais cette quantité n'est pas retirée lors du démarrage de l'exécution. Une flèche de lecture est figurée par des pointillés.*

Remarque 3.9. *Tout trigger déplié peut être considéré comme une ressource dont la valeur initiale est nulle et dont chaque consommation ou production se fait dans une quantité unitaire.*

Deux fonctions F et G se trouvent en *conflit de ressource* lorsqu'elles doivent consommer à un même instant la même ressource, dans les quantités respectives f et g mais que le niveau courant de la ressource, d, permet de satisfaire soit F, soit G mais non F et G[13]. Le formalisme ne prévoit pas de gestion des conflits de ressources ; en particulier, on ne peut définir de priorité d'accès aux ressources.

Remarque 3.10. *CORE propose un mode supplémentaire de consommation des ressources, dit* Acquire Available. *Dans ce mode, il est en effet possible d'effectuer une « pré-réservation » de la ressource si, lors de l'activation de la fonction, son niveau est inférieur à la quantité nécessitée par la fonction. Dans ce cas, la fonction « pré-consomme » la quantité disponible (empêchant donc toute autre fonction d'y accéder) et attend que le reliquat soit disponible. Il est également possible d'avoir des conflits en pré-réservation mais l'outil ne prévoit pas de gestion de ces conflits.*

Dans le cas où la fonction est tuée au cours de l'attente de ses ressources, les quantités pré-consommées ne sont pas restituées.

3.6 Exemples

Cette section présente deux exemples relativement simples de modèles EFFBDs. Le premier est un motif de conception (ou *design pattern*) de terminaison forcée ; le second constitue l'exemple applicatif qui servira de support tout au long de ce document.

D'une manière générale, et indépendamment du formalisme mis en œuvre, il n'existe pas de modélisation unique d'un système : elle dépend du degré de détail souhaité, du niveau d'avancement de la conception, mais également des habitudes et des règles de conception de l'ingénieur système et des autres parties-prenantes. Les modèles proposés dans la suite n'ont donc pas la prétention d'être optimaux, mais bien d'illustrer tel ou tel aspect du formalisme.

13. On a donc $f \leq d$, $g \leq d$ et $f + g > d$.

3.6.1 Modélisation d'une terminaison forcée par dépassement de délai

La conception d'un système communicant nécessite souvent des mécanismes de surveillance ou même d'interdiction du dépassement de délais. On peut ainsi affecter à toute fonction consommant des flux d'entrée un *time-out*, c'est-à-dire un délai maximum d'attente des flux : si le délai est atteint, on interrompt le traitement de la fonction, dont l'exécution est donc omise, puis l'on poursuit normalement la suite de l'exécution du système. En limitant le temps d'attente en réception, on permet ainsi d'éviter qu'une fonction ne se bloque en attendant des flux qui ne seront jamais produits.

La modélisation d'un *time-out* revient à lier à la fonction une horloge, qui démarre à l'activation de la fonction (et qui reste active tant que la fonction ne démarre pas son exécution ou n'est pas tuée), et une valeur δ ($\delta \in \mathbb{R}_{\geq 0}$). Si l'horloge atteint la valeur δ, on tue la fonction et le contrôle passe à son successeur (si certaines ressources ont été consommées en pré-réservation, elles ne sont pas restituées).

Remarque 3.11. *Si les flux d'entrée deviennent disponibles au moment exact de l'échéance du* time-out*, les deux comportements sont possibles*[14] *:*

- *consommation des flux puis exécution normale de la fonction ;*
- *terminaison forcée (sans consommation) de la fonction.*

Ils sont bien sûr en exclusion mutuelle ; en particulier, il n'est pas possible de procéder à la consommation puis d'effectuer la terminaison forcée.

La sémantique des EFFBDs n'est pas assez expressive pour donner une priorité à l'un ou l'autre de ces comportements : en effet, nous verrons dans la suite que, lorsque plusieurs actions discrètes sont possibles à un instant donné, elles peuvent être toutes également *choisies : il n'existe donc pas de priorité ou de hiérarchisation des actions. De même, une gestion des priorités* via *une ressource ne peut fonctionner puisque les priorités dans les conflits de ressources ne sont pas gérées.*

Considérons la fonction `F`, illustrée figure 3.12. Elle est déclenchée par l'item `A` et consomme deux unités de la ressource `R`. À l'issue de son exécution, elle produit 1.5 unité de `R` ainsi que l'item `A'`.

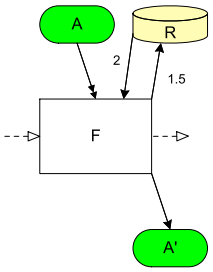


FIGURE 3.12 – Fonction consommatrice

On décompose `F` en deux fonctions :

- `F_C`, d'une durée nulle, représente la consommation des flux d'entrée de `F` ;

14. L'implémentation proposée dans CORE donne toujours la priorité à l'exécution.

– `F_EP`, de même durée que `F`, représente son exécution et la production des flux de sortie.

Enfin, on souhaite adjoindre à `F` un *time-out* de δ unités de temps.

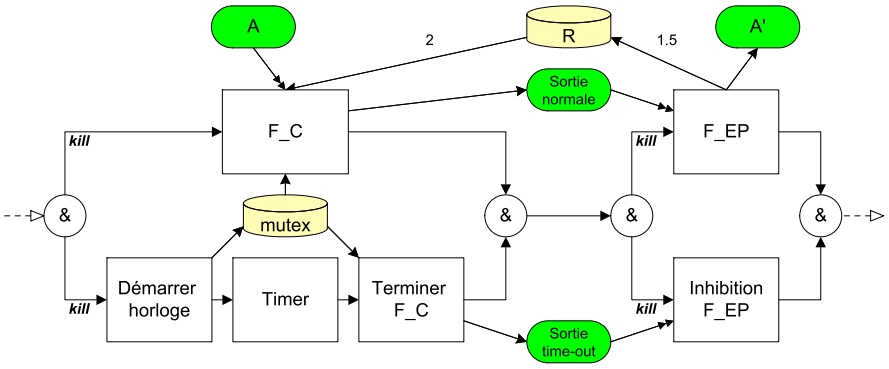


FIGURE 3.13 – Modélisation du *time-out* d'une fonction par des branches de terminaison

La figure 3.13 décrit un motif de terminaison par *time-out* comprenant deux structures parallèles dont toutes les branches portent le modificateur *kill*. La première structure représente l'attente des flux d'entrée ; elle fait apparaître une fonction `Timer` dont la durée d'exécution est de δ unités de temps. Si ce délai est atteint, on tue la branche supérieure et l'on sort de la première structure, en produisant l'item `Sortie time-out`. En revanche, si les entrées deviennent disponibles en moins de δ unités de temps, on exécute complètement la fonction `F_C`, ce qui provoque la terminaison forcée de `Timer` et l'émission de l'item `Sortie normale`.

> **Remarque 3.12.** *Le motif fait apparaître une ressource* `mutex`*, dont la valeur initiale est nulle et qui n'est produite que par la fonction instantanée* `Démarrer horloge`*. Son seul rôle est d'empêcher le démarrage de la fonction* `F_C` *(ce qui provoque la consommation des flux d'entrée) et, dans le même instant, l'exécution de la fonction (instantanée)* `Terminer F_C`*, dont la fin provoque la terminaison de la première structure parallèle. Cette situation ne peut se produire que si les flux deviennent disponibles à la date d'échéance de la fonction* `Timer`*. On notera qu'il y a donc, dans ce cas, conflit d'accès à la ressource entre* `F_C` *et* `Terminer F_C` *mais, aucune priorité n'étant donnée (cf. p. 55), les deux comportements décrits dans la remarque 3.11 sont possibles.*
>
> *Dans le cas où les flux sont disponibles avant l'échéance,* `mutex` *est toujours consommée par* `F_C`*; s'ils sont disponibles après,* `mutex` *est toujours consommée par* `Terminer F_C`*.*

La seconde structure parallèle représente l'exécution de la fonction, qui doit être inhibée si l'on a effectué une sortie par *time-out* : la fonction `Inhibition de F_EP` est ainsi une fonction de durée nulle qui, si elle est activée, empêche l'exécution de `F_EP` et transfère le contrôle à la construction suivante. Ce sont les items supplémentaires `Sortie normale` et `Sortie time-out` qui permettent « d'aiguiller » le flux de contrôle dans la seconde structure parallèle. D'après la remarque 3.11, ces deux items sont en exclusion mutuelle. Ainsi, bien que la seconde construction soit une structure parallèle, une seule de ses branches peut être exécutée.

La figure 3.14 illustre une autre modélisation du *time-out* mettant en œuvre un sous-scénario. Celui-ci décrit le comportement de la fonction décomposée `Consommation avec`

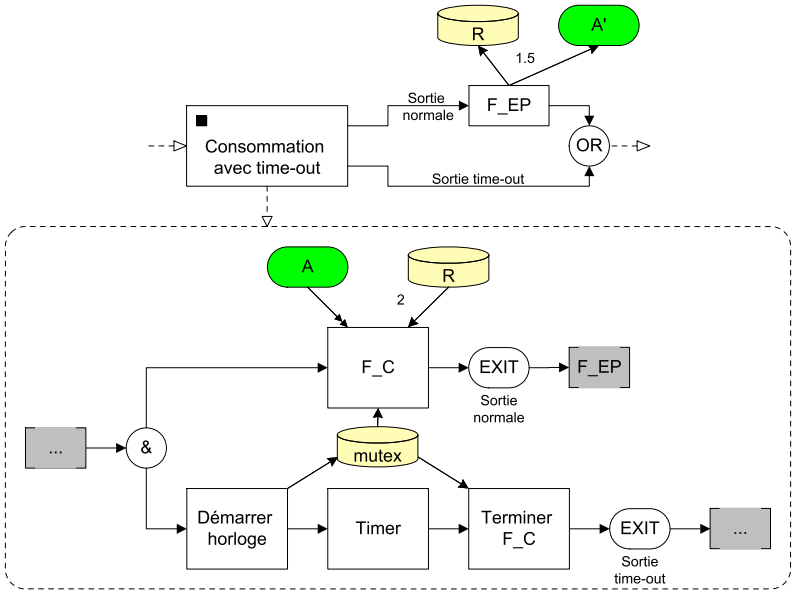


FIGURE 3.14 – Modélisation du *time-out* d'une fonction par une fonction décomposée

`time-out`, possédant deux états de sortie, `Sortie normale` et `Sortie time-out`. Le comportement est similaire au cas précédent et n'est pas détaillé plus avant.

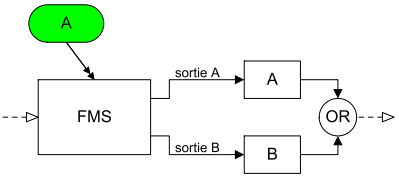
Remarque 3.13. *Ces deux motifs s'étendent sans difficulté au cas où la fonction possède plusieurs états de sortie, comme illustré figure 3.15(a). En effet, il suffit de copier les branches de sortie de la fonction dans une structure de choix qui se place :*

- *à la suite de la fonction* `Inhibition FMS_EP` *pour le premier motif, comme illustré figure 3.15(b) ;*
- *sur la branche de sortie* `Sortie time-out` *pour le second motif (par souci de concision, nous ne donnons pas l'illustration de ce motif, sensiblement similaire aux précédents).*

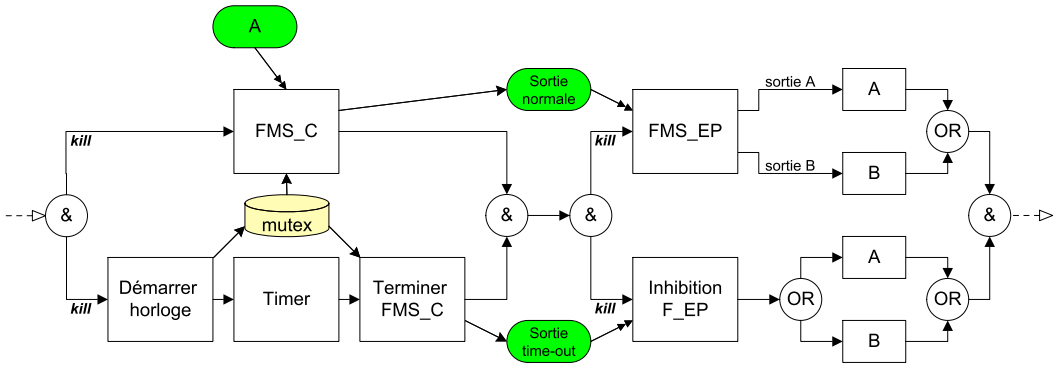
3.6.2 Le problème du passage à niveau

La modélisation d'un passage à niveau ferroviaire (ou PN) est un exercice classique dans le domaine des applications temps-réel [7, 51, 85, 89] ; ce cas d'étude rentre également dans le champ d'application de l'Ingénierie Système. Nous proposons dans cette section une modélisation simple d'un passage à niveau dit « à signalisation lumineuse et sonore à deux demi-barrières » ou PN-SAL2 (le plus répandu). Nous la compléterons par la suite afin d'illustrer les concepts et résultats présentés dans la suite de ce chapitre ainsi que dans les chapitres suivants.

Description du système Les caractéristiques techniques et le comportement des PNs sont régis par un arrêté du Ministère de l'Équipement [56]. Un PN-SAL2 est équipé des éléments suivants :



(a) Fonction multi-sortie consommatrice



(b) Modélisation du *time-out* par branches de terminaison

FIGURE 3.15 – Modélisation d'un *time-out* d'une fonction multi-sortie

- une sonnerie ;
- dans chaque sens de circulation routière :
  - deux feux rouges clignotants de part et d'autre de la chaussée ;
  - une demi-barrière sur la moitié droite de la chaussée.

On définit en amont et autour du passage à niveau proprement dit deux zones (cf. figure 3.16) :

1. la *zone d'annonce*, débutant à *l'origine d'annonce* (point A) ; l'entrée dans cette zone entraîne la mise en route de la procédure de fermeture du passage ;
2. la *zone courte*, s'étendant de part et d'autre du PN ; l'évacuation *complète* de cette zone permet la réouverture du passage.

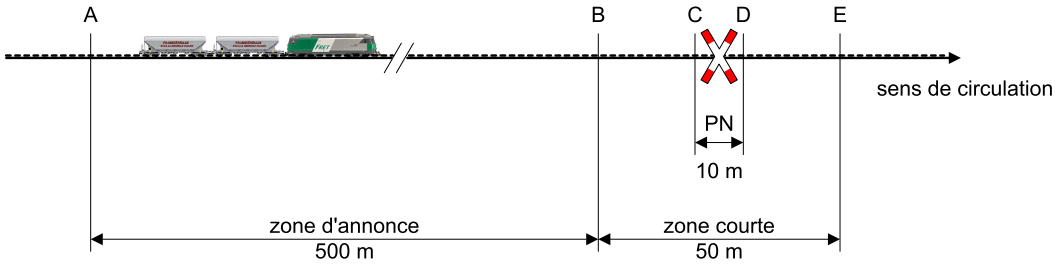


FIGURE 3.16 – Plan schématique d'un passage à niveau

Une *pédale d'annonce* est placée à l'origine d'annonce ; c'est ce dispositif, électromécanique ou électronique, qui détecte le passage du train. Une *pédale de réarmement*, située en sortie de

la zone courte (point E), repère le passage du dernier wagon du convoi [15]. La zone d'annonce est dimensionnée de manière à respecter un *délai d'annonce* entre l'entrée dans la zone et l'arrivée sur le passage (délimité par les points C et D), afin que la circulation routière soit évacuée puis interdite. L'arrêté fixe ce délai à 20 s ; la SNCF l'a porté à 25 s.

La dimension de la zone d'annonce dépend bien évidemment de la vitesse maximale des trains circulant sur la voie : on suppose ici que seuls des trains de fret empruntent la voie et qu'ils ont une vitesse comprise entre 35 km/h ($\approx$ 10 m/s) et 70 km/h ($\approx$ 20 m/s). Une longueur de 500 m est donc suffisante pour assurer le respect du délai d'annonce [16].

> **Remarque 3.14.** *L'inertie d'un convoi de fret est bien plus élevée que celle d'un véhicule routier : à 70 km/h, il faut environ 900 m pour obtenir l'arrêt complet. Ainsi, même si un obstacle est détecté sur le PN lors de l'entrée dans la zone d'annonce, le convoi ne pourra s'arrêter. On comprend donc que la sécurité du système repose uniquement sur le bon fonctionnement des barrières du passage.*

Une fois que le passage en A a été détecté, la séquence des événements est la suivante :

- allumage des feux clignotants et mise en marche de la sonnerie ;
- 5 secondes après le début du clignotement des feux : descente des deux demi-barrières ;
- une fois que les barrières sont en position basse (entre 5 et 10 secondes) : arrêt de la sonnerie ;
- après le passage du train : remontée des barrières ;
- une fois que les barrières sont en position haute (entre 10 et 15 secondes) : extinction des feux clignotants.

Enfin, les convois sont formés d'une motrice et d'une douzaine de wagons, ce qui représente une longueur totale de 190 m. Le tableau 3.1 donne le temps nécessaire pour atteindre chacun des points caractéristiques du PN, l'origine étant prise à la date d'arrivée en A, à la vitesse maximale de 20 m/s.

TABLE 3.1 – Temps de parcours du train sur le passage à niveau (v =20 m/s)

Trajet	Distance (m)	Durée (s)
Tête en A $\rightarrow$ tête en B	500	25
Tête en B $\rightarrow$ tête en C	20	1
Tête en C $\rightarrow$ queue en D	200	10
Queue en D $\rightarrow$ queue en E	20	1

Hypothèses de modélisation La modélisation du passage à niveau et de son *environnement* (c'est-à-dire des trains) nécessite quelques hypothèses supplémentaires, énoncées ci-dessous.

15. Ces dispositifs communiquent entre eux : en particulier, ils déterminent combien d'essieux comporte le convoi et sont capables de repérer par exemple si un wagon s'est détaché.

16. En réalité, les trains de *marchandises* ont une vitesse commerciale comprise entre 80 km/h et 100 km/h, et les trains de *messagerie* entre 120 km/h et 200 km/h. Nos valeurs restent cependant valides puisqu'il est courant que les trains ralentissent avant d'entrée dans la zone d'annonce, surtout si le profil de la voie entraîne une mauvaise visibilité.

Fréquence des trains On suppose qu'il s'agit d'une zone en voie unique et qu'au moins 2 minutes s'écoulent entre le passage de chaque train. Il suffit donc de modéliser un seul train parcourant de façon répétée le réseau. Par conséquent, il est inutile de prévoir une mécanisme de protection d'accès en exclusion mutuelle au PN. Cela implique en outre que les barrières ont toujours le temps de se relever complètement avant l'arrivée du train suivant.

> **Remarque 3.15.** *Si le PN comporte 2 voies ou plus, il est possible qu'un train soit à l'approche alors que les barrières sont en train de se relever suite au passage d'un train dans le sens inverse. Dans ce cas, et au vu des durées de montée et descente des barrières, il peut être nécessaire de forcer leur descente avant d'atteindre la position haute, de manière à respecter le délai d'annonce du second train. Une telle modélisation est proposée par exemple dans [15].*

Allure des convois Nous l'avons vu, la distance de freinage des convois à pleine vitesse est supérieure à la longueur **AE**. Nous supposerons ainsi que les convois gardent une vitesse comprise entre 10 et 20 m/s durant la traversée des deux zones. Cependant, cette vitesse peut ne pas être constante.

Comportement des usagers de la route Nous supposerons que les véhicules routiers respectent le code de la route :

- évacuation des véhicules dès le début de la sonnerie et arrêt de la circulation entrante ;
- aucune circulation entre le début de la descente des barrières et la fin de leur levée.

Sous cette hypothèse, il n'est pas nécessaire de modéliser le comportement des véhicules routiers.

Arrêt du système On ne représente pas de mécanisme de terminaison du système ; à titre indicatif, celui-ci pourrait se déclencher après un certain nombre de passages, après une certaine durée, sur réception d'une commande de l'agent de circulation, ...

Description du modèle La figure 3.17 donne le modèle EFFBD du passage à niveau ; pour simplifier la lecture, nous avons porté sur chaque fonction son intervalle de durée d'exécution lorsque celui-ci est différent de $[0, 0]$ (l'unité de temps est la seconde).

Nous avons adopté l'architecture classique à trois branches parallèles, représentant ;

- *l'environnement* (branche supérieure) : la branche décrit le mouvement du train sur le réseau (la zone en amont du point A et celle en aval du point E sont considérées identiques) ;
- *le contrôleur* : cette branche commande les mouvements du PN à partir de la circulation des trains ;
- *le passage à niveau* (branche inférieure) : la branche décrit le fonctionnement de la sonnerie et des barrières (par souci de lisibilité, nous n'avons pas représenté la gestion des feux clignotants, qui est triviale).

Les trois branches se synchronisent en échangeant des items : ceux-ci représentent les messages issus des capteurs de circulation et les ordres émis par le contrôleur. L'état de la barrière est modélisé par deux ressources, `ouvert` et `fermé` dont les valeurs initiales respectives sont 1 et 0.

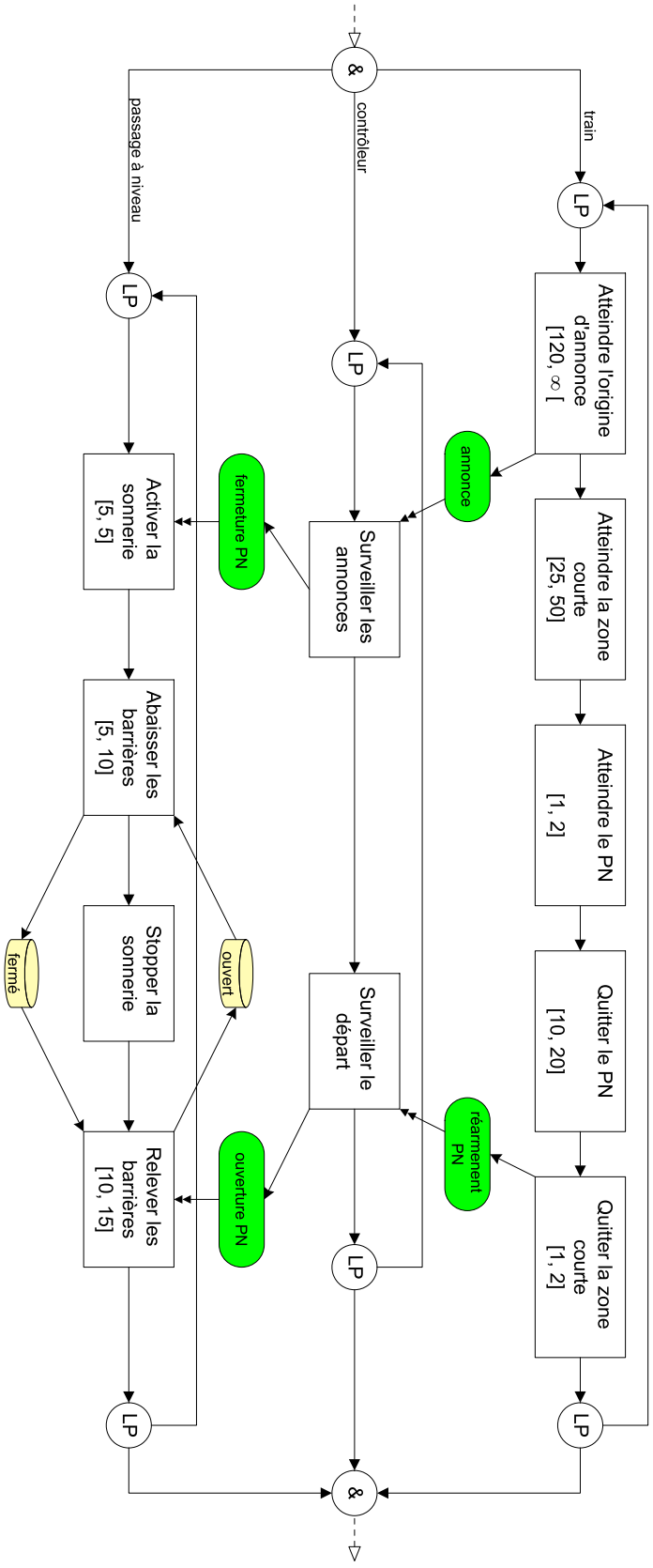


FIGURE 3.17 – Modèle EFFBD d'un passage à niveau

Chapitre 4

Formalisation des EFFBDs

Résumé *Ce chapitre donne la syntaxe et la sémantique formelles des diagrammes EFFBDs, introduites de manière informelle au chapitre précédent. C'est, à notre connaissance, la première fois qu'une telle formalisation est proposée.*

En particulier, la sémantique formelle est donnée sous forme d'un système de transitions temporisé, dont nous avons rappelé la définition au chapitre 2. Devant la richesse d'expressivité des EFFBDs, nous avons choisi de subdiviser notre démarche en trois étapes progressives : dans un premier temps, nous avons en effet défini et décrit un modèle atemporel simplifié pour ensuite prendre en compte l'écoulement du temps et enfin traiter les mécanismes de terminaisons forcées.

Cette étape de formalisation nous a permis d'obtenir plusieurs résultats fondamentaux tels que la définition du caractère borné *d'un EFFBD ou l'expression de conditions suffisantes pour garantir cette propriété. Ces résultats seront essentiels pour la suite de nos travaux, présentée dans les chapitres suivants.*

Sommaire

Nous avons pu le constater tout au long du chapitre précédent : le nombre de constructions disponibles et, plus généralement l'expressivité des diagrammes EFFBDs sont importants. Le comportement d'un modèle EFFBD, quoiqu'intuitif, obéit ainsi à de nombreuses règles d'évolution. C'est pourquoi il nous a paru judicieux de proposer la formalisation des EFFBDs en trois étapes successives, de manière à faciliter la lecture de la description formelle du langage.

Ainsi, nous avons choisi de présenter dans un premier temps une version « épurée » des EFFBDs (section 4.1), où ne figurent ni la modélisation du temps, ni les structures de terminaison forcée, que ce soient les branches parallèles affectées du modificateur *kill* ou les nœuds LE et EXIT [1].

Cette première description sera ensuite complétée de manière à prendre en compte l'écoulement du temps (section 4.2) puis les mécanismes de terminaison forcée (section 4.3). Nous adopterons dans ces différents cas la même architecture de présentation :

- présentation de la syntaxe formelle ;
- introduction (éventuelle) de définitions complémentaires, permettant d'alléger les expressions sémantiques ;
- présentation de la sémantique formelle ;
- illustration de ces différents éléments sur l'exemple du passage à niveau (excepté dans le dernier cas, le modèle ne comprenant aucune structure de terminaison forcée).

Enfin, nous donnerons dans la section 4.4 quelques définitions et propriétés établies à partir de ces descriptions formelles, qui nous seront particulièrement utiles dans les chapitres 5 et 6.

Auparavant, il nous paraît nécessaire de justifier dès à présent les choix que nous avons effectués pour représenter la sémantique des EFFBDs. Notre démarche consiste à écrire dans un premier temps cette sémantique sous forme d'un STT puis de donner la traduction des EFFBDs vers les TPNs. Nous avons en effet cherché à obtenir dans la preuve de l'équivalence entre un modèle EFFBD et sa traduction en TPN un cheminement qui soit aussi direct que possible. C'est pourquoi, sachant que la sémantique des TPNs est donnée sous forme d'un STT, nous avons eu recours à ce formalisme de très bas niveau, suffisamment puissant et expressif pour nos besoins.

Cependant, il serait légitime de s'interroger sur la possibilité d'exprimer directement cette sémantique au moyen, par exemple, des TPNs. Une telle approche de formalisation par traduction vers un modèle formel est d'ailleurs classique [2].

Néanmoins, une telle sémantique serait bien plus difficile à exploiter efficacement. Nous rappelons ainsi, par exemple, que les TA et les TPNs ne sont pas comparables entre eux [3] : appliquer une méthode d'analyse basée par exemple sur les TA nécessiterait des traductions supplémentaires de la sémantique si celle-ci était basée sur les TPNs [4]. Enfin, la sémantique

1. En particulier, la description des sous-scénarios liés à une fonction multi-sortie ne sera pas traitée dans cette première section.

2. Le comportement des *threads* d'un modèle AADL, pour citer un exemple récent, est ainsi exprimé au moyen d'automates hybrides [79].

3. Tout comme différentes « variétés » de TPNs, tels les transition-TPNs et les place-TPNs, ne sont pas comparables entre eux.

4. *A priori*, ces traductions seraient d'ailleurs basées sur l'écriture de la sémantique des modèles sous forme d'un STT.

telle qu'elle est définie aujourd'hui se prête plus facilement à la définition d'extensions permettant, par exemple, de définir des comportements hybrides.

4.1 Les EFFBDs atemporels

Comme précisé en introduction, nous nous restreignons dans un premier temps aux EFFBDs *atemporels (untimed EFFBDs)*, en faisant abstraction du temps et des structures de terminaison forcée.

Les cas triviaux tels que les réplications et les sous-scénarios décrivant des fonctions à sortie simple ne sont pas traités ici. De même, on supposera qu'à chaque fonction ne correspond qu'une seule structure fonctionnelle. Enfin, seuls sont considérés les ressources et les triggers purs, dont on supposera qu'ils sont dépliés comme précisé dans la section 3.5.1. D'après la remarque 3.9, on pourra donc regrouper ces deux notions sous le terme générique de *flux (flow)*.

4.1.1 Syntaxe des EFFBDs atemporels

Nous donnons ci-dessous la définition d'un EFFBD atemporel. Celle-ci fait intervenir la notion *d'arcs de contrôle avants* (resp. *arrières*) : ceux-ci correspondent, sur un diagramme EFFBD, aux flèches reliant deux nœuds et dirigées de gauche à droite (resp. de droite à gauche). On notera que seuls les nœuds IT et LP sont reliés par des arcs arrières.

Ainsi, un nœud AND ouvrant, par exemple, est défini comme étant un objet contenant le symbole `AND` et tel qu'il existe au plus un arc avant y arrivant et $n \geq 2$ arcs avants en partant. De la définition de tous les types de nœuds découle un certain nombre de contraintes sur les arcs de contrôle. Dans un souci de lisibilité, nous avons considéré qu'elles sont suffisamment intuitives pour être omises ici.

Nous introduisons en outre la notion *d'arcs de flux*. Ceux-ci relient une fonction à un flux et sont pondérés par un poids entier, strictement positif[5] (valant 1 lorsque l'arc relie un trigger). Ces arcs appartiennent à trois ensembles distincts :

- $\mathcal{A}_{\mathcal{F},C}$ ensemble des *arcs de consommation* ;
- $\mathcal{A}_{\mathcal{F},L}$ ensemble des *arcs de lecture* ;
- $\mathcal{A}_{\mathcal{F},P}$ ensemble des *arcs de production*.

Soit f une fonction consommant 2 unités de la ressource res ; l'ensemble $\mathcal{A}_{\mathcal{F},C}$ contient donc l'arc noté $(f, 2, res)_C$. On suppose en outre qu'il existe au plus *un seul* arc de chaque type reliant une fonction et un flux donnés.

Définition 4.1 (EFFBD atemporel) *Un* EFFBD atemporel *est un 6-uplet* $\mathcal{E}_U = (\mathcal{N}, \mathcal{F}, \mathcal{A}, iter, n_0, F_0)$ *où :*

- $\mathcal{N}$ *est un ensemble fini et non vide de* nœuds*, formé de la réunion des ensembles suivants*[6] *:*
  - AND_{in} *et* AND_{out}*, l'ensemble des nœuds* AND *ouvrants et fermants ;*

5. On aurait également pu prendre ces poids dans $\mathbb{Q}_{>0}$ sans difficulté supplémentaire.
6. Ces ensembles sont bien sûr disjoints deux à deux.

  - OR_{in} *et* OR_{out}*, l'ensemble des nœuds* OR *ouvrants et fermants ;*
  - LP_{in} *et* LP_{out}*, l'ensemble des nœuds* LP *ouvrants et fermants ;*
  - IT_{in} *et* IT_{out}*, l'ensemble des nœuds* IT *ouvrants et fermants ;*
  - FC*, l'ensemble des fonctions.*
- $\mathcal{F}$ *est un ensemble fini de* flux[7] *;*
- $\mathcal{A}$ *est un ensemble fini et non vide* d'arcs de contrôle *et* de flux *:* $\mathcal{A} = \mathcal{A}_C \cup \mathcal{A}'_C \cup \mathcal{A}_\mathcal{F}$ *avec :*
  - $\mathcal{A}_C \subseteq \mathcal{N} \times \mathcal{N}$ *: ensemble non vide* d'arcs de contrôle avants *;*
  - $\mathcal{A}'_C \subseteq (LP_{out} \times LP_{in}) \cup (IT_{out} \times IT_{in})$ *: ensemble* d'arcs de contrôle arrières *;*
  - $\mathcal{A}_\mathcal{F} \subseteq FC \times \mathbb{N}_{>0} \times \mathcal{F}$ *: ensemble* d'arcs de flux *;*
- $iter \in \mathbb{N}_{>0}^{IT_{in}}$ *est la* fonction d'itération*, donnant pour chaque structure d'itération (réduite à son nœud ouvrant) le nombre d'itérations à effectuer ;*
- $n_0 \in AND_{in} \cup OR_{in} \cup LP_{in} \cup IT_{in} \cup FC$ *est le* nœud initial *;*
- $F_0 \in \mathbb{N}^\mathcal{F}$ *est la fonction associant à chaque flux sa* valeur initiale[8]*.*

Dans la suite, les nœuds ne représentant pas de fonction seront généralement notés par une lettre grecque (surmontée d'une barre pour les nœuds fermants).

4.1.2 Définitions complémentaires

Nous introduisons dans cette section plusieurs définitions complémentaires, qui permettront de simplifier les expressions sémantiques données dans la section suivante.

Définition 4.2 (Prédécesseurs et successeurs) *Soient* $\mathcal{N}$ *un ensemble de nœuds,* n *un élément de* $\mathcal{N}$ *et* $\mathcal{A}_C$ *un ensemble d'arcs de contrôle avants. Les* prédécesseurs *du nœud* n *forment l'ensemble* $Pre(n)$ *défini par :*

$$Pre(n) = \{n' \in \mathcal{N} / (n', n) \in \mathcal{A}_C\}$$

Les successeurs *de* n*, notés* $Post(n)$ *sont défini par :*

$$Post(n) = \{n' \in \mathcal{N} / (n, n') \in \mathcal{A}_C\}$$

On notera que, de par la définition respective des ensembles $\mathcal{A}_C$ et $\mathcal{A}'_C$, l'ensemble $Post(\bar{\imath})$ (où $\bar{\imath}$ est un nœud IT fermant) ne contient pas le nœud IT ouvrant correspondant mais bien, s'il existe, le premier nœud suivant la structure complète d'itération ; la même remarque s'applique pour les nœuds LP.

Définition 4.3 (Chemins) *Soient* $\mathcal{N}$ *un ensemble de nœuds,* $\mathcal{A}_C$ *un ensemble d'arcs de contrôle avants,* n_0 *et* n_m *deux éléments, distincts, de* $\mathcal{N}$ *(*$m \in \mathbb{N}_{>0}$*). Les* chemins de n_0 à n_m *sont définis par :*

$$Paths(n_0, n_m) = \left\{(n_0, \ldots, n_m) \in \mathcal{N}^{m+1} / \forall\, 0 \leq i < m,\ (n_i, n_{i+1}) \in \mathcal{A}_C\right\}$$

7. Comme précisé plus haut, il est formé de la réunion des ensembles $\mathcal{T}$ et $\mathcal{R}$ représentant respectivement les triggers et les ressources du modèle ($\mathcal{T} \cap \mathcal{R} = \emptyset$).

8. On a bien sûr $F_0(t) = 0$ pour tout trigger t de $\mathcal{T}$.

Dans la suite, la notation $(n, n', \ldots, m', m) \in Paths(n, n')$ signifie qu'il existe au moins un chemin débutant par n puis n' et se terminant par m' puis m.

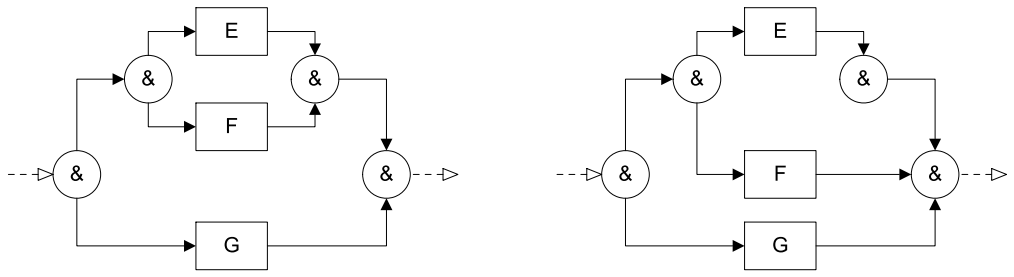
Définition 4.4 (Structures parallèles) *Soient $\alpha \in AND_{in}$ et $\bar{\alpha} \in AND_{out}$ deux nœuds* AND. *La* structure $\langle \alpha, \bar{\alpha} \rangle$ *appartient à AND,* l'ensemble des structures parallèles, *si et seulement si les deux conditions suivantes sont vérifiées :*

$$\forall n \in Post(\alpha) : \begin{cases} n \in Pre(\bar{\alpha}) & (4.1) \\ \text{ou } \exists!\ n' \in Pre(\bar{\alpha}) \text{ t.q. } (\alpha, n, \ldots, n', \bar{\alpha}) \in Paths(\alpha, \bar{\alpha}) & (4.2) \end{cases}$$

$$\forall n' \in Pre(\bar{\alpha}) : \begin{cases} n' \in Post(\alpha) & (4.3) \\ \text{ou } \exists!\ n \in Post(\alpha) \text{ t.q. } (\alpha, n, \ldots, n', \bar{\alpha}) \in Paths(\alpha, \bar{\alpha}) & (4.4) \end{cases}$$

Les sous-conditions 4.1 et 4.3 correspondent au cas où la branche parallèle est vide ou ne comprend qu'un seul nœud (nécessairement dans FC). Les sous-conditions 4.2 et 4.4 garantissent que toute branche partant d'un nœud AND ouvrant arrive au nœud fermant correspondant et réciproquement. Il n'y a donc pas de branche « flottante » ou incomplète.

Exemple 5. *Considérons le modèle illustré figure 4.1(a). Dans la suite, les nœuds* AND *sont notés, de gauche à droite, α, α', $\bar{\alpha}'$ et $\bar{\alpha}$.*



(a) Construction autorisée (b) Construction interdite

FIGURE 4.1 – Structures parallèles emboîtées

Les structures $\langle \alpha, \bar{\alpha} \rangle$ et $\langle \alpha', \bar{\alpha}' \rangle$ appartiennent à AND ; en revanche, ni $\langle \alpha, \bar{\alpha}' \rangle$ ni $\langle \alpha', \bar{\alpha} \rangle$ ne sont des structures parallèles autorisées. En effet, dans le premier cas, il n'existe pas de chemin reliant α à $\bar{\alpha}'$ et passant par G, ce qui contredit les propositions 4.1 et 4.2. Dans le second cas, aucun chemin passant par G et arrivant en $\bar{\alpha}$ ne part de α', ce qui contredit les propositions 4.3 et 4.4.

Considérons à présent le modèle illustré figure 4.1(b). De nouveau, les nœuds AND *sont notés, de gauche à droite, α, α', $\bar{\alpha}'$ et $\bar{\alpha}$. Ce modèle ne contient aucune structure parallèle correctement formée :*

- *$\langle \alpha, \bar{\alpha} \rangle \notin AND$ car au nœud α', successeur de α correspondent* deux *précédesseurs de $\bar{\alpha}$, $\bar{\alpha}'$ et F, ce qui contredit la proposition 4.2 ;*
- *$\langle \alpha', \bar{\alpha}' \rangle \notin AND$ car aucun chemin passant par F ne relie α' à $\bar{\alpha}'$, ce qui contredit 4.1 et 4.2 ;*
- *$\langle \alpha, \bar{\alpha}' \rangle \notin AND$ car aucun chemin ne relie G à $\bar{\alpha}'$, ce qui contredit 4.1 et 4.2 ;*
- *$\langle \alpha', \bar{\alpha} \rangle \notin AND$ car aucun chemin passant par G et arrivant en $\bar{\alpha}$ ne part de α', ce qui contredit 4.3 et 4.4.*

La définition du *contenu* d'une structure parallèle s'en déduit immédiatement. On remarquera que cette définition exclut les nœuds α et $\bar{\alpha}$ du contenu de $\langle \alpha, \bar{\alpha} \rangle$.

Définition 4.5 (Contenu d'une structure parallèle) *Le* contenu d'une structure parallèle $and = \langle \alpha, \bar{\alpha} \rangle$, *noté* $\mathcal{C}(and)$, *est l'ensemble des noeuds* $n \in \mathcal{N}$ *tels qu'il existe un chemin de* α *à* n *mais aucun chemin de* $\bar{\alpha}$ *à* n. *Formellement :*

$$\forall n \in \mathcal{N},\ n \in \mathcal{C}(and) \Leftrightarrow \begin{cases} Paths(\alpha, n) \neq \emptyset \\ Paths(\bar{\alpha}, n) = \emptyset \end{cases}$$

L'ensemble OR des *structures de sélection* ainsi que le contenu $\mathcal{C}(or)$ d'une structure de sélection *or* sont définis de façon similaire. On étend également ces définitions à l'ensemble des structures de fonctions multi-sorties, FC_{MS}, en remplaçant α (respectivement $\bar{\alpha}$) par $f \in FC$ (respectivement $\bar{\omega} \in OR_{out}$) dans la définition précédente, et au contenu d'une fonction multi-sortie f, $\mathcal{C}(f\!\prec\!)$ avec $f\!\prec\! = \langle f, \bar{\omega} \rangle$.

Définition 4.6 (Structures de boucle) *Soient* $\lambda \in LP_{in}$ *et* $\bar{\lambda} \in LP_{out}$ *deux nœuds* LP. *La structure* $lp = \langle \lambda, \bar{\lambda} \rangle$ *appartient à* LP, l'ensemble des structures de boucle, *si et seulement si les trois conditions suivantes sont vérifiées :*

$$(\bar{\lambda}, \lambda) \in \mathcal{A}'_C$$

$$\forall n \in Post(\lambda) : \begin{cases} n \in Pre(\bar{\lambda}) \\ \text{ou } \exists!\ n' \in Pre(\bar{\lambda}) \text{ t.q. } (\lambda, n, \ldots, n', \bar{\lambda}) \in Paths(\lambda, \bar{\lambda}) \end{cases}$$

$$\forall n' \in Pre(\bar{\lambda}) : \begin{cases} n' \in Post(\lambda) \\ \text{ou } \exists!\ n \in Post(\lambda) \text{ t.q. } (\lambda, n, \ldots, n', \bar{\lambda}) \in Paths(\lambda, \bar{\lambda}) \end{cases}$$

L'ensemble IT des *structures d'itération* est défini de façon similaire. Le contenu d'une structure de boucle, $\mathcal{C}(lp)$ ou celui d'une structure d'itération, $\mathcal{C}(it)$, est défini de la même façon que précédemment.

Nous pouvons à présent définir de façon formelle la notion *d'EFFBD bien formé*.

Définition 4.7 (EFFBD bien formé) *Un EFFBD* $\mathcal{E}_U$ *est* bien formé *si et seulement si il respecte toutes les propriétés suivantes :*

- *unicité des structures : tout nœud ne représentant pas une fonction à sortie simple est constitutif d'exactement une structure :*

$$\begin{cases} \forall \alpha \in AND_{in}, \exists!\ \bar{\alpha} \in AND_{out} / \langle \alpha, \bar{\alpha} \rangle \in AND \\ \forall \bar{\alpha} \in AND_{out}, \exists!\ \alpha \in AND_{in} / \langle \alpha, \bar{\alpha} \rangle \in AND \end{cases}$$

  *(*idem *pour les ensembles OR, LP, IT et* FC_{MS}*) ;*
- *continuité : tout nœud différent de* n_0 *possède au moins un prédécesseur.* $\mathcal{E}_U$ *ne contient qu'un seul nœud final, qui n'a pas de successeur :*

$$\begin{cases} \forall n \in \mathcal{N}, Pre(n) = \emptyset \Leftrightarrow n = n_0 \\ \exists!\ n \in \mathcal{N} / Post(n) = \emptyset \end{cases}$$

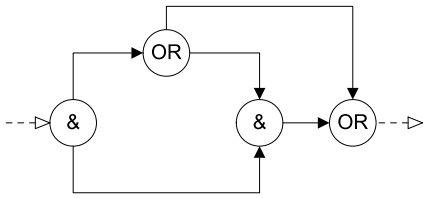


FIGURE 4.2 – EFFBD mal formé

– *imbrication : deux structures quelconques sont soit en séquence, soit entièrement emboîtées l'une dans l'autre :*

$$\forall x, \langle n, \bar{n} \rangle \in AND \cup OR \cup IT \cup LP \cup FC_{MS},\ n \in \mathcal{C}(x) \Leftrightarrow \bar{n} \in \mathcal{C}(x)$$

Exemple 6. *La figure 4.2 illustre un modèle minimaliste ; les nœuds sont notés, de gauche à droite, α, ω, $\bar{\alpha}$ et $\bar{\omega}$. Le modèle est mal formé car il ne respecte pas la propriété d'imbrication. On peut en effet vérifier que l'on a bien $and = \langle \alpha, \bar{\alpha} \rangle \in AND$ et $or = \langle \omega, \bar{\omega} \rangle \in OR$; en revanche, puisque $\mathcal{C}(and) = \{\omega\}$, on a $\omega \in \mathcal{C}(and)$ et $\bar{\omega} \notin \mathcal{C}(and)$. De même, $\bar{\alpha} \in \mathcal{C}(or)$ mais $\alpha \notin \mathcal{C}(or)$.*

Enfin, nous définissons trois opérateurs intuitifs de manipulation des flux de données ; ils nous éviteront de travailler dans la suite directement avec les arcs de flux.

Définition 4.8 (Relation de consommation) *Soient FC un ensemble de fonctions, $\mathcal{F}$ un ensemble de flux et $\mathcal{A}_{\mathcal{F},C}$ un ensemble d'arcs de consommation. La* relation de consommation *$Cons : FC \to \mathbb{N}^{\mathcal{F}}$ est définie pour tous $f_c \in FC$ et $f_l \in \mathcal{F}$ par :*

$$Cons(f_c)[f_l] = \begin{cases} k & \text{si } \exists\ k > 0 \text{ tel que } (f_c, k, f_l)_C \in \mathcal{A}_{\mathcal{F},C} \\ 0 & \text{sinon} \end{cases}$$

Les relations de lecture *Lect* et de production *Prod* sont définies de façon similaire, en remplaçant $\mathcal{A}_{\mathcal{F},C}$ par $\mathcal{A}_{\mathcal{F},L}$ et $\mathcal{A}_{\mathcal{F},P}$ dans la définition ci-dessus.

Remarque 4.1. *Soient $f \in FC$ et $res \in \mathcal{F}$ tels que f consomme 3 unités de res et en lit 5 unités*[9]*. Par convention, on considère que la fonction f nécessite 5 unités de res pour démarrer, pour n'en consommer effectivement que 3. La quantité minimale de ressource nécessaire est donc donnée par* $\max(Cons(f)[res], Lect(f)[res])$.

4.1.3 Sémantique des EFFBDs atemporels

La sémantique comportementale des EFFBDs atemporels est exprimée sous la forme d'un STE ; de façon similaire, nous exprimerons la sémantique des modèles incluant le temps par un STT. Les expressions sémantiques que nous donnons dans cette section sont passablement complexes : elles sont en effet le reflet de la riche expressivité des EFFBDs.

9. $(f, 3, res)_C, (f, 5, res)_L \in \mathcal{A}_{\mathcal{F}}$

Activité des nœuds Avant de définir la sémantique du formalisme, il nous faut au préalable définir la notion *d'activité* d'un nœud, ce qui nous permettra par la suite de spécifier la nature des états du système de transitions. Un nœud peut en effet être *inactif (inactive)*, *activé (enabled)* ou, s'il précède un nœud AND fermant et qu'il a fini son exécution mais que l'on attend la fin des autres branches parallèles, *exécuté (executed)*. Enfin, les nœuds représentant des fonctions possèdent une activité supplémentaire, *en exécution (executing)*.

Définition 4.9 (Énumérations des activités d'un nœud) L'activité $A(n)$ *d'un nœud* n *appartient à l'ensemble :*

- $\mathbb{A} = \{inactive; enabled; executing; executed\}$ *si* $n \in FC$ *;*
- $\mathbb{A}^* = \{inactive; enabled; executed\}$ *sinon.*

> **Remarque 4.2.** *Dans le cas des nœuds fermants* IT *et* LP*, le contrôle revient toujours au nœud ouvrant correspondant (même si, dans le cas de d'itération, le compteur a atteint la valeur maximale). Le comportement de ces nœuds fermants est donc en réalité dirigé par celui des nœuds ouvrants ; en particulier, ils ne peuvent jamais être directement dans l'état executed : si nécessaire, c'est lors du traitement du nœud ouvrant que l'on pourra les placer dans cet état.*

Initialement, tous les nœuds, excepté n_0, sont inactifs ; on définit ainsi l'activité initiale du système, A_0, par :

$$\forall n \in \mathcal{N},\ A_0(n) = \begin{cases} enabled & \text{si } n = n_0 \\ inactive & \text{sinon} \end{cases}$$

La séquence d'activité « normale » d'un nœud non fonctionnel est donc *inactive* puis *enabled* et enfin *executed* ou *inactive*, selon sa nature et celle de son successeur ; pour une fonction, la séquence normale est *inactive* puis *enabled*, puis *executing* et enfin *executed* ou *inactive* [10].

États du système Un *état* du système de transitions définissant la sémantique des EFFBDs atemporels est un triplet $s = (A, C, N)$ représentant *l'activité* des nœuds, les *compteurs* d'itération et le *niveau* des flux.

> **Remarque 4.3.** *La valeur du compteur relatif à une itération donnée est définie comme étant le nombre de fois que la première structure de la branche itérée a été activée. En effet, conformément à ce qui était décrit dans la section 3.3.2, on n'observe non pas le nombre de fois que l'itération s'est achevée mais bien le nombre de fois qu'elle a démarré. Ce choix, non immédiat, est lié au motif de traduction en réseau de* PETRI *et à la définition de la relation d'équivalence entre un modèle EFFBD et sa traduction en TPN, comme nous le verrons par la suite.*
>
> *De même, pour éviter de créer une règle spécifique pour initialiser le compteur lors de la première itération, celui-ci est remis à zéro lors de la sortie de l'itération.*

Écriture de la sémantique L'écriture de la sémantique des EFFBDs, ou plus exactement celle de la relation de transition, fait intervenir des propositions logiques $\mathcal{P}_x$, données à la suite

10. En particulier, une fonction ne peut passer directement de l'état *enabled* à *executed* : elle doit en effet au préalable passer par l'état *executing*.

de la définition de la sémantique et dépendant de la nature du nœud formant la transition courante. Ainsi, la proposition $\mathcal{P}_*$ décrit la règle « générique » selon laquelle :

- le nœud courant est dans l'état *enabled* ;
- tous ses successeurs deviennent activés ;
- le nœud courant devient désactivé ;
- tous les autres nœuds ainsi que les compteurs et les niveaux ne sont pas affectés.

Cette règle s'applique aux nœuds AND et LP ouvrants ainsi qu'aux nœuds OR fermants. On notera que trois règles définissent le comportement d'une fonction :

- $\mathcal{P}_f$ décrit les conditions pour qu'une fonction, qu'elle soit multi-sortie ou non, démarre son exécution (la fonction doit être activée et les flux d'entrée disponibles) ;
- $\mathcal{P}_{\bar{f}}$ décrit les conditions pour qu'une fonction à sortie simple achève son exécution ;
- $\mathcal{P}_{\bar{f}\prec\!\!\!\cdot}$ décrit les conditions pour qu'une fonction multi-sortie achève son exécution [11].

De même, selon que le compteur associé à un nœud IT ouvrant donné a atteint sa valeur maximale ou non [12], on appliquera l'une des deux règles $\mathcal{P}_{\iota=}$ ou $\mathcal{P}_{\iota<}$.

Définition 4.10 (Sémantique d'un EFFBD atemporel) *La* sémantique comportementale d'un EFFBD $\mathcal{E}_U = (\mathcal{N}, \mathcal{F}, \mathcal{A}, iter, n_0, F_0)$ *est un quadruplet* $||\mathcal{E}_U|| = (S, s_0, \mathcal{N}, \rightarrow)$ *où :*

- $S \subseteq \mathbb{A}^{\mathcal{N}} \times \mathbb{N}^{IT_{in}} \times \mathbb{N}^{\mathcal{F}}$ *est l'ensemble des* états *du système ;*
- $s_0 = (A_0, \vec{0}, F_0)$ *est* l'état initial *;*
- $\rightarrow \subseteq S \times \mathcal{N} \times S$ *est la relation de transition (discrète), définie* $\forall n \in \mathcal{N}$ *par :*

$$(A, C, N) \xrightarrow{n} (A', C', N') \quad \Leftrightarrow \quad \begin{cases} \begin{cases} A(n) = enabled \wedge NodeBehavior \\ \text{ou } A(n) = executing \wedge \begin{cases} \mathcal{P}_{\bar{f}} & \text{si } n \notin FC_{MS} \\ \mathcal{P}_{\bar{f}\prec\!\!\!\cdot} & \text{sinon} \end{cases} \end{cases} \\ A'(n) = NextActivity \end{cases}$$

avec :

$$NextActivity \quad = \quad \begin{cases} executed & \text{si } \begin{cases} Post(n) \cap AND_{out} \neq \emptyset \\ n \notin IT_{out} \cup LP_{out} & \text{(cf. rem. 4.2)} \\ (A(n) = enabled) \Rightarrow n \notin FC & \text{(cf. note 10)} \end{cases} \\ executing & \text{si } (n \in FC \wedge A(n) = enabled) \\ inactive & \text{sinon} \end{cases}$$

et :

$$NodeBehavior \quad = \quad \begin{cases} n \in AND_{in} \cup OR_{out} \cup LP_{in} \Rightarrow \mathcal{P}_* \\ n \in AND_{out} \Rightarrow \mathcal{P}_{\bar{\alpha}} \\ n \in OR_{in} \Rightarrow \mathcal{P}_{\omega} \\ n \in IT_{in} \Rightarrow \begin{cases} \mathcal{P}_{\iota<} & \text{si } C(n) < iter(n) \\ \mathcal{P}_{\iota=} & \text{sinon} \end{cases} \\ n \in LP_{out} \cup IT_{out} \Rightarrow \mathcal{P}_{\overline{\lambda\iota}} \\ n \in FC \Rightarrow \mathcal{P}_f \end{cases}$$

11. Il s'agit d'une combinaison de $\mathcal{P}_{\bar{f}}$ et de $\mathcal{P}_{\omega}$, la règle décrivant le comportement d'un nœud OR ouvrant.

12. On peut vérifier que les règles sémantiques garantissent que la valeur courante d'un compteur ne devient jamais strictement supérieure à sa valeur maximale.

NextActivity décrit l'activité résultante du nœud courant ; la plupart du temps, celui-ci devient inactif. *NodeBehavior* décrit les conditions que doit remplir le système pour pouvoir traiter le nœud et les conséquences sur l'état suivant ; cette condition se présente sous forme de propositions en exclusion mutuelle. En notant $\mathcal{N}_{-n}$ l'ensemble $\mathcal{N} \setminus \{n\}$, ces propositions sont :

$$\mathcal{P}_* = \begin{cases} \forall n' \in \mathcal{N}_{-n},\ A'(n') = \begin{cases} enabled & \text{si } n' \in Post(n) \\ A(n') & \text{sinon} \end{cases} \\ C' = C \\ N' = N \end{cases}$$

$$\mathcal{P}_{\bar{\alpha}} = \begin{cases} \forall n' \in \mathcal{N}_{-n}, \begin{cases} (A(n') = executed \ \wedge \ A'(n') = inactive) & \text{si } n' \in Pre(n) \\ A'(n') = enabled & \text{si } n' \in Post(n) \\ A'(n') = A(n') & \text{sinon} \end{cases} \\ C' = C \\ N' = N \end{cases}$$

$$\mathcal{P}_{\omega} = \begin{cases} \exists!\ n_{Select} \in Post(n) \text{ t.q.} \begin{cases} A'(n_{Select}) = enabled \\ \forall n' \in \mathcal{N}_{-n},\ A'(n') = A(n') \end{cases} \\ C' = C \\ N' = N \end{cases}$$

$$\mathcal{P}_{\iota<} = \begin{cases} \forall n' \in \mathcal{N}_{-n},\ A'(n') = \begin{cases} enabled & \text{si } n' \in Post(n) \\ A(n') & \text{sinon} \end{cases} \\ \forall \iota \in IT_{in}, C'(\iota) = \begin{cases} C(\iota) + 1 & \text{si } \iota = n \\ C(\iota) & \text{sinon} \end{cases} \\ N' = N \end{cases}$$

$$\mathcal{P}_{\iota=} = \begin{cases} \forall n' \in \mathcal{N}_{-n},\ A'(n') = \begin{cases} enabled & \text{si } n' \in Post(\bar{n}) \text{ avec } \langle n, \bar{n} \rangle \in IT \\ executed & \text{si } (n' = \bar{n}) \wedge (Post(\bar{n}) \cap AND_{out} \neq \emptyset) \\ A(n') & \text{sinon} \end{cases} \\ \forall \iota \in IT_{\iota}, C'(\iota) = \begin{cases} 0 & \text{si } \iota = n \quad \text{(cf. rem. 4.3)} \\ C(\iota) & \text{sinon} \end{cases} \\ N' = N \end{cases}$$

$$\mathcal{P}_{\overline{\lambda\iota}} = \begin{cases} \forall n' \in \mathcal{N}_{-n},\ A'(n') = \begin{cases} enabled & \text{si } \langle n', n \rangle \in LP \cup IT \quad \text{(cf. rem. 4.2)} \\ A(n') & \text{sinon} \end{cases} \\ C' = C \\ N' = N \end{cases}$$

$$\mathcal{P}_f = \begin{cases} \forall n' \in \mathcal{N}_{-n},\ A'(n') = A(n') \\ C' = C \\ \begin{cases} N \geq \max(Cons(n), Lect(n)) \text{ (cf. rem. 3.9)} \\ N' = N - Cons(n) \end{cases} \end{cases}$$

$$\mathcal{P}_{\bar{f}} = \begin{cases} \forall n' \in \mathcal{N}_{-n},\ A'(n') = \begin{cases} enabled & \text{si } n' \in Post(n) \\ A(n') & \text{sinon} \end{cases} \\ C' = C \\ N' = N + Prod(n) \end{cases}$$

$$\mathcal{P}_{\bar{f}\prec} = \begin{cases} \exists!\ n_{Select} \in Post(n) \text{ t.q. } \begin{cases} A'(n_{Select}) = enabled \\ \forall n' \in \mathcal{N}_n,\ A'(n') = A(n') \end{cases} \\ C' = C \\ N' = N + Prod(n) \end{cases}$$

4.1.4 Application au passage à niveau atemporel

Considérons à nouveau l'exemple du passage à niveau, décrit dans la section 3.6.2 ; dans cette section, nous ferons abstraction des durées des fonctions. Afin d'alléger la description des exécutions, nous adoptons la nomenclature des nœuds et flux donnée dans le tableau 4.1 ci-dessous.

Remarque 4.4. *Pour plus de commodité, nous avons reproduit la figure 3.17 au dos du 4^e de couverture de la version imprimée de ce document. Nous y avons également reporté le nom de chaque nœud.*

TABLE 4.1 – Nomenclature des nœuds et flux du modèle de la figure 3.17

$\alpha/\bar{\alpha}$	: nœud AND ouvrant/fermant
$\lambda_t/\bar{\lambda}_t$	: nœud LP ouvrant/fermant (branche *train*)
f_{Att_OA}	: fonction *Atteindre l'origine d'annonce*
f_{Att_ZC}	: fonction *Atteindre la zone courte*
f_{Att_PN}	: fonction *Atteindre le PN*
f_{Qui_PN}	: fonction *Quitter le PN*
f_{Qui_ZC}	: fonction *Quitter la zone courte*
$\lambda_c/\bar{\lambda}_c$	: nœud LP ouvrant/fermant (branche *contrôleur*)
f_{Sur_an}	: fonction *Surveiller les annonces*
f_{Sur_de}	: fonction *Surveiller le départ*
$\lambda_p/\bar{\lambda}_p$	: nœud LP ouvrant/fermant (branche *passage à niveau*)
f_{Act_so}	: fonction *Activer la sonnerie*
f_{Ab_bar}	: fonction *Abaisser les barrières*
f_{Sto_so}	: fonction *Stopper la sonnerie*
f_{Re_bar}	: fonction *Relever les barrières*
ann	: item *annonce*
$ferPN$	: item *fermeture PN*
$reaPN$	: item *réarmement PN*
$ouvPN$	: item *ouverture PN*
ouv	: ressource *ouvert*
fer	: ressource *fermé*

Enfin, pour distinguer parmi les transitions celles qui décrivent le démarrage d'une fonction (respectivement son arrêt), nous ferons précéder le nom de la transition du symbole $\uparrow$ (respectivement $\downarrow$).

Remarque 4.5. *Cela revient à remplacer dans la définition de la sémantique l'alphabet des actions $\mathcal{N}$ par $\{\mathcal{N} \setminus FC\} \cup \{\{\uparrow;\downarrow\} \times FC\}$ et à adapter l'expression de la relation de transition en conséquence. Cette transformation est triviale mais, à notre sens, complique encore l'expression de la sémantique. De même, nous avons choisi de ne pas faire figurer dans le nom de l'action liée au traitement d'un nœud* OR *ouvrant l'indication de la branche de sélection choisie, de manière à expliciter le non déterminisme du traitement de ce type de nœud.*

L'équation 4.5 ci-dessous représente le début d'une exécution de l'EFFBD atemporel correspondant au modèle du passage à niveau. Le détail de chaque état est donné dans le tableau 4.2 ; nous n'y faisons figurer que les informations « intéressantes » : ainsi, le tableau ne mentionne ni les nœuds inactifs, ni les niveaux de flux lorsque ceux-ci sont nuls.

Remarque 4.6. *On notera que, dans le modèle, aucun nœud ne peut atteindre l'activité executed. En effet, seuls les nœuds* LP *fermants sont susceptibles d'avoir cette activité puisqu'ils précèdent le nœud* AND *fermant ; or, d'après les règles sémantiques énoncées plus haut (dont la règle $\mathcal{P}_{\overline{\lambda_t}}$) et l'expression de NextActivity, les nœuds de ce type ne peuvent jamais devenir executed.*

Les boucles étant infinies, les trois branches ne pourront jamais se synchroniser pour quitter la structure parallèle : l'exécution du modèle est donc bien infinie.

$$s_0 \xrightarrow{\alpha} s_1 \xrightarrow{\lambda_c} s_2 \xrightarrow{\lambda_t} s_3 \xrightarrow{\uparrow f_{Att\_OA}} s_4 \xrightarrow{\downarrow f_{Att\_OA}} s_5 \xrightarrow{\uparrow f_{Sur\_an}} s_6 \ldots \tag{4.5}$$

TABLE 4.2 – Description des états successifs de l'exécution (4.5)

États	**Nœuds activés** $\{n/A(n) = enabled\}$	**Fonctions en exécution** $\{f_c/A(f_c) = executing\}$	**Niveaux des flux** $\{(f_l,k) \in \mathcal{F} \times \mathbb{N}_{>0}/F(f_l) = k\}$
s_0	$\{\alpha\}$	$\emptyset$	$\{(ouv,1)\}$
s_1	$\{\lambda_t;\lambda_c;\lambda_p\}$	$\emptyset$	$\{(ouv,1)\}$
s_2	$\{\lambda_t;\lambda_p;f_{Sur\_an}\}$	$\emptyset$	$\{(ouv,1)\}$
s_3	$\{\lambda_p;f_{Sur\_an};f_{Att\_OA}\}$	$\emptyset$	$\{(ouv,1)\}$
s_4	$\{\lambda_p;f_{Sur\_an}\}$	$\{f_{Att\_OA}\}$	$\{(ouv,1)\}$
s_5	$\{\lambda_p;f_{Sur\_an};f_{Att\_ZC}\}$	$\emptyset$	$\{(ouv,1);(ann,1)\}$
s_6	$\{\lambda_p;f_{Att\_ZC}\}$	$\{f_{Sur\_an}\}$	$\{(ouv,1)\}$

Plusieurs transitions étaient possibles à partir de l'état initial s_0 : nous avons choisi ici d'entrer d'abord dans la boucle de la branche du contrôleur, mais nous aurions pu également choisir les nœuds λ_t ou λ_p comme transitions.

Enfin, cette exécution met en évidence le mécanisme de synchronisation par flux entre les

branches parallèles du modèle : ainsi, la fonction *Surveiller les annonces* est activée dès l'état s_2 mais ne peut s'exécuter qu'après la fin de la fonction *Atteindre l'origine d'annonce* (s_5) et l'envoi du trigger *annonce*.

4.2 EFFBDs temporels sans terminaisons forcées

Nous ajoutons dans cette section la modélisation du temps ; seules les fonctions peuvent laisser le temps s'écouler, le traitement des autres types de nœuds étant en effet instantané. Les éléments présentés dans la section précédente s'étendent ainsi aisément au cas temporisé.

4.2.1 Syntaxe des EFFBDs sans terminaisons forcées

Nous ajoutons simplement à la définition 4.1 les éléments décrivant les durées minimale et maximale de l'exécution d'une fonction.

Définition 4.11 (EFFBD temporel) *Un* EFFBD temporel *est un n-uplet* $\mathcal{E}_T = (\mathcal{E}_U, a, b)$ *où :*

- $\mathcal{E}_U$ *est un EFFBD atemporel ;*
- $a \in \mathbb{N}^{FC}$ *et* $b \in (\mathbb{N} \cup \{\infty\})^{FC}$ *donnent respectivement les durées minimale et maximale de l'exécution des fonctions* ($a \leq b$).

Ici, les bornes des durées sont prises dans $\mathbb{N}$, mais nous aurions également pu les choisir dans $\mathbb{Q}_{\geq 0}$ [13].

4.2.2 Sémantique des EFFBDs sans terminaisons forcées

La sémantique d'un EFFBD temporel est définie par un système de transitions temporisé. Nous définissons une valuation sur l'ensemble des fonctions, que nous notons $\nu \in (\mathbb{R}_{\geq 0})^{FC}$; ν est telle que, pour toute fonction $f_c \in FC$, $\nu(f_c)$ est le temps écoulé depuis que f_c a débuté son exécution. On considère que $\nu(f_c)$ n'a de sens que si la fonction est en exécution.

Par ailleurs, les états du STT sont étendus par rapport au cas atemporel : ce sont à présent des quadruplets (A, C, N, ν) représentant respectivement l'activité des nœuds, les compteurs d'itération, le niveau des flux et la valuation des fonctions.

Enfin, on autorise l'écoulement du temps, modélisé par la relation de transition continue, à partir d'un état s seulement si aucune autre transition discrète, instantanée, n'est possible à partir de cet état. Aucun nœud non fonctionnel n'est alors dans l'état *enabled* et, si une fonction est activée (*enabled*), elle ne peut démarrer son exécution (*executing*) faute d'un ou plusieurs flux d'entrée.

Définition 4.12 (Sémantique d'un EFFBD temporel) *La* sémantique d'un EFFBD temporel $\mathcal{E}_\tau = (\mathcal{N}, \mathcal{F}, \mathcal{A}, iter, n_0, F_0, a, b)$ *est un quadruplet* $||\mathcal{E}_\tau|| = (S, s_0, \mathcal{N}, \rightarrow)$ *où :*

- $S \subseteq \mathbb{A}^{\mathcal{N}} \times \mathbb{N}^{IT_{in}} \times \mathbb{N}^{\mathcal{F}} \times (\mathbb{R}_{\geq 0})^{FC}$ *est l'ensemble des* états *du système ;*

13. Nous verrons même que, s'il ne s'agit d'effectuer que des simulations mais non des vérifications formelles, ces bornes peuvent être choisies dans $\mathbb{R}_{\geq 0}$ (cf. annexe A).

- $s_0 = (A_0, \mathbf{0}, F_0, \vec{0})$ *est* l'état initial ;
- $\rightarrow \subseteq S \times (\mathcal{N} \cup \mathbb{R}_{\geq 0}) \times S$ *est la relation de transition, composée :*
  - *de la relation de transition discrète* $\xrightarrow{n \in \mathcal{N}} \subseteq S \times \mathcal{N} \times S$ *définie par :*

$$(A, C, N, \nu) \xrightarrow{n} (A', C', N', \nu') \Leftrightarrow \begin{cases} \begin{cases} A(n) = enabled \wedge NodeBehavior^\tau \\ \text{ou } A(n) = executing \wedge \begin{cases} \mathcal{P}^\tau_{\bar{f}} & \text{si } n \notin FC_{MS} \\ \mathcal{P}^\tau_{\bar{f} \prec} & \text{sinon} \end{cases} \end{cases} \\ A'(n) = NextActivity \text{ (comme défini ci-dessus)} \end{cases}$$

avec :

$$NodeBehavior^\tau \quad = \quad \begin{cases} n \in AND_{in} \cup OR_{out} \cup LP_{in} \Rightarrow \mathcal{P}^\tau_* \\ n \in AND_{out} \Rightarrow \mathcal{P}^\tau_{\bar{\alpha}} \\ n \in OR_{in} \Rightarrow \mathcal{P}^\tau_{\omega} \\ n \in IT_{in} \Rightarrow \begin{cases} \mathcal{P}^\tau_{\iota <} & \text{si } C(n) < iter(n) \\ \mathcal{P}^\tau_{\iota =} & \text{sinon} \end{cases} \\ n \in LP_{out} \cup IT_{out} \Rightarrow \mathcal{P}^\tau_{\overline{\lambda \iota}} \\ n \in FC \Rightarrow \mathcal{P}^\tau_F \end{cases}$$

  - *de la relation de transition continue* $\xrightarrow{\delta \in \mathbb{R}_{\geq 0}} \subseteq S \times \mathbb{R}_{\geq 0} \times S$ *définie par :*

$$(A, C, N, \nu) \xrightarrow{\delta} (A', C', N', \nu') \Leftrightarrow \begin{cases} \forall n \notin FC,\ A(n) \neq enabled \\ \forall n \in FC,\ (A(n) = enabled) \Rightarrow (N < \max(Cons(n), Lect(n))) \\ \forall n \in FC,\ (A(n) = executing) \Rightarrow (\nu(n) + \delta \leq b(n)) \\ \nu' = \nu + \delta \end{cases}$$

Les conditions temporisées $\mathcal{P}^\tau_x$ sont :

$$\mathcal{P}^\tau_f \quad = \quad \begin{cases} \mathcal{P}_f \\ \forall f \in FC,\ \nu'(f) = \begin{cases} 0 & \text{si } f = n \\ \nu(f) & \text{sinon} \end{cases} \end{cases}$$

$$\mathcal{P}^\tau_{\bar{f}} \quad = \quad \begin{cases} \mathcal{P}_{\bar{f}} \\ a(n) \leq \nu(n) \leq b(n) \\ \nu' = \nu \end{cases}$$

$$\mathcal{P}^\tau_{\bar{f} \prec} \quad = \quad \begin{cases} \mathcal{P}_{\bar{f} \prec} \\ a(n) \leq \nu(n) \leq b(n) \\ \nu' = \nu \end{cases}$$

Les autres conditions sont identiques au cas atemporel, avec l'addition de la contrainte $\nu' = \nu$.

4.2.3 Application au passage à niveau

Dans l'exemple du passage à niveau, nous pouvons à présent prendre en compte l'écoulement du temps. En particulier, par rapport à l'exécution précédente, il faut que tous les nœuds

LP ouvrants soient traités avant de laisser passer la durée (ici prise égale à 3 minutes ou 180 secondes) correspondant à l'exécution de la première fonction, *Atteindre l'origine d'annonce*.

$$s_0 \xrightarrow{\alpha} s_1 \xrightarrow{\lambda_c} s_2 \xrightarrow{\lambda_t} s_3 \xrightarrow{\uparrow f_{Att\_OA}} s_4 \xrightarrow{\lambda_p} s_5 \xrightarrow{180} s_6 \xrightarrow{\downarrow f_{Att\_OA}} s_7 \xrightarrow{\uparrow f_{Att\_ZC}} s_8 \xrightarrow{\uparrow f_{Sur\_an}} s_9 \xrightarrow{\downarrow f_{Sur\_an}} s_{10} \xrightarrow{\uparrow f_{Act\_so}} s_{11} \xrightarrow{5} s_{12} \xrightarrow{\downarrow f_{Act\_so}} s_{13} \ldots \quad (4.6)$$

Les états s_0 à s_3 sont identiques à ceux de l'exécution 4.5, à l'ajout des valuations (nulles) près. Pour faciliter la lecture du tableau 4.3, nous associons à chaque fonction en exécution sa valuation.

TABLE 4.3 – Description des états successifs de l'exécution 4.6

États	**Nœuds activés** $\{n/A(n) = enabled\}$	**Fonctions en exécution** $\{(f_c, \nu(f_c))/A(f_c) = executing\}$	**Niveaux des flux** $\{(f_l, k)/F(f_l) = k\}$
s_0	$\{\alpha\}$	$\emptyset$	$\{(ouv, 1)\}$
s_1	$\{\lambda_t; \lambda_c; \lambda_p\}$	$\emptyset$	$\{(ouv, 1)\}$
s_2	$\{\lambda_t; \lambda_p; f_{Sur\_an}\}$	$\emptyset$	$\{(ouv, 1)\}$
s_3	$\{\lambda_p; f_{Sur\_an}; f_{Att\_OA}\}$	$\emptyset$	$\{(ouv, 1)\}$
s_4	$\{\lambda_p; f_{Sur\_an}\}$	$\{(f_{Att\_OA}, 0)\}$	$\{(ouv, 1)\}$
s_5	$\{f_{Sur\_an}; f_{Act\_so}\}$	$\{(f_{Att\_OA}, 0)\}$	$\{(ouv, 1)\}$
s_6	$\{f_{Sur\_an}; f_{Act\_so}\}$	$\{(f_{Att\_OA}, 180)\}$	$\{(ouv, 1)\}$
s_7	$\{f_{Sur\_an}; f_{Act\_so}; f_{Att\_ZC}\}$	$\emptyset$	$\{(ouv, 1); (ann, 1)\}$
s_8	$\{f_{Sur\_an}; f_{Act\_so}\}$	$\{(f_{Att\_ZC}, 0)\}$	$\{(ouv, 1); (ann, 1)\}$
s_9	$\{f_{Act\_so}\}$	$\{(f_{Att\_ZC}, 0); (f_{Sur\_an}, 0)\}$	$\{(ouv, 1)\}$
s_{10}	$\{f_{Act\_so}; f_{Sur\_de}\}$	$\{(f_{Att\_ZC}, 0)\}$	$\{(ouv, 1); (ferPN, 1)\}$
s_{11}	$\{f_{Sur\_de}\}$	$\{(f_{Att\_ZC}, 0); (f_{Act\_so}, 0)\}$	$\{(ouv, 1)\}$
s_{12}	$\{f_{Sur\_de}\}$	$\{(f_{Att\_ZC}, 5); (f_{Act\_so}, 5)\}$	$\{(ouv, 1)\}$
s_{13}	$\{f_{Sur\_de}; f_{Ab\_bar}\}$	$\{(f_{Att\_ZC}, 5)\}$	$\{(ouv, 1)\}$

Cette exécution décrit donc l'enchaînement des fonctions qui conduit de l'arrivée d'un train à la fermeture des barrières, tel qu'il a été décrit dans la section 3.6.2.

De nouveau, plusieurs transitions étaient possibles à partir des états s_0 ou s_7, pour ne citer que ces deux états. En revanche, seule la fin de la fonction *Activer la sonnerie* était possible après l'état s_{12} puisqu'il restait au moins 20 s avant que la fonction *Atteindre la zone courte* puisse s'achever.

On retrouve également le mécanisme de synchronisation par flux : *Surveiller les annonces* est activée dès l'état s_2 mais ne peut s'exécuter avant s_9 ; *Activer la sonnerie* est activée dès s_5 mais ne se déclenche qu'à l'état s_{11}, c'est-à-dire après la fin de *Surveiller les annonces*.

4.3 EFFBDs temporels avec terminaisons forcées

4.3.1 Syntaxe des EFFBDs

La syntaxe d'un EFFBD temporel avec structures de terminaison forcée (ce que désignera le terme d'EFFBD dans la suite de ce manuscrit) est similaire aux syntaxes présentées dans les sections 4.1.1 et 4.2.1. Elle nécessite cependant quelques éclaircissements préliminaires.

Nœuds implicites Des structures parallèles ou sélectives peuvent ne comprendre que des branches se terminant par un nœud de terminaison forcée, LE ou EXIT. Reprenons ainsi la figure 3.6 p. 50, donnée ci-dessous : les deux branches de la structure de choix contenue dans le sous-scénario sont terminées par un nœud EXIT. Dans ce cas, la structure n'est suivie d'aucune autre puisque tout ce qui est placé après cette structure serait inatteignable par le flux de contrôle. Le nœud AND ou OR fermant n'est alors pas représenté, bien qu'il fasse partie du modèle, de manière à faciliter la lecture du diagramme.

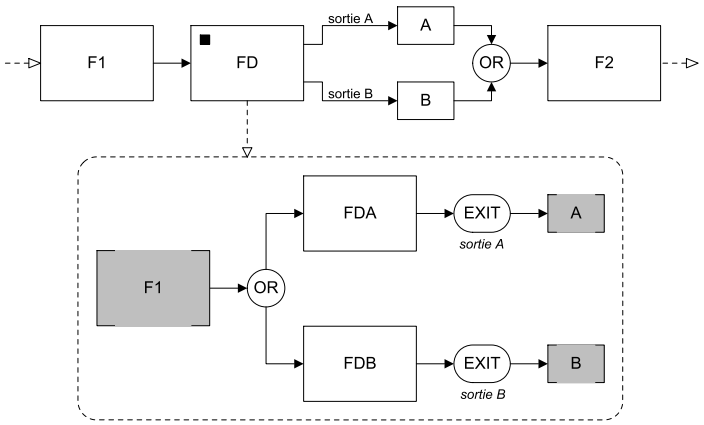


FIGURE 4.3 – Exemple de Fonction décomposée multi-sortie

Inclusion de sous-scénarios Les EFFBDs considérés dans cette section sont susceptibles de contenir des fonctions décomposées multi-sorties et, partant, des sous-scénarios. Nous considérons dans la suite que ces sous-scénarios sont *directement rattachés* au scénario principal par des arcs de contrôle, comme indiqué au moyen des rectangles grisés (cf. par exemple la figure 4.3).

Dans cet exemple, le nœud représentant la fonction `F1` a pour successeur non le nœud `FD` mais bien le nœud OR ouvrant du sous-scénario. De même, le nœud `A` a pour prédécesseur le nœud EXIT lié à la sortie `sortie A` et non le nœud `FD`. Tout se passe donc comme si la fonction FD n'existait pas et que tous les sous-scénarios du modèle étaient ramenés au même niveau hiérarchique.

Pour éviter de surcharger notre propos de descriptions fastidieuses, nous avons préféré ne pas donner la définition formelle de ces arcs de contrôle, dont la construction est cependant intuitive et immédiate.

Structure fonctionnelle d'une fonction décomposée En ce qui concerne les branches de sortie d'une fonction décomposée, nous considérons dans la suite que tout nœud placé entre un nœud EXIT et le nœud OR terminant la fonction décomposée fait partie de la structure fonctionnelle de la fonction décomposée.

Cette structure a autant de points d'entrée qu'il y a de nœuds EXIT dans le sous-scénario ; cependant, le comportement des nœuds EXIT, détaillé dans la section 4.3.3 garantit qu'une seule branche de sortie est activée.

Prise en compte des modificateurs *kill* Les modificateurs *kill*, placés au début d'une branche parallèle, ne peuvent affecter une branche s'achevant par un nœud de terminaison. En effet, le comportement du *kill* serait redondant avec celui du nœud de terminaison. En d'autres termes, il n'affecte que des branches « complètes », c'est-à-dire se terminant sur un nœud AND fermant. Par conséquent, il est équivalent de placer le *kill* au début ou à la fin de la branche parallèle. Sachant en outre que le modificateur n'intervient dans le comportement de la branche qu'à la fin de l'exécution de celle-ci, nous considérerons que c'est en réalité l'arc final d'une branche parallèle « tueuse » qui porte le modificateur (il atteint donc le nœud AND fermant).

On définit alors la fonction K qui, à tout couple $(n, \bar{\alpha}) \in \mathcal{N} \times AND_{out}$ associe la valeur `vrai` si $(n, \bar{\alpha})$ est un arc portant le modificateur *kill* et `faux` sinon [14].

La syntaxe d'un EFFBD s'écrit alors :

Définition 4.13 (EFFBD) *Un* EFFBD *est un 9-uplet* $\mathcal{E} = (\mathcal{N}, \mathcal{F}, \mathcal{A}, iter, n_0, F_0, a, b, K)$ *où :*

- $\mathcal{N}$ *est un ensemble fini et non vide de* nœuds, *formé de la réunion des ensembles suivants* [15] *:*
  - AND_{in}, AND_{out}, OR_{in}, OR_{out}, LP_{in}, LP_{out}, IT_{in}, IT_{out} *et* FC, *définis comme précédemment ;*
  - LE, *l'ensemble des nœuds* LE ;
  - dFC, *l'ensemble des fonctions (multi-sorties) décomposées ;*
  - $EXIT$, *l'ensemble des nœuds* EXIT.
- $\mathcal{F}$, $\mathcal{A}$, $iter$, n_0, F_0, a *et* b *sont définis comme précédemment ;*
- $K \in \mathbb{B}^{\mathcal{N} \times AND_{out}}$ *est la fonction de répartition des modificateurs* kill.

L'ensemble $\mathcal{N}$ comprend *tous* les nœuds du modèle, y compris ceux des éventuels sous-scénarios.

4.3.2 Définitions complémentaires

Avant de pouvoir exprimer la sémantique des EFFBDs, il est nécessaire de modifier et compléter les définitions données dans la section 4.1.2. En premier lieu, nous donnons la définition de *l'ensemble des structures de fonctions multi-sorties décomposées*, dFC_{MS}. Celle-ci est calquée sur les définitions 4.4 et 4.6.

14. En particulier, $K(n, \bar{\alpha}) =$ `faux` si $(n, \bar{\alpha}) \notin \mathcal{A}_C$.
15. Ces ensembles sont bien sûr de nouveau disjoints deux à deux.

On notera dans la suite $\mathcal{C}(df)$ le *contenu d'une fonction décomposée* $df \in dFC$, c'est-à-dire l'ensemble des nœuds de son sous-scénario, en incluant les nœuds EXIT. Réciproquement, la *fonction de référence* d'un nœud EXIT ϵ sera notée $Ref(\epsilon)$.

Définition 4.14 (Structures de fonction multi-sortie) *Soient* $\epsilon_1, \ldots, \epsilon_m \in EXIT$ *et* $\bar{\omega} \in OR_{out}$ *; la* structure de fonction multi-sortie $\langle \epsilon_1, \ldots, \epsilon_m, \bar{\omega} \rangle$ *appartient à l'ensemble* dFC_{MS} *si et seulement si les trois conditions suivantes sont réunies :*

$$\exists!\, df \in dFC \text{ t.q. } \forall i \in \mathbb{N}_{1..m} : \begin{cases} \epsilon_i \in \mathcal{C}(df) & (4.7) \\ \forall df' \in dFC, \epsilon_i \in \mathcal{C}(df') \Rightarrow df \in \mathcal{C}(df') & (4.8) \end{cases}$$

$$\forall \epsilon_i, \forall n \in Post(\epsilon_i) : \begin{cases} n = \bar{\omega} \vee n \in Pre(\bar{\omega}) & (4.9) \\ \text{ou } \exists!\, n' \in Pre(\bar{\omega}) \text{ t.q. } (\epsilon_i, n, \ldots, n', \bar{\omega}) \in Paths(\epsilon_i, \bar{\omega}) & (4.10) \\ \text{ou } n \in LE \cup EXIT & (4.11) \\ \text{ou } \exists\ n_{term} \in LE \cup EXIT \text{ t.q. } Paths(n, n_{term}) \neq \emptyset & (4.12) \end{cases}$$

$$\forall n' \in Pre(\bar{\omega}), \exists!\, i \in \mathbb{N}_{1..m} / \begin{cases} n' = \epsilon_i \vee n' \in Post(\epsilon_i) & (4.13) \\ \text{ou} \exists!\, n \in Post(\epsilon_i) / (\epsilon_i, n, \ldots, n', \bar{\omega}) \in Paths(\epsilon_i, \bar{\omega}) & (4.14) \end{cases}$$

Les propositions 4.7 et 4.8 garantissent que tous les nœuds de sortie ϵ_i correspondent à la même fonction décomposée. En particulier, dans le cas de fonctions emboîtées, un nœud EXIT se réfère toujours à la fonction immédiatement contenante (cf. 4.8).

Par ailleurs, quel que soit i, $Post(\epsilon_i)$ est un singleton ; en outre, les propositions 4.9 à 4.12 sont en exclusion mutuelle. Elles garantissent qu'à chaque branche de sortie correspond au plus un nœud EXIT. De plus, si la branche ne s'achève pas par un nœud de terminaison, il existe exactement un seul nœud EXIT lui correspondant (cf. 4.13 et 4.14).

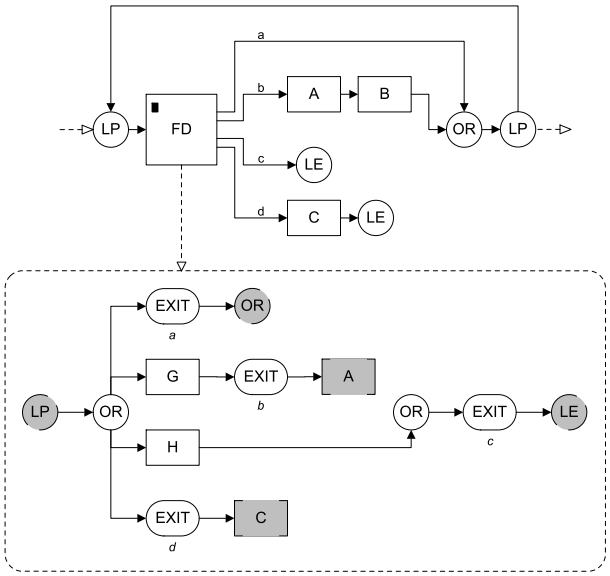


FIGURE 4.4 – Structure de fonction multi-sortie

Exemple 7. *Considérons le modèle illustré figure 4.4. Chacune des quatre branches de sortie* `a` *à* `d` *de la fonction décomposée* `FD` *illustre, dans cet ordre, les propositions 4.9 à 4.12.*

Définition 4.15 (Contenu d'une fonction décomposée multi-sortie) *Le* contenu d'une fonction décomposée multi-sortie $f_{\prec} = \langle \epsilon_1, \ldots, \epsilon_m, \bar{\omega} \rangle$*, noté* $\mathcal{C}(f_{\prec})$*, est défini par :*

$$\forall n \in \mathcal{N},\ n \in \mathcal{C}(f_{\prec}) \Leftrightarrow \begin{cases} \exists i \in \mathbb{N}_{1..m} / Paths(\epsilon_i, n) \neq \emptyset \\ Paths(\bar{\omega}, n) = \emptyset \end{cases}$$

$\mathcal{C}(f_{\prec})$ contient donc les nœuds portés par les branches de sortie de la fonction décomposée.

De même, les définitions des structures parallèles, sélectives, etc. doivent prendre en compte la possibilité pour une branche de se terminer par un nœud LE ou EXIT.

Définition 4.16 (Structures parallèles) *Soient* $\alpha \in AND_{in}$ *et* $\bar{\alpha} \in AND_{out}$ *deux nœuds* AND*. La structure* $\langle \alpha, \bar{\alpha} \rangle$ *appartient à* AND*,* l'ensemble des structures parallèles*, si et seulement si les deux conditions suivantes sont réunies :*

$$\forall n \in Post(\alpha) : \begin{cases} n \in Pre(\bar{\alpha}) & (4.15) \\ \text{ou } \exists!\ n' \in Pre(\bar{\alpha}) \text{ t.q. } (\alpha, n, \ldots, n', \bar{\alpha}) \in Paths(\alpha, \bar{\alpha}) & (4.16) \\ \text{ou } n \in LE \cup EXIT & (4.17) \\ \text{ou } \exists\ n_{term} \in LE \cup EXIT \text{ t.q. } Paths(n, n_{term}) \neq \emptyset & (4.18) \end{cases}$$

$$\forall n' \in Pre(\bar{\alpha}) : \begin{cases} n' \in Post(\alpha) & (4.19) \\ \text{ou } \exists!\ n \in Post(\alpha) \text{ t.q. } (\alpha, n, \ldots, n', \bar{\alpha}) \in Paths(\alpha, \bar{\alpha}) & (4.20) \end{cases}$$

La proposition 4.17 est vraie seulement si le nœud AND ouvrant est immédiatement suivi d'un nœud de terminaison forcée. Conceptuellement, cette construction n'a pas de sens puisqu'on sort de la structure avant même d'avoir pu exécuter ne serait-ce qu'une partie de son contenu. Elle est cependant autorisée dans la suite.

Afin de garantir la propriété de bonne imbrication dans la suite, les structures parallèles excluent de leur contenu les éventuels nœuds EXIT. Le contenu d'une structure parallèle est alors défini comme suit :

Définition 4.17 (Contenu d'une structure parallèle) *Le* contenu d'une structure parallèle $and = \langle \alpha, \bar{\alpha} \rangle$*, noté* $\mathcal{C}(and)$*, est défini par :*

$$\forall n \in \mathcal{N},\ n \in \mathcal{C}(and) \Leftrightarrow \begin{cases} n \notin EXIT \\ Paths(\alpha, n) \neq \emptyset \\ Paths(\bar{\alpha}, n) = \emptyset \end{cases}$$

L'ensemble OR des structures de sélection ainsi que le contenu $\mathcal{C}(or)$ d'une structure de sélection or sont définis de façon similaire.

La définition d'une structure de boucle s'obtient en combinant les définitions 4.6 et 4.16. L'ensemble IT des structures d'itération est défini de façon similaire. Le contenu d'une structure de boucle, $\mathcal{C}(lp)$ ou celui d'une structure d'itération, $\mathcal{C}(it)$, est défini de la même façon que précédemment.

Enfin, nous définissons de façon formelle la *structure de boucle de référence* d'un nœud LE ; nous rappelons que, dans le cas de boucles imbriquées, la référence est la boucle contenante la plus proche.

Définition 4.18 (Boucle de référence d'un nœud LE) *Soient $le \in LE$ et $lp \in LP$. lp est la boucle de référence de le, ce qui sera noté $lp = Ref(le)$, si et seulement si :*

$$\begin{cases} le \in \mathcal{C}(lp) \\ \forall lp' = \langle \lambda', \bar{\lambda}' \rangle \in LP, le \in \mathcal{C}(lp') \Rightarrow \lambda', \bar{\lambda}' \notin \mathcal{C}(lp) \end{cases}$$

Nous pouvons enfin compléter la définition d'un EFFBD bien formé.

Définition 4.19 (EFFBD bien formé) *Un EFFBD $\mathcal{E}$ est* bien formé *si et seulement si il respecte toutes les propriétés suivantes :*

- *unicité des structures : tout nœud est constitutif d'exactement une structure :*

$$\begin{cases} \forall \alpha \in AND_{in}, \exists!\ \bar{\alpha} \in AND_{out} / \langle \alpha, \bar{\alpha} \rangle \in AND \\ \forall \bar{\alpha} \in AND_{out}, \exists!\ \alpha \in AND_{in} / \langle \alpha, \bar{\alpha} \rangle \in AND \end{cases}$$

  *(*idem *pour les ensembles OR, LP, IT, FC_{MS} et dFC_{MS}) ;*
- *continuité : tout nœud différent de n_0 possède au moins un prédécesseur (hormis les nœuds implicites dans AND_{out} et OR_{out}). $\mathcal{E}$ ne contient qu'un seul nœud final*[16]*, qui n'a pas de successeur :*

$$\begin{cases} \forall n \in \mathcal{N}, Pre(n) = \emptyset \Leftrightarrow \begin{cases} n = n_0 \\ \text{ou } n \in AND_{out} \cup OR_{out} \wedge Post(n) = \emptyset \end{cases} \\ \exists!\, n \in \mathcal{N} \setminus LE \cup EXIT \text{ t.q. } Post(n) = \emptyset \end{cases}$$

- *imbrication : deux structures quelconques sont soit en séquence, soit entièrement emboîtées l'une dans l'autre : $\forall c \in AND \cup OR \cup IT \cup LP \cup dFC_{MS}$:*

$$\begin{cases} \forall \langle n, \bar{n} \rangle \in AND \cup OR \cup IT \cup LP, n \in \mathcal{C}(c) \Leftrightarrow \bar{n} \in \mathcal{C}(c) \\ \forall \langle \epsilon_1, \ldots, \epsilon_m, \bar{\omega} \rangle \in dFC_{MS}, \epsilon_1 \in \mathcal{C}(c) \Leftrightarrow \epsilon_2 \in \mathcal{C}(c) \Leftrightarrow \cdots \Leftrightarrow \bar{\omega} \in \mathcal{C}(c) \end{cases}$$

- *tout nœud* LE *node possède une et une seule boucle de référence, située dans le même niveau hiérarchique :*

$$\forall le \in LE, \exists!\ \langle \lambda, \bar{\lambda} \rangle \in LP \text{ t.q. } \begin{cases} Ref(le) = \langle \lambda, \bar{\lambda} \rangle \\ (\exists df \in dFC / le \in \mathcal{C}(df)) \Leftrightarrow (\lambda, \bar{\lambda} \in \mathcal{C}(df)) \end{cases}$$

- *tout chemin dans un sous-scénario se termine sur un nœud* LE *ou* EXIT [17] *:*

$$\forall df \in dFC, \forall n \in \mathcal{C}(df) \setminus LE \cup EXIT, \exists\, n_{term} \in LE \cup EXIT \text{ t.q. } Paths(n, n_{term}) \neq \emptyset$$

16. Ce nœud ne peut être terminal : un nœud LE doit en effet être contenu dans une boucle et un nœud EXIT dans un scénario de plus haut niveau.

17. Nous interdisons en particulier les fonctions décomposées hybrides. Leur prise en compte pourrait cependant se faire sans difficulté supplémentaire.

4.3.3 Sémantique des EFFBDs

À nouveau, les états du STT sont des quadruplets représentant l'activité des nœuds, les compteurs d'itération, les niveaux des flux ainsi que les valuations des fonctions. La sémantique donnée ci-dessous réutilise et adapte un grand nombre des propositions $\mathcal{P}_x^\tau$ données dans la section 4.2.2. Nous introduisons les règles suivantes :

- $\mathcal{P}_{LE}^\tau$ décrit le comportement d'un nœud LE : celui-ci active le nœud qui suit sa boucle de référence, s'il existe, et désactive tous les nœuds contenus par cette boucle (si besoin, il place le nœud LP fermant dans l'état *executed*) ; en outre, les itérations éventuellement contenues sont remises à zéro ;
- $\mathcal{P}_\epsilon^\tau$ décrit le comportement d'un nœud EXIT ; ce comportement est similaire au cas LE.

Par ailleurs, plusieurs nœuds sont susceptibles d'être en dernière position sur une branche parallèle « tueuse ». Il s'agit des nœuds AND, OR, LP et IT [18] fermants et des fonctions à sortie simple (donc non décomposées). Ces nœuds ont un comportement spécifique, noté $\mathcal{P}_x^\kappa$. En particulier, tous les précédesseurs du nœud fermant de la structure parallèle sont placés dans l'état *executed*, de manière à forcer la synchronisation des branches et la sortie de la structure. Nous définissons le booléen $\kappa(n)$ pour tout $n \in \mathcal{N}$ par : $\kappa(n) \Leftrightarrow \exists and = \langle \alpha, \bar{\alpha} \rangle \in AND$ t.q. $K(n, \bar{\alpha})$

Définition 4.20 (Sémantique d'un EFFBD) *La* sémantique d'un EFFBD $\mathcal{E} = (\mathcal{N}, \mathcal{F}, \mathcal{A}, iter, n_0, F_0, a, b, K)$ *est un quadruplet* $||\mathcal{E}|| = (S, s_0, \mathcal{N}, \rightarrow)$ *où :*

- $S \subseteq \mathbb{A}^{\mathcal{N}} \times \mathbb{N}^{IT_{in}} \times \mathbb{N}^{\mathcal{F}} \times (\mathbb{R}_{\geq 0})^{FC}$;
- $s_0 = (A_0, \vec{0}, F_0, \mathbf{0})$;
- $\rightarrow \subseteq S \times (\mathcal{N} \cup \mathbb{R}_{\geq 0}) \times S$, *la relation de transition, est composée :*
  - *de la relation de transition discrète* $\xrightarrow{n \in \mathcal{N}} \subseteq S \times \mathcal{N} \times S$ *définie par :*

$$(A, C, N, \nu) \xrightarrow{n} (A', C', N', \nu') \Leftrightarrow \begin{cases} \left\{ \begin{array}{ll} A(n) = enabled \wedge NodeBehavior^\kappa & \\ \text{ou } A(n) = executing \wedge \begin{cases} \mathcal{P}_{\bar{f}}^\kappa & \text{si } \kappa(n) \\ \mathcal{P}_{\bar{f} \preceq}^\tau & \text{si } n \in FC_{MS} \\ \mathcal{P}_{\bar{f}}^\tau & \text{sinon} \end{cases} \end{array} \right. \\ A'(n) = NextActivity \end{cases}$$

18. Ce sont en réalité les nœuds LE et IT ouvrants qui gèrent le comportement des nœuds LP et IT fermants (cf. remarque 4.2).

avec :

$$NodeBehavior^{\kappa} = \begin{cases} n \in AND_{in} \cup LP_{in} \Rightarrow \mathcal{P}^{\tau}_{*} \\ n \in AND_{out} \Rightarrow \begin{cases} \mathcal{P}^{\kappa}_{\bar{\alpha}} & \text{si } \kappa(n) \\ \mathcal{P}^{\tau}_{\bar{\alpha}} & \text{sinon} \end{cases} \\ n \in OR_{in} \Rightarrow \mathcal{P}^{\tau}_{\omega} \\ n \in OR_{out} \Rightarrow \mathcal{P}^{\kappa}_{*} \\ (n \in IT_{in} \wedge C(n) < iter(n)) \Rightarrow \mathcal{P}^{\tau}_{\iota<} \\ (n \in IT_{in} \wedge C(n) = iter(n)) \Rightarrow \begin{cases} \mathcal{P}^{\kappa}_{*} & \text{si } \kappa(\bar{\iota}) \text{ avec } \langle n, \bar{\iota} \rangle \in IT \\ \mathcal{P}^{\tau}_{\iota=} & \text{sinon} \end{cases} \\ n \in LP_{out} \cup IT_{out} \Rightarrow \mathcal{P}^{\tau}_{\overline{\lambda\iota}} \\ n \in FC \Rightarrow \mathcal{P}^{\tau}_{F} \\ n \in LE \Rightarrow \begin{cases} \mathcal{P}^{\kappa}_{*} & \text{si } \kappa(\bar{\lambda}) \text{ avec } \langle \lambda, \bar{\lambda} \rangle = Ref(n) \\ \mathcal{P}^{\tau}_{LE} & \text{sinon} \end{cases} \\ n \in EXIT \Rightarrow \mathcal{P}^{\tau}_{\epsilon} \end{cases}$$

– *de la relation de transition continue* $\xrightarrow{\delta \in \mathbb{R}_{\geq 0}} \subseteq S \times \mathbb{R}_{\geq 0} \times S$ *définie comme précédemment.*

La définition des propositions est donnée ci-dessous :

$$\mathcal{P}^{\kappa}_{\bar{f}} = \begin{cases} \forall n' \in \mathcal{N}_{-n}, A'(n') = \begin{cases} executed & \text{si } n' \in Pre(\bar{\alpha}) \\ enabled & \text{si } n' = \bar{\alpha} \\ inactive & \text{si } n' \in \mathcal{C}(and) \setminus Pre(\bar{\alpha}) \\ A(n') & \text{sinon} \end{cases} \\ \forall \iota \in IT_{in} \begin{cases} C'(\iota) = 0 & \text{si } \iota \in \mathcal{C}(and) \\ C'(\iota) = C(it) & \text{sinon} \end{cases} \\ N' = N + Prod(n) \\ a(n) \leq \nu(n) \leq b(n) \\ \nu' = \nu \end{cases}$$

$$\mathcal{P}^{\kappa}_{\bar{\alpha}} = \begin{cases} \forall n' \in Pre(\bar{\alpha}), A(n') = executed \\ \mathcal{P}^{\kappa}_{*} \end{cases}$$

$$\mathcal{P}^{\kappa}_{*} = \begin{cases} \forall n' \in \mathcal{N}_{-n}, A'(n') = \begin{cases} executed & \text{si } n' \in Pre(\bar{\alpha}) \\ enabled & \text{si } n' = \bar{\alpha} \\ inactive & \text{si } n' \in \mathcal{C}(and) \setminus Pre(\bar{\alpha}) \\ A(n') & \text{sinon} \end{cases} \\ \forall \iota \in IT_{in} \begin{cases} C'(\iota) = 0 & \text{si } \iota \in \mathcal{C}(and) \\ C'(\iota) = C(it) & \text{sinon} \end{cases} \\ N' = N \\ \nu' = \nu \end{cases}$$

$$\mathcal{P}^{\tau}_{LE} = \begin{cases} \forall n' \in \mathcal{N}_{-n}, A'(n') = \begin{cases} executed & \text{si } (n' = \bar{\lambda} \wedge Post(\bar{\lambda}) \cap AND_{out} \neq \emptyset) \\ enabled & \text{si } n' \in Post(\bar{\lambda}) \wedge n' \neq \lambda \\ inactive & \text{si } n' \in \mathcal{C}(\langle \lambda, \bar{\lambda} \rangle) \\ A(n') & \text{sinon} \end{cases} \\ \forall \iota \in IT_{in} \begin{cases} C'(\iota) = 0 & \text{si } \iota \in \mathcal{C}(\langle \lambda, \bar{\lambda} \rangle) \\ C'(\iota) = C(it) & \text{sinon} \end{cases} \\ N' = N \\ \nu' = \nu \end{cases}$$

$$\mathcal{P}^{\tau}_{\epsilon} = \begin{cases} \forall n' \in \mathcal{N}_{-n}, A'(n') = \begin{cases} enabled & \text{si } n' \in Post(n) \\ inactive & \text{si } n' \in \mathcal{C}(Ref(n)) \\ A(n') & \text{sinon} \end{cases} \\ \forall \iota \in IT_{in} \begin{cases} C'(\iota) = 0 & \text{si } \iota \in \mathcal{C}(Ref(n)) \\ C'(\iota) = C(it) & \text{sinon} \end{cases} \\ N' = N \\ \nu' = \nu \end{cases}$$

4.3.4 Extensions de la sémantique : modélisation des *time-outs*

Nous avons montré dans la section 3.6.1 comment la sémantique existante des EFFBDs pouvait être mise en œuvre de manière à modéliser des *time-outs*. Plutôt que d'utiliser les motifs de conception proposés dans cette section, il serait possible de modifier et de compléter la syntaxe et la sémantique présentées ci-dessus pour y proposer directement des mécanismes de terminaison des fonctions sur échéance.

Nous ne traiterons pas ce cas en détail ici et nous nous bornerons à indiquer les principales modifications qu'il faudrait apporter aux définitions 4.13 et 4.20 :

- définition et ajout dans la syntaxe d'une application associant à chaque fonction f_c ayant des flux d'entrée [19] un réel positif définissant la durée maximale pendant laquelle la fonction peut être continûment *enabled* ;
- ajout dans la définition d'un état du système d'une valuation « timer » μ associant à chaque fonction f_c le temps écoulé depuis sa dernière activation ($\mu(f_c)$ n'a de sens que si f_c est dans l'état *enabled*) ;
- modification de la relation de transition discrète de manière à modéliser les comportements suivants :
  - initialisation des timers des fonctions activées par un nœud (quelconque) ;
  - désactivation d'une fonction dont le timer est arrivé à échéance ;
  - recopie des valeurs des timers dans les autres cas.
- modification de la relation de transition continue de manière à :
  - incrémenter les timers lors de l'écoulement du temps ;
  - garantir qu'aucune fonction dans l'état *enabled* ne peut dépasser son time-out.

19. *i.e.* telle que $Cons(f_c) + Lect(f_c) > \vec{0}$.

4.4 Propriétés des EFFBDs bien formés

Nous donnons dans cette section quelques résultats essentiels sur la structure et le comportement d'un EFFBD. Dans la suite, nous supposerons que tous les EFFBDs sont bien formés.

> **Remarque 4.7.** *Les éditeurs, en majeure partie graphiques, de modèles EFFBDs mis en œuvre dans les ateliers logiciels CORE et* MDWorkbench *sont suffisamment contraints pour garantir le caractère bien formé du modèle. L'étude d'EFFBDs mal formés n'a donc que peu de sens.*

4.4.1 Non réentrance

La syntaxe et la sémantique des EFFBDs garantissent un résultat fondamental :

Proposition 4.1 *Un EFFBD est non réentrant : chaque nœud et chaque structure sont désactivés avant de pouvoir être activés de nouveau.*

Démonstration. Initialement, le modèle ne comprend qu'un nœud activé.

Par ailleurs, les structures parallèles sont les seules à dupliquer le flux de contrôle (toutes les autres structures font simplement transiter le flux). Ainsi, pour qu'un nœud ou une structure déjà active puisse être de nouveau activée, il faudrait que plusieurs branches parallèles convergent vers elle. Or, d'après l'hypothèse de bonne formation des EFFBDs, des branches parallèles ne peuvent converger que sur le nœud AND fermant (lequel, *via* la synchronisation, réduit les flux de contrôle parallèles à un seul).

Aucun nœud ne peut donc être activé alors qu'il est encore en exécution. □

4.4.2 EFFBDs bornés

Le caractère *borné* ou non d'un EFFBD [20] est une propriété comportementale essentielle. Nous la définissons ci-dessous :

Définition 4.21 (EFFBD borné) *Soient $\mathcal{E}$ un EFFBD et $||\mathcal{E}||$ sa sémantique. $\mathcal{E}$ est* borné *si et seulement si $||\mathcal{E}||$ vérifie :*

$$\forall (A, C, N, \nu) \in S,\ \forall f_l \in \mathcal{F},\ \exists k \in \mathbb{N} \text{ tel que } N(f_l) \leq k$$

En effet, les compteurs d'itération gardent des valeurs finies (et même bornées par *iter*) : nous pouvons donc réduire la bornitude de l'EFFBD à celle de ses flux. Selon la terminologie usuelle, on dira que le modèle est k-borné lorsque l'entier k est la valeur de la borne. L'ensemble des EFFBD bornés sera noté $\mathcal{B}_\mathcal{E}$ par la suite.

Nous donnons ci-dessous deux conditions suffisantes (mais non nécessaires) assurant la bornitude d'un EFFBD. Elles sont suffisamment triviales pour que nous ne donnions pas leur preuve détaillée, qui repose sur deux constatations élémentaires :

20. C'est à regret que nous emploierons dans la suite le terme de *bornitude*, faute d'une meilleure traduction de l'anglais *boundedness*.

- seules les fonctions produisant des flux sont capables, lorsqu'elles sont terminées normalement (c'est-à-dire sans être « tuées ») de faire augmenter le niveau courant d'un flux ;
- les boucles LP sont les seuls éléments dont le comportement est potentiellement infini.

C'est donc la seule conjonction de ces deux phénomènes qui peut conduire à l'obtention d'EFFBDs non bornés.

Proposition 4.2 *Un EFFBD ne contenant pas de flux ni/ou de boucles est borné.*

Cette condition est triviale mais bien sûr très restrictive. Nous en donnons une deuxième, plus affinée.

Proposition 4.3 *Un EFFBD tel qu'aucune boucle ne contient des fonctions produisant des flux est borné.*

De nouveau, cette proposition n'est pas nécessaire : considérons ainsi l'exemple du passage à niveau, dont le modèle est donné figure 3.17. Les trois boucles comportent des fonctions productrices et pourtant, le modèle est borné :

- les ressources *ouvert* et *fermé* sont en « exclusion mutuelle » : pour toute exécution du modèle et tout état $s = (A, C, N, \nu)$ de cette exécution, $N(ouvert) + N(fermé) = 1$.
- Un exemplaire, au plus, de l'item *annonce* est disponible à chaque instant : en notant t la date de la première production de cet item, la deuxième production se produira au plus tôt à la date $t + 157$ (ce qui correspond à l'exécution la plus courte de la boucle). Or, entre l'exécution de la fonction *Surveiller les annonces* et son activation suivante, il s'écoule autant de temps qu'entre le début de *Atteindre la zone courte* et la fin de *Quitter la zone courte*, soit entre 37 et 74 secondes. Une consommation de l'item *annonce* s'intercale donc toujours entre deux productions (de même pour l'item *réarmement PN*).
- De même, il faut, au pire, 15 secondes après la production de l'item *annonce* pour que *Relever les barrières* soit prête à consommer *ouverture PN*, qui est produit, au mieux, à $t + 37$. Un raisonnement similaire sur *réarmement PN* et *fermeture PN* montre que les items *fermeture PN* et *ouverture PN* sont toujours immédiatement consommés et ne s'accumulent pas : leur niveau est donc limité à 1 unité.

Le modèle est bien borné (et même *sauf*, c'est-à-dire 1-borné).

Remarque 4.8. *S'il est suffisant de montrer qu'entre deux exécutions successives d'une fonction, le niveau des flux qu'elle produit n'a pas augmenté, cela n'est pas nécessaire pour garantir la bornitude du modèle. Considérons ainsi le modèle illustré figure 4.5. Entre la première et la seconde exécution de la fonction G, le niveau de res (initialement nul) est passé de 1 à 2 unités : le niveau du flux qu'elle produit a bien augmenté.*

Cependant, la présence de la branche parallèle « tueuse » et la fin de la fonction F après 5 unités de temps provoque la terminaison de G et la fin de l'exécution. Le modèle est donc bien borné.

A contrario*, un modèle $\mathcal{E}$ est borné si le modèle $\mathcal{E}_\tau$ obtenu à partir de $\mathcal{E}$ par abstraction des structures de terminaison forcée est borné.*

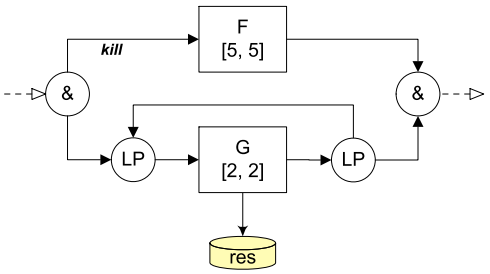


FIGURE 4.5 – Modèle borné

Nous pouvons extrapoler cette dernière remarque en une nouvelle proposition :

Proposition 4.4 *Soient $\mathcal{E}$ un EFFBD et $\mathcal{E}_U$ (respectivement $\mathcal{E}_\tau$) le modèle obtenu à partir de $\mathcal{E}$ par abstraction du temps (respectivement des structures de terminaison forcée). Les trois modèles vérifient les implications suivantes :*

$$\begin{cases} (\mathcal{E}_U \in \mathcal{B}_\mathcal{E}) \Rightarrow (\mathcal{E}_\tau \in \mathcal{B}_\mathcal{E}) \\ (\mathcal{E}_\tau \in \mathcal{B}_\mathcal{E}) \Rightarrow (\mathcal{E} \in \mathcal{B}_\mathcal{E}) \end{cases}$$

Remarque 4.9. *À nouveau, les implications ne sont pas des équivalences : en reprenant l'exemple du passage à niveau, on note que le modèle atemporel sous-jacent est quant à lui non borné. En effet, la branche supérieure n'est pas contrainte par la réception de flux : elle peut donc s'exécuter indéfiniment sans que l'on ait besoin de débuter l'exécution des autres branches (le non écoulement du temps n'impose pas d'exécuter* Surveiller les annonces *avant* Atteindre le PN*, par exemple). Par conséquent, le niveau des flux* annonce *et* réarmement PN *peuvent devenir infinis : le modèle atemporel n'est donc pas borné.*

Il serait légitime de s'interroger sur l'intérêt de travailler avec des modèles non bornés. En effet, les systèmes qui sont modélisés dans le cadre de l'IS sont, généralement, par essence bornés car représentant des systèmes physiques et finis. Les situations de non bornitude résultent donc, pour la plupart, d'une mauvaise conception ou modélisation. Cependant, nous avons précisé dans le premier chapitre de ce mémoire que nos travaux cherchent à être appliqués dès les premières étapes de la conception du système, c'est-à-dire lorsque les erreurs sont les plus nombreuses. La présence de bornes infinies peut donc alerter le concepteur sur un défaut de son modèle.

Se pose alors la question de la *détection* de ces situations non bornées. S'il est « facile » de vérifier qu'un flux donné ne dépasse pas une certaine valeur, fournie par exemple par le concepteur, la mise en évidence de bornes infinies est un problème bien différent, comme nous le verrons dans le chapitre suivant.

Chapitre 5

Traduction des EFFBDs en TPNs

Résumé *Le chapitre précédent a permis d'établir la sémantique formelle des diagrammes EFFBDs sous forme d'un STT. Nous pouvons à présent définir une traduction de ces modèles de haut niveau vers un formalisme de plus bas niveau qui permettra, comme nous le verrons au cours du chapitre suivant, d'effectuer des vérifications formelles sur le comportement du système modélisé.*

Nous présentons ainsi dans ce chapitre les réseaux de Petri *temporels* (Time Petri Nets, *TPNs) et les raisons qui nous ont amenés à choisir ce formalisme. Par ailleurs, la transformation que nous proposons est* structurelle *: nous donnons ainsi l'ensemble des motifs de traduction des différents éléments du langage EFFBD vers les TPNs.*

Après avoir montré l'équivalence, au sens de la bisimulation temporelle (forte), d'un modèle EFFBD quelconque et de sa traduction en TPN, nous avons été en mesure de démontrer un certain nombre de résultats essentiels sur les EFFBDs, dont la décidabilité de problèmes classiques appliqués aux diagrammes EFFBDs.

Sommaire

La syntaxe et la sémantique formelles des EFFBDs étant désormais établies, nous sommes en mesure de proposer la traduction d'un modèle EFFBD Σ vers un modèle de plus bas niveau σ, selon la démarche générale illustrée figure 1.2. Ce sont les réseaux de PETRI temporels *(Time* PETRI *Nets* ou TPNs) que nous avons retenus pour exprimer les modèles σ. Les raisons de ce choix ainsi que la présentation formelle des TPNs seront données section 5.1.

Nous avons opté pour une *traduction structurelle* des EFFBDs en TPNs : nous donnons ainsi dans la section 5.2 les différents *motifs élémentaires* de traduction, ainsi que les règles de connexion des motifs entre eux. Nous présenterons, pour conclure cette section, le réseau correspondant au modèle du passage à niveau.

Enfin, nous établirons dans la section 5.3 *l'équivalence*, au sens de la bisimulation temporelle, entre un modèle EFFBD Σ et sa traduction σ (ce qu'en introduction, nous avions noté $\Sigma \sim \sigma$). Forts de ce résultat fondamental, nous pourrons alors prouver un certain nombre de propriétés, dont l'indécidabilité de la bornitude des diagrammes EFFBDs.

5.1 Les réseaux de PETRI temporels

5.1.1 Généralités

Nous l'avons vu dans le premier chapitre : la conception et le développement sûrs de systèmes dynamiques complexes nécessitent en particulier leur modélisation selon un formalisme rigoureux et non ambigu. Dans ce but, de nombreux langages formels ont été développés depuis le milieu du siècle dernier. Parmi ceux-ci, les réseaux de PETRI temporels constituent un outil de conception et d'analyse puissant, particulièrement adapté à la description de systèmes dynamiques, discrets, distribués et communicants.

Le langage, à la fois graphique et mathématique, découle des réseaux de PETRI « classiques » [62] par l'ajout de contraintes temporelles, sous forme d'un intervalle, sur l'occurrence des événements. Plus précisément, nous nous plaçons ici dans le cadre des réseaux de PETRI *T-temporels* [14, 55] (c'est-à-dire associant les informations temporelles aux transitions), en *temps dense* et munis de la *sémantique forte* et *mono-serveur*. De ce fait, nous ne considérons pas :

- les réseaux de PETRI *temporisés (Timed* PETRI *Nets)*, où les contraintes temporelles sont données par une unique valeur, représentant un *délai minimal* (ou exact dans le cas d'une évolution « au plus tôt ») de franchissement des transitions [64] ; on notera que les réseaux temporisés constituent un sous-ensemble des réseaux temporels ;
- les réseaux *A-temporels* [36] (respectivement *P-temporels* [45]), associant les informations temporelles aux arcs (respectivement aux places) ;
- un écoulement *discret* du temps (cf. en particulier les travaux de M. MAGNIN [53]) puisque, dans le cas des EFFBDs, le temps est dense ;
- une sémantique *faible*, où les bornes temporelles supérieures peuvent être dépassées (les conditions temporelles représentent donc les instants où les événements *peuvent* se produire, sans obligation).

Remarque 5.1. *La sémantique forte traduit* l'urgence*, caractéristique des systèmes temps-réel en général et des systèmes modélisés par les EFFBDs en particulier. On peut par ailleurs mon-*

trer que, dans ce cas, les réseaux A-, P- et T-temporels ne sont plus équivalents : en particulier, les réseaux A-temporels sont strictement plus expressifs que les deux autres classes [19]. En revanche, l'hypothèse de la sémantique forte dans le cas des modèles A- et P-temporels conduit à la « péremption » (ou la mort*) des jetons : une telle situation est peu adaptée à la modélisation des comportements représentés par un EFFBD, d'où notre choix des réseaux T-temporels.*

Si les TPNs paraissent donc particulièrement adaptés à la traduction des EFFBDs, nous avons également envisagé, lors de nos premières recherches, le recours aux automates temporisés ou TA (pour *Timed Automata* [6]). Cependant, la modélisation du parallélisme, largement présent dans les EFFBDs, nous a paru plus naturelle avec les TPNs qu'avec les TA. En outre, on peut montrer qu'un TA ne comprenant pas d'autre variable que les horloges ne peut représenter les situations non bornées ; or, nous avons montré en conclusion du chapitre précédent que de telles situations peuvent se produire avec les EFFBDs. Signalons enfin que c'est au sein du laboratoire de l'IRCCyN qu'est développé Roméo, un atelier logiciel de conception et d'analyse des TPNs, permettant en particulier leur vérification formelle [32]. Cet élément ne peut bien sûr à lui seul justifier notre choix mais la proximité avec les développeurs du logiciel a représenté un avantage non négligeable lors des phases d'implémentation successives.

Remarque 5.2. *Que le lecteur ne s'y trompe pas : nous ne cherchons pas à présenter les TPNs et les TA comme antagonistes. Au contraire, ces deux formalismes sont intimement liés et complémentaires. Rappelons ainsi que, si les TA sont strictement plus expressifs, en termes de bisimulation temporelle, que les TPNs (bornés), ils ont la même expressivité en termes d'acceptation de langage temporisé [12].*

5.1.2 Syntaxe et sémantique des TPNs

Nous donnons ci-dessous la syntaxe formelle des TPNs.

Définition 5.1 (Réseau de Petri temporel) *Un* réseau de Petri temporel *(étiqueté) est un n-uplet* $\mathcal{T} = (P, T, {}^\bullet(.), (.)^\bullet, M_0, a, b, A, \Lambda)$ *où :*

- $P = \{p_1, ..., p_m\}$ *est un ensemble fini de* places *;*
- $T = \{t_1, ..., t_n\}$ *est un ensemble fini de* transitions *;*
- ${}^\bullet(.) \in (\mathbb{N}^P)^T$ *est la* fonction d'incidence arrière [1] *;*
- $(.)^\bullet \in (\mathbb{N}^P)^T$ *est la* fonction d'incidence avant *;*
- $M_0 \in \mathbb{N}^P$ *est le* marquage initial *;*
- $a \in (\mathbb{N})^T$ *(respectivement* $b \in (\mathbb{N} \cup \{\infty\})^T$*) est la fonction associant à chaque transition sa* date de tir au plus tôt *(respectivement* au plus tard*) ; a et b vérifient $a \leq b$;*
- A *est un ensemble* d'actions *;*
- $\Lambda : T \to A$ *est la* fonction d'étiquetage.

Remarque 5.3. *Les poids des arcs sont ici choisis dans* $\mathbb{N}$ *; il est possible, sans difficulté supplémentaire, de les prendre dans* $\mathbb{Q}_{\geq 0}$ *(cf. remarque 5 du chapitre 4).*

1. $\forall t \in T$, $\forall p \in P$, ${}^\bullet t(p)$ est le poids de l'arc (amont) reliant la place p à la transition t (nul s'il n'existe pas).

Les places modélisent généralement un état du système alors que les transitions représentent des événements ou la validation de conditions ; l'évolution du système est modélisée par le transit de *jetons* entre les places. Un *marquage* du réseau est alors un vecteur $M \in \mathbb{N}^P$ tel que pour toute place $p \in P$, $M(p)$ représente le nombre de jetons dans la place p.

Par ailleurs, une transition t est dite *sensibilisée* par le marquage M si toute place en amont de t contient au moins autant de jetons qu'indiqué par le poids de l'arc reliant cette place à t, *i.e.* $M \geq {}^\bullet t$. On notera dans ce cas $t \in enabled(M)$. La transition t peut alors être *tirée* si elle a été continûment sensibilisée depuis une durée au moins égale à $a(t)$. Le choix de la sémantique forte impose que t *doit* être tirée au plus tard à la date $b(t)$ (à moins d'avoir été désensibilisée entre temps par le tir d'une transition).

Enfin, la transition t est dite *nouvellement sensibilisée* par le tir d'une transition t_f depuis le marquage M, ce que nous noterons $\uparrow enabled(t, M, t_f)$, si elle est sensibilisée par le marquage $M - {}^\bullet t_f + t_f{}^\bullet$ mais ne l'était pas par le marquage $M - {}^\bullet t_f$. Formellement :

$$\uparrow enabled(t, M, t_f) = \begin{cases} {}^\bullet t \leq M - {}^\bullet t_f + t_f{}^\bullet \\ (t = t_f) \vee ({}^\bullet t > M - {}^\bullet t_f) \end{cases}$$

Par extension, l'ensemble des transitions nouvellement sensibilisées par le tir de la transition t_f depuis le marquage M sera noté $\uparrow enabled(M, t_f)$ dans la suite. On remarquera qu'il définit l'ensemble des transitions dont les horloges sont remises à zéro par le tir de t_f.

> **Remarque 5.4.** *Nous considérons ici la sémantique mono-serveur intermédiaire, sans multi-sensibilisation des transitions. En effet, la définition de $\uparrow enabled(t, M, t_f)$ considère qu'il n'existe qu'une seule horloge par transition sensibilisée et que lorsqu'une transition t est tirée, elle sera soit désensibilisée, soit* nouvellement *sensibilisée, indépendamment du* marquage intermédiaire *$M - {}^\bullet t$ (d'où la condition $t = t_f$ dans la définition ci-dessus).*
>
> *Le lecteur est invité à consulter [11] pour une discussion plus approfondie des différentes sémantiques et de leurs relations.*

La sémantique des TPNs est alors définie comme un STT dont les états sont formés par l'association d'un marquage M et de la valuation des horloges ν, $\nu(t)$ représentant le temps écoulé depuis la dernière sensibilisation de la transition t :

Définition 5.2 (Sémantique d'un TPN) *La sémantique d'un TPN $\mathcal{T}$ est un quadruplet $||\mathcal{T}|| = (Q, q_0, A, \rightarrow)$ tel que :*

- $Q \subseteq \mathbb{N}^P \times (\mathbb{R}_{\geq 0})^T$;
- $q_0 = (M_0, \vec{0})$;
- $\rightarrow \subseteq Q \times \{A \cup \mathbb{R}_{\geq 0}\} \times Q$ *est la* relation de transition *composée :*
  - *de la relation de transition discrète définie pour tout $t_f \in T$ par :*

$$(M, \nu) \xrightarrow{\Lambda(t_f)} (M', \nu') \Leftrightarrow \begin{cases} t_f \in enabled(M) \\ a(t_f) \leq \nu(t_f) \leq b(t_f) \\ M' = M - {}^\bullet t_f + t_f{}^\bullet \\ \forall t \in T,\ \nu'(t) = \begin{cases} 0 & \text{si } t \in \uparrow enabled(M, t_f) \\ \nu(t) & \text{sinon} \end{cases} \end{cases}$$

– *de la relation de transition continue définie pour tout* $\delta \in \mathbb{R}_{\geq 0}$ *par :*

$$(M,\nu) \xrightarrow{\delta} (M,\nu') \Leftrightarrow \begin{cases} \nu' = \nu + \delta \\ \forall t \in T,\ t \in enabled(M) \Rightarrow \nu'(t) \leq b(t) \end{cases}$$

Exemple 8. *Considérons le réseau illustré figure 5.1. Les places sont figurées par des cercles, contenant éventuellement des jetons ; les transitions par des rectangles. Contrairement à la convention usuelle, l'intervalle temporel qui n'est pas mentionné est égal à* $[0,0]$ *et non* $[0,\infty[$. *Les arcs sont représentés par des flèches pondérées (lorsqu'il n'est pas mentionné, le poids d'un arc vaut 1).*

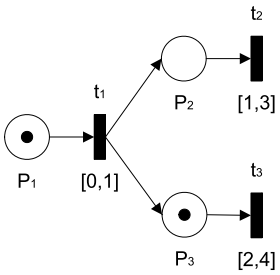


FIGURE 5.1 – Exemple de réseau de PETRI temporel

Le marquage initial est $M_0 = (1,0,1)_{(P_1,P_2,P_3)}$ *; les transitions* t_1 *et* t_3 *sont donc initialement sensibilisées mais le tir de* t_3 *ne peut se produire qu'après celui de* t_1. *Nous donnons ci-dessous le début de l'exécution du réseau ; les états sont représentés par les couples de triplets* $(M(p_i), \nu(t_i))$.

$$\begin{pmatrix} 1 & 0 \\ 0 & \times \\ 1 & 0 \end{pmatrix} \xrightarrow{0.75} \begin{pmatrix} 1 & 0.75 \\ 0 & \times \\ 1 & 0.75 \end{pmatrix} \xrightarrow{t_1} \begin{pmatrix} 0 & \times \\ 1 & 0 \\ 2 & 0.75 \end{pmatrix} \xrightarrow{1.5} \begin{pmatrix} 0 & \times \\ 1 & 1.5 \\ 2 & 2.25 \end{pmatrix} \xrightarrow{t_3} \begin{pmatrix} 0 & \times \\ 1 & 1.5 \\ 1 & 0 \end{pmatrix} \xrightarrow{0.5} \cdots$$

Le symbole $\times$ *signifie que la transition n'est pas sensibilisée et que la valeur de l'horloge associée n'a pas d'influence dans l'évolution du réseau. On notera que le tir de* t_1 *ne remet pas l'horloge associée à* t_3 *à zéro puisque* t_3 *était déjà sensibilisée par* M_0 *:* $t_3 \notin \uparrow enabled(M_0, t_1)$. *En revanche, le tir de* t_3 *depuis le marquage* $(0,1,2)$ *conduit à sa nouvelle sensibilisation et donc à la remise à zéro de son horloge.*

5.1.3 Propriétés et résultats principaux

Étant donné un TPN $\mathcal{T}$ muni de sa sémantique $||\mathcal{T}||$, un certain nombre de problèmes peuvent être étudiés ; citons les plus usuels :

– *accessibilité de marquage* : « étant donné un marquage M, existe-t-il une valuation ν telle que $(M,\nu) \in Q$? », soit « $\exists \nu \in (\mathbb{R}_{\geq 0})^T / (M,\nu) \in Q$ » ;
– *bornitude* : « $\exists k \in \mathbb{N}, \forall (M,\nu) \in Q, \forall p \in P, M(p) \leq k$ » ;
– *k-bornitude* : étant donné $k \in \mathbb{N}$, « $\forall (M,\nu) \in Q, \forall p \in P, M(p) \leq k$ » ;
– *accessibilité d'état* : étant donné un état q, « $q \in Q$ » ;
– *vivacité* : « $\forall t \in T, \forall q \in Q, \exists \tau = (t_0, t_1, \dots) \in T^*, q' \in Q,\ q \xrightarrow{\tau.t} q'$ ».

Les auteurs de [44] ont montré que les TPNs (en temps dense) ont le même pouvoir d'expressivité que les machines de TURING. Par conséquent, l'accessibilité du marquage est un problème indécidable. *A fortiori*, les problèmes de la bornitude, de l'accessibilité d'état et de vivacité sont également indécidables. En revanche, la k-bornitude est un problème décidable : elle peut en effet être décidée *via* le calcul d'une abstraction finie de l'espace d'états (telle que le graphe des classes d'états présenté dans [14]).

Dans le cas des TPNs bornés, tous les problèmes ci-dessus deviennent décidables.

Remarque 5.5. *D'une manière générale, un certain nombre de résultats « intéressants » s'appliquent aux réseaux bornés ou même saufs* (i.e. 1*-bornés). De nombreuses recherches ont alors été menées pour établir des conditions suffisantes de bornitude des TPNs. En particulier, la bornitude du réseau atemporel sous-jacent forme une condition suffisante (mais non nécessaire) à la bornitude du TPN. Nous retrouvons ainsi des propriétés similaires aux EFFBDs (cf. la proposition 4.4).*

5.1.4 Introduction des arcs de vidange et de lecture

Bien que leur expressivité soit importante, les TPNs ne permettent pas une modélisation immédiate des comportements de remise à zéro ou de lecture sans consommation. Pour pallier ces déficiences, deux types d'arcs ont été ajoutés au formalisme classique :

- les arcs de *vidange* (*reset arcs* [24]) reliant une place p à une transition t, ils ne modifient pas la condition de sensibilisation de la transition. En revanche, lorsque celle-ci est tirée, tous les jetons contenus par p sont supprimés ;
- les arcs de lecture ou de test *(read arcs* [21]), reliant également une place p à une transition t, ils empêchent la sensibilisation de t si p ne contient pas au moins autant de jetons que donné par le poids de l'arc. En revanche, lors du tir de t, les jetons contenus par p ne sont pas consommés.

Formellement, nous obtenons :

Définition 5.3 (Réseau de PETRI temporel avec arcs de vidange et de lecture) *Un* réseau de PETRI temporel avec des arcs de vidange et de lecture *est un n-uplet* $\mathcal{T} = (P, T, {}^\bullet(.), (.)^\bullet, {}^\blacklozenge(.), {}^\lozenge(.), M_0, a, b, A, \Lambda)$ *où :*

- $P, T, {}^\bullet(.), (.)^\bullet, M_0, a, b, A, \Lambda$ *sont définis comme précédemment ;*
- ${}^\blacklozenge(.) \in (\{0,1\}^P)^T$ *est la* fonction de *reset ;*
- ${}^\lozenge(.) \in \mathbb{N}^P)^T$ *est la* fonction de lecture *;*

Soit t une transition, nous notons $\rho(t)$ le vecteur défini par :

$$\forall p \in P,\ \rho(t)[p] = \max({}^\blacklozenge t(p) * M(p), {}^\bullet t(p))$$

$\rho(t)[p]$ représente le nombre de jetons qui seront retirés de la place p lors du tir de t, en tenant compte d'un éventuel arc de vidange. Les définitions de $enabled(M)$ et $\uparrow enabled(t, M, t_f)$ sont modifiées comme suit :

$$enabled(M) = \{t \in T \ /\ M \geq \max({}^\bullet t, {}^\lozenge t)\}$$

$$\uparrow enabled(t, M, t_f) = \begin{cases} \max({}^{\bullet}t, {}^{\diamond}t) \leq M - \rho(t_f) + t_f{}^{\bullet} \\ (t = t_f) \vee (\max({}^{\bullet}t, {}^{\diamond}t) > M - \rho(t_f)) \end{cases}$$

En remarquant que seule la relation de transition discrète est affectée par l'introduction de ces arcs, la sémantique est alors définie par :

Définition 5.4 (Sémantique d'un TPN avec arcs de vidange et de lecture) *La sémantique d'un TPN $\mathcal{T}$ avec arcs de vidange et de lecture est un quadruplet $||\mathcal{T}|| = (Q, q_0, A, \rightarrow)$ tel que :*

- $Q \subseteq \mathbb{N}^P \times (\mathbb{R}_{\geq 0})^T$;
- $q_0 = (M_0, \vec{0})$;
- $\rightarrow \subseteq Q \times \{A \cup \mathbb{R}_{\geq 0}\} \times Q$ *est la* relation de transition *composée :*
  - *de la relation de transition discrète définie pour tout $t_f \in T$ par :*

$$(M, \nu) \xrightarrow{\Lambda(t_f)} (M', \nu') \Leftrightarrow \begin{cases} t_f \in enabled(M) \\ a(t_f) \leq \nu(t_f) \leq b(t_f) \\ M' = M - \rho(t_f) + t_f{}^{\bullet} \\ \forall t \in T,\ \nu'(t) = \begin{cases} 0 & \text{si } t \in \uparrow enabled(M, t_f) \\ \nu(t) & \text{sinon} \end{cases} \end{cases}$$

  - *de la relation de transition continue définie comme précédemment.*

Exemple 9. *Considérons le réseau illustré figure 5.2. Les arcs de vidange (respectivement* de lecture*) sont figurés par des flèches dont l'extrémité est un losange plein (respectivement* blanc*). Sur cet exemple, on obtient donc ${}^{\blacklozenge}t_1(P_i) = (0,1,0,0)_{(P_1,P_2,P_3,P_4)}$ et ${}^{\diamond}t_2 = (0,0,2,0)$. De même, $\rho(t_1) = (2,3,0,0)$ à partir du marquage initial $M_0 = (3,3,1,1)$.*

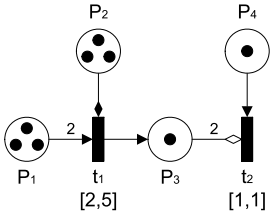


FIGURE 5.2 – Exemple de réseau de PETRI temporel avec arc de vidange

Une exécution possible du réseau est :

$$\begin{pmatrix} 3 \\ 3 \\ 1 \\ 1 \end{pmatrix}, \begin{matrix} \\ 0 \\ \times \\ \end{matrix} \xrightarrow{2.75} \begin{pmatrix} 3 \\ 3 \\ 1 \\ 1 \end{pmatrix}, \begin{matrix} \\ 2.75 \\ \times \\ \end{matrix} \xrightarrow{t_1} \begin{pmatrix} 1 \\ 0 \\ 2 \\ 1 \end{pmatrix}, \begin{matrix} \\ \times \\ 0 \\ \end{matrix} \xrightarrow{1} \begin{pmatrix} 1 \\ 0 \\ 2 \\ 1 \end{pmatrix}, \begin{matrix} \\ \times \\ 1 \\ \end{matrix} \xrightarrow{t_2} \begin{pmatrix} 1 \\ 0 \\ 2 \\ 0 \end{pmatrix}, \begin{matrix} \\ \times \\ \times \\ \end{matrix}$$

Enfin, les arcs de vidange ou de lecture ne modifient pas les résultats de décidabilité des réseaux bornés [53].

5.2 Motifs de traduction

Il nous est rapidement apparu naturel de procéder à une transformation structurelle des EFFBDs, c'est-à-dire en traduisant chaque flux et chaque type de nœud par un motif de conception élémentaire en TPN. Le réseau final est alors obtenu par la connexion des différents motifs entre eux, comme présenté dans la section 5.2.2. Afin de faciliter la lecture, la plupart des places et des transitions sont notées par $p(n)$ ou $t(n)$. Nous conserverons dans le reste du document les notations introduites dans la section 5.2.1.

Les EFFBDs atemporels sont traduits par des réseaux de Petri, les EFFBDs temporels sans structure de terminaison forcée par des TPNs et les EFFBDs comprenant ces structures par des TPNs avec arcs de vidange.

5.2.1 Description des motifs

Motifs des flux Un flux A est traduit par une unique place $pFlux(A)$, dont le marquage initial vaut $F_0(A)$.

Motifs des nœuds Chaque motif traduisant un nœud n est un TPN (sans arc de vidange ou de lecture) $\mathcal{T}^n = ((P^n, T^n, {}^\bullet(.)^n, (.)^{\bullet n}, M_0^n, a_\mathcal{T}^n, b_\mathcal{T}^n, n, \Lambda^n))$. Les figures 5.3 à 5.10 donnent les motifs correspondant aux différents éléments présentés dans les chapitres 3 et 4.

Dans ces motifs, les flèches pointillées dont l'extrémité est pleine représentent les arcs réalisant la connexion entre les autres motifs du modèle. Dans le cas des motifs bouclant (cf. figures 5.6 et 5.7) cependant, l'arc modélisant le retour du contrôle du nœud fermant au nœud ouvrant est en traits pleins. De plus, certaines traductions font intervenir des arcs de vidange et de lecture (cf. figures 5.4 et 5.8) ; ceux-ci ne font pas à proprement parler partie d'un motif spécifique mais plutôt du réseau complet. Nous détaillons donc leur construction dans la section 5.2.2.

On notera que tous les motifs de nœuds ouvrants débutent par une seule place et que tous les nœuds fermants s'achèvent par une seule transition. C'est pourquoi nous symboliserons les motifs correspondant aux constructions contenues dans la structure étudiée par une place reliée par un arc (dont l'extrémité est blanche) à une transition, ces trois éléments étant également en pointillés.

Structure parallèle La figure 5.3 donne le motif d'une structure parallèle $\langle \alpha, \bar{\alpha} \rangle$ à m branches. Les places en amont de $t(\bar{\alpha})$ réalisent la synchronisation entre les branches permettant la sortie de la structure parallèle. Ces places sont notées $pSynch(\bar{\alpha}, n_i)$ où n_i est le dernier nœud de la i^e branche parallèle.

Structure parallèle avec *kill* Le motif d'une structure parallèle dont la dernière branche [2] porte le modificateur *kill* est donné figure 5.4. Très similaire au motif précédent, il comprend en outre un arc « générique », noté ←--◆. Cet arc traduit le fait qu'un arc de vidange additionel relie toute place du motif de tout nœud contenu par les autres branches parallèles à la

2. Bien évidemment, le motif s'étend aux cas où une ou plusieurs autres branches portent le modificateur *kill*.

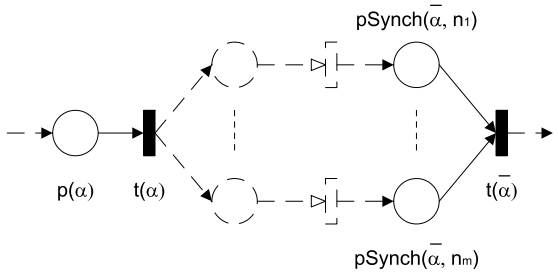


FIGURE 5.3 – Motif d'une structure parallèle $\langle \alpha, \bar{\alpha} \rangle$

transition de sortie de la dernière construction de la branche tueuse (ici notée $t\kappa$). En outre, s'il existe une place p dans les autres branches parallèles dont le marquage initial q est non nul [3], on ajoute un arc de poids q entre la transition $t\kappa$ et p, de manière à *réinitialiser* le motif auquel p appartient. La construction exacte des ces arcs additionnels est donnée section 5.2.2.

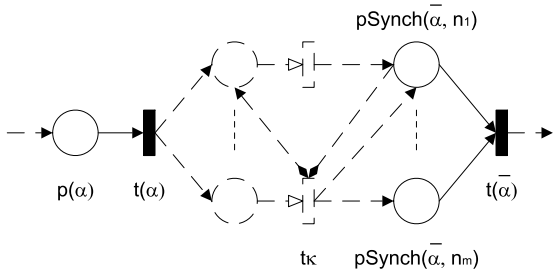


FIGURE 5.4 – Motif d'une structure parallèle $\langle \alpha, \bar{\alpha} \rangle$ avec *kill*

Enfin, on notera que, après le tir de $t\kappa$, toutes les places en amont de $t(\bar{\alpha})$ contiennent exactement un jeton : nous verrons dans la suite que cela correspond dans le modèle EFFBD au comportement selon lequel on force tous les prédécesseurs du nœud $\bar{\alpha}$ à être dans l'état *executed* (cf. section 4.3.3).

Structure de sélection La figure 5.5 donne le motif d'une structure de sélection $\langle \omega, \bar{\omega} \rangle$ à m branches ($m \geq 2$). D'une manière similaire au motif d'une structure parallèle, n_i représente le premier nœud de la i^e branche de sélection.

Structure de boucle et sortie de boucle La figure 5.6 donne le motif d'une structure de boucle $\langle \lambda, \bar{\lambda} \rangle$ contenant un nœud de sortie de boucle le [4].

Tout comme dans le cas des branches parallèles avec *kill*, on ajoute un arc de vidange entre toutes les autres places contenues dans la boucle et la transition $t(le)$ et, le cas échéant, des arcs permettant la réinitialisation des motifs contenus par la boucle. On notera que la

3. D'après le tableau 5.1 donné p. 103, et comme le nœud initial ne peut être contenu dans une quelconque structure, nous verrons qu'il ne peut s'agir que de structures d'itération.

4. Le motif s'étend sans difficulté au cas où la boucle contient plusieurs nœuds LE.

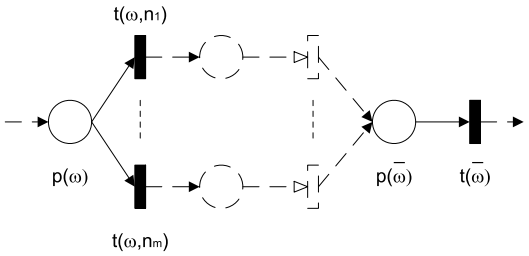


FIGURE 5.5 – Motif d'une structure de sélection $\langle \omega, \bar{\omega} \rangle$

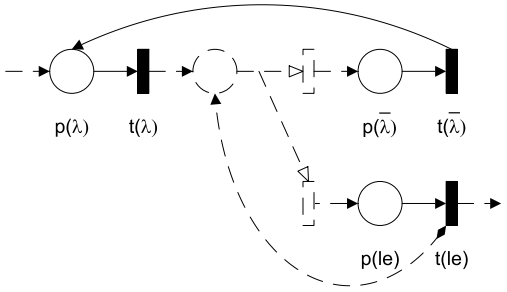


FIGURE 5.6 – Motif d'une structure de boucle $\langle \lambda, \bar{\lambda} \rangle$ et d'une sortie de boucle *le*

transition $t(\bar{\lambda})$ n'a pas d'autre successeur que $p(\lambda)$: en effet, nous avons vu que le contrôle passe toujours du nœud LP fermant au nœud ouvrant correspondant. En revanche, l'arc de connexion partant de $t(le)$ sera bien sûr relié à la première place du motif du successeur de $\bar{\lambda}$.

Structure d'itération Le motif d'une structure d'itération $\langle \iota, \bar{\iota} \rangle$ est donné 5.7. La place $pMin(\iota)$ garantit que l'itération sera effectuée au moins i_M fois (avec $i_M = iter(\iota)$) ; la place $pMax(\iota)$, dont le marquage initial vaut i_M, garantit que l'on pourra effectuer l'itération au plus i_M fois.

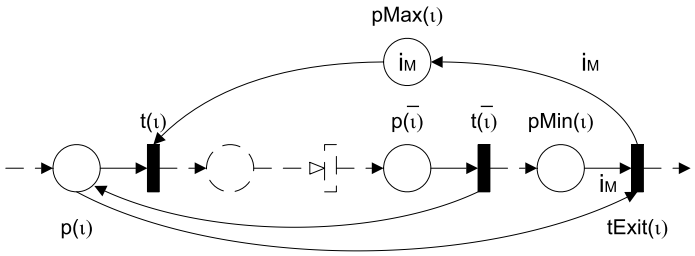


FIGURE 5.7 – Motif d'une structure d'itération $\langle \iota, \bar{\iota} \rangle$

À l'issue de la dernière itération, le tir de $t(\bar{\iota})$ place un jeton dans $p(\iota)$ mais la transition $t(\iota)$ ne pourra plus être tirée, faute de jetons dans $pMax(\iota)$: l'arc situé entre $p(\iota)$ et $tExit(\iota)$ permet alors de vider $p(\iota)$.

Remarque 5.6. *Dans ce motif, on remet le compteur à zéro, en plaçant i_M jetons dans $pMax(\iota)$, à la sortie de la structure : on retrouve donc le comportement annoncé dans la remarque 4.3.*

Fonction à sortie simple La figure 5.8 donne le motif de traduction d'une fonction à sortie simple f, manipulant les trois flux L, C et P tels que $Lect(f)[L] = qL$, $Cons(f)[C] = qC$ et $Prod(f)[P] = qP$. Le franchissement de la transition $\uparrow f$ marque le début de l'exécution de la fonction f, $\downarrow f$ la fin. La place $pW(f)$ correspond donc à l'attente des flux d'entrée, $pE(f)$ à l'exécution de la fonction. On notera que la transition $\downarrow f$ est la seule (avec ses équivalents dans le motif d'une fonction multi-sortie non décomposée) dont l'intervalle temporel est potentiellement différent de $[0, 0]$, puisque $a^f_{\mathcal{T}}(\downarrow f) = a(f)$ et $b^f_{\mathcal{T}}(\downarrow f) = b(f)$.

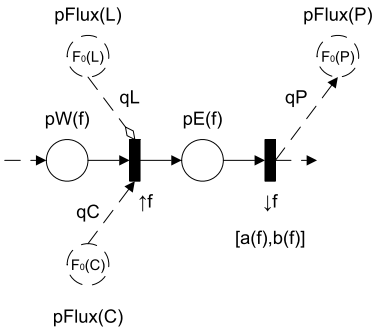


FIGURE 5.8 – Motif d'une fonction à sortie simple f

Fonction multi-sortie non décomposée Le motif précédent est modifié de façon à prendre en compte les m branches de sortie de la fonction f ($m \geq 2$). On remarquera que le motif, donné figure 5.9, peut être vu comme une composition des motifs des figures 5.5 et 5.8.

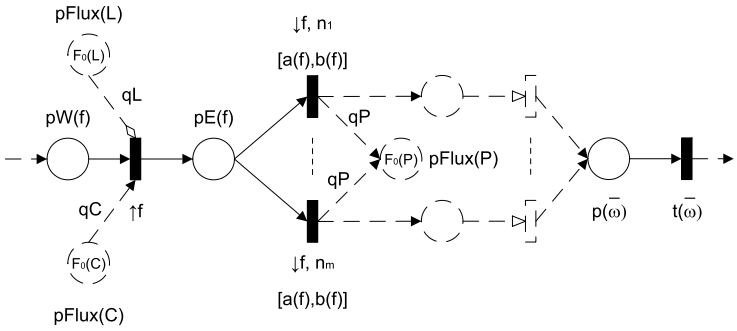


FIGURE 5.9 – Motif d'une fonction multi-sortie non décomposée $\langle f, \bar{\omega} \rangle$

Fonction multi-sortie décomposée La figure 5.10 donne le motif d'une fonction décomposée ayant m branches de sortie ($m \geq 2$). Le premier motif en pointillé (à gauche) représente

la traduction du sous-scénario. Les transitions de sortie des motifs des nœuds EXIT sont notés $t(\epsilon, 1)$ à $t(\epsilon, m)$; pour chaque nœud ϵ_i, un arc de vidange est ajouté entre chaque place du sous-scénario (y compris les places $t(\epsilon, j)$, $i \neq j$) et la transition $t(\epsilon, i)$ (de même, on ajoute le cas échéant des arcs permettant la réinitialisation des motifs contenus par le sous-scénario).

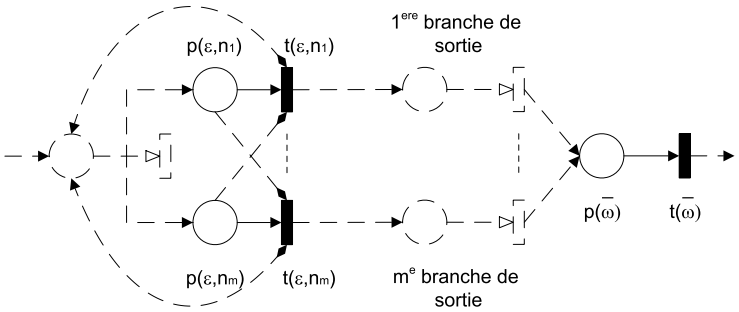


FIGURE 5.10 – Motif d'une fonction multi-sortie décomposée $\langle \epsilon_1, \ldots, \epsilon_m, \bar{\omega} \rangle$

Description des fonctions M_0 et Λ Lorsqu'il n'est pas nul, le marquage initial des places des différents motifs est donné par le tableau 5.1. Nous y avons conservé les notations des figures 5.3 à 5.10.

TABLE 5.1 – Marquages initiaux des motifs de traduction

Nœud n	**Place**	**M_0^n**	**Remarque**
AND_{in}	$p(\alpha)$	1	ssi $n = n_0$
OR_{in}	$p(\omega)$	1	
LP_{in}	$p(\lambda)$	1	
IT_{in}	$p(\iota)$	1	
	$pMax(\iota)$	i_M	avec $i_M = iter(n)$
FC	$pW(f)$	1	ssi $n = n_0$

Enfin, la fonction d'étiquetage des motifs est définie par :

$$\forall n \in \mathcal{N},\ \forall t \in T^n,\ \Lambda(t) = n$$

En particulier, dans le cas d'un nœud OR ouvrant ω, le tir de toute transition $t(\omega, n_i)$ correspond à l'action ω : on garde donc explicitement le non déterminisme évoqué par la remarque 4.5 p. 75.

5.2.2 Construction du réseau complet

Le réseau final $\mathcal{T} = (P, T, {}^\bullet(.), (.)^\bullet, {}^\blacklozenge(.), (.)^\diamond, M_0, a_\mathcal{T}, b_\mathcal{T}, \mathcal{N}, \Lambda)$ est obtenu par la conjonction des motifs des nœuds et des flux auxquels sont ajoutés les arcs de connexion entre motifs ainsi que les éventuels arcs de vidange et de lecture.

Places et transitions En notant $P_{\mathcal{N}} = \cup_{n\in\mathcal{N}}(P^n)$ et $P_{\mathcal{F}} = \cup_{A\in\mathcal{F}}(pFlux(A))$ on obtient donc $P = P_{\mathcal{N}} \cup P_{\mathcal{F}}$. De même, $T = \cup_{n\in\mathcal{N}} T^n$.

Nous adoptons dans la suite les notations suivantes :

- $\forall p \in P_{\mathcal{N}}$, $\pi(p)$ est l'unique nœud tel que $p \in P^{\pi(p)}$;
- $\forall t \in T$, $\tau(t)$ est l'unique nœud tel que $t \in T^{\tau(t)}$;
- $\forall p \in P_{\mathcal{F}}$, $\varphi(p)$ est l'unique flux tel que $p = pFlux(\varphi(p))$.

Arcs d'amont La fonction ${}^\bullet(.)$, regroupant les arcs d'amont internes aux motifs et les arcs marquant la consommation de flux, est définie pour tout $t \in T$ par :

$$\forall p \in P_{\mathcal{N}},\quad {}^\bullet t(p) = \begin{cases} {}^\bullet t^{\tau(t)}(p) & \text{si } \tau(t) = \pi(p) \\ 0 & \text{sinon} \end{cases}$$

$$\forall p \in P_{\mathcal{F}},\quad {}^\bullet t(p) = \begin{cases} k & \text{si } (f = \tau(t) \in FC) \wedge (Cons(f)[\varphi(p)] = k) \wedge (t = tW(f)) \\ 0 & \text{sinon} \end{cases}$$

Arcs d'aval Comme mentionné plus haut, la fonction $(.)^\bullet$ regroupe :

- les arcs d'aval internes aux motifs ;
- les arcs de connexion entre motifs ;
- les arcs marquant la production de flux ;
- les arcs permettant la réinitialisation des itérations tuées par des motifs de terminaison forcée.

Nous définissons ainsi une fonction booléenne $Connect \subseteq \mathbb{B}^{T\times P}$ qui, à chaque transition t et à chaque place p, associe `vrai` si et seulement si t doit être reliée à p pour marquer la connexion de deux motifs. L'ensemble des cas possibles étant assez large, nous avons préféré présenter dans le tableau 5.2 les conditions pour que la transition t soit effectivement reliée à la place p en fonction du type de nœud modélisé [5]. Le symbole - signifie que $Connect(t,p)$ vaut toujours `faux`.

Les conditions (1) à (9) sont décrites ci-après ; on notera que les conditions (1), (5) et (7) sont similaires, de même que les conditions (2), (6) et (8) et les conditions (4) et (9).

(1) $n' \in Post(n)$

(2) $p = pSynch(\bar{\alpha}, n)$ avec $\bar{\alpha} = n'$

(3) $t = t(\omega, n')$ avec $\omega = n$

(4) $\langle n', n\rangle \in LP$

(5) $n' \in Post(\bar{\lambda})$ avec $\bar{\lambda} \,/\, \langle \lambda, \bar{\lambda}\rangle = Ref(n)$

(6) $p = pSynch(\bar{\alpha}, \bar{\lambda})$ avec $\bar{\alpha} = n'$ et $\bar{\lambda} \,/\, \langle \lambda, \bar{\lambda}\rangle = Ref(n)$

(7) $n' \in Post(\bar{\iota})$ avec $\langle n, \bar{\iota}\rangle \in IT$

(8) $p = pSynch(\bar{\alpha}, \bar{\iota})$ avec $\bar{\alpha} = n'$ et $\langle n, \bar{\iota}\rangle \in IT$

(9) $\langle n', n\rangle \in IT$

5. Pour faciliter la lecture, nous ne prenons pas en considération les fonctions à sortie multiple non décomposées : il s'agit en effet d'une combinaison des cas FC et OR_{in}.

TABLE 5.2 – Règles de connexion entre deux motifs

			$n' = \pi(p)$							
			$AND_{in} \cup OR_{in} \cup LE \cup FC \cup EXIT$	AND_{out}	OR_{out}	LP_{in}	LP_{out}	IT_{in}		IT_{out}
								pn'	$pMin\,n'$	
$n = \tau(t)$	AND_{in}		(1)	(1)	-	(1)	-	(1)	-	-
	$AND_{out} \cup OR_{out} \cup FC$		(1)	(2)	(1)	(1)	(1)	(1)	-	(1)
	OR_{in}		(3)	-	(3)	(3)	-	(3)	-	-
	LP_{in}		(1)	-	-	(1)	(1)	(1)	-	-
	LP_{out}		-	-	-	(4)	-	-	-	-
	LE		(5)	(6)	(5)	(5)	(5)	(5)	-	(5)
	IT_{in}	tn	(1)	-	-	(1)	-	(1)	-	(1)
		$tExit\,n$	(7)	(8)	(7)	(7)	(7)	(7)	-	(7)
	IT_{out}		-	-	-	-	-	(9)	(9)	-
	$EXIT$		(1)	-	(1)	(1)	-	(1)	-	-

La condition (1) représente la connexion « usuelle » d'un nœud avec son successeur (les conditions (5) et (7) prennent en compte les règles sémantiques propres aux structures de boucle et d'itération). Les conditions (2), (6) et (8) décrivent « l'aiguillage correct » entre un nœud AND fermant et ses prédécesseurs. De façon similaire, la condition (3) traite de la bonne connexion entre un nœud OR ouvrant et ses successeurs. Enfin, la condition (4) (resp. (9)) décrit les arcs liant les motifs des deux nœuds d'une structure LP (resp. IT).

Par ailleurs, on dira qu'un nœud n est *terminant* s'il est le dernier [6] d'une branche parallèle portant le modificateur *kill*. Par extension, une transition t sera terminante, ce que nous notons $isKiller(t) = \texttt{vrai}$ si $\tau(t)$ est terminant.

Remarque 5.7. *Dans le cas de l'itération, seule la transition $tExit(\iota)$ peut être terminante. Dans tous les autres cas, et sachant que ni un nœud de OR_{in} ni une fonction de FC_{MS} ne peuvent être le dernier nœud d'une branche parallèle, la transition est bien unique.*

6. Dans le cas d'une itération, si le nœud fermant est le dernier de la branche, alors seul son nœud ouvrant est terminant.

Plus généralement, nous définissons la fonction booléenne $isKilled(p,t)$ pour tous $t \in T$ et $p \in P$ par :

$$isKilled(p,t) = \texttt{vrai} \Leftrightarrow \begin{cases} \begin{cases} Paths(\pi(p),\tau(t)) = \emptyset \\ isKiller(t) \\ \pi(p) \in \mathcal{C}(\langle \alpha, \bar{\alpha} \rangle) \vee p = pSynch(\bar{\alpha}, n_i) \text{ avec } n_i \neq \tau(t) \end{cases} & (5.1) \\ \text{ou} \begin{cases} \tau(t) \in LE \\ \pi(p) \in \mathcal{C}(Ref(\tau(t)) \setminus \{\tau(t)\} \end{cases} & (5.2) \\ \text{ou} \begin{cases} \tau(t) \in EXIT \\ \pi(p) \in \mathcal{C}(Ref(\tau(t)) \setminus \{\tau(t)\} \end{cases} & (5.3) \end{cases}$$

Les conditions 5.1, 5.2 et 5.3 s'appliquent respectivement à la terminaison par *kill*, nœud LE et nœud EXIT. La sous-condition $Paths(\pi(p),\tau(t)) = \emptyset$ rappelle que les constructions situées sur la même branche parallèle que l'élément tueur n'ont pas besoin d'être tuées puisqu'elles sont nécessairement déjà inactives. La fonction est donc vraie si la place p doit être vidée par le tir de la transition t ; si $p = pMax(\iota)$, la place doit être réinitialisée.

Munis de ces notations, nous pouvons alors définir $(.)^\bullet$ pour tout $t \in T$ par :

$$\forall p \in P_{\mathcal{N}}, \quad t^\bullet(p) = \begin{cases} t^{\bullet\tau(t)}(p) & \text{si } \tau(t) = \pi(p) \\ 1 & \text{si } Connect(t,p) \vee (isKiller(t) \wedge p = pSynch(\bar{\alpha}, n_i)) \\ iter(\pi(p)) & \text{si } \iota = \pi(p) \in IT_{in} \wedge p = pMax(\iota) \wedge isKilled(p,t) \\ 0 & \text{sinon} \end{cases}$$

$$\forall p \in P_{\mathcal{F}}, \quad t^\bullet(p) = \begin{cases} k & \text{si } (f = \tau(t) \in FC) \wedge (Prod(f)[\varphi(p)] = k) \wedge (t =\downarrow f) \\ 0 & \text{sinon} \end{cases}$$

On notera que les différentes conditions sont en exclusion mutuelle.

Arcs de vidange Des définitions précédentes, nous pouvons déduire immédiatement la définition de ${}^\blacklozenge(.)$ pour tout $t \in T$:

$$\forall p \in P_{\mathcal{N}}, \quad {}^\blacklozenge t(p) = \begin{cases} 1 & \text{si } isKilled(p,t) \\ 0 & \text{sinon} \end{cases}$$

$$\forall p \in P_{\mathcal{F}}, \quad {}^\blacklozenge t(p) = 0$$

Arcs de lecture La fonction ${}^\lozenge(.)$ se définit immédiatement pour tout $t \in T$ par :

$$\forall p \in P_{\mathcal{N}}, \quad {}^\lozenge t(p) = 0$$

$$\forall p \in P_{\mathcal{F}}, \quad {}^\lozenge t(p) = \begin{cases} k & \text{si } (f = \tau(t) \in FC) \wedge (Lect(f)[\varphi(p)] = k) \wedge (t =\uparrow f) \\ 0 & \text{sinon} \end{cases}$$

Fonctions M_0, $a_\mathcal{T}$, $b_\mathcal{T}$ et Λ Le marquage initial du réseau complet est défini par :

$$\begin{aligned} \forall p \in P_\mathcal{N}, \quad M_0(p) &= M_0^{\pi(p)}(p) \\ \forall p \in P_\mathcal{F}, \quad M_0(p) &= F_0(\varphi(p)) \end{aligned}$$

Enfin, les fonctions $a_\mathcal{T}$, $b_\mathcal{T}$ et Λ sont définies pour tout $t \in T$ par :

$$\begin{aligned} a_\mathcal{T}(t) &= a_\mathcal{T}^{\tau(t)}(t) \\ b_\mathcal{T}(t) &= b_\mathcal{T}^{\tau(t)}(t) \\ \Lambda(t) &= \tau(t) \end{aligned}$$

Exemple 10. *La section 5.2.3 présente le réseau issu de la traduction du modèle du passage à niveau. Cependant, ce modèle ne présente aucune structure de terminaison forcée : nous avons donc souhaité présenter dans un exemple simple la mise en œuvre des arcs de vidange/réinitialisation.*

Considérons ainsi le modèle illustré figure 5.11. La structure d'itération $\langle \iota, \bar{\iota} \rangle$ est incluse dans une structure parallèle $\langle \alpha, \bar{\alpha} \rangle$ dont la branche supérieure est affectée du modificateur kill.

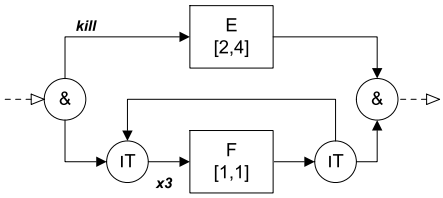


FIGURE 5.11 – Terminaison forcée d'une structure d'itération

En suivant les règles de traduction et de composition énoncées plus haut, la traduction de ce modèle conduit au réseau illustré figure 5.12. Pour faciliter la lecture, les arcs de vidange sont représentés en bleu ; les arcs permettant la réinitialisation de l'itération et le placement forcé des jetons dans les prédécesseurs de $\bar{\alpha}$ en pointillés rouges.

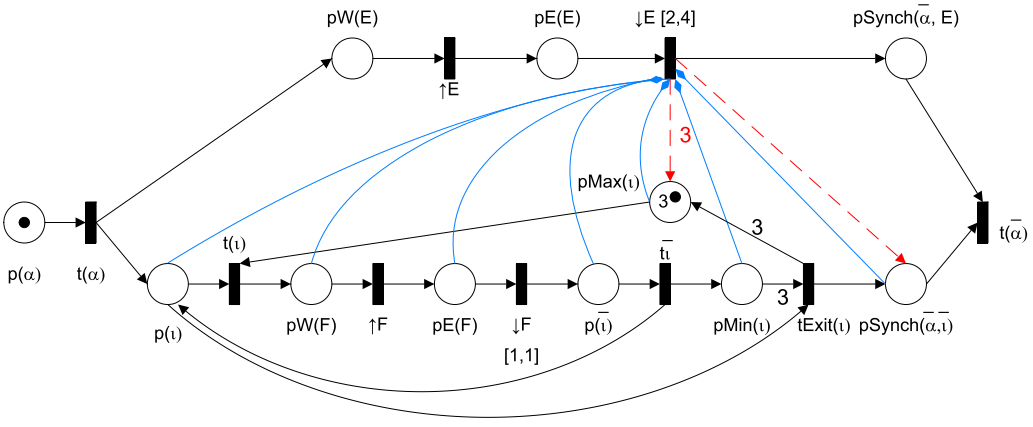


FIGURE 5.12 – Traduction du modèle de la figure 5.11

La fonction E est susceptible de forcer la terminaison de $\langle \iota, \bar{\iota} \rangle$ et de F ; par conséquent, on ajoute des arcs de vidange entre $\downarrow E$ et les places $p(\iota)$, $pW(F)$, $pE(F)$, $p(\bar{\iota})$, $pMin(\iota)$, $pSynch(\bar{\alpha}, E)$ et $pSynch(\bar{\alpha}, \bar{\iota})$, ainsi qu'un arc de poids 3 entre $\downarrow E$ et $pMax(\iota)$.

5.2.3 Application au problème du passage à niveau

Nous donnons figure 5.13 le réseau de PETRI temporel issu de la traduction du modèle du passage à niveau. Nous gardons la nomenclature présentée dans le tableau 4.1.

Afin de faciliter la lecture, les échanges de flux sont représentés par des arcs pointillés gris. Nous avons ajouté à la suite de la transition $t(\bar{\alpha})$ (qui est la « dernière » transition du réseau) une place « terminale » $pFinal$ définie par :

$$\begin{cases} M_0(pFinal) = 0 \\ \forall t \in T \begin{cases} {}^\bullet t(pFinal) = \begin{cases} 1 & \text{si } t = tFinal \\ 0 & \text{sinon} \end{cases} \\ t^\bullet(pFinal) =^{\blacklozenge} t(pFinal) =^{\lozenge} t(pFinal) = 0 \end{cases} \end{cases}$$

où $tFinal$ est la transition finale du réseau ; en notant $nFinal$ le nœud final du modèle EFFBD, cette transition est égale à :

- $t(le)$ si $nFinal \in LP_{out}$ et le est un nœud LE contenu par la boucle terminée par $nFinal$ [7] ;
- $tExit(\iota)$ si $nFinal \in IT_{out}$ et ι est le nœud IT ouvrant l'itération terminée par $nFinal$;
- $t(n)$ dans les autres cas (c'est-à-dire $n \in AND_{out} \cup OR_{out} \cup FC$).

On peut noter que les trois places $pSynch(\bar{\alpha}, \bar{\lambda}_i)$ n'ont pas de transition en amont : elles ne pourront donc jamais contenir de jetons. *A fortiori*, la transition $t(\bar{\alpha})$ ne pourra jamais être franchie et la place $pFinal$ restera toujours vide. Si l'on considère que la présence d'un jeton dans la place $pSynch(\bar{\alpha}, \bar{\lambda}_i)$ signifie que le dernier nœud de la branche i est dans l'état *executed*, on retrouve exactement le comportement décrit dans la remarque 4.6 p. 75.

Plus généralement, nous définissons dans la section suivante une relation entre le comportement d'un EFFBD et celui du TPN issu de sa traduction ; nous montrerons qu'il s'agit d'une bisimulation temporelle.

5.3 Propriétés et résultats

5.3.1 Définition de la relation $\sim$

Définition 5.5 *Soient $\mathcal{E} = (\mathcal{N}, \mathcal{F}, \mathcal{A}, iter, n_0, F_0, a, b, K)$ un EFFBD et $\mathcal{T} = (P, T, {}^\bullet(.), (.)^\bullet, {}^{\blacklozenge}(.), {}^{\lozenge}(.), M_0, a_\mathcal{T}, b_\mathcal{T}, \mathcal{N}, \Lambda)$ le TPN issu de la traduction de $\mathcal{E}$. Soient $||\mathcal{E}|| = (S, s_0, \mathcal{N}, \rightarrow_\mathcal{E})$ et $||\mathcal{T}|| = (Q, q_0, A, \rightarrow_\mathcal{T})$ leur sémantique respective. Nous définissons la relation $\sim \subseteq S \times Q$*

7. Il peut donc y avoir plusieurs transitions finales dans le réseau, mais une seule sera tirable.

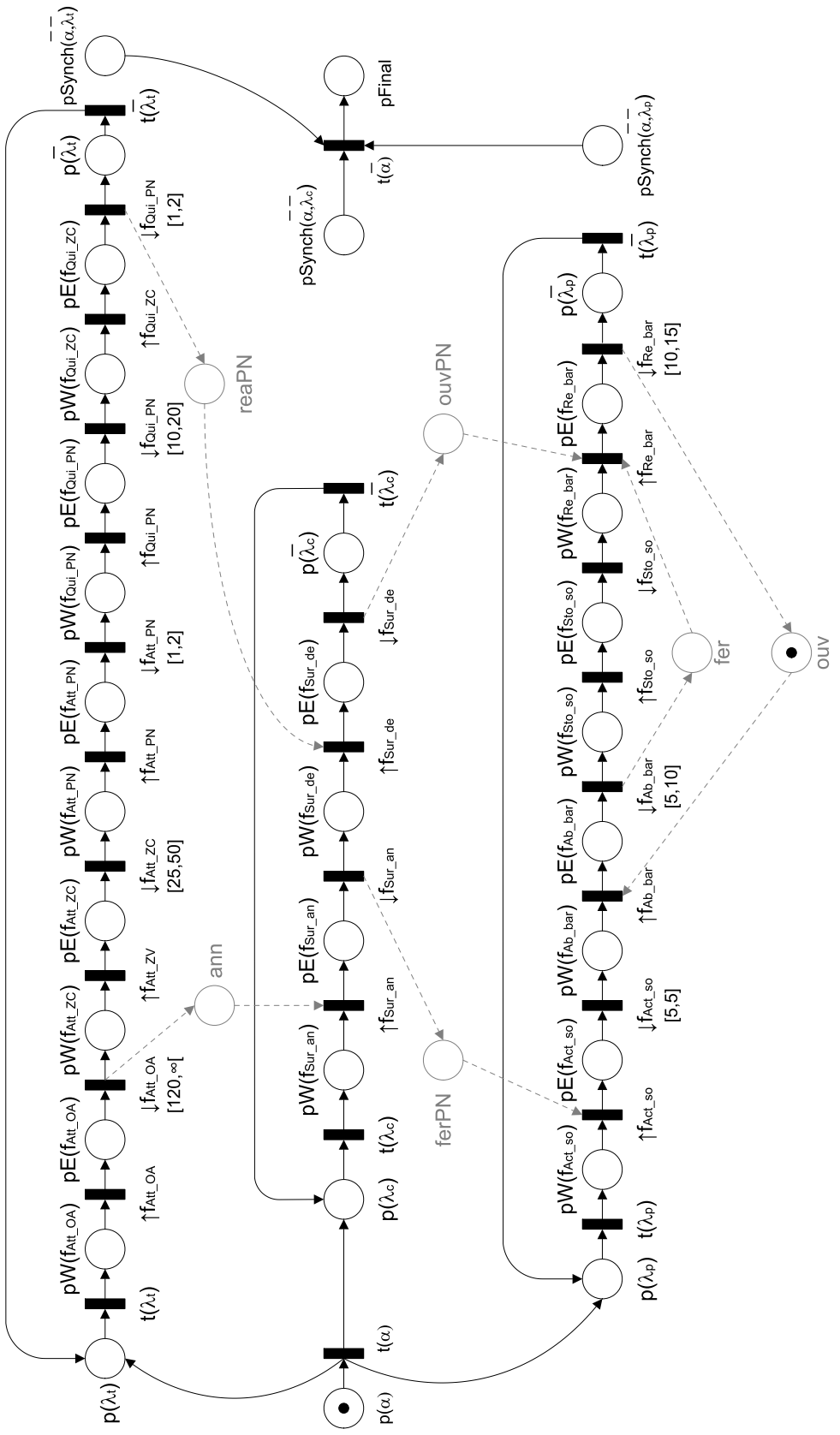


FIGURE 5.13 – Traduction du modèle de la figure 3.17

pour tous $s = (A, C, N, \nu) \in S$ *et* $q = (M, v) \in Q$ *par* $s \sim q$ *si et seulement si :*

$$\begin{cases} \forall n \in \mathcal{N} : A(n) = enabled \Leftrightarrow \begin{cases} M(pW(n)) = 1 & sin \in FC \\ \sum_{n' \in Pre(n)} M(pSynch(n, n')) \geq 1 & sin \in AND_{out} \\ M(p(n)) = 1 & sinon \end{cases} & (5.4) \\ \forall f \in FC : A(n) = executing \Leftrightarrow M(pE(f)) = 1 & (5.5) \\ \forall n \in \mathcal{N} / \exists\, \bar{\alpha} \in AND_{out} \cap Post(n) : A(n) = executed \Leftrightarrow M(pSynch(\bar{\alpha}, n) = 1 & (5.6) \\ \forall \iota \in IT_{in}, \forall k \in \mathbb{N} : C(\iota) = k \Leftrightarrow M(pMax(\iota)) = iter(\iota) - k & (5.7) \\ \forall F \in \mathcal{F}, \forall k \in \mathbb{N} : N(F) = k \Leftrightarrow M(pFlux(F)) = k & (5.8) \\ \forall f \in FC, \forall x \in \mathbb{R}_{\geq 0} : \nu(f) = x \Leftrightarrow v(\downarrow f) = x & (5.9) \\ & (5.10) \end{cases}$$

Les conditions 5.4 à 5.9 découlent directement de la sémantique des EFFBDs et de la construction des motifs en TPNs. On notera la forme particulière de la condition portant sur les valeurs des compteurs (équation 5.7) : elle s'explique par le motif d'une structure d'itération et par la définition de sa sémantique.

5.3.2 Bisimulation temporelle

Proposition 5.1 *La relation* $\sim$ *est une bisimulation temporelle (forte).*

Le schéma général de la preuve consiste à démontrer chacun des points des définitions 2.10 et 2.11 ; nous noterons :

$\mathbf{R_0}$ équivalence des états initiaux ;

$\mathbf{R}_\delta$ propagation de l'équivalence par écoulement du temps ;

$\mathbf{R}_\sigma$ propagation de l'équivalence par tir d'une action.

La démarche est basée sur une induction structurelle : des bisimulations « élémentaires » sont établies en considérant chaque type de transition possible. Nous ne donnons ici que les cas des nœuds AND ouvrants et du passage du temps ; les autres transitions suivent les mêmes arguments et nous en laissons l'étude au lecteur. Seuls les EFFBDs bien formés sont considérés ici : les propriétés d'unicité des structures, de continuité et d'imbrication garantissent que les bisimulations élémentaires se propagent immédiatement à toute combinaison (valide) de transitions.

Démonstration. Soient $\mathcal{E}$, $\mathcal{T}$, $||\mathcal{E}||$, $||\mathcal{T}||$ et $\sim$ définis comme ci-dessus.

$\mathbf{R_0}$ la relation $s_0 \sim q_0$ est triviale ;

$\mathbf{R}_\delta$ Soient $s = (A, C, N, \nu)$, $s' = (A', C', N', \nu') \in S$ et $\delta \in \mathbb{R}_{>0}$[8] tels que $s \xrightarrow{\delta}_{\mathcal{E}} s'$. Soit $q = (M, v) \in Q$ tel que $s \sim q$. Il vient :

- de la définition de $||\mathcal{E}||$, $\nu' = \nu + \delta$ et $\forall f \in FC$[9] telle que $A(f) = executing$, $\nu(f) + \delta \leq b(f)$;

8. Si $\delta = 0$, le cas est trivial.

9. Pour alléger la démonstration, on suppose ici que les fonctions sont à sortie simple. Le raisonnement s'étend bien sûr au cas multi-sortie.

- d'après la définition de $\sim$ et le motif de traduction d'une fonction, $\nu(f) = v(\downarrow f)$ et $b(f) = b_{\mathcal{T}}(\downarrow f)$;
- $enabled(M) = \{\downarrow f / A(f) = executing\}$ car les nœuds $n \notin FC$ ne sont pas activés et $\forall f' \in FC$ telle que $A(n) = enabled$, f' attend des flux d'entrée, *i.e.* $\uparrow f' \notin enabled(M)$;
- par conséquent, $\forall t \in enabled(M)$, $v(t) + \delta \leq b_{\mathcal{T}}(t)$.

Soit $q' = (M, v + \delta)$. Trivialement, on obtient $q \xrightarrow{\delta}_{\mathcal{T}} q'$ et $s' \sim q'$.

Réciproquement, soient $q = (M, v)$, $q' = (M, v') \in Q$ et $\delta \in \mathbb{R}_{>0}$ tels que $q \xrightarrow{\delta}_{\mathcal{T}} q'$. Soit $s = (A, C, N, \nu) \in S$ tel que $s \sim q$. Il vient :

- de la définition de $||\mathcal{T}||$, $v' = v + \delta$ et $\forall t \in enabled(M)$, $v(t) + \delta \leq b_{\mathcal{T}}(t)$;
- d'après la définition de $\sim$ et le motif d'une fonction, $\forall f \in FC$ $\nu(f) = v(\downarrow f)$ et $b(f) = b_{\mathcal{E}}(\downarrow f)$;
- puisque $\delta > 0$, seules les transitions $\downarrow f$ sont sensibilisées (avec $f \in FC$) *i.e.* $\forall n \notin FC$, $A(n) \neq enabled$ et $\forall f' \in FC$, si $A(f') = enabled$ alors l'absence de flux d'entrée empêche f' d'être dans l'état *executing* ;
- par conséquent, $\forall f \in FC$, $A(f) = executing \Rightarrow \nu(f) + \delta \leq b(f)$.

Soit $s' = (A', C', N', \nu + \delta) \in S$. Trivialement, on obtient $s \xrightarrow{\delta}_{\mathcal{E}} s'$ et $s' \sim q'$.

$\mathbf{R}_\sigma$ Soient $s = (A, C, N, \nu)$, $s' = (A', C', N', \nu') \in S$ et $\alpha \in AND_{in}$ tels que $s \xrightarrow{\alpha}_{\mathcal{E}} s'$. Soit $q = (M, v) \in Q$ tel que $s \sim q$. Il vient :

- par définition de $||\mathcal{E}||$, $A(\alpha) = enabled$, tout nœud contenu dans la structure parallèle est inactif, $A'(\alpha) = inactive$[10] et $\forall n \in Post(\alpha)$, $A'(n') = enabled$;
- d'après la définition de $\sim$ et des motifs de traduction, $M(p(\alpha)) = 1$, d'où $t(\alpha) \in enabled(M)$;
- enfin, $v(t(\alpha)) = 0$;

Soit $q' = (M', v)$ tel que $M'(p(\alpha)) = 0$, $\forall n \in Post(\alpha)$, $M'(p(n)) = 1$ et $M'(p) = M(p)$ pour toutes les autres places. Trivialement, $q \xrightarrow{\alpha}_{\mathcal{T}} q'$. D'après les motifs de traduction, l'intervalle de tir de toute transition nouvellement sensibilisée par M' est $[0, 0]$: on obtient donc $s' \sim q'$.

Réciproquement, soient $q = (M, v)$, $q' = (M, v') \in Q$ et $\alpha \in AND_{in}$ tels que $q \xrightarrow{\alpha}_{\mathcal{T}} q'$. Soit $s = (A, C, N, \nu) \in S$ tel que $s \sim q$. Il vient :

- par définition de $||\mathcal{T}||$, $M(p(\alpha)) = 1$ et $\forall p$ telle que $t(\alpha)^\bullet(p) = 1$, $M'(p) = 1$;
- d'après la définition de $\sim$ et des motifs de traduction, $A(\alpha) = enabled$.

Soit $s' = (A', C, N, \nu) \in S$ tel que $A'(\alpha) = inactive$, $\forall n \in Post(\alpha)$, $A'(n) = enabled$. Trivialement, $s \xrightarrow{\alpha}_{\mathcal{E}} s'$. D'après les motifs de traduction, la valuation de toute transition $\downarrow f$ pour $f \in FC$ n'est pas affectée par le tir de $t(\alpha)$: on a donc $s' \sim q'$.

Puisque les nœuds sont soit en séquence, soit complètement emboîtés, les relations de bisimulation élémentaires sont propagées tout au long de l'EFFBD. La relation binaire $\sim$ est donc une bisimulation temporelle forte. □

10. Si l'une des branches parallèles était vide, on pourrait avoir $A'(\alpha) = executed$; nous supposerons pour simplifier que ce n'est pas le cas ici mais notre raisonnement s'appliquerait également.

5.3.3 Résultats complémentaires

De ce résultat fondamental, pierre d'angle de notre démarche de vérification formelle, découlent plusieurs propositions importantes.

Bornitude des TPNs obtenus Les comportements d'un EFFBD $\mathcal{E}$ et de sa traduction $\mathcal{T}$ sont intimement liés : en particulier, la bornitude de $\mathcal{E}$ entraîne celle de $\mathcal{T}$.

Proposition 5.2 *Soient $\mathcal{E}$ un EFFBD borné et F_{max} sa borne*[11]*. Soit m l'entier défini par :*

$$m = \max(F_{max}, \max_{\iota \in IT_{in}} iter(\iota))$$

Soit $\mathcal{T}$ le TPN traduisant $\mathcal{E}$. $\mathcal{T}$ est m-borné.

Démonstration. Considérons le marquage de chaque place du réseau :

- chaque motif de nœud est *sauf* ou, dans le cas d'un nœud IT ouvrant ι, i_M-borné (avec $i_M = iter(\iota)$) :
  - par construction, les poids des arcs sont égaux à 1 ou i_M ;
  - les propriétés de bisimulation et de bonne formation des EFFBDs excluent que l'on rentre dans un motif (*i.e.* que le marquage des places $p(n)$ ou $pE(f)$ augmente) tant que l'on en n'est pas sorti (*i.e.* tant que le marquage n'est pas nul dans les places autres que $pMax(\iota)$) ;
  - la place d'entrée de n_0 (notée $pE(n_0)$ ou $p(n_0)$ selon qu'il s'agisse d'une fonction ou non) a un marquage initial égal à 1 ; les places $pMax(\iota)$ et $pMin(\iota)$ sont, par construction, i_M-bornées ; toutes les autres places sont initialement vides ;
- les places $pFlux(A)$ sont F_{Max}-bornées par définition.

$\mathcal{T}$ est donc m-borné. □

Résultats de décidabilité Nous avons rappelé dans la section 5.1.3 la décidabilité de la k-bornitude des TPNs. Grâce à la proposition 5.1, le corollaire suivant s'en déduit :

Corollaire 5.1 *Soient $\mathcal{E}$ un EFFBD borné, $(S, s_0, \mathcal{N}, \rightarrow)$ sa sémantique et $k \in \mathbb{N}$. Le problème suivant est décidable :*

$$\forall (A, C, FN, \nu) \in S, \forall F \in \mathcal{F}, \quad N(F) \leq k$$

Par conséquent, il est toujours possible de vérifier si le niveau d'un flux quelconque reste toujours en deçà d'une certaine limite, fixée *a priori* par le concepteur du système. On conçoit ainsi l'importance d'un tel résultat lorsqu'il s'agit de dimensionner le système.

De même, à partir des résultats de décidabilité établis sur les TPNs bornés et des propositions 5.1 et 5.2, nous pouvons établir le corollaire suivant :

Corollaire 5.2 *Soient $\mathcal{E}$ un EFFBD borné et $(S, s_0, \mathcal{N}, \rightarrow)$ sa sémantique. Les problèmes suivants sont décidables :*

11. Définie, nous le rappelons, comme étant le plus petit entier k tel que le niveau d'un flux quelconque reste toujours inférieur ou égal à k.

- accessibilité d'une activité [12] : *« étant donnés un nœud $n \in \mathcal{N}$ et une activité $act \in \mathbb{A}$, $A(n) = act$ »* ;
- accessibilité d'un état : *« étant donné un état s, $s \in S$ »*.

Par ailleurs, les auteurs de [20] ont montré que tout TPN borné peut être traduit vers un automate temporisé qui lui est (faiblement) temporellement bisimilaire (cf. remarque 2.4). Par conséquent, un EFFBD borné peut également être transformé en un TA, les deux modèles étant en bisimulation temporelle faible. Il serait alors possible d'appliquer les outils propres aux TA pour effectuer des analyses supplémentaires sur les EFFBDs. Similairement, il pourrait s'avérer intéressant d'étendre le formalisme des EFFBDs pour prendre en compte des entités « hybrides » telles que des taux continus de consommation et de production de flux, etc. et de décrire une transformation équivalente vers les automates hybrides [37], de manière à profiter des résultats établis sur ces modèles.

Enfin, nous verrons dans le chapitre suivant comment adapter les résultats de décidabilité liés à la vérification formelle de certaines logiques temporelles sur les TPNs bornés à la vérification des EFFBDs.

12. On peut bien sûr étendre ce qui suit à l'accessibilité d'un niveau de flux ou d'une valeur de compteur.

Chapitre 6

Vérification des EFFBDs

Résumé *La formalisation des EFFBDs et leur traduction vers les TPNs marquent la première étape vers leur vérification formelle. Celle-ci nécessite en outre la définition d'une logique adaptée à la fois aux exigences de la sûreté de fonctionnement et au formalisme EFFBD ; elle doit également se prêter à une traduction vers une logique spécifique aux TPNs puisque ce sont ces modèles bas niveau qui servent de support aux vérifications effectives.*

Nous introduisons ainsi dans ce chapitre la famille de logiques temporelles quantitatives TCTL (Timed Computation Tree Logic). *Nous donnons en particulier la syntaxe et la sémantique des logiques TPN-TCTL et EFFBD-TCTL permettant d'exprimer des propriétés comportementales sur les TPNs et les EFFBDs. Nous montrons alors l'équivalence entre une propriété donnée sur un modèle EFFBD et sa traduction exprimée sur le TPN résultant, justifiant ainsi notre démarche générale donnée au chapitre 1. Le chapitre se conclut sur la démonstration des résultats de décidabilité et de complexité algorithmique du model checking d'EFFBD-TCTL.*

Nous verrons dans le dernier chapitre comment étendre la démarche générale de vérification pour prendre en compte l'architecture dysfonctionnelle du système.

Sommaire

Après avoir décrit le système par des modèles formels de haut et bas niveau, nous avons cherché un formalisme capable d'exprimer les propriétés comportementales à vérifier. Notre choix s'est rapidement porté sur les logiques de type *Timed Computation Tree Logic* ou TCTL [5]. Elles permettent en effet d'exprimer des propriétés complexes telles que, pour reprendre notre exemple applicatif, « l'arrivée d'un train dans la zone d'annonce conduit toujours à la fermeture complète des barrières en moins de 25 secondes ».

La section 6.1 présente ainsi la logique TCTL et son extension aux réseaux de Petri temporels, TPN-TCTL. La section 6.2 donne la syntaxe et la sémantique formelles de la logique EFFBD-TCTL, que nous illustrerons sur l'exemple du passage à niveau. Nous établissons également l'équivalence entre une formule EFFBD-TCTL Φ, exprimée sur un modèle EFFBD Σ, et sa traduction φ en logique TPN-TCTL sur le TPN σ résultant de la traduction de Σ. Ce résultat fondamental, qui complète la démarche illustrée figure 1.2, nous a permis d'établir certains résultats de décidabilité et de complexité algorithmique, détaillés dans la section 6.3.

Remarque 6.1. *Les langages tels que les TA et les TPNs offrent la possibilité de vérifier des propriétés comportementales par l'emploi* d'observateurs *ramenant la vérification à un problème d'accessibilité d'un état (discret). Un observateur peut être vu comme un sous-modèle composé avec le modèle à vérifier. Il ne doit bien sûr pas perturber le comportement du modèle initial, ce qui nécessite une bonne connaissance du langage et du système.*

Cette technique se heurte à trois difficultés principales : en premier lieu, il n'existe en général aucune technique de génération automatique d'observateurs. De plus, il peut être complexe de ramener l'expression d'une propriété à un problème d'accessibilité. Enfin, l'observateur peut avoir une taille comparable à celle du modèle à vérifier, ce qui accroît d'autant le coût de la vérification. C'est pourquoi nous n'avons pas considéré le développement d'une telle technique pour les EFFBDs, d'autant que le recours aux logiques temporelles s'est révélé suffisamment efficace.

6.1 Logiques temporelles quantitatives

6.1.1 Les logiques CTL* et CTL

La logique TCTL dérive de la logique de branchement temporelle CTL* [26, 63], dont nous rappelons la définition ci-dessous[1].

Définition 6.1 (CTL*) *Le langage* CTL* *est l'ensemble des* formules d'état φ_S *exprimées par des* formules de chemin φ_P *définies inductivement sur un ensemble de variables propositionnelles ap par :*

$$\begin{array}{lcl} \varphi_S & ::= & ap \mid \neg\varphi_S \mid \varphi_S \wedge \varphi_S \mid \forall\varphi_P \mid \exists\varphi_P \\ \varphi_P & ::= & \varphi_S \mid \neg\varphi_P \mid \varphi_P \wedge \varphi_P \mid \bigcirc\varphi_P \mid \varphi_P \,\mathcal{U}\, \varphi_P \end{array}$$

Une formule CTL* fait donc intervenir les quantificateurs de chemin $\forall$ et $\exists$ ainsi que les

1. Par souci de lisibilité, nous ne donnerons pas la sémantique des logiques CTL*, CTL et TCTL, mais uniquement celle de TPN-TCTL et EFFBD-TCTL.

opérateurs temporels `next` [2] (noté $\bigcirc$) et `until` [3] (noté $\mathcal{U}$). En particulier, elle autorise toute combinaison de quanticafieurs de chemin et d'opérateurs temporels [4].

De façon classique, on utilise les notations suivantes :

- $\Diamond\varphi \Leftrightarrow \texttt{vrai}\,\mathcal{U}\,\varphi$;
- $\exists\Box\varphi \Leftrightarrow \neg(\forall\Diamond\neg\varphi)$ et $\forall\Box\varphi \Leftrightarrow \neg(\exists\Diamond\neg\varphi)$

La logique CTL *(Computation Tree Logic)* est le fragment de CTL* tel qu'un opérateur temporel est toujours précédé d'un opérateur de chemin [22]. Elle est définie inductivement par :

$$CTL \quad ::= \quad ap \,|\, \neg\varphi \,|\, \varphi \wedge \varphi \,|\, \forall \bigcirc\varphi \,|\, \exists \bigcirc\varphi \,|\, \forall\varphi\,\mathcal{U}\,\varphi \,|\, \exists\varphi\,\mathcal{U}\,\varphi$$

avec $\varphi \in CTL$.

La logique CTL est suffisamment expressive pour décrire des propriétés telles que « si un train est présent sur le passage à niveau, alors la barrière est nécessairement fermée », ce qui s'écrirait $\forall\Box(train\ sur\ le\ PN \Rightarrow barrière\ fermée)$.

6.1.2 La logique TCTL

Les logiques CTL* et CTL ne permettent pas de quantifier temporellement les propriétés logiques. C'est pour pallier cette limitation qu'a été développée la logique TCTL [4, 5]. Celle-ci peut être vue comme une extension de CTL privée de l'opérateur `next`, ce qui est généralement noté $\text{CTL}_{-\bigcirc}$, où des intervalles de temps sont ajoutés aux opérateurs temporels, de manière à traduire les contraintes de temps sur les formules.

Définition 6.2 (TCTL) *La* logique TCTL *est définie inductivement par :*

$$TCTL ::= ap \,|\, \neg ap \,|\, \varphi \wedge \varphi \,|\, \forall\varphi\,\mathcal{U}_{\mathcal{I}}\varphi \,|\, \exists\varphi\,\mathcal{U}_{\mathcal{I}}\varphi$$

où ap est une proposition atomique, φ une formule de TCTL et $\mathcal{I}$ un intervalle de $\mathbb{R}_{\geq 0}$ à bornes entières de la forme $[n,m]$, $]n,m]$, $[n,m'[$ ou $]n,m'[$ avec $n,m \in \mathbb{N}$ et $m' \in \mathbb{N} \cup \{\infty\}$.

Les définitions des opérateurs $\Box$ et $\Diamond$ sont étendues de façon similaire [5]. De même, on définit l'opérateur de réponse bornée `leads to`, noté $\rightsquigarrow_{\mathcal{I}}$, par :

$$\varphi \rightsquigarrow_{\mathcal{I}} \psi \Leftrightarrow \forall\Box(\varphi \Rightarrow \forall\Diamond_{\mathcal{I}}\psi)$$

La propriété « lorsqu'un train approche, la barrière s'abaisse toujours en moins de 25 secondes » s'écrit alors : $train\ à\ l'approche \rightsquigarrow_{[0,25]} barrière\ fermée$.

2. Intuitivement, $\bigcirc\varphi$ signifie que φ est vraie dans l'état suivant.
3. Intuitivement, $\varphi\mathcal{U}\psi$ signifie que ψ est vraie à partir d'un certain état et que φ est vraie dans tous les états intermédiaires.
4. Dans le cas des systèmes en temps dense, cet opérateur n'a bien sûr plus de sens.
5. Lorsque cela n'introduit pas d'ambiguïté, nous écrirons $\Box$ pour $\Box_{[0,\infty[}$ (*idem* pour $\Diamond$).

6.1.3 La logique TPN-TCTL

Les auteurs de [18, 31] ont proposé une extension de TCTL adaptée aux TPNs, nommée TPN-TCTL. Les intervalles temporels $\mathcal{I}$ représentent des contraintes de temps sur une séquence de tirs. Les propositions atomiques sont des contraintes d'exclusion mutuelle généralisées ou GMECs (pour *Generalized Mutual Exclusion Constraints*), introduites dans [35] et généralisées dans [31]. Elles définissent des ensembles de marquages ou, plus généralement, des propriétés d'exclusion mutuelle, éventuellement pondérées.

Définition 6.3 (GMEC) *Soit $P = \{p_1, \ldots, p_n\}$ un ensemble de places. Une* GMEC *est définie inductivement par :*

$$GMEC ::= \left(\sum_{i=1}^{n} a_i * M(p_i)\right) \bowtie c \,|\, \neg\varphi \,|\, \varphi \wedge \varphi$$

où $a_i \in \mathbb{Z}$, $c \in \mathbb{N}$ et $\varphi \in GMEC$.

Nous donnons, à titre d'illustration, trois GMECs classiques :

$$\bigwedge_{i=1}^{n}(M(p_i) = m_i) \quad (6.1) \qquad \bigwedge_{i=1}^{n}(M(p_i) \leq k) \quad (6.2) \qquad \left(\sum_{p_i \in P'} M(p_i)\right) \leq 1 \quad (6.3)$$

Les équations 6.1 à 6.3 traduisent respectivement l'accessibilité du marquage $(m_i) \in \mathbb{N}^P$, la k-bornitude des places et la présence d'au plus un jeton dans le sous-ensemble des places $P' \subseteq P$, ce qui traduit une situation d'exclusion mutuelle.

La logique TPN-TCTL se définit alors comme suit.

Définition 6.4 (TPN-TCTL) *La* logique TPN-TCTL *est définie inductivement par :*

$$TPN - TCTL ::= GMEC \,|\, \forall\varphi\,\mathcal{U}_{\mathcal{I}}\varphi \,|\, \exists\,\varphi\,\mathcal{U}_{\mathcal{I}}\varphi$$

où φ est une formule de TPN-TCTL et $\mathcal{I}$ un intervalle de $\mathbb{R}_{\geq 0}$ à bornes entières défini comme précédemment.

La sémantique (ou satisfaisabilité) de TPN-TCTL est définie sur le STT donnant la sémantique du TPN. Dans cette définition, les exécutions possibles d'un TPN $\mathcal{T}$ sont notées $\sigma = \langle (s_0, v_0) \xrightarrow{\delta_1} (s'_0, v'_0) \xrightarrow{\sigma_1} (s_1, v_1) \ldots \rangle \in [\![\mathcal{T}]\!]$.

Définition 6.5 (Satisfaisabilité de TPN-TCTL) *Soit $\mathcal{T}$ un TPN et $||\mathcal{T}|| = (Q, q_0, A, \rightarrow)$ sa sémantique. La valeur de vérité d'une formule φ de TPN-TCTL pour un état $(M, v) \in Q$ est déterminée inductivement par :*

$$\begin{array}{lcl}
(M,v) \vDash GMEC & \Leftrightarrow & M \vDash GMEC \\
(M,v) \vDash \neg\varphi & \Leftrightarrow & (M,v) \nvDash \varphi \\
(M,v) \vDash \varphi \wedge \psi & \Leftrightarrow & ((M,v) \vDash \varphi) \wedge ((M,v) \vDash \psi) \\
(M,v) \vDash \exists\,\varphi\,\mathcal{U}_{\mathcal{I}}\psi & \Leftrightarrow & \exists\,\sigma \in [\![\mathcal{T}]\!] \text{ t.q } U_{\mathcal{I}}(\sigma, \varphi, \psi) \\
(M,v) \vDash \forall\varphi\,\mathcal{U}_{\mathcal{I}}\psi & \Leftrightarrow & \forall\sigma \in [\![\mathcal{T}]\!],\ U_{\mathcal{I}}(\sigma, \varphi, \psi)
\end{array}$$

où la condition $U_{\mathcal{I}}(\sigma, \varphi, \psi)$ est définie par :

$$U_{\mathcal{I}}(\sigma, \varphi, \psi) \quad = \quad \begin{cases} (s_0, v_0) = (M, v) \\ \forall i \in \mathbb{N}_{1..n},\ \forall \delta \in [0, \delta_i[: (s_{i-1}, v_{i-1} + \delta) \vDash \varphi \\ (\sum_{i=1}^{n} \delta_i) \in \mathcal{I} \\ (s_n, v_n) \vDash \psi \end{cases}$$

Par extension, nous noterons $\mathcal{T} \vDash \varphi$ si et seulement si l'état initial du réseau vérifie φ *i.e.* $(M_0, \vec{0}) \vDash \varphi$.

Remarque 6.2. *Cette définition de la sémantique de l'opérateur $\mathcal{U}_{\mathcal{I}}$ correspond à l'interprétation* non stricte. *En effet, l'état (s_0, v_0) doit vérifier la propriété φ. Il serait bien sûr possible de définir un opérateur strict, noté $\mathcal{U}^S$, qui vérifie alors $\varphi \mathcal{U}_{\mathcal{I}} \psi \Leftrightarrow \varphi \wedge (\varphi \mathcal{U}^S_{\mathcal{I}} \psi)$.*

Cette logique a été développée afin de permettre un model checking de TCTL sur les TPNs qui soit *natif*. En effet, mis à part l'emploi d'observateurs [6] ou une traduction vers les TA, par exemple, aucune méthode n'avait jusqu'alors été proposée pour la vérification de telles propriétés sur les TPNs [66].

Remarque 6.3. *Le model checking de TPN-TCTL passe par l'exploration de l'espace d'états du modèle. Or, dans le cas du temps continu, l'ensemble des états accessibles est infini, même si le réseau est borné. Il est donc nécessaire de recourir à des* méthodes symboliques *permettant de calculer une abstraction finie de l'espace d'états, tout en préservant les propriétés à vérifier.*

Ainsi, dans l'outil ROMÉO, *c'est la notion de* zones *qui a été retenue pour effectuer le model checking* à la volée *d'une sous-classe de TPN-TCTL [31, 32]. Nous reviendrons sur ces méthodes symboliques lorsque nous évoquerons dans la conclusion de ce mémoire la possibilité d'effectuer le model checking d'EFFBD-TCTL à la volée.*

Nous rappelons sans les détailler quelques résultats fondamentaux sur le model checking de TPN-TCTL établis dans [18, 31].

Théorème 6.1 *La satisfaisabilité de TPN-TCTL est indécidable.*

Ce théorème découle de l'indécidabilité du model checking de TCTL [4].

Théorème 6.2 *La satisfaisabilité de TPN-TCTL est décidable sur les TPNs bornés.*

Théorème 6.3 *Le model checking de TPN-TCTL sur les TPNs bornés est PSPACE-complet* [7].

En particulier, il existe un algorithme de complexité PSPACE décidant toute formule de TPN-TCTL sur les TPNs bornés.

6. En tenant compte des restrictions mentionnées plus haut.
7. En considérant, de façon classique, que la taille d'un TPN est la somme $|P| + |T|$.

6.2 La logique EFFBD-TCTL

6.2.1 Syntaxe et sémantique

D'une façon similaire, nous avons développé une extension de TCTL aux EFFBDs. Celle-ci est fortement inspirée de TPN-TCTL, de manière à tirer parti du travail présenté dans les chapitres précédents ainsi que des résultats donnés dans la section 6.1.3. Ainsi, les propositions atomiques des formules EFFBD-TCTL sont des contraintes EFFBD-GMECs, définies sur un état discret (A, C, N) du modèle EFFBD.

Définition 6.6 (EFFBD-GMEC) *Soient $\mathcal{E}$ un EFFBD et (A, C, N) un état discret d'une exécution de $\mathcal{E}$. Une* contrainte EFFBD-GMEC *est définie inductivement par :*

$$EFFBD-GMEC ::= (A(n) = act) \mid \sum_{\iota \in ITin} a_\iota C(\iota) \bowtie c \mid \sum_{F \in \mathcal{F}} b_F N(F) \bowtie d \mid \neg\varphi \mid \varphi \wedge \varphi$$

où $n \in \mathcal{N}$, $act \in \mathbb{A}$, $a_\iota, b_F \in \mathbb{Z}$, $c, d \in \mathbb{N}$ et $\varphi \in EFFBD-GMEC$.

Tout comme dans la section 6.1.3, nous donnons ci-dessous trois propriétés génériques exprimées en langage naturel et dans la logique EFFBD-TCTL :

- « Les fonctions f_1 à f_n sont en exécution » : $\bigwedge_{i=1}^{n}(A(f_i) = executing)$;
- « Tous les flux sont k-bornés » : $\bigwedge_{F \in \mathcal{F}}(N(F) \leq k)$;
- « L'exécution des fonctions f_1 à f_n est en exclusion mutuelle » :

$$\bigwedge_{i=1}^{n} \left((A(f_i) = executing) \Rightarrow \bigwedge_{j \neq i} A(f_j) \neq executing \right)$$

Dans la suite, nous employons la notation $\downdownarrows \mathcal{E}$ pour signifier que l'exécution de l'EFFBD $\mathcal{E}$ est arrivée à son terme (*i.e.* au nœud final). Dans ce cas, aucun nœud ne peut être dans l'état *enabled*[8] ou *executed*[9]. De même, aucune fonction ne peut être *executing*. On a donc $\downdownarrows \mathcal{E} \Leftrightarrow \bigwedge_{n \in \mathcal{N}} (A(n) = inactive)$.

Définition 6.7 (EFFBD-TCTL) *La* logique EFFBD-TCTL *est définie inductivement par :*

$$EFFBD-TCTL ::= EFFBD-GMEC \mid \neg\varphi \mid \varphi \wedge \psi \mid \forall\varphi \mathcal{U}_{\mathcal{I}} \psi \mid \exists\varphi \mathcal{U}_{\mathcal{I}} \psi$$

où φ et ψ sont deux formules d'EFFBD-TCTL et $\mathcal{I}$ un intervalle de $\mathbb{R}_{\geq 0}$ à bornes entières défini comme précédemment.

La sémantique de cette logique est identique à celle donnée par la définition 6.4 en remplaçant respectivement M par A, C, N et $GMEC$ par $EFFBD-GMEC$. Par extension, nous noterons $\mathcal{E} \vDash \varphi$ si et seulement si l'état initial du modèle $\mathcal{E}$ vérifie φ *i.e.* $(A_0, \mathbf{0}, F_0, \vec{0}) \vDash \varphi$.

Nous donnons ci-dessous l'expression de quelques propriétés de sûreté et de vivacité :

- « La fonction f ne s'exécute jamais » : $\forall\Box(A(f) \neq executing)$;
- « Le modèle $\mathcal{E}$ s'exécute toujours en moins de $x \in \mathbb{N}$ unités de temps » : $\forall\Diamond_{[0,x]} \downdownarrows \mathcal{E}$;
- « L'itération ι se produit au plus deux fois d'affilée avant reset » : $\forall\Box(C(\iota) \leq 2)$.

8. Dans le cas contraire, le modèle peut encore évoluer ou est bloqué dans un état intermédiaire.
9. Puisqu'au moins un nœud de AND_{out} serait alors *enabled*.

6.2.2 Définition et propriétés de la relation $\approx$

La traduction d'une propriété EFFBD-TCTL, exprimée sur un EFFBD $\mathcal{E}$, vers une propriété TPN-TCTL exprimée sur le TPN issu de la traduction de $\mathcal{E}$ s'obtient inductivement à partir des définitions 5.5, 6.3 et 6.6, par la traduction des EFFBD-GMECs en GMECs. En effet, les deux logiques ont la même syntaxe et la même sémantique, à la distinction GMEC/EFFBD-GMEC près. Par souci de lisibilité, nous avons choisi d'omettre la définition de cette traduction.

Remarque 6.4. *Nous avons montré dans le chapitre précédent que si l'EFFBD est bien formé, toutes les places du TPN résultant sont sauves, à l'exception des places $pMin(.)$, $pMax(.)$ et $pFlux(.)$. Ainsi, pour tout nœud non fonctionnel n et toute fonction (non décomposée) f :*

$$\neg(M(p(n)) = 1) \Leftrightarrow M(p(n)) = 0$$
$$\neg(M(pW(f)) = 1) \Leftrightarrow M(pW(f)) = 0$$
$$\neg(M(pE(f)) = 1) \Leftrightarrow M(pE(f)) = 0$$

De même, pour tout nœud n précédant un nœud AND *fermant $\bar{\alpha}$, on obtient :*

$$\neg(M(pSynch(\bar{\alpha}, n)) = 1) \Leftrightarrow M(pSynch(\bar{\alpha}, n)) = 0$$

Enfin, la place finale vérifie également $\neg(M(pFinal) = 1) \Leftrightarrow M(pFinal) = 0$. Il est ainsi possible de simplifier l'expression de la négation des équations 5.4 à 5.6, données p. 110.

Remarque 6.5. *Compte tenu de la remarque 6.4, et d'après les définitions précédentes, on peut montrer que l'exécution du modèle EFFBD arrive à son terme si et seulement si le réseau équivalent contient un jeton dans la place finale* i.e. $\downdownarrows \mathcal{E} \Leftrightarrow M(pFinal) = 1$.

Nous notons $\approx$ la relation liant une propriété EFFBD-TCTL et sa traduction dans la logique TPN-TCTL. En utilisant la relation de bisimulation temporelle existant entre un EFFBD et sa traduction en TPN, la proposition suivante se démontre immédiatement.

Proposition 6.1 *Soient $\mathcal{E}$ un EFFBD bien formé et $\mathcal{T}$ tel que $\mathcal{E} \sim \mathcal{T}$. Soient $\varphi_{\mathcal{E}}$ une propriété EFFBD-TCTL définie sur $\mathcal{E}$ et $\varphi_{\mathcal{T}}$ une propriété TPN-TCTL définie sur $\mathcal{T}$ telle que $\varphi_{\mathcal{E}} \approx \varphi_{\mathcal{T}}$:*

$$\mathcal{E} \vDash \varphi_{\mathcal{E}} \Leftrightarrow \mathcal{T} \vDash \varphi_{\mathcal{T}}$$

Par conséquent, notre démarche générale de vérification formelle des modèles de haut niveau est *correcte*. En particulier, il est possible d'employer un model checker de propriétés TPN-TCTL pour vérifier des propriétés EFFBD-TCTL. Puisque les différentes transformations sont réalisées sans perte d'information sur les comportements, les résultats d'analyse peuvent s'exprimer en termes de nœuds, flux,... ce qui répond à notre première problématique. Cet aspect pratique sera développé dans les sections suivantes ainsi que dans l'annexe A.

6.2.3 Exemples

À partir de l'exemple du passage à niveau, nous pouvons à présent décrire trois propriétés de sûreté et de vivacité :

P_1 « Lorsqu'un train est présent sur le passage à niveau, les barrières sont fermées »

P_2 « Les barrières ne restent pas fermées plus de 90 s[10] »

P_3 « Il s'écoule au moins 25 s entre l'activation de la sonnerie et le passage du train »

Ces propriétés s'écrivent respectivement :

Φ_1 $\forall\Box\ (A(f_{Qui\_PN}) = executing) \Rightarrow (N(fer) = 1)$

Φ_2 $N(fer) = 1 \rightsquigarrow_{[0,90]} N(ouv) = 1$

Φ_3 $\neg\exists\Diamond(A(f_{Act\_so}) = executing \Rightarrow \exists\Diamond_{[0,25[}A(Qui\_PN) = executing)$

et, sur le réseau équivalent :

φ_1 $\forall\Box\ (M(pE(Qui\_PN)) \geq 1) \Rightarrow (M(pFlux(fer)) = 1)$

φ_2 $M(pFlux(fer)) = 1 \rightsquigarrow_{[0,90]} M(pFlux(ouv)) = 1$

φ_3 $\neg\exists\Diamond(M(pE(Act\_so)) = 1 \Rightarrow \exists\Diamond_{[0,25[}M(pE(Qui\_PN)) = 1)$

Remarque 6.6. *Dans le cas général, la transformation $P \rightarrow \Phi$ est loin d'être triviale. Ainsi, une formule telle que Φ_3, quoique ne faisant intervenir que deux fonctions et un délai, est déjà passablement complexe. Dans le cadre de notre démarche d'IS, nous avons ainsi été amenés à proposer dans notre outil logiciel un nombre limité de* classes de propriétés. *Nous donnons le détail de ces classes dans la section 6.3.3.*

Les trois propriétés ci-dessous sont vérifiées. Bien évidemment, la modélisation du système et des propriétés sont suffisamment simples pour effectuer ces vérifications « à la main », sans recourir à la transformation vers les TPNs. Le but de cet exemple est bien d'illustrer le principe de notre démarche, non sa puissance.

Considérons à présent le cas où le passage comporte deux voies telles qu'un seul train circule sur chacune[11]. Le modèle présenté plus haut est modifié en dédoublant la branche `train`[12].

La propriété P_1 n'est plus vérifiée : en effet, si le second train atteint son origine d'annonce lorsque le premier vient de quitter la zone courte, la barrière ne peut encore prendre en compte l'ordre de fermeture puisqu'elle est en train de se lever et que la fonction n'est pas interruptible. En effet, il s'écoule au pire (*i.e.* au plus long) 30 secondes entre le début de la remontée des barrières et leur abaissement complet. Comme le train prend au pire (*i.e.* au plus court) 26 secondes pour arriver sur le PN, la propriété n'est pas vérifiée.

D'un point de vue pratique, le model checker TPN-TCTL de Roméo fournit une trace (non temporisée) mettant en évidence le non respect de la propriété φ_1. De façon similaire, notre outil de vérification intégré aux ateliers Kimono et Omotesc fournit une trace atemporisée « de haut niveau » où ne figurent, pour plus de lisibilité, que les débuts d'exécution des fonctions (cf. annexe A). Pour des raisons de lisibilité, nous ne donnons pas ces traces ici, qui décrivent simplement le passage successif de deux trains, comme esquissé plus haut.

Remarque 6.7. *Le défaut du modèle étendu à deux trains peut être contourné en modifiant*

10. Cette durée comprend le temps de remontée des barrières, de manière à garder en particulier un trafic routier fluide.

11. Cette hypothèse évite d'avoir à concevoir et modéliser un dispositif d'accès au PN en exclusion mutuelle.

12. Ou, de manière équivalente, en encadrant la boucle $\langle \lambda_t, \bar{\lambda}_t \rangle$ par une réplication dont le compteur vaut 2 et la branche de contrôle est vide.

la fonction Relever les barrières *de manière à pouvoir l'interrompre à l'annonce d'un nouveau passage (voir par exemple le modèle donné dans [31]).*

D'une manière plus générale, l'outil de vérification permet d'aiguiller le concepteur sur les défauts de son modèle[13]*, mais l'analyse des résultats et le choix d'une solution aux problèmes mis en évidence restent en majeure partie à la charge de l'ingénieur système.*

6.3 Propriétés et résultats complémentaires

6.3.1 Résultats de décidabilité

De la proposition 6.1 et du théorème 6.1, nous pouvons immédiatement déduire :

Proposition 6.2 *La satisfaisabilité de la logique EFFBD-TCTL est indécidable.*

De même, en utilisant la proposition 5.2, il vient :

Proposition 6.3 *La satisfaisabilité d'EFFBD-TCTL est décidable sur les EFFBDs bornés.*

6.3.2 Détermination de la complexité algorithmique

Dans la suite, nous considérons que la taille d'un EFFBD est donnée par la somme $|\mathcal{N}| + |\mathcal{F}|$. Par ailleurs, et d'après les motifs donnés au chapitre 5, le TPN équivalent a :

- $|P| \leq (1 + |AND_{out}|)|\mathcal{N}| + |FC| + 2|IT_{in}|$ places ;
- $|T| \leq (1 + |OR_{in}| + |FC|)|\mathcal{N}| + |FC| + |IT_{in}|$ transitions.

Ainsi, en notant $|\mathcal{E}|$ et $|\mathcal{T}|$ la taille des deux modèles, il vient : $|\mathcal{T}| = \mathcal{O}(|\mathcal{E}|)$. De même, la transformation d'une formule d'EFFBD-TCTL vers une formule de TPN-TCTL est également polynomiale en la longueur de la formule.

En utilisant le théorème 6.3, nous obtenons :

Proposition 6.4 *Il existe un algorithme de complexité PSPACE décidant EFFBD-TCTL sur les EFFBDs bornés.*

Par ailleurs, il est possible de montrer qu'un problème PSPACE-complet (à savoir l'évaluation d'une *Quantified Boolean Formula* ou QBF) peut se réduire polynomialement à la vérification d'une formule EFFBD-TCTL sur un EFFBD borné :

Proposition 6.5 *Le model checking d'EFFBD-TCTL sur les EFFBDs bornés est PSPACE-difficile.*

Démonstration. Nous adoptons la démarche suivie dans [5] et [31] consistant à ramener le problème de l'évaluation d'une QBF à la vérification d'une propriété EFFBD-TCTL. Ce problème est en effet « canoniquement » PSPACE-complet [33].

Une QBF est une formule de la forme $\varphi = Q_1 x_1 \dots Q_n x_n \, \beta(x_1, \dots, x_n)$ où Q_i est un quantifieur ($Q_i \in \{\forall, \exists\}$) et β est une formule booléenne non quantifiée sur les variables propositionnelles x_i.

13. En supposant qu'il puisse et/ou sache exprimer les propriétés à vérifier, comme mentionné dans la remarque 6.6.

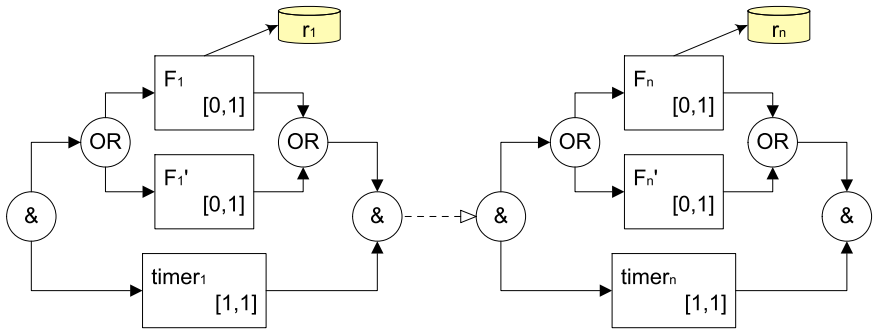


FIGURE 6.1 – Encodage d'une QBF par un EFFBD

Considérons le modèle $\mathcal{E}$ illustré figure 6.1. Il se compose de n motifs élémentaires ; l'exécution de chaque motif prend exactement une unité de temps. Chacune des ressources r_i a un niveau initial nul ; le niveau de la ressource r_i, après exécution du modèle, vaut une unité si et seulement si la fonction F_i a été exécutée, au détriment de la fonction F_i'.

Ainsi, à partir de l'état initial, il existe 2^n chemins atteignant un état final, les ressources permettant de mémoriser le chemin suivi. Dans l'état final, la formule β peut être transformée par une contrainte EFFBD-GMEC γ en remplaçant chaque occurrence de x_i par $N(r_i) = 1$ et $\neg x_i$ par $N(r_i) = 0$.

Par conséquent, $\Downarrow\!\!\Downarrow \mathcal{E} \wedge \gamma$ est vraie si et seulement s'il existe une valuation des variables x_i telle que β est vraie. Considérons alors la formule $\Phi = Q_1 \Diamond_{[1,1]} \dots Q_n \Diamond_{[1,1]} (\Downarrow\!\!\Downarrow \mathcal{E} \wedge \gamma)$. Φ est vraie si et seulement si φ est vraie.

Comme le modèle $\mathcal{E}$ a une taille égale à $8\ n$, nous avons réduit polynomialement un problème notoirement PSPACE-complet au model checking d'une propriété EFFBD-TCTL sur un EFFBD borné. Ce model checking est donc bien PSPACE-difficile. □

Remarque 6.8. *Tout comme dans [31], nous avons choisi pour notre démonstration un motif « explicitement temporisé », mais cela n'était pas nécessaire. Considérons en effet le modèle illustré figure 6.2.*

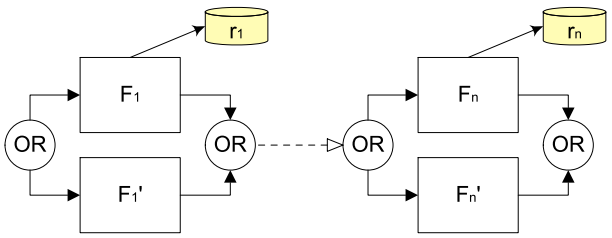


FIGURE 6.2 – Encodage d'une QBF par un EFFBD atemporel

Un raisonnement similaire à ce qui précède montre que la QBF φ peut être encodée par la formule $\Phi' = Q_1 \Diamond \dots Q_n \Diamond (\Downarrow\!\!\Downarrow \mathcal{E} \wedge \gamma)$, toutes les autres définitions étant conservées. Informellement, Φ' est une formule EFFBD-CTL$_{-\bigcirc}$. Le model checking d'EFFBD-CTL$_{-\bigcirc}$ sur les EFFBDs atemporels bornés est donc également PSPACE-difficile.

Puisque les EFFBDs atemporels sont contenus par les EFFBDs temporels et que EFFBD-CTL$_{-\bigcirc}$ est incluse dans EFFBD-TCTL, le model checking d'une formule EFFBD-TCTL sur les EFFBDs (temporels) bornés est également PSPACE-difficile.

Des propositions 6.4 et 6.5 découle immédiatement le résultat suivant :

Proposition 6.6 *Le model checking d'EFFBD-TCTL sur les EFFBDs bornés est PSPACE-complet.*

6.3.3 Propriétés logiques usuelles

Comme évoqué plus haut, les logiques de type TCTL, même adaptées au langage EFFBD, restent délicates à manipuler. C'est pourquoi, dans notre outil logiciel, nous avons choisi de n'offrir que six classes de propriétés logiques, exprimées en language naturel (cf. annexe A).

Ce nombre peut bien sûr paraître limité au regard de la richesse sémantique des différents modèles. Cependant, le choix de ces classes répond au exigences formulées par les cahiers des charges des projets KIMONO et OMOTESC et, plus généralement, aux situations usuellement rencontrés en Ingénierie Système[14]. Bien évidemment, le travail de formalisation du langage EFFBD et de la logique EFFBD-TCTL (ainsi que leur traduction respective vers des modèles de plus bas niveau) ouvre la voie à la définition d'autres classes de propriétés.

Nous détaillons chacune des six classes de propriétés ci-dessous, en précisant leur traduction dans les différentes logiques[15]. Comme précédemment, P est l'expression en langage naturel, Φ la proposition EFFBD-TCTL et φ la proposition TPN-TCTL.

Exécution complète Cette propriété permet de vérifier qu'aucun blocage ou boucle infinie ne peut arriver durant l'exécution d'un modèle $\mathcal{E}$:

P « L'exécution de $\mathcal{E}$ arrive toujours à son terme »
Φ $\forall\Diamond(\downdownarrows\mathcal{E})$
φ $\forall\Diamond(M(pFinal)=1)$

Exécution complète bornée Soit I un intervalle réel à bornes entières de la forme $[a,b]$ ou $[a,\infty[$.

P « L'exécution de $\mathcal{E}$ arrive toujours à son terme et sa durée appartient à $\mathcal{I}$ »
Φ $\forall\Diamond_{\mathcal{I}}(\downdownarrows\mathcal{E})$
φ $\forall\Diamond_{\mathcal{I}}(M(pFinal)=1)$

Exécution complète bornée d'un sous-scénario Soient $x\in\mathbb{N}$ et $\mathcal{E}'$ un sous-scénario attaché au modèle $\mathcal{E}$. Soient n_1 le premier nœud du sous-scénario et $\mathcal{N}'$ l'ensemble des nœuds de $\mathcal{E}'$. De même, soient p_1 la première place de la traduction de $\mathcal{E}'$ et P' l'ensemble des places de cette traduction.

14. Ajoutons que, de par le nombre limité de paramètres laissés à la définition du concepteur, la saisie des propriétés ne nécessite qu'une interface basique.

15. Lorsque cela est possible, nous avons simplifié les formules TPN-TCTL comme précisé dans la remarque 6.4.

P « La durée de l'exécution de $\mathcal{E}'$ appartient toujours à $[0, x]$ »

Φ $A(n_1) = \mathit{enabled} \rightsquigarrow_{[0,x]} \bigwedge_{n' \in \mathcal{N}'} A(n') = \mathit{inactive}$

φ $M(p_1) = 1 \rightsquigarrow_{[0,x]} \sum_{p' \in P'} M(p') = 0$[16]

Exclusion mutuelle Soient $f_1, \ldots, f_n$ des fonctions.

P « À tout instant, une seule fonction f_i, au plus, est en exécution »

Φ $\forall\Box \bigwedge_{i=1}^{n} \left((A(f_i) = \mathit{executing}) \Rightarrow \bigwedge_{j \neq i} A(f_j) \neq \mathit{executing} \right)$

φ $\forall\Box \sum_{i=1}^{n} M(pE(f_i)) \leq 1$

Réponse bornée Soient f et g deux fonctions et $x \in \mathbb{N}$.

P « L'exécution de f entraîne toujours celle de g en moins de x unités de temps »

Φ $A(f) = \mathit{executing} \rightsquigarrow_{[0,x]} A(g) = \mathit{executing}$

φ $M(pE(f)) = 1 \rightsquigarrow_{[0,x]} M(pE(g)) = 1$

Bornitude d'un flux Soient $F \in \mathcal{F}$ et $k \in \mathbb{N}$.

P « Le niveau de F reste toujours inférieur ou égal à x unités »

Φ $\forall\Box(N(F) \leq x)$

φ $\forall\Box(M(pFlux(F)) \leq x)$

16. On suppose ici, pour simplifier, que le sous-scénario ne contient aucune itération ; la propriété s'étend bien sûr à ce cas.

Chapitre 7

Vérification des Modèles Défaillants

Résumé *En réponse à notre première problématique, nous avons défini et construit dans les chapitres précédents un outil de vérification formelle des architectures fonctionnelles. Nous avons alors cherché à étendre ces résultats de manière à analyser des architectures* dysfonctionnelles*, c'est-à-dire pour lesquelles des pannes affectent des éléments du système.*

Afin d'introduire le contexte de cette seconde problématique, nous présentons tout d'abord deux outils usuels en SdF*, les AMDECs et les arbres de défaillances (AdDs). Nous décrivons ensuite la syntaxe et la sémantique formelles des EFFBDs avec défaillances ainsi que l'extension d'EFFBD-TCTL correspondante. Enfin, nous donnons la traduction de ces modèles vers les TPNs et TPN-TCTL et étendons les résultats des chapitres précédents.*

Le chapitre se conclut une dernière fois sur l'application de ces résultats au problème du passage à niveau.

Sommaire

Ce chapitre propose une extension des travaux développés précédemment prenant en compte l'effet de pannes sur le comportement du système. Nous y avons adopté une structure similaire à l'ensemble des chapitres précédents. Ainsi, la section 7.1 précise le contexte général de l'analyse dysfonctionnelle en illustrant deux techniques fondamentales en Ingénierie Système. La section 7.2 donne, de façon informelle puis formelle, la description des EFFBDs comportant des défaillances ainsi que l'extension de la logique EFFBD-TCTL. La section 7.3 étend les résultats des chapitres 5 et 6 au cas des pannes.

7.1 Analyse des architectures dysfonctionnelles en IS

7.1.1 Introduction

Nous donnons dans cette section deux méthodes majeures employées en Ingénierie Système pour la description et l'analyse des architectures dysfonctionnelles. C'est à dessein que nous y consacrons une entière section : en effet, c'est l'étude de ces outils de SdF qui a permis de dégager les cas de pannes applicables aux EFFBDs, présentés dans les sections suivantes [1]. En outre, ces outils et nos travaux sont largement complémentaires. Rappelons ainsi que nous ne cherchons pas à nous substituer aux méthodes dédiées à la sûreté de fonctionnement mais à les compléter en étendant les résultats et les outils que nous avons développés pour l'analyse des architectures fonctionnelles.

Signalons enfin que d'autres méthodes de SdF, non spécifiques à l'IS, sont de plus en plus appliquées à ce domaine. Ainsi, ces dernières années ont vu l'émergence du langage AADL et sa mise en œuvre dans des projets d'Ingénierie Système [29]. En parallèle, des travaux de recherche ont été menés afin de transformer des modèles AADL vers les GSPNs (*Generalized Stochastic* Petri *Nets*) [67]. L'application de la théorie des chaînes de Markov, en particulier, permet alors d'effectuer des analyses de fiabilité quantitatives.

> **Remarque 7.1.** *Les techniques d'analyse des architectures fonctionnelles et dysfonctionnelles sont exactement contemporaines. En effet, les EFFBDs, les AMDECs et les arbres de défaillances ont tous trois été développés ou appliqués (sous une forme éventuellement simplifiée) dans les années 1960, par ou pour des organismes pionniers en IS tels que la NASA et l'US* Air Force.

7.1.2 Analyse des modes de défaillance, de leurs effets et de leur criticité

Présentation générale L'AMDEC [2] est une méthode d'analyse inductive cherchant à identifier les défaillances affectant le système pour ensuite les hiérarchiser selon leur niveau de criticité [42, 84]. Elle consiste à recenser pour chaque constituant ou chaque fonction ses *modes de défaillance* (dont nous donnons quelques exemples ci-dessous), les *causes* et les *effets* possibles de ces défaillances ainsi que leur *criticité* [3]. Une AMDEC est généralement complétée par la description des moyens de *détection* et des *actions* de réduction du risque.

1. Précisons que la plupart des ces études ont été menées par A. Faisandier et T. Renard, de MAP système, dans le cadre du projet Omotesc.
2. Traduit par le terme FMECA *(Failure Modes, Effects and Criticality Analysis)*.
3. La criticité est généralement calculée comme le produit du coefficient de probabilité d'occurrence par le coefficient de gravité.

Le tableau 7.1 illustre, de façon partielle, la méthode appliquée au problème du passage à niveau. Les valeurs de criticité données ici sont arbitraires ; de même, nous ne faisons figurer ni les moyens de détection ni les actions préventives ou correctives.

TABLE 7.1 – AMDEC partielle du passage à niveau

Fonction	Mode de défaillance	Causes	Effets	Criticité
Relever la barrière	Ne se lève pas	Non réception du flux *ouverture PN* Blocage mécanique	Arrêt de la circulation routière	20
	Se lève trop lentement	Moteur sous-alimenté Frottements	Ralentissement du débit routier Détérioration de la barrière	10

Cette méthode, simple et systématique, concerne tous les secteurs d'activité de l'IS[4] ; cependant, sa construction conduit à un volume d'information très important et souvent non homogène. De plus, elle ne met pas en évidence les combinaisons de pannes entraînant la défaillance globale du système. Elle reste néanmoins un outil privilégié de définition des modes de défaillances et de traçabilité des évolutions du système.

> **Remarque 7.2.** *La construction d'une AMDEC est souvent précédée d'une Analyse Préliminaire des Risques (APR ou* Preliminary Hazard Analysis*) qui permet de limiter l'AMDEC aux seuls cas « intéressants » (phases de vie principales, scénarios critiques,...).*

Modes de défaillance On définit en général quatre modes de défaillance fonctionnelle génériques :

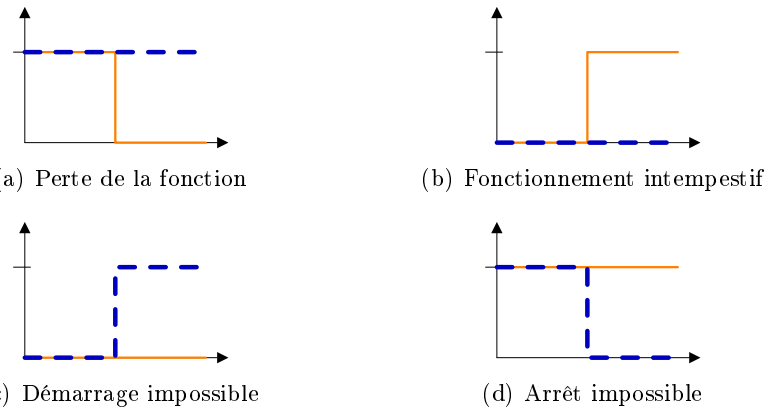
- perte de la fonction (ou *arrêt prématuré*) ;
- fonctionnement intempestif ;
- démarrage impossible ;
- arrêt impossible.

Il existe également un mode dit de *fonctionnement dégradé*. Celui-ci peut représenter un grand nombre de comportements : modification de la durée d'exécution, des quantités de flux échangés, etc.

La figure 7.1 donne une vue schématique de ces modes ; le fonctionnement attendu est représenté par des pointillés sombres et le comportement réel par une ligne continue claire.

> **Remarque 7.3.** *Pour reprendre l'exemple du tableau 7.1, le mode* Ne se lève pas *correspond donc à un* démarrage impossible *et le mode* Se lève trop lentement *à un* fonctionnement dégradé.

4. Dans le domaine logiciel, cependant, on applique plus volontiers l'Analyse des Effets des Erreurs du Logiciel (AEEL).



(a) Perte de la fonction (b) Fonctionnement intempestif

(c) Démarrage impossible (d) Arrêt impossible

FIGURE 7.1 – Modes génériques de défaillance fonctionnelle

Nous verrons que, dans la définition des cas de panne donnée section 7.2.1, nous avons cherché à respecter ces différents modes génériques.

7.1.3 Arbres de défaillances

Présentation générale Les AMDECs ne permettant pas d'analyser les combinaisons de défaillances, elles sont souvent complétées par les arbres de défaillances (AdDs) ou *Fault Trees* [87]. Ceux-ci proposent une analyse déductive des causes provoquant les *événements redoutés*[5].

Un AdD est représenté par un arbre dont la racine est *l'événement redouté* et les feuilles les *événements élémentaires* ; ceux-ci sont reliés par les opérateurs booléens classiques[6]. En reprenant l'exemple du passage à niveau, la figure 7.2 illustre, de manière très partielle, la combinaison des événements élémentaires e_i conduisant à l'événement redouté E.

L'analyse des *coupes minimales (cutsets)* met alors en évidence les ensembles de causes conduisant à la défaillance du système. Enfin, l'adjonction de probabilités de défaillance permet de mener des analyses quantitatives de fiabilité.

> **Remarque 7.4.** *Il est possible de dériver un* AdD *d'un EFFBD ; un tel outil a d'ailleurs été implémenté et intégré à l'atelier* Omotesc.

Prise en compte du temps Par défaut, un AdD ne permet pas de représenter la séquentialité des événements conduisant à une défaillance. Plusieurs extensions ont alors été proposées pour combler ce manque. En particulier, les auteurs de [69] ont formalisé et étendu la sémantique des AdDs de manière à prendre en compte non seulement les relations booléennes entre les événements mais également les relations de causalité. Cette sémantique s'applique donc à la description de systèmes dynamiques.

Ces résultats ont été développés dans [81] où la sémantique des AdDs est exprimée dans la logique CTL. L'emploi d'un model checker de CTL permet alors de vérifier la complétude

5. Dans le contexte des AdDs, un *événement* n'est pas nécessairement instantané.
6. La plupart des définitions introduisent également des portes de vote n/m.

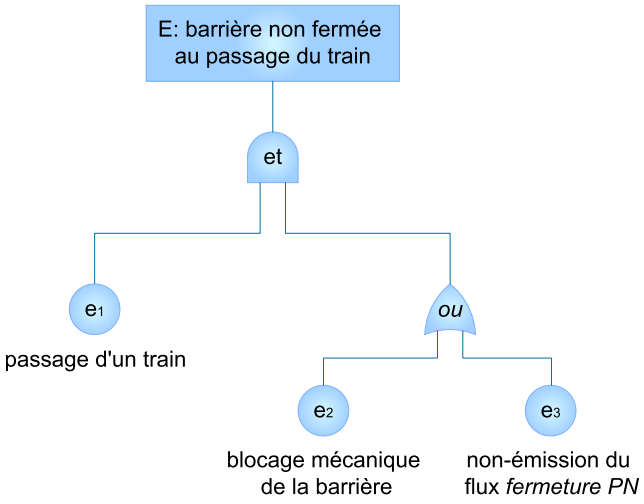


FIGURE 7.2 – Exemple d'arbre de défaillances

du modèle ainsi que des propriétés de sûreté et de fiabilité. Enfin, la prise en compte de l'écoulement quantifié du temps se fait par le biais d'observateurs.

> **Remarque 7.5.** *En réalité, ces observateurs ne permettent de décrire que trois types d'événements temporels. La complétude de ces cas par rapport à la théorie des* AtDDs *n'est pas précisée.*

7.2 Syntaxe et sémantique des EFFBDs avec défaillances

7.2.1 Description informelle des défaillances et de leurs effets

Les défaillances affectent des fonctions ou des flux ; elles se subdivisent en sept catégories modifiant chacune une règle sémantique de l'élément concerné. Les pannes définies sur les fonctions sont :

(*i*) *non activation* : lorsque le flux de contrôle atteint la fonction, elle n'est pas activée (ni, *a fortiori*, exécutée) et le flux de contrôle n'est pas transmis à son successeur ;

(*ii*) *durée infinie* : une fois son exécution démarrée, la fonction ne peut s'arrêter (à moins d'être tuée par une structure de terminaison) ;

(*iii*) *durée modifiée* : l'exécution de la fonction démarre et se termine normalement mais sa durée appartient à un nouvel intervalle $[a', b']$;

(*iv*) *non production des flux de sortie* : à l'issue de l'exécution, la fonction ne produit pas ses flux de sortie mais passe le contrôle à la structure suivante ;

(*v*) *non transfert du contrôle* : à la fin de son exécution, la fonction produit ses flux se sortie mais ne transfère pas le contrôle à son successeur.

Une panne de type (*i*) correspond au mode de défaillance *démarrage impossible* ; le type (*ii*) à un *arrêt impossible*. Le type (*iii*) peut être vu comme une *perte de la fonction* (si la durée

d'exécution effective est inférieure à la durée minimale théorique a) ou, plus généralement, à un *fonctionnement dégradé*, tout comme la panne (iv). Enfin, la panne de type (v) peut être vue comme un *démarrage impossible* des successeurs de la fonction touchée.

Seules les pannes de type (iii), (iv) et (v) peuvent affecter simultanément une même fonction puisqu'une panne de type (i) (respectivement (ii)) masque toutes les autres pannes (respectivement celles de type (iii) à (v)). On supposera en outre qu'une seule panne de chaque type, au plus, peut toucher une même fonction. Enfin, on considère que les pannes n'affectent pas les fonctions décomposées, puisqu'il suffit de les reporter sur les fonctions du sous-scénario.

> **Remarque 7.6.** *Plus généralement, on peut distinguer les pannes* permanentes *des pannes* fugaces*, ces dernières pouvant ne pas s'exprimer à chaque exécution de la fonction. On suppose dans la suite que toutes les pannes sont permanentes ; la fugacité des pannes peut être modélisée comme suit.*
>
> *Soit F une fonction affectée d'une panne permanente P et d'une panne fugace Q. Le comportement de F peut être modélisé par une structure de choix à deux branches. La première porte la fonction F_P, affectée de la seule panne P. La seconde porte la fonction $F_{P,Q}$, affectée des pannes (permanentes) P et Q. Les trois fonctions ont bien évidemment la même durée et les mêmes flux d'entrée et de sortie.*
>
> *Nous ne donnons pas ici les règles de dépliage des pannes, qui s'étendent au cas multi-sortie, celles-ci étant suffisamment intuitives*[7].

Les pannes affectant les flux sont les suivantes :

(vi) *niveau initial modifié :* la quantité initiale du flux est donnée par un nouvel entier ;

(vii) *quantité produite modifiée :* la quantité de flux produite par la fonction est donnée par un nouvel entier[8].

Ces deux derniers cas de panne permettent de modéliser des défaillances sur les liens entre les fonctions (perte d'information, saturation ou rémanence,...) mais également une forme de *déclenchement intempestif* des fonctions. Considérons ainsi le modèle illustré figure 7.3 (les défaillances ne faisant pas partie du formalisme EFFBD originel, les notations ne sont pas normalisées). Les trois fonctions ont une durée d'exécution unitaire ; l'item A, qui devrait avoir un niveau initial de 0, est affecté d'une panne de type (vi) telle que $F_0(A) = 1$.

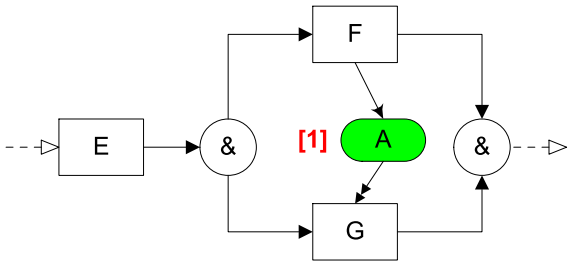
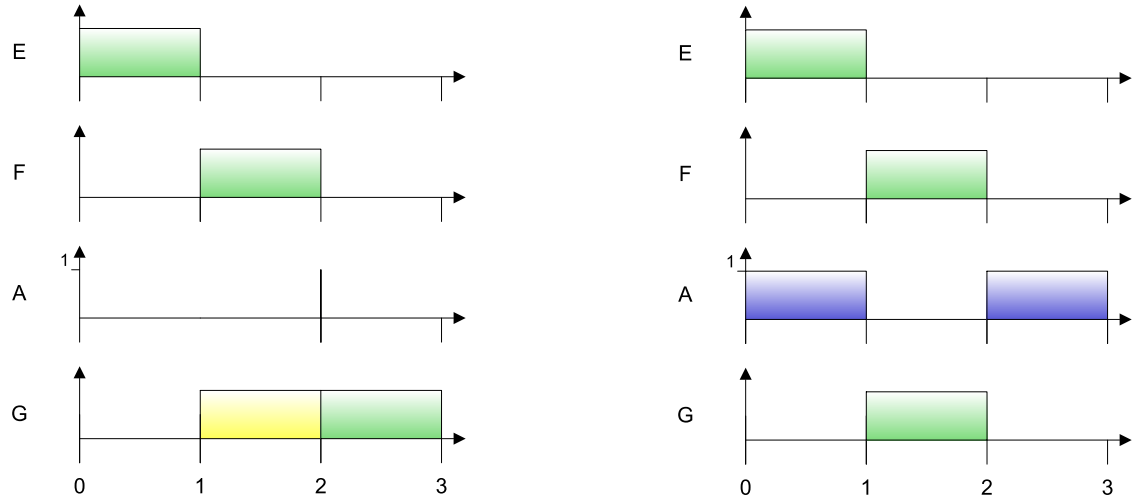


FIGURE 7.3 – Modèle comportant une panne sur le niveau initial d'un flux

7. Précisons cependant que, dans ce cas uniquement, les éventuels items d'entrée ne sont pas dépliés.

8. Cette panne, qui peut être vue comme un raffinement de la panne (iv), affecte donc un couple fonction-flux ; de façon similaire, on pourrait définir des pannes sur les quantités consommées ou lues.

Les résultats sont illustrés figure 7.4. L'item A étant présent dès le début, la fonction G n'est plus bloquée en attente de F et son déclenchement à la date $t = 1$ est donc bien intempestif par rapport au cas non défaillant.



(a) Comportement sans panne (b) Comportement avec panne

FIGURE 7.4 – Effet d'une panne définie sur le niveau initial d'un flux

Remarque 7.7. *La fonction G est nécessairement activée avant d'être déclenchée (instantanément) par A et le comportement du modèle vérifie encore la propriété de non réentrance. Plus généralement, on notera que l'action de toutes les pannes est synchrone avec les exécutions des fonctions du système.*

7.2.2 Syntaxe et sémantique formelles

Définitions préliminaires Dans la suite de ce chapitre, nous adoptons les notations et définitions suivantes :

- $\mathbb{D} = \{noAct; infDur; modDur; noOutput; noControl\}$ est *l'ensemble des types de pannes* affectant les fonctions ;
- $\mathcal{D}_{FC} \subseteq FC \times \mathbb{D}$ est *l'ensemble des pannes* affectant les fonctions de FC ;
- $\mathcal{D}_{noAct}$ est l'ensemble des fonctions ayant une panne de type (i) (non activation) ; il est défini par :
$$\mathcal{D}_{noAct} = \{f \in FC / (f, noAct) \in \mathcal{D}_{FC}\}$$
- de façon similaire, les ensembles $\mathcal{D}_{infDur}$, $\mathcal{D}_{modDur}$, $\mathcal{D}_{noOutput}$ et $\mathcal{D}_{noControl}$ définissent les ensembles des fonctions affectées d'une panne de type (ii) à (v) ;
- $\mathcal{D}_{F_0} \subseteq \mathcal{F}$ est l'ensemble des flux de $\mathcal{F}$ affectés d'une panne de type (vi) ;
- $\mathcal{D}_{Prod} \subseteq FC \times \mathcal{F}$ est l'ensemble des couples fonction-flux affectés d'une panne de type (vii).

Remarque 7.8. *Soient f une fonction et φ un flux. $(f, \varphi) \in \mathcal{D}_{Prod}$ n'a de sens que s'il existe $k \in \mathbb{N}_{>0}$ tel que $(f, k, \varphi) \in \mathcal{A}_{\mathcal{F},P}$, c'est-à-dire si φ est effectivement produit par f.*

Syntaxe des EFFBDs comportant des pannes La définition d'un EFFBD avec pannes (ce que désignera le terme d'EFFBD dans la suite) s'écrit alors :

Définition 7.1 (EFFBD avec pannes) *Un EFFBD avec pannes est un n-uplet* $\mathcal{E}_{\mathcal{D}} = (\mathcal{E}, \mathcal{D}_{FC}, a', b', \mathcal{D}_{F_0}, F'_0, \mathcal{D}_{Prod}, \mathcal{A}'_{\mathcal{F},P})$ *où :*

- $\mathcal{E}$ *est un EFFBD sans panne ;*
- $\mathcal{D}_{FC}$,$\mathcal{D}_{F_0}$ *et* $\mathcal{D}_{Prod}$ *sont définis comme ci-dessus ;*
- $a' \in \mathbb{N}^{\mathcal{D}_{modDur}}$ *et* $b' \in (\mathbb{N} \cup \{\infty\})^{\mathcal{D}_{modDur}}$ *donnent respectivement les durées minimale et maximale de l'exécution des fonctions dont la durée est modifiée par une panne de type* (iii) *;*
- $F'_0 \in \mathbb{N}^{\mathcal{D}_{F_0}}$ *donne la nouvelle valeur initiale des flux impactés par une panne de type* (vi) *;*
- $\mathcal{A}'_{\mathcal{F},P} \subseteq FC \times \mathbb{N} \times \mathcal{F}$ *est l'ensemble des arcs de production donnant pour chaque couple de* $\mathcal{D}_{Prod}$ *la nouvelle quantité produite.*

Remarque 7.9. *Puisque toutes les pannes sont permanentes, nous aurions pu directement intégrer l'effet des pannes dans la définition de l'EFFBD (*i.e. *fusionner les fonctions a et a', par exemple). Ce faisant, nous aurions perdu la traçabilité des pannes, pour une complexité à peine plus faible. En outre, la définition des pannes comme une « sur-couche » ménage plus facilement la possiblité d'étentre les cas de pannes et leurs effets.*

Définitions complémentaires Les définitions des précédesseurs, successeurs, relations de consommation et de lecture, proposées au chapitre 4 restent valables ; la relation de production est étendue comme suit. Soient $f \in FC$ et $\varphi \in \mathcal{F}$:

- si $f \in \mathcal{D}_{noOutput}$, alors $Prod(f)[\varphi] = 0$;
- sinon :
  - s'il existe k' tel que $(f, k', \varphi) \in \mathcal{A}'_{\mathcal{F},P}$, alors $Prod(f)[\varphi] = k'$;
  - sinon :
    - s'il existe k tel que $(f, k, \varphi) \in \mathcal{A}_{\mathcal{F},P}$, alors $Prod(f)[\varphi] = k$;
    - sinon, $Prod(f)[\varphi] = 0$.

Définition d'un état Une panne de type (ii) ou (iii), affectant la durée de la fonction, se manifeste exactement pendant que celle-ci est dans l'état *executing*. De même, une panne de type (iv), (v) ou (vii) se produit lorsque la fonction touchée passe de l'activité *executing* à *inactive* (ou *executed*). Enfin, les pannes de type (vi) se manifestent uniquement dans l'état initial.

Ainsi, seules les pannes de type *non activation* (i) nécessitent la définition d'une nouvelle activité, $failed$. Nous notons alors $\mathbb{A}^+ = \mathbb{A} \cup \{failed\}$.

L'état d'un EFFBD est de nouveau défini comme un quadruplet $s = (A, C, N, \nu)$ représentant l'activité des nœuds, les compteurs d'itération, le niveau des flux et les valuations des

fonctions[9]. L'état initial s'écrit $s_0 = (A_0, \vec{0}, \varphi_0, \mathbf{0})$ où le nouveau niveau initial est défini par :

$$\forall \varphi \in \mathcal{F},\ \varphi_0(\varphi) = \begin{cases} F'_0(\varphi) & si\ \varphi \in \mathcal{D}_{F_0} \\ F_0(\varphi) & sinon \end{cases}$$

Sémantique formelle La sémantique formelle des EFFBDs avec pannes est fortement inspirée de la définition 4.20 ; on notera que le temps ne s'écoule plus pour les fonctions dont l'exécution est infinie et qu'elles ne peuvent quitter l'état *executing* (à moins d'être tuées).

Définition 7.2 (Sémantique d'un EFFBD avec pannes) *La sémantique d'un EFFBD $\mathcal{E}_\mathcal{D}$ est un quadruplet $||\mathcal{E}_\mathcal{D}|| = (S, s_0, \mathcal{N}, \rightarrow)$ où :*

- *$S \subseteq (\mathbb{A}^+)^\mathcal{N} \times \mathbb{N}^{IT_{in}} \times \mathbb{N}^\mathcal{F} \times (\mathbb{R}_{\geq 0})^{FC}$;*
- *s_0 est défini comme ci-dessus ;*
- *$\rightarrow \subseteq S \times (\mathcal{N} \cup \mathbb{R}_{\geq 0}) \times S$, la relation de transition, est composée :*
  - *de la relation de transition discrète $\xrightarrow{n \in \mathcal{N}} \subseteq S \times \mathcal{N} \times S$ définie par :*

$$(A, C, N, \nu) \xrightarrow{n} (A', C', N', \nu') \Leftrightarrow \begin{cases} \begin{cases} A(n) = enabled \wedge NodeBehavior^\mathcal{D} \\ \text{ou } A(n) = executing \wedge \begin{cases} \mathcal{P}_{\tilde{f}}^{\kappa\mathcal{D}} & \text{si } \kappa(n) \\ \mathcal{P}_{\tilde{f}_{\prec}}^{\mathcal{D}} & \text{si } n \in FC_{MS} \\ \mathcal{P}_{\tilde{f}}^{\mathcal{D}} & \text{sinon} \end{cases} \end{cases} \\ A'(n) = NextActivity \end{cases}$$

  - *de la relation de transition continue $\xrightarrow{\delta \in \mathbb{R}_{\geq 0}} \subseteq S \times \mathbb{R}_{\geq 0} \times S$ définie par :*

$$(A, C, N, \nu) \xrightarrow{\delta} (A', C', N', \nu') \Leftrightarrow$$
$$\begin{cases} \forall n \notin FC,\ A(n) \neq enabled \\ \forall n \in FC,\ (A(n) = enabled) \Rightarrow (N < \max(Cons(n), Lect(n))) \\ \forall n \in FC,\ (A(n) = executing) \Rightarrow \begin{cases} \nu(n) + \delta \leq b(n) & \text{si } n \notin \mathcal{D}_{infDur} \cup \mathcal{D}_{modDur} \\ \nu(n) + \delta \leq b'(n) & \text{si } n \in \mathcal{D}_{modDur} \end{cases} \\ \forall f \in FC,\ \nu'(f) = \begin{cases} \nu(f) + \delta & si\ n \notin \mathcal{D}_{infDur} \\ 0 & sinon \end{cases} \end{cases}$$

$NodeBehavior^\mathcal{D}$ dérive de $NodeBehavior^\kappa$ en ajoutant à chaque proposition $\mathcal{P}_x$ la description des pannes de type (i), comme illustré dans le cas générique $\mathcal{P}_*^\mathcal{D}$:

$$\mathcal{P}_*^\mathcal{D} = \begin{cases} \forall n' \in \mathcal{N}_{-n},\ A'(n') = \begin{cases} enabled & \text{si } n' \in Post(n) \wedge n' \notin \mathcal{D}_{noAct} \\ failed & \text{si } n' \in Post(n) \wedge n' \in \mathcal{D}_{noAct} \\ A(n') & \text{sinon} \end{cases} \\ C' = C \\ N' = N \\ \nu' = \nu \end{cases}$$

9. Il est possible d'étendre cette définition de façon à représenter non seulement les valeurs effectives des durées et des niveaux mais également les valeurs théoriques (*i.e.* hors panne). C'est d'ailleurs ce que l'outil de simulation que nous avons implémenté propose (cf. annexe A).

La condition $\mathcal{P}_{\bar{f}}^{\mathcal{D}}$ s'écrit :

$$\mathcal{P}_{\bar{f}}^{\mathcal{D}} = \begin{cases} n \notin \mathcal{D}_{infDur} \\ \forall n' \in \mathcal{N}_{-n},\ A'(n') = \begin{cases} enabled & \text{si } n' \in Post(n) \wedge n' \notin \mathcal{D}_{noAct} \\ failed & \text{si } n' \in Post(n) \wedge n' \in \mathcal{D}_{noAct} \\ A(n') & \text{sinon} \end{cases} \\ C' = C \\ N' = N + Prod(n) \\ \begin{cases} a(n) \leq \nu(n) \leq b(n) & \text{si } n \notin \mathcal{D}_{modDur} \\ a'(n) \leq \nu(n) \leq b'(n) & \text{sinon} \end{cases} \\ \nu' = \nu \end{cases}$$

Les conditions $\mathcal{P}_{\bar{f}}^{\kappa\mathcal{D}}$ et $\mathcal{P}_{\bar{f}\prec\lambda}^{\mathcal{D}}$ sont modifiées de façon similaire.

7.2.3 Extension de la logique EFFBD-TCTL

Les définitions 6.6 et 6.7 s'étendent immédiatement au cas des pannes en remplaçant $\mathbb{A}$ par $\mathbb{A}^+$ dans l'écriture des EFFBD-GMECs. À la suite des éléments présentés dans la section 6.3.3, il est alors possible de définir de nouvelles classes de propriétés usuelles [10]. Soient f, $f_1 \ldots f_n$ et g des fonctions (f et les fonctions f_i pouvant tomber en panne) et x un entier :

P_1 « f ne tombe jamais en panne »

P_2 « f ne reste jamais en panne plus de x unités de temps d'affilée »

P_3 « La panne de f déclenche l'exécution de g en moins de x unités de temps »

P_4 « À tout instant, une seule fonction f_i, au plus, est en panne »

Ces propriétés s'écrivent respectivement :

Φ_1 $\forall\Box A(f) \neq failed$

Φ_2 $A(f) = failed \rightsquigarrow_{[0,x]} A(f) \neq failed$

Φ_3 $A(f) = failed \rightsquigarrow_{[0,x]} A(g) \neq executing$

Φ_4 $\forall\Box \bigwedge_{i=1}^{n} \left((A(f_i) = failed) \Rightarrow \bigwedge_{j \neq i} A(f_j) \neq failed \right)$

7.3 Traduction vers les TPNs et TPN-TCTL

Nous étendons dans cette section la plupart des résultats présentés dans les deux chapitres précédents.

10. Dans la suite, toutes les pannes sont implicitement de type (i).

7.3.1 Évolution des motifs de traduction

Les nouveaux motifs de traduction vers les TPNs sont donnés ci-dessous. Pour faciliter la lecture, nous ne donnons pas la description formelle des règles de connexion entre les motifs (introduites dans la section 5.2.2).

Non activation Le motif d'une fonction f affectée d'une panne de type (i) est composé d'une simple place $pFail(f)$. Elle forme le point d'entrée du motif, qui n'a donc pas de transition de sortie. Si f est contenue par une structure de terminaison, un arc de vidange relie $pFail(f)$ à la transition idoine.

Durée infinie La figure 7.5 illustre le motif d'une fonction f comportant une panne de type (ii). Ce motif dérive de celui illustré figure 5.8 (p. 102) par ajout de la place $p\infty(f)$, toujours vide. Cette place inhibe la transition $\downarrow f$, maintenant ainsi le jeton dans $pE(f)$ et traduisant l'exécution infinie de la fonction. Le motif s'étend bien sûr au cas multi-sortie.

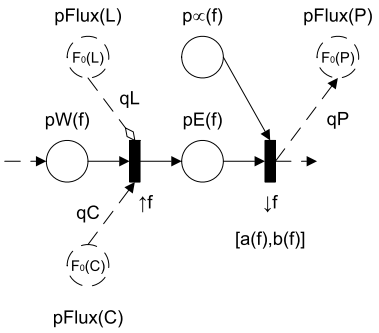


FIGURE 7.5 – Motif d'une fonction f affectée d'une panne de durée infinie

Remarque 7.10. *La transition $\downarrow f$ n'étant jamais sensibilisée, sa valuation reste nulle, ce qui est conforme à la règle sémantique selon laquelle le temps ne s'écoule plus pour la fonction en panne.*

Durée modifiée Le motif de la figure 5.8 est adapté en remplaçant l'intervalle de la transition $\downarrow f$ par $[a'(f), b'(f)]$.

Non production des flux de sortie Les arcs reliant la transition $\downarrow f$ aux places modélisant les flux de sortie sont supprimés, modifiant ainsi le calcul des arcs d'aval donnés p. 104.

Non transfert du contrôle L'arc reliant la transition $\downarrow f$ à la place d'entrée du motif suivant est supprimé dans le calcul des arcs d'aval de connexion.

Niveau initial modifié Soit φ un flux atteint d'une panne de type (vi). Le marquage initial de la place $pFlux(\varphi)$ vaut $F'_0(\varphi)$.

Quantité produite modifiée Soient $(f, \varphi) \in \mathcal{D}_{Prod}$ et k' tel que $(f, k', \varphi) \in \mathcal{A}'_{\mathcal{F},P}$. Le poids de l'arc reliant $\downarrow f$ à $pFlux(\varphi)$ vaut k'.

7.3.2 Propriétés

Propriétés structurelles des EFFBDs Comme nous l'avons vu dans la section 7.2.1, les modèles avec pannes vérifient de nouveau la propriété de non réentrance, pour peu qu'ils soient bien sûr bien formés. En effet, les pannes ne font pas « apparaître » des flux de contrôle mais les font éventuellement disparaître. La conséquence sur le caractère borné des EFFBDs avec pannes est immédiate.

Proposition 7.1 *Soit $\mathcal{E}_\mathcal{D}$ un EFFBD comportant des éléments en panne. Si l'EFFBD sous-jacent (*i.e. *sans panne) est borné, alors $\mathcal{E}_\mathcal{D}$ est borné.*

La réciproque est bien sûr fausse.

Propriétés de la traduction vers les TPNs La définition de la relation $\sim$, donnée dans la section 5.3.1 est étendue de manière à prendre en compte l'activité $failed$, en ajoutant l'équivalence suivante à la définition 5.5 :

$$\forall f \in \mathcal{D}_{noAct} : \ A(n) = failed \Leftrightarrow M(pFail(f)) = 1$$

La propriété suivante se démontre alors immédiatement :

Proposition 7.2 *La relation $\sim$ étendue aux pannes est une bisimulation temporelle forte.*

Propriétés du model checking d'EFFBD-TCTL étendu aux pannes La définition de la relation $\approx$ étendue aux pannes se déduit immédiatement de ce qui précède. Il vient alors la proposition suivante :

Proposition 7.3 *Soient $\mathcal{E}_\mathcal{D}$ un EFFBD et $\mathcal{T}_\mathcal{D}$ sa traduction en TPN tel que $\mathcal{E}_\mathcal{D} \sim \mathcal{T}_\mathcal{D}$. Soient $\varphi_{\mathcal{E}_\mathcal{D}}$ une propriété EFFBD-TCTL définie sur $\mathcal{E}_\mathcal{D}$ et $\varphi_{\mathcal{T}_\mathcal{D}}$ une propriété TPN-TCTL définie sur $\mathcal{T}_\mathcal{D}$ telle que $\varphi_{\mathcal{E}_\mathcal{D}} \approx \varphi_{\mathcal{T}_\mathcal{D}}$.*

$$\mathcal{E}_\mathcal{D} \vDash \varphi_{\mathcal{E}_\mathcal{D}} \Leftrightarrow \mathcal{T}_\mathcal{D} \vDash \varphi_{\mathcal{T}_\mathcal{D}}$$

En d'autres termes, notre démarche générale de vérification formelle des modèles EFFBDs étendus aux pannes est correcte, ce qui répond à notre seconde problématique.

Enfin, les résultats de décidabilité et de complexité établis dans les sections 6.3.1 et 6.3.2 sont encore valides : le model checking de la logique EFFBD-TCTL étendue aux pannes est décidable sur les EFFBDs bornés et le problème est PSPACE-complet.

7.3.3 Application au problème du passage à niveau

Description et modélisation des pannes Nous définissons deux pannes sur le modèle du passage à niveau :

- le train peut rester bloqué sur le passage à niveau (D_1) ;
- suite à une mauvaise modélisation mécanique, la barrière met toujours 30% de temps de plus que prévu pour s'abaisser (D_2).

D_1 se traduit par une panne fugace de durée infinie sur la fonction *Quitter le PN*. Cette fonction est alors dédoublée en une nouvelle fonction, notée f'_{Qui_PN} qui porte la panne permanente $(f'_{Qui\_PN}, infDur) \in \mathcal{D}_{FC}$.

D_2 s'écrit quant à elle $(f_{Ab\_bar}, modDur) \in \mathcal{D}_{FC}$ avec $a'(f_{Ab\_bar}) = 7$ et $b'(f_{Ab\_bar}) = 13$.

> **Remarque 7.11.** *L'intérêt principal de définir des pannes de type (i) est de concevoir en parallèle des mécanismes de confinement ou de recouvrement des fautes. Ces notions, quoique particulièrement intéressantes, sont en dehors de notre champ de réflexion actuel. C'est pourquoi nous n'illustrons pas ici le model checking de la logique EFFBD-TCTL étendue aux pannes.*

Description et vérification des propriétés Considérons de nouveau les deux propriétés de sûreté et de vivacité suivantes :

P_1 « Lorsqu'un train est présent sur le passage à niveau, les barrières sont fermées »

P_2 « Les barrières ne restent pas fermées plus de 90 s »

Compte tenu de la fugacité de (D_1), ces propriétés s'écrivent respectivement :

Φ_1 $\forall\Box\ (A(f_{Qui\_PN}) = executing \vee A(f'_{Qui\_PN}) = executing) \Rightarrow (N(fer) = 1)$

Φ_2 $N(fer) = 1 \rightsquigarrow_{[0,90]} N(ouv) = 1$

et, sur le réseau équivalent :

φ_1 $\forall\Box\ (M(pE(f_{Qui\_PN})) + M(pE(f'_{Qui\_PN})) \geq 1) \Rightarrow (M(pFlux(fer)) = 1)$

φ_2 $M(pFlux(fer)) = 1 \rightsquigarrow_{[0,90]} M(pFlux(ouv)) = 1$

La propriété P_1 est de nouveau vérifiée : le comportement de D_1 la vérifie et les paramètres temporels du modèle sont assez robustes pour autoriser une imprécision de 30 % sur la durée de la descente. En revanche, puisque l'occurrence de D_1 bloque le modèle avec la barrière abaissée, la propriété P_2 n'est pas vérifiée.

Chapitre 8

Conclusion : Bilan et Perspectives

8.1 Bilan des travaux

Après plus d'un demi-siècle d'existence, l'Ingénierie Système s'impose aujourd'hui comme une méthodologie essentielle de conception et de réalisation des systèmes complexes. L'un de ses défis majeurs reste la poursuite et la maîtrise de la sûreté de fonctionnement. Dans ce contexte, le recours à des méthodes formelles telles que le model checking de propriétés logiques apparaît prometteur. Cependant, la complexité (au sens commun mais aussi algorithmique) de cette technique limite généralement sa mise en œuvre de façon efficace dans les projets d'IS.

Il est ainsi apparu nécessaire de concevoir et de développer un outil de vérification formelle d'architectures fonctionnelles qui reste utilisable dans un contexte d'Ingénierie Système, démarche interdisciplinaire par excellence. Le processus retenu comprend quatre étapes successives :

- modélisation du système et des propriétés à vérifier selon des formalismes dits de *haut niveau* ;
- transformation de ces modèles Σ et Φ vers des équivalents de plus bas niveau σ et φ ;
- vérification des modèles bas niveau au moyen de techniques et d'outils spécifiques ;
- présentation des résultats en termes d'éléments du modèle Σ.

Nous avons retenu les EFFBDs comme modèle d'entrée : en effet, ces diagrammes sont à la fois génériques et suffisamment expressifs pour représenter les systèmes considérés ici. Par ailleurs, nous avons choisi les réseaux de Petri temporels comme modèle de bas niveau. Leur structure syntaxique et sémantique (dont la représentation naturelle du parallélisme) ainsi que les nombreux travaux menés sur ce modèle le rend en effet particulièrement adapté à notre démarche. En ce qui concerne les propriétés à vérifier, elles sont exprimées par la logique TCTL adaptée respectivement aux EFFBDs et aux TPNs.

Dans un premier temps, nous avons proposé une sémantique des EFFBDs sous la forme d'un système de transitions temporisé. À notre connaissance, il s'agit là de la première démarche de formalisation de ces modèles. En effet, bien qu'implémentés dans plusieurs ateliers de conception, ce qui assurait une sémantique opérationnelle *de facto*, aucune syntaxe ou sémantique formelle n'en avait été donnée. Cette première formalisation nous a permis de définir la classe des *EFFBDs bornés*, sur lesquels nous avons pu établir certains résultats théoriques.

À partir de ces résultats, nous avons pu définir une transformation structurelle des EFFBDs vers les TPNs. Nous avons montré que cette transformation préserve le comportement temporel des modèles en prouvant qu'il existe une relation de bisimulation temporelle forte entre un modèle EFFBD et sa traduction. Ce résultat fondamental a permis de montrer, en particulier, que la traduction d'un EFFBD borné conduit à l'obtention d'un TPN borné.

En parallèle, nous avons défini la logique EFFBD-TCTL à partir de TCTL ; les propositions atomiques représentent des contraintes d'exclusion mutuelle sur les états du modèle (activité des nœuds, niveau des flux,...). Nous avons alors pu définir sa traduction vers la logique TPN-TCTL, son équivalent sur les TPNs. En d'autres termes, nous avons montré qu'il était équivalent de vérifier la propriété Φ sur le modèle Σ et φ sur σ, justifiant ainsi notre démarche générale. Par ailleurs, nous avons établi que le model checking de la logique EFFBD-TCTL sur les EFFBDs bornés est un problème décidable et PSPACE-complet.

Enfin, nous avons étendu ces différents résultats aux architectures dysfonctionnelles. Nous avons ainsi donné la définition et la sémantique des cas de pannes pouvant affecter les fonctions et les flux d'un modèle EFFBD. Nous avons également étendu la logique EFFBD-TCTL de manière à pouvoir exprimer de nouvelles propriétés « dysfonctionnelles ». Les principaux résultats de bornitude, décidabilité et complexité établis sur les EFFBDs sans défaillance ont ensuite pu être adaptés au cas dysfonctionnel.

D'un point de vue pratique, ces travaux ont donné lieu à l'implémentation au sein d'un atelier de conception d'un outil de simulation et de vérification des EFFBDs étendus aux pannes [1]. Le module de simulation fait appel à un moteur écrit à partir de notre définition de la sémantique des EFFBDs alors que la vérification utilise le model checker de ROMÉO *via* les traductions définies plus haut.

Afin de cacher la complexité de la transformation à l'utilisateur, celui-ci ne manipule que des éléments de haut niveau (fonctions, flux, etc.), ce qui ne nécessite donc pas la connaissance du modèle TPN sous-jacent. Cet outil, développé dans le cadre de deux programmes d'étude amont (PEAs) successifs pour la DGA sera opérationnel et développé industriellement à la fin de l'année 2009.

> **Remarque 8.1.** *Les outils* KIMONO *et* OMOTESC *sont encore en phase de développement, de déploiement ou d'évaluation. Le retour d'expérience s'est pour l'instant limité à des discussions itératives avec l'équipe-projet et les architectes système de la DGA afin, notamment, de définir les classes de propriétés à vérifier ou les types de pannes à modéliser.*

1. Dans le cas dysfonctionnel, seul le module de simulation est actuellement opérationel.

8.2 Perspectives

Un certain nombre de pistes de réflexion se sont dessinées à l'issue de ce travail de thèse. À plusieurs reprises au cours de ce manuscrit, nous avons en particulier dû limiter notre champ d'étude. La prise en compte des *time-outs* ou de la consommation des flux en pré-réservation, par exemple, font ainsi partie des éléments esquissés dans ce document et qui mériteraient d'être développés. De même, il serait intéressant d'étendre et de compléter les cas de pannes proposés ainsi que les propriétés-types définies dans l'outil de vérification. Pour cela, nous comptons notamment sur le retour d'expérience des utilisateurs des plate-formes KIMONO et OMOTESC.

Par ailleurs, la formalisation des EFFBDs ouvre la voie à leur transformation vers d'autres modèles formels dans l'optique de mener des analyses complémentaires. Ainsi, en complétant la description des pannes par des informations stochastiques, il serait envisageable de définir une transformation vers les GSPNs *(Generalized Stochastic* PETRI *Nets)* afin de réaliser des études de fiabilité quantitatives. De façon similaire, la sémantique des EFFBDs pourrait être étendue de façon à prendre en compte des transferts continus de flux, tout au long de l'exécution d'une fonction, par exemple. L'analyse de ces modèles passerait alors par une traduction vers un langage tel que les automates hybrides [37].

Enfin, les définitions formelles des EFFBDs et de la logique EFFBD-TCTL ménagent la possibilité de construire un model checker directement basé sur les EFFBDs, ce qui éviterait alors le recours aux TPNs. Il serait alors nécessaire de définir – tout comme dans le cas des TPNs – une abstraction de l'espace d'états préservant les propriétés à vérifier [2]. Cette première étape permettrait ensuite d'écrire les algorithmes effectuant (éventuellement à la volée) la vérification des propriétés élémentaires définissant inductivement la logique EFFBD-TCTL.

2. Cette abstraction serait vraisemblablement basée sur la notion de *zone*.

Annexe A

Implémentation des outils de simulation et de vérification

Les travaux présentés dans le corps de cette thèse ont donné lieu à plusieurs implémentations, effectuées dans le cadre des PEAs Kimono et Omotesc ; cette annexe offre une vue d'ensemble de ces développements. Les spécifications ont été mises au point en collaboration avec les membres du projet [1] tandis que la réalisation a presqu'exclusivement été assurée par Sodius. Plus spécifiquement, j'ai assuré la majeure partie de la conception et de la réalisation de l'infrastructure (moteurs de transformation et de simulation), les autres membres de l'équipe-projet étant à l'origine de la superstructure (intégration à MDWorkbench, intefaces homme-machine, etc.).

Nous illustrons dans un premier temps quelques aspects des méta-modèles qui ont servi de support aux outils (section A.1). La section A.2.1 décrit le module de simulation par chronogrammes tandis que la section A.2.2 présente le module de vérification formelle.

A.1 Méta-modélisation et MDA/MDE

Les activités de Sodius sont largement axées sur l'ingénierie dirigée par les modèles (*Model Driven Architecture/Engineering* ou MDA/MDE), un paradigme de plus en plus présent en Ingénierie Système [30]. Afin de tirer partie des outils qui y sont développés, dont l'atelier MDWorkbench, nous avons également adopté cette démarche dont les fondements sont rappelés ci-dessous.

A.1.1 Éléments de la démarche MDA/MDE

L'approche MDA, apparue au début de la décennie, s'articule autour du paradigme « tout est modèle » [30, 46]. Elle se structure selon une architecture pyramidale en quatre niveaux d'abstraction successifs, illustrée tableau A.1.

Plusieurs langages de niveau M3 ont été développés, dont le MOF *(Meta Object Facility)*, standard de l'OMG, et ECore, défini et mis en œuvre dans le projet Eclipse EMF *(*Eclipse *Modeling Framework)*. MDWorkbench étant basé sur Eclipse, c'est donc ce dernier qui a été retenu pour la description et la manipulation des méta-modèles donnés plus bas.

Remarque A.1. *L'un des intérêts majeurs de l'approche MDA est de permettre la transformation automatisée des modèles en manipulant leur méta-modèle respectif, par le biais de règles*

1. Et plus spécifiquement avec MAP système.

TABLE A.1 – Niveaux d'abstraction de l'approche MDA

Niveau	Entité	Description
M3	Méta-méta-modèle	langage (auto-défini) décrivant l'écriture d'un méta-modèle
M2	Méta-modèle	langage définissant l'écriture d'un modèle
M1	Modèle	représentation d'un système
M0	Système	éléments du monde réel

de transformation[2]*. Cependant, notre traduction vers les TPNs est trop complexe pour être automatisée. Nous avons donc choisi d'implémenter manuellement les transformations, sous forme de classes et méthodes* JAVA*, les modèles étant quant à eux au format XML*[3] *ou XMI*[4]*.*

A.1.2 Description des méta-modèles

Modélisation des EFFBDs Au cours du projet, deux méta-modèles d'Ingénierie Système ont été développés : le premier, `defense`, est issu des réflexions menées autour de KIMONO et OMOTESC et comprend de ce fait des classes (ou *méta-types*) spécifiques au domaine de la défense. Le second, `systems`, vise une application plus généraliste.

Tous deux décrivent en particulier la syntaxe des EFFBDs, avec quelques différences. Ainsi, `systems` garde la distinction entre item et ressource, supprimée dans `defense`. En outre, la durée d'une fonction dans `systems` est précisée *via* une structure de données pouvant représenter une constante, un intervalle ou plus généralement une loi probabiliste ; `defense` se limite à la définition par un intervalle.

Nous donnons ci-dessous des extraits – pour la plupart simplifiés – de ces méta-modèles. En particulier, et lorsque cela ne gêne pas la compréhension, nous omettrons les rôles dans les relations entre méta-types. La figure A.1 représente ainsi la méta-modélisation des structures de contrôle. On notera qu'en parallèle des structures de contrôle (`Construct`), la notion de branche est explicitement modélisée. Une branche référence une suite (ordonnée) de SequenceConstructs (dont héritent les structures AND, OR, LP, IT et FC) et, éventuellement, une structure LE ou EXIT. Le calcul des prédécesseurs et des successeurs revient alors à explorer les contenus et les conteneurs des branches.

> **Remarque A.2.** *Les deux méta-modèles comportent des classes et des attributs supplémentaires. En particulier, nous y avons modélisé les réplications ainsi que des facteurs de probabilité de sélection des branches de type* `SelectBranch` *et* `FunctionExitBranch`*, comme proposé dans la section 3.2.2*

La figure A.2 représente les relations de consommation, lecture et production entre une fonction et un flux. La modélisation fait apparaître une classe `Quantity` portant la valeur de la quantité échangée[5].

Enfin, la figure A.3 décrit les différentes propriétés vérifiables par le moteur, telles qu'elles ont été données dans la section 6.3.3.

2. Notons que la méta-modélisation ne permet pas de capturer directement la sémantique comportementale

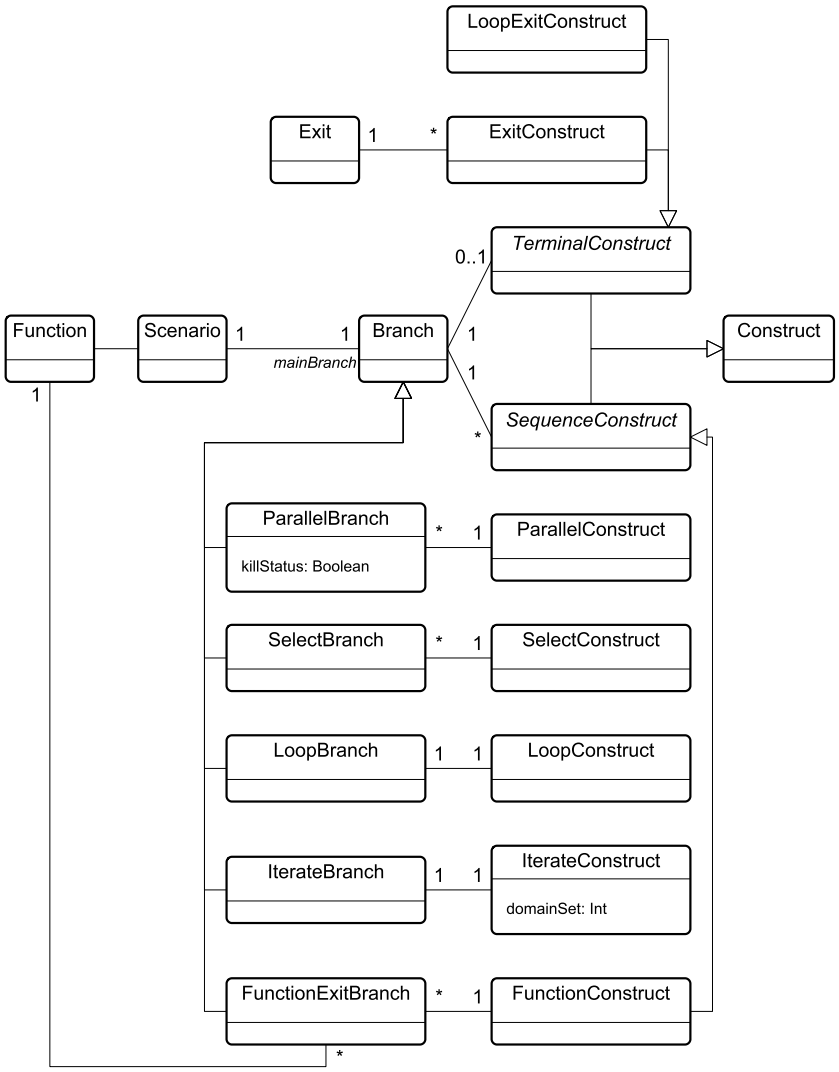


FIGURE A.1 – Structures de contrôle des EFFBDs

Description des chronogrammes Le moteur de simulation, décrit dans la section A.2.1, fournit une suite d'états s qui reste encore difficilement interprétable. Nous avons choisi d'abstraire les états en étapes (ou *steps*) de manière à ne représenter que l'évolution des fonctions et des niveaux des flux[6], comme cela est proposé dans le logiciel CORE. Le méta-modèle correspondant, `timeline`, est illustré figure A.4.

Pour plus de simplicité, nous n'y avons pas fait figurer la prise en compte des défaillances. En particulier, il est possible de représenter pour chaque fonction les durées d'exécution

des modèles [68].

3. *eXtensible Markup Language.*
4. *XML Metadata Interchange.*
5. Il s'agit ici d'un entier ; dans le cas de la simulation, cette valeur (de même que les quantités initiales) peut être décrite par un flottant.
6. Compte-tenu de la note précédente, ceux-ci peuvent être non entiers.

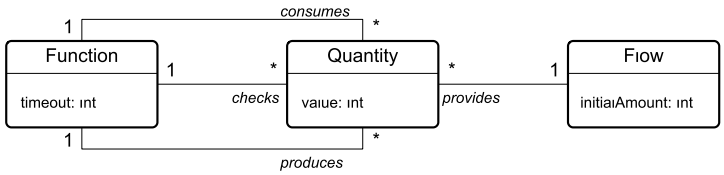


FIGURE A.2 – Relations de consommation, lecture et production

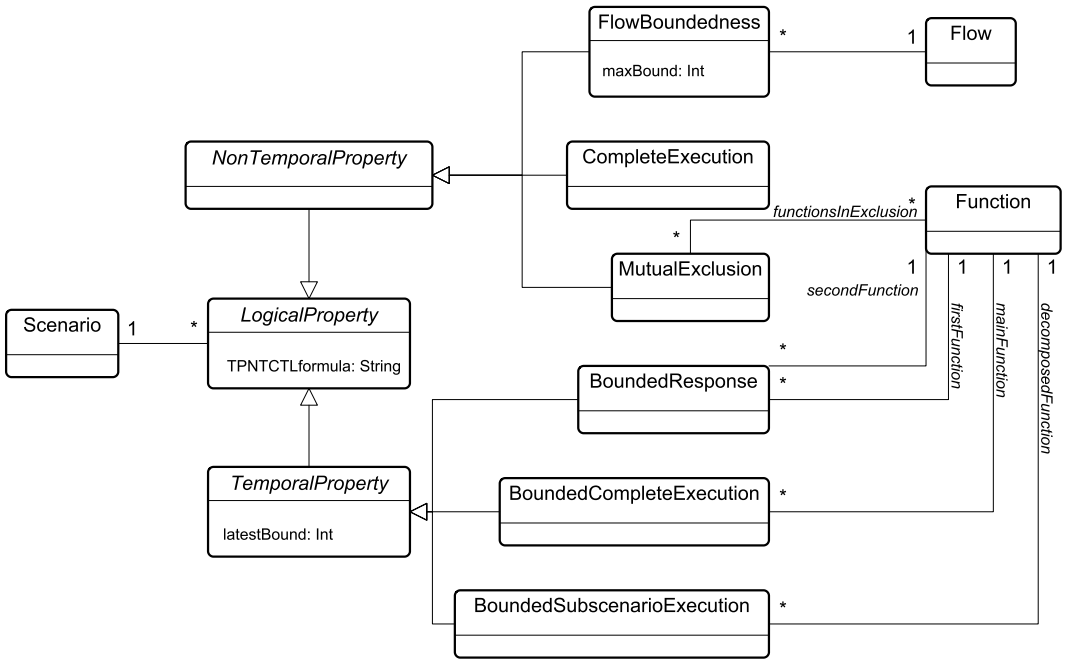


FIGURE A.3 – Méta-modélisation des propriétés vérifiables

théoriques et effectives (ou, pour chaque flux, les niveaux attendus et réels) ainsi que la liste des erreurs qui ont été activées.

Description des TPNs Le moteur de vérification, décrit dans la section A.2.2, crée des modèles *conformes* au méta-modèle `TPN`, illustré figure A.5, et qui peuvent être lus par ROMÉO à des fins de vérification formelle.

Pour simplifier, ce méta-modèle ne comprend pas les informations graphiques, telles que la couleur et la position, disponibles dans ROMÉO.

A.2 Description des outils de simulation et de vérification

A.2.1 Module de simulation

Principe général L'algorithme général du moteur de simulation par chronogramme se décompose en cinq étapes :

– lecture du modèle XMI (conforme aux méta-modèles `defense` ou `systems`) ;

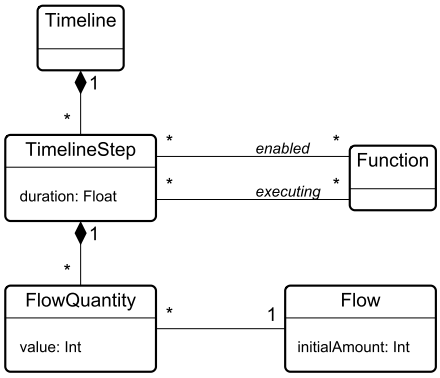


FIGURE A.4 – Extrait du méta-modèle `timeline`

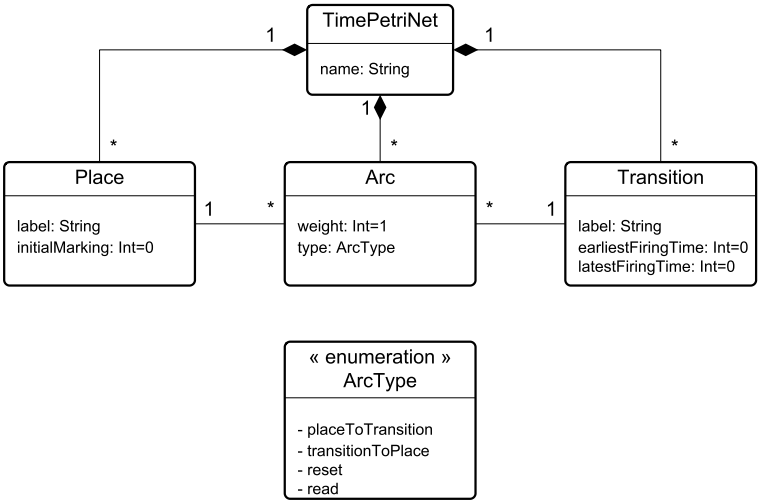


FIGURE A.5 – Extrait du méta-modèle `TPN`

- dépliage des réplications, items partagés, etc. et transformation des objets de type `Construct` en nœuds ;
- calcul de l'état initial ;
- calcul des états successifs jusqu'à atteindre un état final (ou le nombre maximal d'itérations) ;
- transformation des états en *steps* par combinaison d'états [7].

Le moteur est en outre capable de suspendre la simulation pour demander explicitement à l'utilisateur de sélectionner une branche de choix ou d'arbitrer un conflit de ressources.

Enfin, l'interface graphique repose en partie sur un modeleur de diagrammes GANTT [8] ; elle propose les opérations courantes de sauvegarde, filtrage des éléments affichés, sélection de la fenêtre temporelle, mode pas-à-pas etc.

7. Pour des raisons de lisibilité, nous ne donnons pas les règles de transformation en *steps*.
8. Il s'agit de l'outil BIRT CHART ENGINE, basé sur ECLIPSE.

Simulation du passage à niveau L'exécution du modèle du passage à niveau étant infinie, la simulation est stoppée après un certain nombre d'états. La figure A.6 illustre l'évolution du système lors du premier passage du train, entre 100 et 200 secondes. Dans cette simulation, le train circule à une vitesse moyenne approximative de 12,5 m/s (45 km/h) et déclenche la pédale d'annonce à $t \approx 130$ s.

À nouveau, l'exécution d'une fonction est représentée sur fond vert et l'attente d'un flux sur fond jaune. Les ressources sont figurées en noir mais, pour faciliter la lecture, les items ne sont pas représentés.

Les cinq premières lignes du chronogramme correspondent à la branche `train`, les deux suivantes à la branche `contrôleur` et les six dernières à la branche `passage à niveau`. Sur ce simple cycle, on retrouve bien évidemment les différentes propriétés annoncées dans la section 6.2.3.

A.2.2 Module de vérification

Ce module fait partie des premiers développements que nous avons réalisés ; il repose sur les méta-modèles `defense` et `TPN`.

Principe général L'algorithme du moteur de vérification se décompose en cinq étapes principales :

- lecture du modèle et transformation en un modèle conforme au méta-modèle `TPN` ;
- lecture et transformation de la propriété ;
- appel au model checker de ROMÉO (`mercutio-tctl`) ;
- lecture de la valeur de vérité de la formule de TPN-TCTL et de la trace éventuellement produite ;
- transformation de la trace en une séquence d'exécutions de fonctions et affichage des résultats d'analyse.

> **Remarque A.3.** *La trace fournie par* ROMÉO *n'est pas temporisée ; la trace produite par notre outil ne l'est donc pas non plus. Il serait bien évidemment possible d'ajouter des informations temporelles ; le model checker que nous nous proposons de développer comportera vraisemblablement une telle fonctionnalité (cf. 8.2). La combinaison de cet outil avec le moteur de simulation permettra enfin de simuler l'exécution ayant conduit à la non vérification de la propriété, de manière à en faciliter l'analyse.*

Vérification d'une propriété du passage à niveau Le système bouclant de façon infinie, une propriété de type « Exécution complète » ou `CompleteExecution` sera nécessairement fausse. La vérification d'une telle propriété sur le modèle du PN conduit ainsi à une réponse négative et au rapport d'analyse donné figure A.7. La trace fournie correspond à un cycle du passage à niveau, mettant en évidence la boucle infinie.

Pour compléter l'analyse, le moteur laisse l'accès au modèle XML représentant le réseau ainsi qu'à la propriété TPN-TCTL. Il est ainsi possible, à partir du moteur de simulation de ROMÉO, de rejouer l'exécution sur le TPN sous-jacent. Compte tenu de la remarque précé-

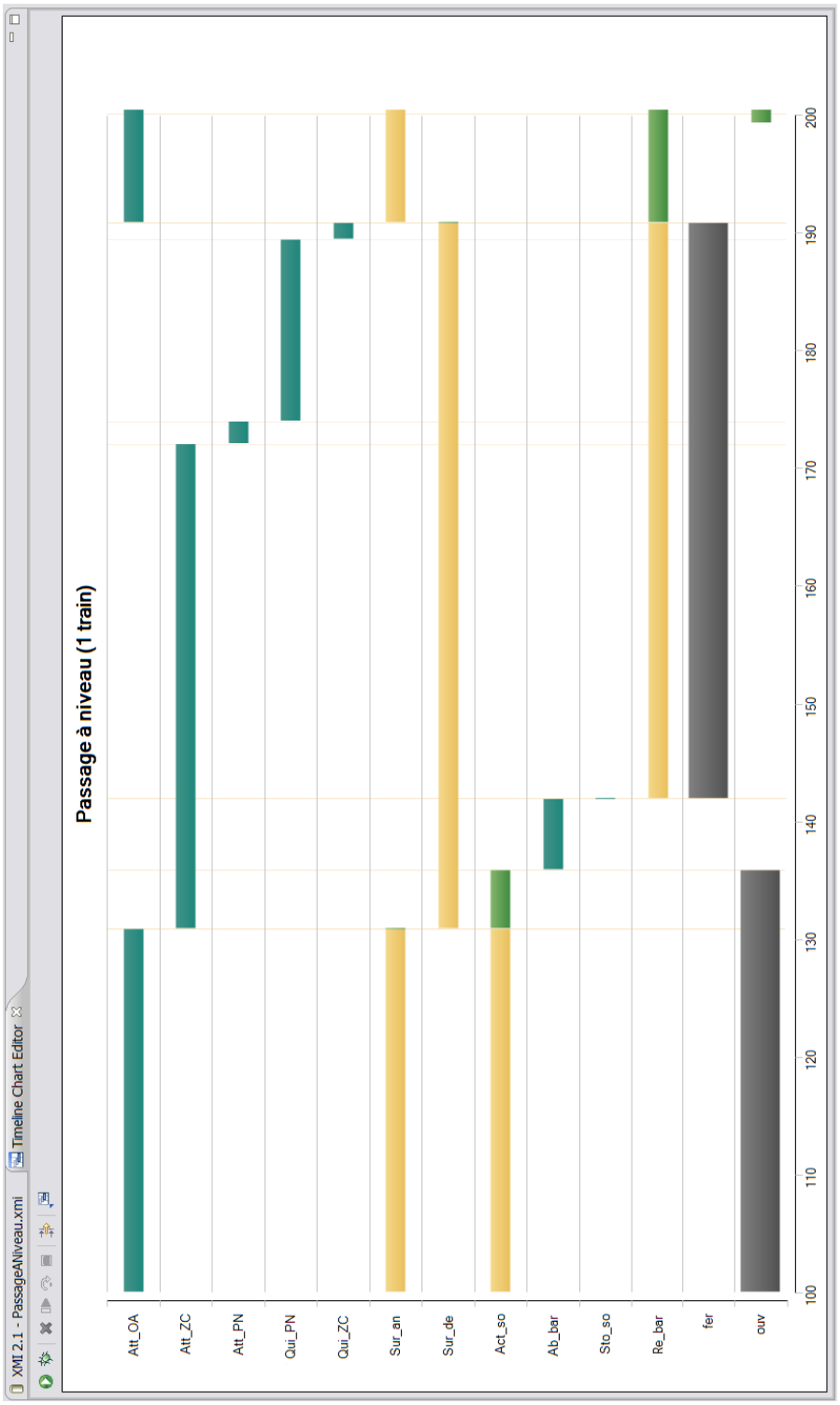


FIGURE A.6 – Chronogramme du passage à niveau

```
Analysis Report of the Logical Property

Test of the property "executionComplete"
Property Type: Complete Execution
The "executionComplete" property isn't verified.

Executed Function: "Att_OA"
Executed Function: "Sur_an"
Executed Function: "Act_so"
Executed Function: "Ab_bar"
Executed Function: "Sto_so"
Executed Function: "Att_ZC"
Executed Function: "Att_PN"
Executed Function: "Qui_PN"
Executed Function: "Qui_ZC"
Executed Function: "Sur_de"
Executed Function: "Re_bar"
Executed Function: "Att_OA"
```

FIGURE A.7 – Vérification d'une propriété d'exécution finie

dente, cette fonctionnalité pourra être supprimée avec le futur model checker écrit directement sur les EFFBDs.

Bibliographie

[1] T. Abdoul, J. Champeau, P. Dhaussy, P.-Y. Pillain et J.-C. Roger : AADL execution semantics transformation for formal verification. *In 13th IEEE International Conference on on Engineering of Complex Computer Systems (ICECCS '08)*, p. 263–268, mars 2008. 23

[2] M. W. Alford : A requirements engineering methodology for real-time processing requirements. *IEEE Transactions on Software Engineering*, 3(1):60–69, jan. 1977. 43

[3] M. W. Alford : Strengthening the systems/software interface for real-time systems. *In 2nd International Symposium of the National Council on Systems Engineering*. NCOSE, 1992. 43

[4] R. Alur, C. A. Courcoubetis et D. L. Dill : Model-checking for real-time systems. *In 5th IEEE Symposium on Logic in Computer Science*, p. 414–425, Philadelphia, PA, juin 1990. 24, 118, 120

[5] R. Alur, C. A. Courcoubetis et D. L. Dill : Model-checking in dense real-time. *Information and Computation*, 104:2–34, 1993. 26, 117, 118, 124

[6] R. Alur et D. L. Dill : A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, avr. 1994. 23, 94

[7] R. Alur, T. A. Henzinger et P.-H. Ho : Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, mars 1996. 58

[8] C. André, F. Mallet et R. de Simone : Modeling of immediate vs. delayed data communications : from AADL to UML MARTE. *In ECSI Forum on Specification & Design Languages (FDL)*, p. 249–254, sept. 2007. 21, 22

[9] C. André, F. Mallet et R. D. Simone : Modeling time(s) in UML. Rapport de recherche ISRN I3S/RR-2007-16-FR, I3S, Sophia-Antipolis, mai 2007. 21

[10] A. Avizienis, J.-C. Laprie, B. Randell et C. Landwehr : Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, 2004. 26

[11] B. Bérard, F. Cassez, S. Haddad, D. Lime et O. H. Roux : Comparison of different semantics for time Petri nets. *In Automated Technology for Verification and Analysis (ATVA'05)*, vol. 3707 de *Lecture Notes in Computer Science*, p. 293–307. Springer, oct. 2005. 95

[12] B. Bérard, F. Cassez, S. Haddad, D. Lime et O. H. Roux : Comparison of the expressiveness of timed automata and time Petri nets. *In 3rd International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2005)*, vol. 3829 de *Lecture Notes in Computer Science*, p. 211–225. Springer, sept. 2005. 94

[13] B. Berthomieu, J.-P. Bodeveix, C. Chaudet, S. D. Zilio, M. Filali et F. Vernadat : Formal verification of AADL specifications in the Topcased environment. *In 14th Ada-Europe International Conference on Reliable Software Technologies*, vol. 5570 de *Lecture Notes in Computer Science*, p. 207–221. Springer, juin 2009. 23

[14] B. Berthomieu et M. Diaz : Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, 1991. 93, 97

[15] B. Berthomieu et F. Vernadat : State class constructions for branching analysis of time Petri nets. *In 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, vol. 2619 de *Lecture Notes in Computer Science*, p. 442–457. Springer, avr. 2003. 61

[16] Bioastronautic Section, Space System Organization, General Electric : Integrated medical and behavioral laboratory measurement system, phase B 2. Volume 3: System concept and design. Appendix B: function flow block diagrams. Rap. tech., NASA, fév. 1968. http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19770077080_1977077080.pdf. 43

[17] C. Bock : UML 2 activity model support for Systems Engineering functional flow diagram. *Systems Engineering*, 6:249–265, 2003. 21

[18] H. Boucheneb, G. Gardey et O. H. Roux : TCTL model checking of time Petri nets. *Journal of Logic and Computation*, 2009. à paraître. 119, 120

[19] M. Boyer et O. H. Roux : On the compared expressiveness of arc, place and transition time Petri nets. *Fundamenta Informaticae*, 88(3):225–249, 2008. 94

[20] F. Cassez et O. H. Roux : Structural translation from time Petri nets to timed automata. *Journal of Systems and Software*, 29(1):1456–1468, 2006. 26, 113

[21] S. Christensen et N. D. Hansen : Coloured Petri nets extended with place capacities, test arcs and inhibitor arcs. *In 14th International Conference on Application and Theory of Petri Nets (ICATPN'93)*, vol. 691 de *Lecture Notes in Computer Science*, p. 186–205. Springer, juin 1993. 97

[22] E. M. Clarke, E. A. Emerson et A. P. Sistla : Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, avr. 1986. 24, 118

[23] E. M. Clarke, O. Grumberg et D. A. Peled : *Model checking*. MIT Press, 1999. 24

[24] C. Dufourd, A. Finkel et P. Schnoebelen : Reset nets between decidability and undecidability. *In 25th International Colloquium on Automata, Languages and Programming (ICALP'98)*, vol. 1443 de *Lecture Notes in Computer Science*, p. 103–115. Springer, juil. 1998. 97

[25] Electronic Industries Alliance (EIA) : *ANSI/EIA-632-1999: Processes for Engineering a System*. EIA, Arlington, VA, avr. 1998. 18

[26] E. A. Emerson et J. Srinivasan : Branching time temporal logic. *In Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency (proceedings of the REX Workshop)*, vol. 354 de *Lecture Notes in Computer Science*, p. 123–172. Springer, juin 1989. 117

[27] D. Evrot : *Contribution à la vérification d'exigences de sécurité : application au domaine de la machine industrielle*. Thèse de doctorat, Université Henri Poincaré, Nancy I, juil. 2008. 22

[28] D. Evrot, J.-F. Pétin et G. Morel : Combining SysML and formals methods for safety requirements verification. *Insight Journal of INCOSE*, 11(3):21–22, 2008. 22

[29] P. H. Feiler, B. A. Lewis et S. Vestal : The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems. *In IEEE Symposium on Computer Aided Control Systems Design (CACSD 2006)*, p. 1206–1211, nov. 2006. 22, 131

[30] D. S. Frankel : *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, jan. 2003. 147

[31] G. Gardey : *Contribution à la vérification et au contrôle des systèmes temps réel – Application aux réseaux de Petri temporels et aux automates temporisés*. Thèse de doctorat, École Centrale de Nantes, déc. 2005. 119, 120, 124, 125

[32] G. Gardey, D. Lime, M. Magnin et O. H. Roux : Roméo: A tool for analyzing time Petri nets. *In 17th International Conference on Computer-Aided Verification (CAV'05)*, vol. 3576 de *Lecture Notes in Computer Science*, p. 418–423. Springer, juil. 2005. 26, 94, 120

[33] M. R. Garey et D. S. Johnson : *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co Ltd, jan. 1979. 124

[34] J.-Y. Girard : Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987. 21

[35] A. Giua, F. DiCesare et M. Silva : Generalized mutual exclusion contraints on nets with uncontrollable transitions. *In IEEE International Conference on Systems, Man and Cybernetics*, vol. 2, p. 974–979. IEEE Press, oct. 1992. 119

[36] H.-M. Hanisch : Analysis of place/transition nets with timed arcs and its application to batch process control. *In 14th International Conference on Application and Theory of Petri Nets (ICATPN'93)*, vol. 691 de *Lecture Notes in Computer Science*, p. 282–299. Springer, juin 1993. 93

[37] T. A. Henzinger : The theory of hybrid automata. *In 11th Annual IEEE Symposium on Logic in Computer Science (LICS96)*, p. 278–292. IEEE Computer Society Press, 1996. 113, 145

[38] T. A. Henzinger, Z. Manna et A. Pnueli : Timed transition systems. *In Real-Time: Theory in Practice (proceedings of the REX Workshop)*, vol. 600 de *Lecture Notes in Computer Science*, p. 226–251. Springer, juin 1991. 23, 35

[39] E. Herzog : *An approach to Systems Engineering tools data representation and exchange*. Thèse de doctorat, Linköpings Universitet, Linköping, avr. 2004. 25

[40] Institute of Electrical and Electronics Engineers (IEEE) : *IEEE Std 1220^{TM}- 2005: IEEE Standard for Application and Management of the Systems Engineering Process*. IEEE, New York, NY, sept. 2005. 11, 17, 18, 19

[41] International Council on Systems Engineering (INCOSE) : *Systems Engineering Handbook: a guide for system life cycle processes and activities*. INCOSE, Seattle, WA, 3^{e} édn, juin 2006. 17, 21

[42] International Electrotechnical Commision (IEC) : *IEC 60812:Analysis techniques for system reliability - Procedure for failure mode and effects analysis (FMEA)* . IEC, jan. 2006. 131

[43] International Organization for Standardization (ISO) : *ISO/IEC 15288-2008: Systems Engineering – System Life-Cycle Processes*. ISO, mars 2008. 18

[44] N. D. Jones, L. H. Landweber et Y. E. Lien : Complexity of some problems in Petri nets. *Theoretical Computer Science*, 4(3):277–299, juin 1977. 97

[45] W. Khansa, P. Aygalinc et J.-P. Denat : Structural analysis of P-time Petri nets. *In Symposium on Discrete Events and Manufacturing Systems (CESA'96)*, p. 127–136, Lille, juil. 1996. 93

[46] A. G. Kleppe, J. Warmer et W. Bast : *MDA Explained - The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman, Boston, MA, 2003. 147

[47] A. Knutilla, C. Schlenoff, S. Ray, S. T. Polyakand, A. Tate, S. C. Cheah et R. C. Anderson : Process Specification Language: an analysis of existing representations. Rap. tech., National Institute of Standards and Technology, Manufacturing Engineering Laboratory, 1998. 43

[48] L. Lamport : Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977. 24

[49] J.-C. Laprie : *Dependability: basic concepts and terminology*. Dependable computing and fault-tolerant systems. Springer, 1992. 26

[50] J.-F. le Tallec et J. DeAntoni : Multi-level designing approach. *In 3rd Junior Researcher Workshop on Real-Time Computing (JRWRTC 09)*, oct. 2009. à paraître. 22

[51] N. G. Leveson et J. L. Stolzy : Safety analysis using Petri nets. *IEEE Transactions on Software Engineering*, 13(3):386–397, 1987. 58

[52] J. Long : Relationships between common graphical representations in Systems Engineering. *In 5th International Symposium of the INCOSE*, St. Louis, MO, juil. 1995. INCOSE. Updated July 2002. 11, 20, 21, 43, 44

[53] M. Magnin : *Réseaux de Petri à chronomètres – Temps continu et temps discret*. Thèse de doctorat, École Centrale de Nantes, déc. 2007. 38, 93, 98

[54] J.-P. Meinadier : *Ingénierie et intégration des systèmes*. Études et logiciels informatiques. Hermès, Paris, 1998. 17, 18, 20, 49

[55] P. M. Merlin : *A study of recoverability of computing systems*. Thèse de doctorat, University of California, Irvine, CA, 1974. 26, 93

[56] Ministère de l'Équipement, du Logement, des Transports et de la Mer : Arrêté du 18 mars 1991 relatif au classement, à la réglementationet à l'équipement des passages à niveau. *Journal Officiel du 14 avril 1991*, 11:67–69, avr. 1991. 58

[57] M. K. Molloy : Performance analysis using stochastic Petri nets. *IEEE Transaction on Computers*, 31(9):913–917, sept. 1982. 22

[58] Object Management Group (OMG) : *UML Profile for MARTE, Beta 2*. Object Management Group, juin 2008. http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf. 21

[59] Object Management Group (OMG) : *OMG Unified Modeling Language™ (OMG UML) version 2.2*. Object Management Group, fév. 2009. http://www.omg.org/spec/UML/2.2. 21

[60] D. W. Oliver, T. P. Kelliher et J. G. Keegan Jr. : *Engineering complex systems with models and objects*. McGraw-Hill, juin 1997. http://www.incose.org/ProductsPubs/DOC/EngComplexSys.pdf. 43

[61] M. Paludetto et A. Benzina : UML et réseaux de Petri : Étude de cas et évaluations temporelles. *Journal Européen des Systèmes Automatisés*, 36(8):1155–1178, 2002. 21

[62] C. A. Petri : *Kommunikation mit Automaten.* Thèse de doctorat, Institut für instrumentelle Mathematik, Bonn, 1962. 23, 93

[63] A. Pnueli : The temporal logic of programs. *In 18th Annual Symposium on Foundations of Computer Science*, p. 46–57. IEEE Press, nov. 1977. 117

[64] C. Ramchandani : *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets.* Thèse de doctorat, Massachusetts Institute of Technology, Cambridge, MA, 1974. 93

[65] X. Renault, F. Kordon et J. Hugues : Adapting models to model checkers, a case study: Analysing AADL using time or colored Petri nets. *In IEEE International Symposium on Rapid System Prototyping*, p. 26–33. IEEE Computer Society, juin 2009. 23

[66] O. H. Roux : *Vérification des réseaux de Petri temporels et à chronomètres.* Habilitation à Diriger les Recherches (HDR), Université de Nantes, déc. 2005. 120

[67] A.-E. Rugina : *Dependability modeling and evaluation – From AADL to stochastic Petri nets.* Thèse de doctorat, Institut National Polytechnique de Toulouse, nov. 2007. 22, 131

[68] D. D. Ruscio, F. Jouault, I. Kurtev, J. Bézivin et A. Pierantonio : Extending AMMA for supporting dynamic semantics specifications of DSLs. Rap. tech. 06.02, Laboratoire d'Informatique de Nantes-Atlantique (LINA), avr. 2006. 149

[69] G. Schellhorn, A. Thums et W. Reif : Formal fault tree semantics. *In 6th Biennal World Conference on Integrated Design and Process Technology (IDPT 2002)*, Pasadena, CA, juin 2002. 133

[70] C. Seidner : Étude de la traduction de diagrammes de type EFFBD en réseaux de Petri temporels. Mémoire de D.E.A., École Centrale de Nantes, sept. 2006. 28

[71] C. Seidner : Étude des représentations haut-niveau en Ingénierie Système et de leur aptitude à supporter des vérifications formelles. Séminaire bibliographique de D.E.A., École Centrale de Nantes, mai 2006. 20, 25

[72] C. Seidner, J.-P. Lerat et O. H. Roux : Usability of formal verification on EFFBD models: applying Petri nets to Systems Engineering issues. *In 17th International Symposium of the INCOSE*, San Diego, CA, juin 2007. INCOSE. 29

[73] C. Seidner, J.-P. Lerat et O. H. Roux : Behavior diagrams model checking: formal methods applied to Systems Engineering and Design. *In 6th Annual Conference on Systems Engineering Research (CSER'08)*, Los Angeles, CA, avr. 2008. University of Southern California. 29

[74] C. Seidner, J.-P. Lerat et O. H. Roux : Usability and usefulness of formal verification in a system design process. *In 18th International Symposium of the INCOSE*, Utrecht, juin 2008. INCOSE. 29

[75] C. Seidner et O. H. Roux : On the formal verification of EFFBD models using a structural translation to time Petri nets. Rapport interne RI20073 3695, IRCCyN, Nantes, oct. 2007. `http://www.irccyn.ec-nantes.fr/~seidner/SR07_TechReport.pdf`. 29

[76] C. Seidner et O. H. Roux : Formal methods for Systems Engineering behavior models. *IEEE Transactions on Industrial Informatics*, 4(4):280–291, nov. 2008. 29

[77] J. Shi et M. Törngren : An overview of UML2 and brief assessment from the viewpoint of embedded control systems development. Rap. tech., Mechatronics Lab, Dpt. of Machine Design, Royal Institute of Technology, Stockholm, mars 2005. 21

[78] J. F. Skipper : Assessing the suitability of UML for capturing and communicating systems engineering design models. Rap. tech., Vitech Corp., juil. 2002. 21

[79] Society of Automotive Engineers (SAE) : *AS5506 : Architecture Analysis & Design Language (AADL)*. SAE, jan. 2009. 22, 65

[80] SysML Finalization Task Force : *OMG Systems Modeling Language (OMG SysML^{TM}) version 1.1*. Object Management Group, nov. 2008. http://www.omg.org/spec/SysML/1.1. 22, 45

[81] A. Thums et G. Schellhorn : Model checking FTA. *In 12th International Formal Methods Europe Symposium (FME 2003)*, vol. 2805 de *Lecture Notes in Computer Science*, p. 739–757. Springer, sept. 2003. 133

[82] United States Air Force : *MIL-STD-499 : Systems Engineering management*. US Department of Defense, juil. 1969. 17, 18, 43

[83] United States Air Force : *MIL-STD-499B : Systems Engineering*. US Department of Defense, mai 1994. 18

[84] United States Navy : *MIL-STD-1629A : Procedures for Performing a Failure Mode, Effects and Critically Analysis*. US Department of Defense, nov. 1980. 131

[85] L. van den Berg, P. Strooper et K. Winter : Introducing time in an industrial application of model-checking. *In 12th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2007)*, vol. 4916 de *Lecture Notes in Computer Science*, p. 56–67. Springer, juil. 2008. 58

[86] T. Vergnaud, B. Zalila et J. Hugues : Ocarina : a compiler for the AADL. Rap. tech., École Nationale Supérieure des Télécommunications, Paris, juin 2006. 23

[87] W. Vesely, F. Goldberg, N. Roberts et D. Haasl : *NUREG-0492 :Fault tree handbook*. U.S. Nuclear Regulatory Commission, Washington, DC, jan. 1981. 133

[88] Vitech Corp. : *COREsim User Guide – release 3.0*. Vienna, VA, déc. 2000. http://www.vitechcorp.com/support/docs/30/COREsm30.pdf. 43

[89] W. Yi, P. Pettersson et M. Daniels : Automatic verification of real-time communicating systems by constraint-solving. *In 7th International Conference on Formal DescriptionTechniques (FORTE'94)*, vol. 6, p. 223–238, Berne, oct. 1994. 58

Vérification des EFFBDs : Model checking en Ingénierie Système

Résumé : l'Ingénierie Système (IS) est une méthodologie pluridisciplinaire de conception et de mise en œuvre des systèmes complexes. La maîtrise de la Sûreté de Fonctionnement est un processus essentiel de l'IS et, dans sa poursuite, le recours à des méthodes formelles telles que le *model checking* se heurte généralement à des difficultés d'utilisation.

Dans cette thèse CIFRE, effectuée en collaboration avec l'IRCCyN et SODIUS, nous avons cherché à concevoir un outil de vérification formelle d'architectures fonctionnelles qui soit immédiatement utilisable dans des démarches de conception d'IS. Dans ce but, nous transformons des modèles et des propriétés comportementales *haut niveau* vers des équivalents *bas niveau* sur lesquels sont effectuées les vérifications formelles, le résultat étant présenté en termes haut niveau.

Plus particulièrement, nous avons choisi comme modèles d'entrée les diagrammes EFFBDs : ils constituent un outil de modélisation largement utilisé en IS et adapté aux contraintes du model checking. Nous en avons établi la syntaxe et la sémantique formelles ; nous avons alors pu décrire une transformation vers les réseaux de PETRI temporels (TPNs) dont nous avons prouvé qu'elle préserve le comportement temporel des modèles. Parallèlement, nous avons décrit une logique temporelle quantitative adaptée aux EFFBDs et sa traduction vers une logique correspondante sur les TPNs ; nous avons alors pu établir la complexité de son model checking.

Ces différents résultats théoriques nous ont enfin permis de réaliser un outil de simulation et de vérification d'architectures fonctionnelles et dysfonctionnelles (c.-à-d. modélisant des fonctions défaillantes), déployé et utilisé industriellement.

Mots-clés : vérification formelle, réseaux de PETRI temporels, Ingénierie Système, Sûreté de Fonctionnement, architecture fonctionnelle, EFFBD.

EFFBDs Verification: Model checking in Systems Engineering

Abstract: Systems Engineering (SE) is an interdisciplinary and methodological approach for the design and operation of complex systems. Safety Engineering is a major SE process, yet the use of formal methods such as *model checking*, however powerful they may be, is hampered by their intrinsic complexity.

Our research work, supported by an industrial partnership between the IRCCyN lab and SODIUS, aimed at designing a tool which is directly usable during the SE design phase and which formally verifies functional models. To that end, *high-level* models and behavioral properties are transformed into *low-level* equivalents on which formal verifications are performed; analysis results are then expressed on the *high-level* models. To be specific, we considered EFFBDs as input models; this modeling language is indeed widely used in SE and adapted to model checking constraints. We formally established their syntax and semantics, then we were able to describe a translation into time PETRI nets (TPNs) which we proved as preserving the model temporal behavior. Simultaneously, we described a quantitative temporal logic adapted to the EFFBDs and its translation into a corresponding logic on TPNs; we then established the computational complexity of its model checking.

These successive theoretical results led us to develop a simulation and verification software tool that can analyze both functional and dysfunctional architectures (i.e. modeling functions failures); this tool is deployed and operated in an industrial context.

Keywords: formal verification, time PETRI nets, Systems Engineering, Safety Engineering, functionnal architecture, EFFBD.