REPUBLIQUE DU CAMEROUN
Paix – Travail – Patrie
****
UNIVERSITE DE YAOUNDE I
FACULTE DES SCIENCES
DEPARTEMENT DE INFORMATIQUE
****
CENTRE DE RECHERCHE ET DE FORMATION DOCTORALE EN SCIENCES, TECHNOLOGIES ET GEOSCIENCES
Laboratoire LIRIMA

REPUBLIC OF CAMEROUN
Peace – Work – Fatherland
****
UNIVERSITY OF YAOUNDE I
FACULTY OF SCIENCE
DEPARTMENT OF COMPUTER SCIENCES
****
POSTGRADUATE SCHOOL OF SCIENCE, TECHNOLOGY AND GEOSCIENCES

Synthèse automatique de systhèmes de contrôle-commande spécifier en Graf-cet sur multi-cibles microcontrôleurs

THESE
Présentée en vue de l'obtention du Doctorat/Ph.D en Informatique

Par : **NZEBOP NDENOKA Gérard**
Master en Informatique

Sous la direction de
TCHUENTE Maurice
Professeur, Université de Yaoundé I
SIMEU Emmanuel
HDR UNIVERSITE DE GRENOPLE ALPES, CNRS

Année Académique : 2019-2020

BIBLIOTHEQUE UNIVERSITAIRE UYI

RÉPUBLIQUE DU CAMEROUN
PAIX-TRAVAIL-PATRIE
****
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR
****
UNIVERSITÉ DE YAOUNDÉ I
****
CENTRE DE RECHERCHE ET DE FORMATION DOCTORALE EN SCIENCES, TECHNOLOGIES ET GEOSCIENCES
****

REPUBLIC OF CAMEROON
PEACE-WORK-FATHERLAND
****
MINISTRY OF HIGHER EDUCATION
****
THE UNIVERSITY OF YAOUNDE I
****
POSTGRADUATE SCHOOL OF SCIENCE, TECHNOLOGY AND GEO-SCIENCES
****

DÉPARTEMENT D'INFORMATIQUE
DEPARTMENT OF COMPUTER SCIENCE

ATTESTATION DE CORRECTION DE LA THÈSE DE DOCTORAT/PhD

Nous soussignés, **Pr. FOUDA NDJODO Marcel** et **Pr. TCHUENTE Maurice**, membres du jury de soutenance de la thèse de Doctorat/PhD présentée par Monsieur **NZEBOP NDENOKA Gérard**, Matricule **08U0539**, intitulée: «**Synthèse automatique de systèmes de contrôle-commande spécifiés en Grafcet sur multi-cibles microcontrôleurs**» et soutenue le **14 juillet 2020** en vue de l'obtention du diplôme de **Doctorat/PhD en Informatique**, attestons que toutes les corrections demandées par le jury de soutenance en vue de l'amélioration de ce travail, ont été effectuées.

En foi de quoi la présente attestation lui est délivrée pour servir et valoir ce que de droit.

Fait à Yaoundé, le 16/07/2020

Président

Pr FOUDA NDJODO Marcel

Rapporteur

Pr. TCHUENTE Maurice

Liste Protocolaire

Division de la Programmation et du Suivi des Activités Académiques
Liste des enseignants permanents
Année académique 2018-2019 (Par Département et par Grade)
Date d'actualisation : 19 Février 2019

Administration		
DOYEN	**TCHOUANKEU Jean Claude**	Professeur
VICE-DOYEN DP-SAA	**DONGO Etienne**	Professeur
VICE-DOYEN DSSE	**AJEAGAH Gidéon**	Professeur
VICE-DOYEN DRC	**ABOSSOLO Monique**	Professeur
Chef DAASR	**MBAZE MEVA'A L.**	Professeur
Chef DAF	**NDOYE FOE Marie C. F.**	Professeur
I-Département de Biochimie (BC) (37)		
NOMS ET PRENOMS	**Grades**	**Observations**
1-**FEKAM B. F.**	Professeur	En Poste
2-**MBACHAM W.**	Professeur	En Poste
3-**MOUNDIPA F.P.**	Professeur	Chef de Dept
4-**BENG née NINTCHOM P.V.**	Professeur	En Poste
5-**OBEN E. J.**	Professeur	En Poste
6-**ACHU Merci BIH**	Maître de Conférences	En Poste
7-**ATOGHO M. B.**	Maître de Conférences	En Poste
8-**BELINGA N.F.M.**	Maître de Conférences	Chef DAF / FS
9-**BIGOGA JUDE**	Maître de Conférences	En Poste
10-**BOUDJEKO T.**	Maître de Conférences	En Poste

11-**EFFA ONOMO P.**	Maître de Conférences	En Poste
12-**FOKOU Élie**	Maître de Conférences	En Poste
13-**KANSCI G.**	Maître de Conférences	En Poste
14-**NANA Louise Epse WAKAM**	Maître de Conférences	En Poste
15-**NGONDI J. L.**	Maître de Conférences	En Poste
16-**NGUEFACK J.**	Maître de Conférences	En Poste
17-**NJAYOU F.N.**	Maître de Conférences	En Poste
18-**AKINDEH MBUH N.**	Chargé de Cours	En Poste
19-**AZANTSA K. G.**	Chargé de Cours	En Poste
20-**BEBOY E.S.N.**	Chargé de Cours	En Poste
21-**DAKOLE D. C.**	Chargé de Cours	En Poste
22-**DJOKAM T. R.**	Chargé de Cours	En Poste
23-**DJUIDJE N.M.**	Chargé de Cours	En Poste
24-**DJUIKWO N.R.**	Chargé de Cours	En Poste
25-**DONGMO L.J.B.**	Chargé de Cours	En Poste
26-**EWANE Cécile A.**	Chargé de Cours	En Poste
27-**FONKOUA M.**	Chargé de Cours	En Poste
28-**BEBEE F.**	Chargé de Cours	En Poste
29-**KOTUE K. C.**	Chargé de Cours	En Poste
30-**LUNGA K. P.**	Chargé de Cours	En Poste
31-**MANANGA M.J.**	Chargé de Cours	En Poste
32-**MBONG A.M.M.**	Chargé de Cours	En Poste
33-**MOFOR T. C.**	Chargé de Cours	Insp Chef Serv. MINESUP
34-**PACHANGOU**	Chargé de Cours	En Poste
35-**PALMER M. N.**	Chargé de Cours	En Poste
36-**TCHANA K.A.**	Chargé de Cours	En Poste
37-**MBOUCHE F.M.**	Assistant	En Poste
II - Département de Biologie et Physio. Animales (BPA) (43)		
1-**BILONG B. C.F.**	Professeur	Chef de Dept.
2-**DIMO Théophile**	Professeur	En Poste
3-**DJIETO Lordon C.**	Professeur	En Poste
4-**ESSOMBA N.M.**	Professeur	VDoyen/FMSB/UYI
5-**FOMENA A.**	Professeur	En Poste
6-**KAMGANG R.**	Professeur	C.S. MINRESI
7-**KAMTCHOUING**	Professeur	En Poste
8-**NJAMEN D**	Professeur	En Poste
9-**NJIOKOU Flobert**	Professeur	En Poste
10-**NOLA Moïse**	Professeur	En Poste

11-**TAN Paul**	Professeur	En Poste
12-**TCHUEM T.L.**	Professeur	Coord. Progr. MIN-SANTE
13-**AJEAGAH G.A.**	Maître de Conférences	VICE-DOYEN / DSSE
14-**DZEUFIET D.P.**	Maître de Conférences	En Poste
15-**FOTO M. S.**	Maître de Conférences	En Poste
16-**JATSA M.H.**	Maître de Conférences	En Poste
17-**KEKEUNOU S.**	Maître de Conférences	En Poste
18-**MEGNEKOU R.**	Maître de Conférences	En Poste
19-**MONY N. R.**	Maître de Conférences	En Poste
20-**NGUEGUIM T.**	Maître de Conférences	En Poste
21-**TOMBI J.**	Maître de Conférences	En Poste
22-**ZEBAZE T.S.H.**	Maître de Conférences	En Poste
23-**ALENE D.C.**	Chargé de Cours	En Poste
24-**ATSAMO A.D.**	Chargé de Cours	En Poste
25-**BELLET E. O.R.**	Chargé de Cours	En Poste
26-**BILANDA D.C.**	Chargé de Cours	En Poste
27-**DJIOGUE Séfirin**	Chargé de Cours	En Poste
28-**DONFACK M.**	Chargé de Cours	En Poste
29-**GOUNOUE K. R.**	Chargé de Cours	En Poste
30-**KANDEDA K. A.**	Chargé de Cours	En Poste
31-**LEKEUFACK F.**	Chargé de Cours	En Poste
32-**MAHOB R.J.**	Chargé de Cours	En Poste
33-**MBENOUN M.P.**	Chargé de Cours	En Poste
34-**MOUNGANG N.**	Chargé de Cours	En Poste
35-**MVEYO N.Y.P.**	Chargé de Cours	En Poste
36-**NGOUATEU K.**	Chargé de Cours	En Poste
37-**NGUEMBOK**	Chargé de Cours	En Poste
38-**NJUA YAFI C.**	Chargé de Cours	Chef Div. UBA
39-**NOAH E. O. V.**	Chargé de Cours	En Poste
40-**TADU Zéphirin**	Chargé de Cours	En Poste
41-**YEDE**	Chargé de Cours	En Poste
42-**ETEME E. S.**	Assistant	En Poste
43-**KOGA M. D.**	Assistant	En Poste
III - Département de Biologie et Physio. Végétale (BPV) (26)		
1-**AMBANG Zachée**	MProfesseur	Chef Division/UYII
2-**BELL JOSEPH M.**	Professeur	En poste
3-**MOSSEBO D.C.**	Professeur	En Poste
4-**YOUMBI E.**	Professeur	Chef Dépt.
5-**ZAPFACK Louis**	Professeur	En Poste
6-**ANGONI H.**	Maître de Conférences	E n Poste

7-**BIYE ELVIRE H.**	Maître de Conférences	En Poste
8-**DJOCGOUE P.F.**	Maître de Conférences	En Poste
9-**KENGNE N.I.M.**	Maître de Conférences	En Poste
10-**MALA A. W.**	Maître de Conférences	En Poste
11-**MBARGA B.M.**	Maître de Conférences	CT/UDs
12-**MBOLO Marie**	Maître de Conférences	En Poste
13-**NDONGO B.**	Maître de Conférences	CE/MINRESI
14-**NGONKEU M.E.**	Maître de Conférences	En Poste
15-**TSOATA Esaïe**	Maître de Conférences	En Poste
16-**GOMANDJE C.**	Chargé de Cours	En Poste
17-**MAFFO M.N.L.**	Chargé de Cours	En Poste
18-**MAHBOU S. G.**	Chargé de Cours	En Poste
19-**NGALLE B. H.**	Chargé de Cours	En Poste
20-**NGOUO Lucas V.**	Chargé de Cours	En Poste
21-**NOUKEU K.A.**	Chargé de Cours	En Poste
22-**ONANA J. M.**	Chargé de Cours	En Poste
23-**NSOM ép. Pial**	Chargé de Cours	Expert. Nat./UNESCO
24-**TONFACK L.B.**	Chargé de Cours	En Poste
25-**DJEUANI A.C.**	Assistant	En Poste
26-**NNANGA M.R.**	Assistant	En Poste
IV - Département de Chimie Inorganique (CI) (32)		
1-**AGWARA O.M.**	Professeur	Vice Rect. Univ Bda
2-**ELIMBI Antoine**	Professeur	En Poste
3-**MELO C. U. F.**	Professeur	Rect. Univ. Ngdere
4-**GHOGOMU M.P.**	Professeur	Ministre Chargé de Miss. PR
5-**NANSEU C.P.**	Professeur	En Poste
6-**NDIFON T. P.**	Professeur	CT MINRESI/Chef de Dept
7-**NDIKONTAR M.**	Professeur	Vice-Doyen/UBda
8-**NENWA Justin**	Professeur	En Poste
9-**NGAMENI E.**	Professeur	DOYEN FS UDs
10-**BABALE D. D.**	Maître de Conférences	Chargée Mission P.R.
11-**DJOUFAC W.E.**	Maître de Conférences	En Poste
12-**KAMGANG Y.G.**	Maître de Conférences	En Poste
13-**KEMMEGNE M.**	Maître de Conférences	En Poste
14-**KONG SAKEO**	Maître de Conférences	En Poste
15-**NGOMO H. M.**	Maître de Conférences	Vice Chancelor/UB
16-**NJIOMOU C.**	Maître de Conférences	En Poste
17-**NJOYA Dayirou**	Maître de Conférences	En Poste

18-**YOUNANG Elie**	Maître de Conférences	En Poste
19-**ACAYANKA Elie**	Chargé de Cours	En Poste
20-**BELIBI B.P.D.**	Chargé de Cours	CS/ ENS Bertoua
21-**CHEUMANI Y.**	Chargé de Cours	En Poste
22-**EMADACK A.**	Chargé de Cours	En Poste
23-**KENNE D. G.**	Chargé de Cours	En Poste
24-**KOUOTOU D.**	Chargé de Cours	En Poste
25-**MAKON T. B.**	Chargé de Cours	En Poste
26-**MBEY Jean A.**	Chargé de Cours	En Poste
27-**NCHIMI N. K.**	Chargé de Cours	En Poste
28-**NDI NSAMI J.**	Chargé de Cours	En Poste
29-**NEBA nee N.**	Chargé de Cours	Insp. de Ser. MINFEM
30-**NYAMEN L.D.**	Chargé de Cours	En Poste
31-**PABOUDAM G.**	Chargé de Cours	En Poste
32-**TCHAKOUTE K.**	Chargé de Cours	En Poste
V - Département de Chimie Organique (CO) (32)		
1-**DONGO Etienne**	Professeur	VD PSAA
2-**GHOGOMU T. R.**	Professeur	Dir. IBAF/UDS
3-**NGOUELA S.A.**	Professeur	En Poste
4-**NKENFACK A. E.**	Professeur	Chef de Dept
5-**NYASSE B.**	Professeur	Directeur/UN
6-**PEGNYEMB D.E.**	Professeur	Directeur/ MINESUP
7-**WANDJI Jean**	Professeur	En Poste
8-**Alex de T. A.**	Maître de Conférences	DEPE/ Rectorat/UYI
9-**EYONG O. K.**	Maître de Conférences	Chef Service DPER
10-**FOLEFOC G.N.**	Maître de Conférences	En poste
11-**KEUMEDJIO F.**	Maître de Conférences	En Poste
12-**KEUMOGNE M.**	Maître de Conférences	En Poste
13-**KOUAM Jacques**	Maître de Conférences	En Poste
14-**MBAZOA D.C.**	Maître de Conférences	En Poste
15-**MKOUNGA P.**	Maître de Conférences	En Poste
16-**NGO MBING J.**	Maître de Conférences	Sous/Direct. MINERESI
17-**NOUNGOUE T.**	Maître de Conférences	En Poste
18-**TABOPDA K. T.**	Maître de Conférences	En Poste
19-**TCHOUANKEU J**	Maître de Conférences	Doyen FS/UYI
20-**TIH née NGO B.**	Maître de Conférences	En Poste
21-**YANKEP E.**	Maître de Conférences	En Poste
22-**AMBASSA P.**	Chargé de Cours	En Poste
23-**FOTSO W. G.**	Chargé de Cours	En Poste

24-**KAMTO E.L.**	Chargé de Cours	En Poste
25-**MVOT A. C.**	Chargé de Cours	En Poste
26-**NGOMO Orléans**	Assistant	En Poste
27-**NGONO B.D.S.**	Chargé de Cours	En Poste
28-**NOTE L. O.**	Chargé de Cours	Chef Service/MINESUP
29-**OUAHOUO W.B.**	Chargé de Cours	En Poste
30-**TAGATSING F.**	Chargé de Cours	En Poste
31-**ZONDENDEGOUMBA**	Chargé de Cours	En Poste
32-**NGNINTEDO D.**	Assistant	En Poste
VI - Département d'Informatique (IN)(26)		
1-**ATSA ETOUNDI Roger**	Professeur	Chef Div. Minesup
2-**FOUDA NDJODO Marcel Laurent**	Professeur	C/D ENS/Chef IGA.MINESUP
3-**TCHUENTE Maurice**	Professeur	PCA Univ. Yaoundé II
4-**NDOUNDAM R.**	Maître de Conférences	En Poste
5-**ABESSOLO A.G.**	Chargé de Cours	En Poste
6-**AMINOU H.**	Chargé de Cours	Chef de Dept
7-**DJAM X.Y.**	Chargé de Cours	En Poste
8-**KAMGUEU P.O.**	Chargé de Cours	En Poste
9-**KOUOKAM K.E.**	Chargé de Cours	En Poste
10-**MELATAGIA Y.**	Chargé de Cours	En Poste
11-**MONTHE D.V.**	Chargé de Cours	En Poste
12-**MOTO M. S.A.**	Chargé de Cours	En Poste
13-**OLLE O.G.D**	Chargé de Cours	C/D Enset. Ebolowa
14-**TAPAMO K.H.M**	Chargé de Cours	En Poste
15-**TINDO Gilbert**	Chargé de Cours	En Poste
16-**TSOPZE Norbert**	Chargé de Cours	En Poste
17-**WAKU K. Jules**	Chargé de Cours	En Poste
18-**BAYEM J. N.**	Assistant	En Poste
19-**DOMGA K. R.**	Assistant	En Poste
20-**EBELE Serge C.**	Assistant	En Poste
21-**HAMZA Adamou**	Assistant	En Poste
22-**JIOMEKONG A.**	Assistant	En Poste
23-**KAMDEM K. C.**	Assistant	En Poste
24-**MAKEMBE S. O.**	Assistant	En Poste
25-**MEYEMDOU N.**	Assistant	En Poste
26-**NKONDOK M.**	Assistant	En Poste

VII - Département de Mathématique(MA) (39)		
1-**BITJONG N.**	Professeur	En Poste
2-**DOSSA C. M.**	Professeur	En Poste
3-**AYISSI Raoul D.**	Maître de Conférences	Chef de Dept
4-**EMVUDU W. Y.S.**	Maître de Conférences	CD Info/ Chef Div. MINESUP
5-**NKUIMI J. C.**	Maître de Conférences	En Poste
6-**NOUNDJEU P.**	Maître de Conférences	En Poste
7-**TCHAPNDA N.S.**	Maître de Conférences	Dir. AIMS Rwanda
8-**AGHOUKENG J.**	Chargé de Cours	Chef Cellule MINPLAMAT
9-**CHENDJOU G.**	Chargé de Cours	En Poste
10-**DJIADEU N. M.**	Chargé de Cours	En Poste
11-**DOUANLA Y. H.**	Chargé de Cours	En Poste
12-**FOMEKONG C.**	Chargé de Cours	En Poste
13-**KIANPI Maurice**	Chargé de Cours	En Poste
14-**KIKI M. A.**	Chargé de Cours	En Poste
15-**MBAKOP G. M.**	Chargé de Cours	En Poste
16-**MBANG Joseph**	Chargé de Cours	En Poste
17-**MBEHOU M.**	Chargé de Cours	En Poste
18-**MBELE B. M. L.**	Chargé de Cours	En Poste
19-**MENGUE M. D.**	Chargé de Cours	En Poste
20-**NGUEFACK B.**	Chargé de Cours	En Poste
21-**NIMPA P.R.**	Chargé de Cours	En Poste
22-**POLA D.E.**	Chargé de Cours	En Poste
23-**TAKAM SOH P.**	Chargé de Cours	En Poste
24-**TCHANGANG R**	Chargé de Cours	En Poste
25-**TCHOUNDJA E.**	Chargé de Cours	En Poste
26-**TETSADJIO T.**	Chargé de Cours	En Poste
27-**TIAYA T.N.A.**	Chargé de Cours	En Poste
28-**MBIAKOP H.G.**	Assistant	En Poste
VIII - Département de Microbiologie (MB) (12)		
1-**ESSIA N.J.J.**	Professeur	DRV/IMPM
2-**ETOA F. Xavier**	Professeur	Chef de Dpt/FS/UYI Recteur Univ. UDla
5-**BOYOMO Onana**	Chargé de Cours	En Poste
3-**NWAGA D.M.**	Maître de Conférences	En Poste
7-**NYEGUE M.A.**	Chargé de Cours	En Poste
8-**RIWOM Sara H.**	Chargé de Cours	En Poste
9-**SADO K.S.L.**	Chargé de Cours	En Poste

9-**ASSAM A. J. P.**	Chargé de Cours	En Poste
4-**BODA Maurice**	Chargé de Cours	En Poste
10-**BOUGNOM B.P.**	Chargé de Cours	En Poste
6-**ESSONO O. G.**	Chargé de Cours	En Poste
11-**NJIKI BIKOÏ**	Chargé de Cours	En Poste
12-**TCHIKOUA R.**	Chargé de Cours	En Poste
IX - Département de Physique (PH) (39)		
1-**BEN-BOLIE G.H.**	Professeur	En Poste
2-**ESSIMBI Z. B.**	Professeur	Vice-Doyen DRC/UY1
3-**KOFANE T.C.**	Professeur	En Poste
4-**NDJAKA J.M.B.**	Professeur	Chef de Dept
5-**NJANDJOCK N.**	Professeur	Sous Dir. MINRESI
6-**NJOMO Donatien**	Professeur	En Poste
7-**PEMHA Elkana**	Professeur	En Poste
8-**TABOD C.T.**	Professeur	Doyen/UBda
9-**TCHAWOUA C.**	Professeur	En Poste
10-**WOAFO Paul**	Professeur	En Poste
11-**BIYA MOTTO F.**	Maître de Conférences	DG/HYDRO Mekin
12-**BODO Bernard**	Maître de Conférences	En Poste
13-**DJUIDJE K.G.**	Maître de Conférences	En Poste
14-**EKOBENA F.H.**	Maître de Conférences	Chef Division. UN
15-**EYEBE F. J.S.**	Maître de Conférences	En Poste
16-**FEWO Serge I.**	Maître de Conférences	En Poste
17-**HONA Jacques**	Chargé de Cours	En Poste
18-**MBANE Biouele**	Maître de Conférences	En Poste
19-**NANA E. S. G.**	Maître de Conférences	Director Students/Aff. UB
20-**NANA N.B.**	Maître de Conférences	En Poste
21-**NOUAYOU R.**	Maître de Conférences	En Poste
22-**SAIDOU**	Maître de Conférences	Sous Directeur/Minresi
23-**SIEWE S. M.**	Maître de Conférences	En Poste
24-**SIMO Elie**	Maître de Conférences	En Poste
25-**VONDOU D.A.**	Maître de Conférences	En Poste
26-**WAKATA née B.**	Maître de Conférences	Sous Directeur/ MINESUP
27-**ZEKENG S.S.**	Maître de Conférences	En Poste
28-**ABDOURAHIMI**	Chargé de Cours	En Poste
29-**EDONGUE H.**	Chargé de Cours	En Poste
30-**ENYEGUE A N.**	Chargé de Cours	En Poste
31-**FOUEDJIO D.**	Chargé de Cours	Chef Cell. MINADER

32-**MBINACK C.**	Chargé de Cours	En Poste
33-**MBONO S.Y.C.**	Chargé de Cours	En Poste
34-**MELI J. L.**	Chargé de Cours	En Poste
35-**MVOGO A.**	Chargé de Cours	En Poste
36-**NDOP Joseph**	Chargé de Cours	En Poste
37-**OBOUNOU M.**	Chargé de Cours	DA/Univ Inter Etat/-Sangmalima
38-**WOULACHE R.**	Chargé de Cours	En Poste
39-**CHAMANI R.**	Chargé de Cours	En Poste
X - Département de Sciences de la Terre(ST) (43)		
1-**BITOM D.L.**	Professeur	Doyen FASA/UDs
2-**FOUATEU R.**	Professeur	En Poste
3-**KAMGANG P.**	Professeur	En Poste
4-**MEDJO EKO R.**	Professeur	CT UY2
5-**NDJIGUI P.D.**	Professeur	Chef de Dept
6-**NKOUMBOU C.**	Professeur	En Poste
7-**NZENTI J.-P.**	Professeur	En Poste
8-**ABOSSOLO A.M.**	Maître de Conférences	Vice-Doyen / DRC
9-**GHOGOMU R.T.**	Maître de Conférences	CD/UMa
10-**MOUNDI A.**	Maître de Conférences	CT/ MINIMDT
11-**NDAM N.J.R.**	Maître de Conférences	En Poste
12-**NGOS III Simon**	Maître de Conférences	DAAC/Uma
13-**NJILAH I.K.**	Maître de Conférences	En Poste
14-**ONANA Vincent**	Maître de Conférences	En Poste
15-**BISSO Dieudonné**	Maître de Conférences	Dir. P. Barage Memvelé
16-**EKOMANE E.**	Maître de Conférences	En Poste
17-**GANNO S.**	Maître de Conférences	En Poste
18-**NYECK Bruno**	Maître de Conférences	En Poste
19-**TCHOUANKOUE**	Maître de Conférences	En Poste
20-**TEMDJIM R.**	Maître de Conférences	En Poste
21-**YENE A.J.Q.**	Maître de Conférences	Chef Div. /MINTP
22-**ZO'O ZAME P.**	Maître de Conférences	DG/ART
23-**ANABA O. A. B.**	Chargé de Cours	En Poste
24-**BEKOA Etienne**	Chargé de Cours	En Poste
25-**ELISE SABABA**	Chargé de Cours	En Poste
26-**ESSONO Jean**	Chargé de Cours	En Poste
27-**EYONG J. T.**	Chargé de Cours	En Poste
28-**FUH C. G.y**	Chargé de Cours	Sec. D'Etat/MINMIDT
29-**LAMILEN B.D.**	Chargé de Cours	En Poste
30-**MBESSE C. O.**	Chargé de Cours	En Poste

31-**MBIDA YEM**	Chargé de Cours	En Poste
32-**METANG Victor**	Chargé de Cours	En Poste
33-**MINYEM D.L.**	Chargé de Cours	CD/Uma
34-**MOUAFO Lucas**	Chargé de Cours	En Poste
35-**NGO B.R.N.**	Chargé de Cours	En Poste
36-**NGO B. L.M.**	Chargé de Cours	En Poste
37-**NGUETCHOUA**	Chargé de Cours	CEA/MINRESI
38-**NOMO N. E.**	Chargé de Cours	En Poste
39-**NTSAMA A. J.**	Chargé de Cours	En Poste
40-**TCHAKOUNTE**	Chargé de Cours	Chef.cell/MINRESI
41-**TCHAPTCHET**	Chargé de Cours	En Poste
42-**TEHNA N.**	Chargé de Cours	En Poste
43-**TEMGA J.P.**	Chargé de Cours	En Poste

Répartition Chiffrée Des Enseignants Permanents Par Département (19 Février 2019)

Department	**Nombre d'enseignants**				
	Pr	**MC**	**CC**	**ASS**	**Total**
BCH	5 (1)	12 (6)	19 (11)	1 (1)	37 (19)
BPA	12 (1)	10 (5)	20 (07)	2 (0)	44 (13)
BPV	5 (0)	10 (2)	9 (04)	2 (2)	26 (9)
CI	9 (1)	9 (2)	14 (3)	0 (0)	32 (6)
CO	7 (0)	14 (4)	10 (4)	1 (0)	32 (8)
IN	2 (0)	1 (0)	13 (0)	10 (3)	26 (3)
MAT	2 (0)	4 (1)	19 (1)	2 (0)	27 (2)
MIB	1 (0)	5 (2)	5 (1)	0 (0)	12 (3)
PHY	10 (0)	17 (2)	11 (3)	1 (0)	39 (5)
ST	7 (1)	15 (1)	21 (5)	1 (0)	43 (7)
Total	**61 (4)**	**97 (25)**	**141 (39)**	**19 (6)**	**318 (75)**

- **Pr**=Professeur : 61 (4)
- **MC**=Maître de Conférences : 97 (25)
- **CC**=Chargé de Cours : 141 (39)
- **ASS**=Assistant : 19 (6)
- (·)=Nombre de femmes.

Dédicaces

Je dédie ce travail à :

- ma maman Marie PÉDATÉ MEMONG
- mon épouse Rachel KENNE, mes enfants Michelle Angie, Océanne Elsa et Joseph Eraste
- mes oncles et tantes, frères et sœurs, cousins et cousines
- mon feu papa Élie PÉDATÉ, dit NDENOKA

Remerciements

Il est primordial de préciser qu'un travail de thèse ne s'accomplit jamais de manière individuelle, mais grâce à l'aide précieuse de plusieurs personnes, qui, chacune dans son domaine et à sa manière, contribue à l'aboutissement du projet. Que ce soit par leur savoir-faire, leur aide technique, leurs conseils, leur expérience, leur vision de la recherche, ou, tout simplement, leur soutien moral, leur amitié ou leur amour, toutes ces personnes méritent d'être chaleureusement remerciées, et c'est ce que j'essaie de faire dans les lignes qui suivent.

Le travail présenté dans cette thèse a été effectué dans le cadre du laboratoire LIRIMA [1], équipe IDASCO [2], au département d'Informatique de l'Université de Yaoundé 1 (UY1), en collaboration avec l'équipe RMS [3] du laboratoire TIMA [4] de l'Université Grenoble Alpes (UGA), sous l'encadrement conjoint du **Professeur Emmanuel SIMEU**, enseignant-chercheur à l'Institut Polytechnique de Grenoble (Grenoble INP) de L'UGA, et du **Professeur Maurice TCHUENTE**, enseignant-chercheur à l'UY1.

1. Laboratoire International de Recherche en Informatique et Mathématiques Appliquées
2. Informatique Distribuée pour l'Analyse des Systèmes Complexes
3. Reliable Mixed signal Systems
4. Techniques de l'Informatiques de la Microélectronique pour l'Architecture des systèmes intégrés

Remerciements particuliers

C'est de Dieu que viennent la vie, la santé, la sécurité, la paix , ainsi que la sagesse et l'intelligence ***(Bible LSG/Proverbes 2, 6)***. C'est également de Lui que viennent le vouloir et le faire ***(Philippiens 2, 13)***. Il est créateur de toutes choses existantes et toutes ses œuvre louent Son Nom. Il n'y a point d'erreur en Lui.

Moi aussi je rends d'innombrables actions de grâce au Rois des rois pour ses bontés et sa miséricorde qu'Il renouvelle chaque jour envers moi. C'est selon son plaisir que ce travail a été effectif. Il m'a entouré de personnes appropriées pour qu'ensemble, nous menions à bout ce joyaux intellectuel.

« *. . . La louange, la gloire, la sagesse, l'action de grâces, l'honneur, la puissance, et la force, soient à notre Dieu, aux siècles des siècles ! Amen !* » ***(Apocalypse 7, 12)***.

Remerciements

Je voudrais remercier de prime abord M. Emmanuel SIMEU, Maître de Conférences à l'UGA, Directeur de l'équipe RMS du laboratoire TIMA, qui a défini et proposé ce sujet de recherche. Je le remercie profondément pour sa grande sollicitude et sa patience. Il n'a ménagé aucun effort pour donner des orientations et procéder au recadrage de ce travail chaque fois que nécessaire. Pour tout cela, je lui suis infiniment reconnaissant.

Mes remerciements vont intimement à l'endroit de M. Maurice TCHUENTE, Professeur à l'UY1, qui a accepté co-diriger mes travaux de recherche, depuis le Master jusqu'à la thèse. Il a consacré du temps et consenti d'énormes efforts, avec des contacts importants, pour que je puisse mener ce travail à bout. Sa minutie, sa patience et ses encouragements ont été déterminants pour l'aboutissement de ce travail houleux.

Toute ma gratitude va également à l'endroit des membres du jury, pour avoir accepté de faire partie du jury de ma soutenance de thèse.

Je remercie M. Rshdee ALHAKIM, ancien postdoc du laboratoire TIMA, pour sa précieuse contribution dans le cadre de ce travail. C'est lui qui a débuté le développement de l'environnement *GrafcetConverter* dans le cadre de l'encadrement du stage d'ingénieur de conception de Michael LANDOT, étudiant polytechnicien. C'est cet environnement qui sera modifié et étendu par nous pour réaliser la synthèse multi-cibles du Grafcet par matrices de codage. En outre, c'est grâce au soutien de Rshdee que j'ai pu prendre en main la programmation du microcontrôleur *dsPIC30F4013* qu'équipait la carte *EASYdsPIC4A*. Tout ceci est inoubliable.

Je remercie M. Serge STINCKWHICH, Directeur de recherche à l'IRD/UMMISCO[5]. Grâce aux séminaires qu'il a organisés sur la méta-programmation et la méta-modélisation, j'ai pu appréhender la philosophie IDM. Je lui suis reconnaissant pour sa disponibilité et de nombreuses échanges chaque fois constructives que j'ai eues avec lui. Il était assisté par son collaborateur M. Nick PAPOULIAS à qui mes remerciement s'adressent en même temps. Ses orientations m'ont en outre permis de réaliser le profilage des codes générés pour des cibles microcontrôleurs.

Je remercie M. Valery M. MONTHE D. , Chargé de cours à l'Université de Yaoundé 1, pour son expertise qu'il a mise à ma disposition pour mieux appréhender l'approche IDM. Son initiation et ses encouragements m'ont permis de m'en imprégner et de résoudre l'épineux problème de la synthèse multi-cibles IDM sur microcontrôleurs.

Je remercie les membres du laboratoire LIRIMA et ceux de l'équipe IDASCO en particulier, pour leur collaboration.

5. Unité de Modélisation Mathématiques et Informatique des Systèmes Complexes

Toute ma gratitude va également à l'endroit de l'Université de Yaoundé I qui a disposé des ressources matérielles et humaines efficaces pour mon éducation en son sein depuis des années. Le dynamisme du Département d'Informatique de la Faculté des Sciences de cette institution est expressive à travers des enseignants qualifiés et disponibles que je remercie vivement.

Je remercie mes collègues du Master IMA, et en particulier mes collaborateurs Aboubakar SIDIKI, Clive EPAH, Césaire KUETE et Casimir SOFEU pour leurs soutiens et constants encouragements.

J'exprime ma gratitude à tous les membres de ma famille, mes oncles et tantes, mes cousins et cousines, qui ont su me soutenir dans les moments difficiles et me motiver par leur affection. Un merci particulier à maman Élise PÉDATÉ MATANG, à mes frères et sœurs Sylvie K., Francis F., Norbert F., Isabelle M., Bruno AY., Adèle T., Bénédicte F., Dorette M., Débora C., Merveille, mes cousins Jean FONKOENG et Jonas PANQUIPHIRI. Il en est de même de Salomon K., Odile T., Esther K., Clément T., Pascal K. et Desmond T. Je pense aussi et surtout à Michelle S. K., Émilie A., Patrick J.J., Pierre K., Victor D., Crescence D., Guy R. K., Elie N., Roddy K. et Blaise N.

Je remercie également mes frères et sœurs en Christ, de près comme de loin, pour leur assistance et leur encadrement à travers de nombreuses prières formulées à Dieu.

Je ne saurai terminer sans remercier tous ceux qui de près ou de loin ont contribué à la réalisation de ce travail. Je ne saurais vous énumérer tous, car vous êtes assez nombreux.

Résumé

Les systèmes embarqués sont omniprésents dans les systèmes modernes et ont en leur cœur un système de contrôle-commande (SCC) dont la conception nécessite précision et minutie. Ces SCCs peuvent être spécifiés à l'aide de plusieurs langages standards dont le Grafcet (CEI 60848), devenu un formalisme international très utilisé. En tant que langage formel de spécification, la mise en œuvre manuelle du Grafcet est sujette à des erreurs et coûteuse en temps, d'où l'intérêt des méthodes automatiques de génération du code à partir de modèles Grafcet. Généralement, les chercheurs proposent de transformer le Grafcet en code PLC du standard CEI 61131-3, ce qui rend onéreuse la solution de contrôle du fait du coût élevé des PLCs. Cependant, le développement continu en ingénierie du logiciel facilite l'émergence et la prolifération rapide d'une grande variété de processeurs à faible coût pour l'exécution de programmes dans des applications embarquées complexes. Ainsi, pour réduire le coût de la solution de contrôle, l'utilisation des microprocesseurs peut être préférée aux PLCs ordinaires dans certaines applications embarquées. La principale difficulté réside alors dans la production du logiciel de contrôle pour une cible de microprocesseur choisie, étant donnée la grande diversité des microcontrôleurs existants. Notre objectif est donc d'exploiter le langage Grafcet pour proposer un environnement de programmation des SCCs basés sur les microcontrôleurs. Pour cela, il importe de trouver une modélisation prenant en compte tous les aspects du Grafcet, ainsi que les principales caractéristiques des microcontrôleurs ciblés. Ensuite, il est question de procéder par raffinements successifs de niveaux d'abstraction partant d'un modèle Grafcet pour aboutir à une implémentation sur le microcontrôleur pris en entrée. Dans un premier temps, nous avons proposé un modèle matriciel du Grafcet, ainsi qu'une représentation des caractéristiques des cibles microcontrôleurs, lesquels ont été mis en œuvre à travers un environnement de synthèse multi-cibles. Cet environnement combine la génération de code d'interpréteurs de modèles Grafcet principalement basée sur l'algorithme d'interprétation avec la génération de code par équations algébriques du Grafcet. Un profilage des codes générés en Arduino est réalisé sur le microcontrôleur *ATmega328P* au terme duquel la durée estimée du cycle de scrutation des programmes générés est de l'ordre de 10 ms. Cependant, cette première approche a présenté des limites

principalement tributaires à la prise en compte exhaustive des éléments du Grafcet dans les matrices de codage, et la mise en œuvre dans un environnement généraliste. On peut noter un manque de généricité, des difficultés de maintenance et d'extension des solutions de synthèse. Pour cela, nous avons réalisé une synthèse multi-cibles du Grafcet par approche IDM, basée sur un métamodèle de Grafcet proposé et un métamodèle des microcontrôleurs, lequel capture toutes les caractéristiques utiles à la génération de code. Les instances de ces deux métamodèles ont ensuite servi en entrée du processus de transformation, formalisé par des règles de transformation et réalisé pour la génération du code de contrôle. Pour simplifier et faciliter l'édition des modèles Grafcet valides, les expressions Grafcet ont été étudiées et formalisées à travers une grammaire algébrique hors-contexte, dont le parseur obtenu est intégré aux processus d'édition des modèles et de génération de code. Pour la mise en œuvre de l'approche, nous avons développé une plateforme basée sur *Éclipse EMF*, le générateur d'analyseurs syntaxiques *ANTLR*, le langage *OCL* et le moteur de génération de code *Acceleo*. Initialement, la validation des solutions proposées s'est faite par simulation et par expérimentation sur la réalisation d'un contrôleur de feux de signalisation à un carrefour. Le contrôleur utilisé à cet effet est basé sur le microcontrôleur *EASYdsPIC4A*, alors que l'application est émulée à l'aide d'une carte électronique d'extension. Pour finir, nous avons réalisé une étude de cas sur un système d'approvisionnement autonome en eau domestique au cœur duquel a été développé et implémenté un modèle de commutation des sources d'énergie. Après génération du code Arduino correspondant par approche IDM, il a été mis en œuvre sur le microcontrôleur *Atmega1280*. Il en ressort que par rapport à l'approche de synthèse par codage matriciel du Grafcet, l'approche IDM présente de nombreux avantages: pas besoin de calculer les matrices de codage, possibilité de vérifier la conformité du modèle au langage Grafcet et grande flexibilité dans la représentation des expressions du modèle Grafcet. Il existe des pistes d'amélioration du présent travail dont voici quelques une identifiées : la prise en compte des structures hiérarchiques du Grafcet dans la modélisation, la réalisation d'un éditeur graphique de Grafcet basé sur le métamodèle proposé et la génération de code pour des microcontrôleurs pouvant fonctionner dans des environnements perturbés.

Mots clés : système embarqué, microcontrôleur, Grafcet, multi-cibles, Ingénierie Dirigée par les Modèles, vérification de modèle, génération de code.

Abstract

Embedded systems are ubiquitous in modern systems and have a built-in logic controller that requires precision and meticulous design. These SCCs can be specified using several standard languages including Grafcet (IEC 60848), which has become a widely used international formalism. As a formal specification language, the manual implementation of the Grafcet is prone to errors and costly in time, hence the interest of automatic methods of generating code from Grafcet models.. Generally, the researchers propose to transform the Grafcet into PLC code of IEC 61131-3 standard, which makes the control solution still expensive because of the high cost of the PLCs. However, the continuous development in software engineering facilitates the emergence and rapid proliferation of a wide variety of low-cost processors for running programs in complex embedded applications. Thus, to reduce the cost of the control solution, the use of microprocessors may be preferred to ordinary PLCs in some embedded applications. The main difficulty lies in the production of the control software for a chosen microprocessor target, given the wide variety of existing microcontrollers. Our goal is to use the Grafcet language to provide a programming environment for microcontroller based logic controllers. For that, it is important to find a modeling taking into account all the aspects of the Grafcet, as well as the main characteristics of the microcontrollers targeted. Then, it will be a question of proceeding by successive refinements of abstraction levels starting from a Grafcet model to result in an implementation on a chosen microcontroller. First, we proposed a Grafcet matrix model, as well as a representation of the characteristics of the microcontroller targets, which were implemented through a multi-target synthesis environment. This environment combines the Grafcet model interpreter code generation mainly based on the interpretation algorithm with Grafcet code generation, according to the model of Grafcet algebraic equations. A profiling of the codes generated in Arduino is carried out on the ATmega328P microcontroller at the end of which the estimated duration of the scanning cycle of the programs generated is of the order of 10 ms. However, this first solution presented limits mainly dependent on the exhaustive taking into account of Grafcet elements in a matrix code and the implementation in a general environment. It is about the lack of genericity, the difficulties of maintenance and extension of the

solutions of synthesis. For this, we carried out a Grafcet multi-target synthesis by IDM approach, based on a proposed Grafcet metamodel and a metamodel of microcontrollers, which captures all the characteristics useful for code generation. The instances of these two metamodels were then used as inputs of the transformation process (formalized by transformation rules) performed for the generation of the control code. To make it easier to edit Grafcet models, Grafcet expressions have been studied and formalized through a free-context algebraic grammar, whose parser is integrated into the model editing and code generation process. For the implementation of the approach, we have developed a platform based on Eclipse EMF, the generator of ANTLR parsers, the OCL language and the Acceleo code generation engine. Initially, the validation of the proposed solutions was done by simulation and experimentation on the realization of a traffic light controller at a crossroads. The controller is then based on the EASYdsPIC4A microcontroller and the application is emulated using an expansion card. Finally, we carried out a case study on an autonomous domestic water supply system at the heart of which was developed and realized a switching model of energy sources, implemented on the Atmega1280 microcontroller after the generation of the corresponding Arduino code. It emerges that compared to Grafcet's matrix-based synthesis approach, the IDM approach has many advantages: no need to calculate the encoding matrices, possibility to check the conformity of the model to the Grafcet language and great flexibility in the representation of Grafcet model expressions. However, there are ways to improve this work, here are some of them identified: taking into account Grafcet's hierarchical structures in modeling, the production of a Grafcet graphic editor based on the proposed metamodel and the code generation for microcontrollers that can operate in disturbed environments.

Keywords : embedded system, microcontroller, Grafcet, multi-target, Model Driven Engineering, model verification, code generation.

Table des matières

Table des figures

Liste des tableaux

Liste des Abréviations

μC	Microcontrôleur
ADEPA	Agence pour le Développement de la Productique Appliquée à l'industrie
AP	Automate Programmable
API	Automate Programmable Industriel
AST	Arbre de Syntaxe Abstraite
CEI	Commisssion Electrotechnique Internationale
CLP	Contrôleur Logique Programmable
DOM	Document Object Model
DSL	Domain Specific Language
DSML	Domain Specific Modeling Language
EBNF	Extended Backus-Naur Form
EDI	Environnement de Développement Intégré
EMF	Eclipse Modeling Framework
FE	Falling Edge (Front descendant)
FPGA	Field-Programmable Gate Array
GMF	Graphical Modeling Framework
GPML	General Purpose Modeling Language
GSA	Graphe de Situations Acessibles
IDM	Ingénierie Dirigée par les Modèles
IEC	International Electrotechnical Commisssion
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MOF	Meta Object Facility
OMG	Object Management Group
PDM	Platform Description Model
PIM	Platform Independent Model
PLC	Programmable Logic Controller
PSM	Platform Specific Model
QVT	Query View Transformation

RdP	Réseaux de Petri
RE	Rising Edge (Front montant)
RPN	Reverse Polish Notation
SCC	Système De Contrôle Commande
SE	Système Embarqué
SED	Système à Evénements Discrets
SFC	Sequential Function Chart
SIA	Sequential based Interpretation Algorithm
SIPN	Singnal Interpreted Petri Net
SPL	Software Product Line
SR	Scan Rate (PLC)
ST	Scan Time (PLC)
XML	Extensible Markup Language

Introduction générale

Un système embarqué (SE) est une association entre le matériel électronique et le logiciel, conçu pour réaliser une fonction dédiée [2]. Il constitue un sous-système enfoui dans un système plus large avec lequel il est interfacé, et pour lequel il réalise des fonctions particulières de contrôle. Les systèmes embarqués sont devenus omniprésents dans notre environnement et jouent un rôle déterminant dans l'innovation industrielle de secteurs aussi divers que l'automobile, la médecine, le spatial, l'armée, la téléphonie, l'aéronautique, la domotique ... et occupent une place prépondérante dans notre vie quotidienne. Un SE a en son sein un système de contrôle-commande (SCC) (*partie commande*) qui a pour rôle de recevoir des signaux représentant l'état de son environnement (*partie opérative*) et d'en envoyer d'autres qui représentent des ordres. Ce fonctionnement est propre aux systèmes à événements discrets (SED).

Avec l'avènement de l'électronique numérique, la mise en œuvre des SCCs nécessite l'usage de contrôleurs qui sont généralement réalisés à base de circuits dédiés ASIC (Application Specific Integrated Circuit) ou d'automates programmables (AP), en anglais Programmable Logic Controller (PLC). Les ASICs sont des solutions matérielles et logicielles développées pour répondre à des besoins précis de contrôle/commande pour une application particulière bien définie. Cette approche est particulièrement adaptée pour des équipements produits en très grands volumes pour lesquels le coût important de l'étude et du développement des modules de contrôle/commande est absorbé dans le volume de production. Cependant, la modularité des automates programmables les rend particulièrement adaptés pour un déploiement sur des applications en exemplaires uniques ou de faible volume. Mais le coût élevé des PLCs industriels reste un frein significatif à leur utilisation pour solutionner les problèmes de contrôle/commande dans des systèmes embarqués à bas coût pour des applications grand public.

Par ailleurs, le développement rapide de la technologie des systèmes intégrés et l'émergence de la nanotechnologie ont rendu disponible et accessible à faible coût une grande variété de microcontrôleurs qui offrent une solution idoine pour intégrer à bas coût des systèmes de contrôle/commande complexes dans des systèmes embarqués pour des applications grand public.

Contrairement aux PLCs qui disposent de standards en termes de langages de programmation (IEC 61131-3 [1]) ainsi que des environnements de développement propriétaires (UNITY, SoMachine, RSLOGIX 5000, Simatic, ... par exemple), les microcontrôleurs existent dans une très grande diversité, avec des langages propriétaires de bas-niveau développés par les fabricants, allant de l'Assembleur au C, avec ses dérivés. Or, malgré les efforts déployés par les fabricants de microcontrôleurs pour proposer des langages de haut-niveau standardisés, le développement des logiciels embarqués basés sur la technologie des microcontrôleurs reste une affaire d'ingénieurs initiés qui doivent à cet effet déployer d'énormes efforts, avec un temps de production conséquent, ce qui accroit considérablement les coûts.

La conception et le développement d'un environnement logiciel permettant de faciliter la spécification des applications de contrôle/commande et leur synthèse sur des cibles microcontrôleurs devient un défi stratégique de premier plan pour les systèmes embarqués à bas coût. Le problème abordé dans ce travail est de trouver une démarche qui, partant d'une description de haut niveau - à l'instar des modèles Grafcet - de l'application et d'une spécification du microcontrôleur cible, permet de générer le code de contrôle de l'application pour le microcontrôleur cible spécifié en entrée.

Pour aborder ce problème, certains ont proposé de partir d'un langage de spécification de haut niveau et graphique, obtenu par extension des réseaux de Petri comme le SIPN [27] et le Grafcet [4, 9]. C'est ainsi que le système étudié dans [27] prend en entrée un modèle en langage SIPN pour générer un code en langage assembleur du microcontrôleur PIC16F84, alors que celles proposées en [9] et [44] utilisent un modèle en langage Grafcet pour générer un code C dédié aux microcontrôleurs C-programmés. Cependant, le langage SIPN utilisé dans la première approche se sert d'un langage d'entrée (le SIPN) qui n'est pas facile d'usage, contrairement au langage Grafcet qui est graphique et très utilisé par les automaticiens du fait de sa lisibilité [2, 27, 55].

Conçu depuis 1977 par l'Association française pour la cybernétique économique, le Grafcet (standard CEI 60848 [16]) est un langage graphique de spécification sous forme de diagrammes fonctionnels en séquence. Il va bénéficier de nombreux travaux consacrés à la formalisation de sa sémantique. Après R. David [19], F. Charbonnier et al. [12] prouvent la contrôlabilité d'un système spécifié par Grafcet en construisant l'automate fini correspondant à l'aide duquel peut être justifié aisément l'accessibilité de tous les états. Jean-Marc Roussel et al. [53] ainsi que J. provost et al. [50] traitent par ailleurs de la vérification et la validation des modèles spécifiés en Grafcet. De même, les travaux de P. Le Parc et al. [36], de F. Schumacher et al. [59, 56] vont contribuer à la sémantique formelle des aspects hiérarchiques et temporels du Grafcet. Dans la plupart des cas, c'est les RdPs duquel est inspiré le Grafcet qui sert de cadre d'étude du langage (après transformation des modèles Grafcet en RdP).

Initialement, le Grafcet vise la spécification des SCCs. Mais en tant

qu'outil de spécification formelle, la simplicité et la généricité de ce langage vont motiver de nombreux travaux qui œvrent pour en faire un langage de programmation. Il y a eu la mise en œuvre manuelle du Grafcet [58, 32] en utilisant principalement les équations algébriques Grafcet [37], ce qui est coûteux en temps et sujet à de nombreuses erreurs. Des méthodes [58, 57] seront alors proposées pour transformer automatiquement les modèles Grafcet en code PLC (CEI 61131-3 [1]). Comme dans ce premier essai, le code est généré en langage SFC et le grafcet pris en entrée est dépourvu d'aspects structurels, F. Schumacher et al. vont proposer une amélioration [32] où le code est généré en langage ST (PLC), laquelle facilite la maintenance et l'évolution de la solution. Outre la génération du code en langages PLC, certains travaux se sont penchés sur la génération du code microcontrôleur. Carla Ferreira et al. [25] présentent une méthode de génération automatique de code C ou Palasm (circuit PLD) pour des contrôleurs spécifiés en Grafcet. C'est aussi le cas de Bayó-Puxan et al. [4] qui proposent une méthodologie de compilation du Grafcet pour les microcontrôleurs programmés en C.

De toutes les solutions existantes sur la transformation du Grafcet en code, il ressort que la synthèse multi-cibles sur microcontrôleur n'a pas été abordée. Les caractéristiques des cibles microcontrôleurs ne sont donc pas prises en compte, et les règles de transformation en code ne sont pas présentées. On note aussi que les dernières méthodes ne tiennent pas compte de plusieurs aspects du langage Grafcet pourtant importantes pour l'expressivité des modèles Grafcet, en l'occurrence des contraintes temporelles, des événements front montant/descendant, la multiplicité et la typologie des actions associées aux étapes, ainsi que la vérification de modèles avant la génération de code.

Toutes ces approches adressent des cibles spécifiques et le code généré nécessite de nombreuses modifications pour fonctionner sur d'autres microcontrôleurs, puisque n'étant pas pris en compte dans la modélisation.

Comme le Grafcet est un langage de modélisation spécifique à un domaine[6], nous proposons ici d'utiliser les méthodes et outils de l'Ingénierie Dirigée par les Modèles (IDM) [55, 31] pour en faire un langage dédié de modélisation (DSML). Celui-ci permettra de construire la spécification Grafcet du contrôleur d'un système sous forme de modèle, de procéder à sa vérification, puis, une fois valide, cette spécification sera transformée automatiquement en un code exécutable sur la cible microcontrôleur choisie en entrée. Puisqu'en IDM «tout est modèle» et qu'on se sert d'un outillage appelé *métamodèle* ou *syntaxe abstraite* du DSML pour spécifier des modèles de modèles, il est nécessaire de proposer un métamodèle pour la synthèse du Grafcet.

Le choix de la cible doit alors être optionnel pour l'utilisateur, lequel n'a pas besoin de connaissances en programmation des microcontrôleurs. Un modèle de description de la cible (métamodèle microcontrôleur) devra

6. Celui des automaticiens qui développent les SCCs

être proposé et la synthèse se fera alors automatiquement par raffinement successif de niveaux d'abstractions pour aboutir à l'artefact logiciel qu'est le code. Les langages d'entrée sont décrits formellement à l'aide de méta-modèles, tandis que le processus de synthèse est décrit à l'aide de règles de transformations. Tous ces outils sont supportés par les environnements IDM qui offrent aussi des formats standards pour la représentation et la persistance des modèles.

Le plan de la thèse se présente comme suit :

Le **chapitre** 1 donne un **État de l'art sur la mise en œuvre des SCCs**. Il fait ressortir dans un premier temps la définition des concepts clés du domaine, la description du standard IEC 60848 (Grafcet) et la présentation des contrôleurs programmables. Dans un second temps, il est présenté les principales solutions existantes et leurs limites au regard de la mise en œuvre multi-cibles microcontrôleurs des SCCs à partir du langage de spécification fonctionnelle de haut-niveau Grafcet.

Au **chapitre** 2, il s'agit de la **Génération de code grafcet multi-cibles par matrices de codage**. Ici, un modèle dit *modèle matriciel du Grafcet* est présenté, exprimant une représentation aussi bien de la structure statique du Grafcet que sa dynamique d'évolution. Toute synthèse à travers ce modèle mathématique garantit le critère d'indépendance du modèle vis-à-vis des plateformes cibles. Ensuite, une mise en œuvre de ce modèle est faite à travers une plateforme de synthèse développée dans un environnement généraliste (non IDM). En plus de la simulation de modèles proposée dans la plateforme de synthèse, une application sur les feux de signalisation est faite en guise de validation de l'approche. Le profilage des programmes générés est enfin réalisé grâce au calcul de la durée du cycle de scrutation à l'aide du microcontrôleur *Atmega328P* qu'équipe la carte *Arduino UNO*.

Quant au **chapitre** 3, il s'intéresse à la **Métamodelisation Grafcet et la prise en compte IDM des expressions**. Cette approche vient pallier aux difficultés rencontrées lors de la mise en œuvre de la solution par matrices de codage dans un environnement généraliste de programmation. Une étude des concepts Grafcet est alors réalisée avec une formalisation sous forme d'un métamodèle Grafcet. Dans cette syntaxe abstraite Grafcet, un accent particulier est mis sur les expressions arithmétiques et/ou logiques Grafcet pour lesquelles une formalisation sous la forme de grammaire algébrique est proposée, mise en œuvre et intégrée à la plateforme de synthèse développée. Pour garantir la cohérence et la conformité des modèles Grafcet édités, des règles sont énoncées et formalisées au sein du métamodèle Grafcet à l'aide du langage *OCL*.

Le **chapitre** 4 porte sur la **Génération de code Grafcet multi-cibles par approche IDM pour microcontrôleurs C-programmés**. Dans un premier temps, après avoir analysé les microcontrôleurs C-programmés, nous proposons un langage pour leur description. Il est formalisé à travers un métamodèle qui capture l'ensemble des concepts clés relatifs à l'architecture de base du microcontrôleur et les éléments du langage de programmation

C utiles à la génération du code. Dans un second temps, nous décrivons la transformation de modèles qui réalise la génération de code multi-cibles. Elle prend en entrée le modèle Grafcet qui représente le comportement du contrôleur ainsi qu'un modèle du microcontrôleur choisi pour générer un code C valable pour cette cible. Certaines conditions (ou *guards*) doivent être remplies sur les deux modèles en entrée pour qu'il soit possible de réaliser la spécification Grafcet sur la cible microcontrôleur choisie.

Le **chapitre** 5 présente une **Application à la génération par approche IDM de code Arduino pour le contrôleur d'un système multi-énergie**. Cette application réalise une commutation entre différentes sources d'énergie ayant des coûts différents pour servir d'énergie de pompage dans le but de garantir la présence de l'eau à divers étages d'un bâtiment. De l'étude générale du problème et de la modélisation faite, il découle une architecture du système et un mécanisme de pompage garantissant une réduction significative des pertes d'énergie potentielle, et par conséquent de l'énergie électrique. Pour ce système, nous produisons un modèle Grafcet de spécification à partir duquel la génération par approche IDM d'un code correspondant est faite en langage *Arduino* pour le microcontrôleur *Atmega1280* dont le modèle est pris en entrée.

Chapitre Un

Etat de l'art sur la mise en œuvre des Système de Contrôle-Commande

Sommaire

1.1 Introduction

Nous présentons ici la notion de Système de Contrôle Commande (SCC), avec les outils de réalisation, au nombre desquels les APIs et les microcontrôleurs. Vu l'intérêt de la spécification formelle pour la réalisation

des SCCs, les principaux modèles de spécification sont présentés, partant des modèles traditionnels aux langages de haut-niveau. Parmi ces langages, il y a le Grafcet qui, inspiré des réseaux de Petri, permet de spécifier facilement les SCCs. Nous étudions alors les travaux consacrés à la synthèse du code du contrôleur spécifié en Grafcet.

1.2 SCC et outils

Un système embarqué est une combinaison de matériel et de logiciels informatique, et éventuellement de composants mécaniques ou autres, conçues pour remplir une fonction spécifique [2]. Les ordinateurs se distinguent de ces systèmes par le fait qu'ils sont conçus pour accomplir des tâches diverses. Un système est qualifié d'embarqué lorsqu'il est à la fois électronique et informatique, autonome et dédié à une tâche précise, fonctionnant souvent en temps réel, possédant une taille limitée et ayant une consommation énergétique restreinte. Il constitue un sous-système, enfoui dans un dispositif plus large avec lequel il est interfacé. Ces systèmes sont devenus omniprésents dans notre environnement et jouent un rôle déterminant dans l'innovation industrielle de secteurs aussi divers que l'automobile, la médecine, le spatial, l'armée, la téléphonie, l'aéronautique, la domotique ... Ils permettent de réaliser des automatismes industriels et possèdent en leur sein un SCC.

1.2.1 Automatisme industriel et SCC

Un automatisme industriel est un sous-ensemble de machines destinées à remplacer l'être humain dans des tâches, en général simples et répétitives mais réclamant précision et rigueur. Cet automatisme industriel vise à répondre au besoin d'élaboration des produits, de l'énergie ou de l'information à coût réduit pour les utilisateurs [12]. Cette élaboration est obtenue au moyen de dispositifs opératifs plus ou moins mécanisés, appelés partie opérative, et grâce à l'intervention d'opérateurs humains pour assurer la coordination des dispositifs opératifs [36]. La partie opérative d'un système automatisé est le sous-ensemble de l'automatisme qui effectue les actions physiques (transformer et adapter l'énergie, transmettre les efforts, agir sur la matière d'œuvre). Elle est généralement composée d'actionneurs, de capteurs, d'effecteurs et d'un bâti.

Un actionneur est l'organe fournissant la force nécessaire à l'exécution d'un travail ordonné par une unité de commande distante.

Un capteur est un composant de la chaîne d'acquisition dans une chaîne fonctionnelle et permettant de prélever une information sur le comportement de la partie opérative en la transformant en une information exploitable par la partie commande[1]. On distingue alors des capteurs à contact

1. http://philippe.berger2.free.fr/automatique/cours/cpt/les_capteurs.htm. *(Accé-*

et des capteurs à proximité, qui sont d'un point de vue applicatif soit des capteurs mécaniques, électriques ou pneumatiques. Selon la grandeur physique mesuré, on peut avoir des capteurs de température, de luminosité, de présence, de gaz, de courant, de débit, de déplacement, de distance, de force, de pression, de son, ... [2]

Les effecteurs sont des dispositifs d'entrée/sortie permettant d'échanger des informations entre l'opérateur et le système. **Un bâti** quand à lui est un assemblage de pièces pour former un support.

1.2.1.1 Système de contrôle commande

Au départ, la partie opérative d'un automatisme industriel reçoit les ordres uniquement de l'opérateur et les exécute. Mais de la répétition de certaines tâches, va naître la nécessité d'intercaler entre l'opérateur et la partie opérative un SCC dont le rôle est de prendre certaines décisions qui ne requièrent pas l'intervention de l'opérateur. Ce système reçoit en entrée des informations de la partie opérative à travers des capteurs ainsi que des consignes et des ordres issus de l'opérateur [36], et il a pour sorties les commandes générées dans la partie commande et envoyées à la partie opérative par l'intermédiaire des actionneurs sous forme d'ordres à exécuter, ainsi que des informations transmises à l'opérateur pour le renseigner (Figure 1.1). Le but d'un automatisme logique est de réaliser les changements demandés à l'entrée en modifiant l'état des sorties.

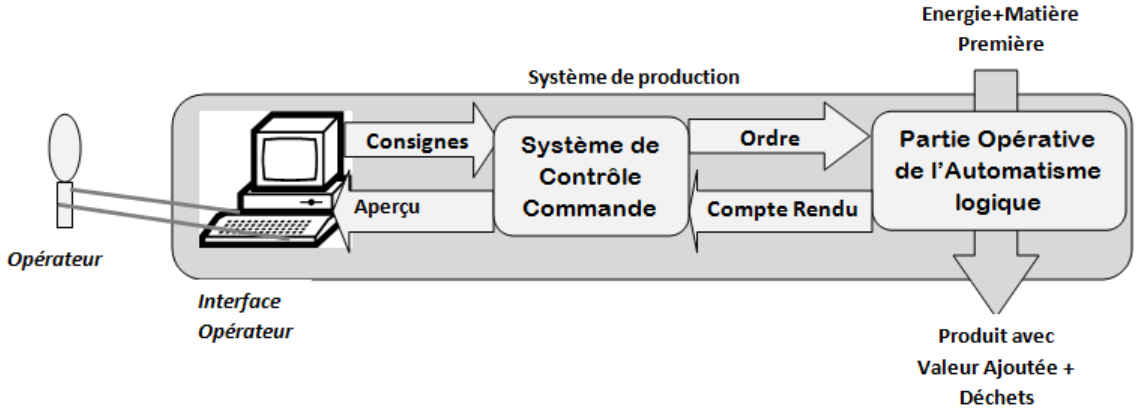


FIG. 1.1: Automatisme logique avec SCC

Un système de contrôle commande (SCC) est donc un système à événements discrets (SED) dont le rôle est de contrôler le comportement d'un processus qui est lui-même vu comme un système à événements discrets, prenant en compte l'état de ce processus et d'autres informations

dée le 27/12/ 2018)

2. https://fr.wikipedia.org/wiki/Capteur

en provenance d'un opérateur ou d'un autre système [19]. Le contrôle de la commande relève des processus dont les mesures et les actions sont de type logique (ayant deux valeurs possibles : $vrai$ ou $faux$). Pour un SCC, une mesure est une entrée tandis qu'une action est une sortie. L'état du SCC peut changer si une condition définie est vraie après la survenue d'un événement. Nous décrivons en 1.2.1.2 ce qu'est un SED.

Un SCC peut être chargé de contrôler une séquence d'événements, de maintenir constante une certaine variable ou de suivre un changement prévu [8]. Par exemple, le système de commande d'une perceuse automatique peut démarrer la descente du foret lorsque la pièce est en position, démarrer le perçage lorsque le foret arrive sur la pièce, stopper le perçage lorsque la profondeur du trou souhaitée est atteinte, retirer le foret, arrêter la perceuse, puis attendre que la pièce suivante arrive en position avant de répéter le processus. Les entrées de ces systèmes de commande peuvent provenir d'interrupteurs, qui sont ouverts ou fermés [8]. Par exemple, la présence de la pièce peut être signalée par son contact avec un interrupteur, qui se ferme, ou par d'autres capteurs, par exemple de température ou de débit. Le contrôleur peut être chargé de commander un moteur pour déplacer un objet vers un certain emplacement ou pour tourner une vanne, ou encore d'allumer ou d'éteindre un radiateur.

1.2.1.2 Système à événements discrets

Un système à événements discrets (SED) [3] est un système qui exprime un phénomène qui n'est pas continu dans le temps [13, 19]. Par exemple, lorsqu'on remplit d'eau une citerne ou bien quand on utilise cette eau, son évolution peut être décrite par le niveau d'eau qui varie, comme l'illustre la figure 1.2(a). Il s'agit d'un phénomène continu. Cependant, en se focalisant sur trois niveaux d'eau (Low, $Middle$ et $Hight$) marqués sur cette citerne, leur description est discrète, tel que présenté en 1.2(b). Les valeurs sont discrètes et chaque niveau peut être caractérisé par une variable booléenne, pour dire si oui ou non l'eau est au moins à ce niveau. On dit que l'état d'un système à événement discret peut toujours être décrit par un nombre fini de variables booléennes [19, 13].

La figure 1.2(c) montre l'évolution de l'état ($true$ ou $false$) de deux variables booléennes x_1 et x_2 qui décrit les trois niveaux d'eau: $x_1 = 1$ si l'eau dans la citerne a atteint le niveau $Middle$ et $x_1 = 0$ sinon (Low), $x_2 = 1$ si l'eau dans la citerne a atteint le niveau $Hight$ et $x_2 = 0$ sinon. On remarque bien que $x_2 = 1$ entraine $x_1 = 1$.

Quand on s'intéresse à l'état de x_1 et x_2 qui décrivent les trois niveaux d'eau (Low, $Middle$ et $Hight$), on a un système à événements discrets. Le passage de x_1 de 0 à 1 ($\uparrow x_1$) est un événement qui survient exactement à l'instant où l'eau atteint le niveau $Middle$ dans la citerne (figure 1.2(d)). On

3. ou Descrete Event System

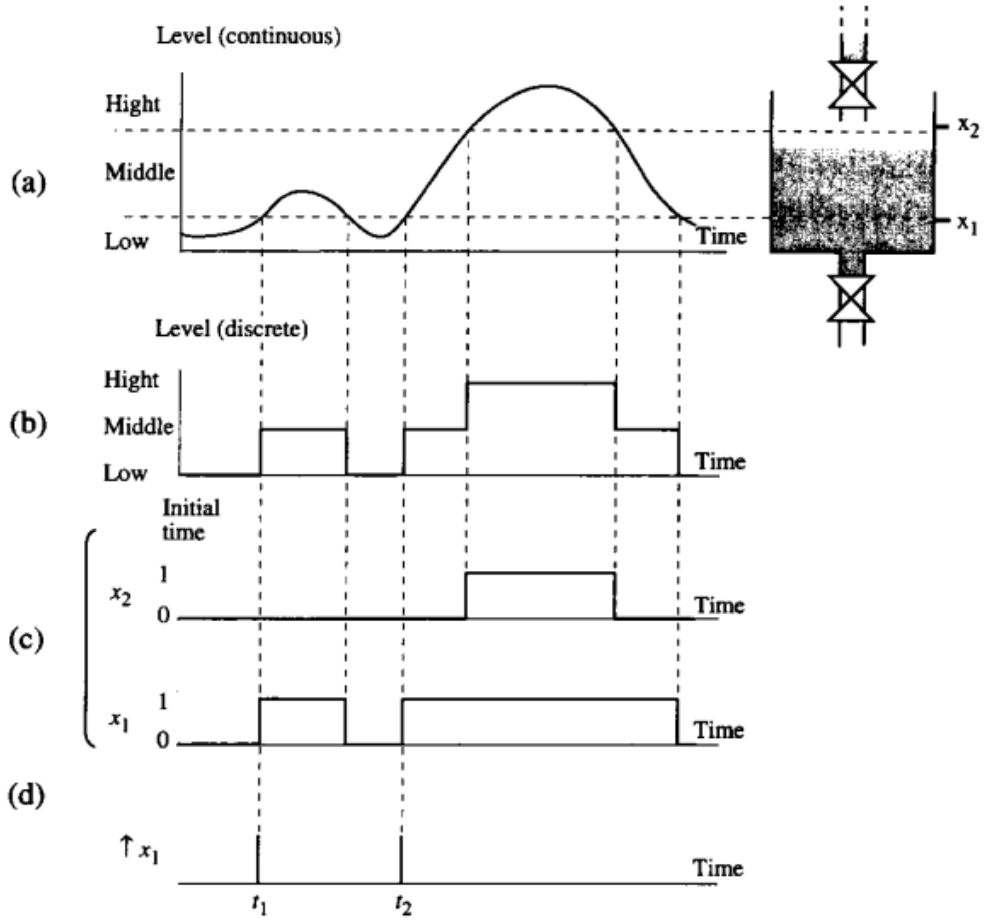


FIG. 1.2: Illustration d'un SED

retient qu'un SED est un système qui satisfait les deux propriétés suivantes :

- l'espace d'état est discret,
- le changement d'état est déclenché par un événement.

1.2.2 Les contrôleurs logiques programmables

L'automatisation de nombreux processus différents, tels que la commande de machines ou lignes d'assemblage en usine, se fait grâce à l'utilisation des ordinateurs de petite taille constitués généralement d'automates programmables industriels (API)[4].

Avant l'apparition des premiers APIs, la partie commande était implémentée à l'aide des composants de logique câblée tels que les relais électroniques, les relais pneumatiques et des combinaisons des portes logiques. Avec ces technologies, les problèmes complexes étaient difficilement solvables et l'automatisme difficilement modifiable [19]. L'utilisation de la logique programmée va se répandre et s'imposer très rapidement du fait de la vulga-

4. en anglais *Programmable Logic Controllers (PLC)*

risation des microprocesseurs désormais rendus accessibles, du fait fait de leur coût et de leurs performantes.

L'architecture de base de ces contrôleurs peut être décrite par la figure 1.3 [8].

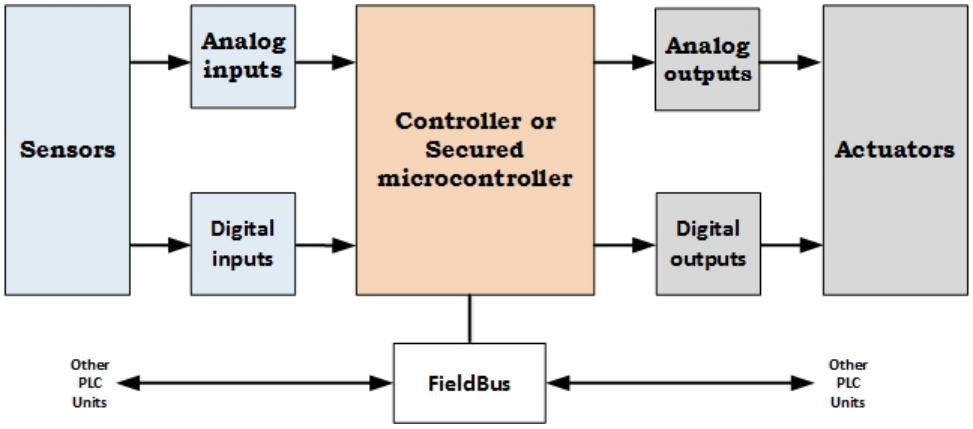


FIG. 1.3: Architecture de base des CLPs

Les entrées/sorties (I/O) constituent l'interface entre un contrôleur et le monde qui l'entoure. Les exemples les plus simples sont les commutateurs/switch (entrées) et les LEDs (sorties). Cependant, les périphériques ou composants d'entrées/sorties sont des éléments matériels qui assurent l'interfaçage entre le processeur et le monde extérieur.

Nous les présentons ci-après. Une comparaison entre les PLCs et les microcontrôleurs sera ensuite faite pour motiver le choix des cibles microcontrôleurs.

1.2.2.1 Les microprocesseurs et microcontrôleurs

Un microprocesseur est un processeur miniaturisé et dont les composant sont intégrés sur un même circuit. Il est constitué d'un morceau de silicium contenant un processeur polyvalent.

Le terme microprocesseur est généralement réservé à une puce contenant un processeur puissant qui n'a pas été conçu avec à l'esprit un besoin particulier de calcul. Ces puces constituent généralement la base des ordinateurs personnels et des postes de travail haut de gamme. Un microcontrôleur ressemble beaucoup à un microprocesseur, à la différence qu'il a été conçu spécifiquement pour être utilisé dans des systèmes embarqués. Les microcontrôleurs incluent généralement un processeur, une mémoire (une petite quantité de RAM, une ROM ou les deux) et d'autres périphériques dans le même circuit intégré. De façon sommaire, un microcontrôleur est un microprocesseur avec des entrées/sorties internes et possédant des mémoires. On retient que :

Microprocesseur : CPU + Unité de contrôle + composants d'E/S (broches).

Microcontrôleur : microprocesseur + éléments de stockage (RAM, ROM, mémoire FLASH).

Il y a une différence entre microcontrôleur et carte à microcontrôleur. Un même microcontrôleur peut équiper des cartes à microcontrôleur différentes, produits par des fabricants différents. Par exemple, l'Atmega328 est un microcontrôleur qui équipe les cartes à microcontrôleur Arduino Uno, Arduino Nano, Arduino Mega, etc.

1.2.2.2 Les automates programmables industriels

Un automate programmable industriel (API / PLC) ou simplement automate programmable (AP) est un ordinateur numérique industriel qui a été renforcé et adapté pour le contrôle de processus de fabrication, tels que le contrôle de machines sur des chaînes de montage en usine, des dispositifs robotiques, ou pour toute activité nécessitant un contrôle extrêmement fiable et une programmation aisée. Il est ainsi conçu pour contrôler des processus séquentiels utilisant des entrées et des sorties numériques ou analogiques dans une atmosphère agressive (par exemple, des distorsions électromagnétiques, des vibrations mécaniques, du bruit de signal, etc.) [4].

Un API est une forme particulière de contrôleur à microprocesseur qui utilise une mémoire programmable pour stocker les instructions et qui implémente différentes fonctions, qu'elles soient logiques, de séquencement, de temporisation, de comptage ou arithmétiques, pour commander les machines et les processus [8].

Les APIs ont d'abord été développés dans le secteur de la construction automobile pour fournir des contrôleurs flexibles, robustes et facilement programmables, destinés à remplacer les relais, les minuteries et les séquenceurs câblés. Depuis lors, ils ont été largement adoptés en tant que contrôleurs d'automatisation à haute fiabilité adaptés aux environnements difficiles. On dit qu'un API est un exemple de système temps réel "dur". A nos jours, les APIs sont largement utilisés dans plusieurs types d'industries (la métallurgie, la manufacture, la chimie, le papier, la nutrition, le transport, ...). Ils sont d'un coût relativement élevé, facilement réalisables et d'applications diverses. Ils ont apporté significativement à l'automatisation industrielle : ils effectuent quotidiennement les tâches les plus ingrates, répétitives et dangereuses. Parfois, ces automatismes sont d'une telle rapidité et d'une telle précision, qu'ils réalisent des actions impossibles pour un être humain. Aujourd'hui, ils peuvent gérer quelques centaines d'entrées/Sorties (avec des possibilités d'extension) et offrent un large éventail de fonctionnalités, y compris les relais de contrôle de base, le contrôle de mouvement, le contrôle de processus et de réseaux complexes, ainsi que leur utilisation dans les systèmes de contrôle distribués. Certains automatismes peuvent être reliés par des réseaux de communication, d'autres peuvent prendre en compte des fonctions additionnelles telles que le calcul et la régularisation [19].

L'environnement de réalisation d'un PLC est décrit par la figure 1.4.

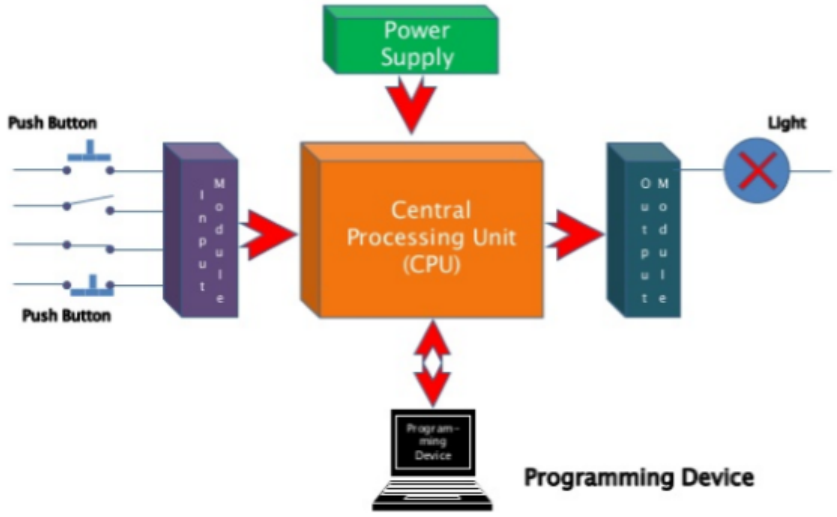


FIG. 1.4: Environnement de réalisation du PLC

Comme le montre la figure 1.5, un API effectue un cycle d'opérations répétitif appelé cycle de balayage ou de scrutation (*scan cycle* en anglais). Durant ce cycle, il lit les entrées, exécute son programme, puis met à jour les sorties. Une caractéristique générale d'un PLC est la durée moyenne d'un cycle balayage (*scan time* en anglais, ST) [8]. Une autre grandeur qui est associée au ST est la vitesse de balayage du PLC (*scan rate* en anglais, SR), qui est le nombre de cycles de balayage exécutés par unité de temps ($ST = 1/SR$).

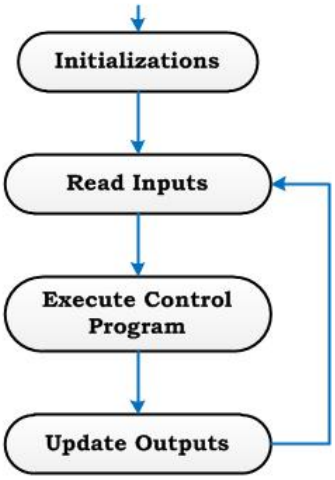


FIG. 1.5: Cycle de base d'un PLC

Les APIs sont conçus pour être exploités par des ingénieurs dont les connaissances en informatique et langages de programmation peuvent être limitées. La création et la modification des programmes de l'API ne sont

TAB. 1.1: Quelques exemples de PLCs

N°	PLC	Prix ($)	SR Max	ST Min
1	Sick CLV490-1010 -FS- [a]	7435	1,200 Hz	0.833 ms
2	6ES7 431-1KF20-0AB0 -FS- [b]	1606	20404 Hz	0.416 ms
3	6ES5 385-8MA11 -NEW- [c]	355	500 Hz	2 ms
4	6SE3 221-0BC40 -NEW- [d]	860	50 - 60 Hz	20 - 16.67 ms
5	6SE6 420-2AB12-5AA1 -FS- [e]	229	47-63Hz	21.28 - 15.87 ms
6	H2-DM1 [f]	299	?	?
7	H2-DM1E [g]	399	?	?

[a] http://www.plc-mall.com/various/barcodescanner/sick-clv490-1010-fs-/
[b] http://www.plc-mall.com/siemens-s7-400/analog-i-o-modules/6es7-431-1kf20-0ab0-fs-/
[c] http://www.plc-mall.com/siemens-s5/special-i-o-modules/6es5-385-8ma11-new-/
[d] http://www.plc-mall.com/siemens-drives/micromaster/6se3-221-0bc40-new-/
[e] http://www.plc-mall.com/siemens-drives/micromaster/6se6-420-2ab12-5aa1-fs-/
[f] http://www.automationdirect.com/adc/Shopping/Catalog/Programmable_Controllers/Do-more_Series_PLCs_(Micro_Modular_-a-_Stackable)/Do-more_H2_(Micro_Modular_PLC)/CPUs/H2-DM1
[g] http://www.automationdirect.com/adc/Shopping/Catalog/Programmable_Controllers/Do-more_Series_PLCs_(Micro_Modular_-a-_Stackable)/Do-more_H2_(Micro_Modular_PLC)/CPUs/H2-DM1E

pas réservées aux seuls informaticiens. Les concepteurs de l'API l'ont préprogrammé pour que la saisie du programme de commande puisse se faire à l'aide d'un langage simple et intuitif.

Plusieurs PLCs sont programmés au départ en utilisant les diagrammes Ladder. Nous présenterons en 1.3.3 les langages dédiés à leur programmation.

Le tableau 1.1 présente quelques APIs avec leur prix, leur fréquence maximale de scrutation (SR max) et la durée minimale d'un cycle de scrutation (ST min).

En raison de sa flexibilité, le PLC est devenu la solution la plus étendue dans le domaine de l'automatisation. C'est pourquoi la puissance de calcul, la fréquence et la fiabilité des automates ont augmenté considérablement [4].

1.2.2.3 Les circuits FPGA

FPGA signifie *Field-Programmable Gate Array*. Cette technologie permet de construire des machines à matériel entièrement adaptable ou (re)configurable [54], ce qui permet de les reprogrammer à volonté afin d'accélérer notablement certaines phases de calculs. L'avantage de ce genre de circuit est sa grande souplesse qui permet de les réutiliser à volonté dans des algorithmes différents en un temps très court.

A la base, un FPGA est constitué d'une matrice de cellules logiques

dont chacune est capable de réaliser une fonction, choisie parmi plusieurs possibles. Le choix de la fonction à réaliser dans une cellule se fait par programmation. De même, les interconnexions entre les cellules sont programmables également. Selon le mode de programmation, on peut réaliser le type RAM ou bien le type anti-fusibles. Les FPGA constituent une architecture de prédilection pour les développements parallèles.

1.2.3 Microcontrôleurs vs. PLCs

Les microcontrôleurs et les APs sont tous des contrôleurs de systèmes automatisés. Mais face à la nécessité de contrôler un système réel donné, il y a un choix à faire entre les deux. Pour faire face à ce choix, quelques considérations sont nécessaires:

- **Coût de production** : C'est la principale contrainte. Plusieurs concepteurs de système embarqué travaillent dessus.
- **Puissance de traitement (Processeur)**: C'est la charge de travail que la puce principale peut gérer. Les millions d'instructions par seconde (MIPS) constituent un moyen courant de comparer la puissance de traitement. Si deux processeurs par ailleurs similaires ont des évaluations de 25 MIPS et 40 MIPS, on dit que ce dernier est le plus puissant. Cependant, d'autres caractéristiques importantes du processeur doivent être prises en compte. L'un est la largeur du registre.
- **Largeur du registre**: elle va généralement de 8 à 64 bits. Les ordinateurs universels d'aujourd'hui utilisent exclusivement des processeurs 32 et 64 bits, mais les systèmes embarqués sont encore principalement construits avec des processeurs moins coûteux de 4, 8 et 16 bits.
- **La mémoire**: la quantité de mémoire (ROM et RAM) requise pour contenir le logiciel exécutable et les données qu'il manipule. La quantité de mémoire requise peut également affecter la sélection du processeur. En général, la largeur de registre d'un processeur établit la limite supérieure de la quantité de mémoire à laquelle il peut accéder.
- **Le nombre d'unités**: le cycle de production prévu. Le compromis entre les coûts de production et les coûts de développement dépend principalement du nombre d'unités devant être produites et vendues.
- **La consommation électrique**: quantité d'énergie utilisée pendant le fonctionnement. Ceci est extrêmement important, en particulier pour les appareils portables alimentés par batterie. Une mesure commune utilisée pour comparer les exigences d'alimentation des périphériques portables est mW/MIPS (milliwatts par MIPS).
- **Le coût de développement**: coût des processus de conception du matériel et des logiciels, appelé ingénierie non récurrente (NRE).
- **La durée de vie**: durée prévue d'utilisation du produit. La durée de vie requise ou attendue affecte toutes sortes de décisions de conception, de la sélection des composants matériels au coût de développement

et de production du système. Combien de temps le système doit-il continuer à fonctionner (en moyenne)? Un mois, une année ou une décennie?

– **La fiabilité**: le produit final doit être fiable. S'il s'agit d'un jouet pour enfants, il ne fonctionnera peut-être pas correctement à 100% du temps, mais s'il s'agit d'un système de freinage antiblocage pour une voiture, il vaut mieux qu'il fasse ce qu'il est supposé faire à chaque fois.

Le tableau 1.2 présente une comparaison entre les principales propriétés des APIs et des microcontrôleurs.

Il en ressort que de façon générale pour les PLCs:

– ils coûtent chers comparativement aux microcontrôleurs,
– ils sont très fiables, rapides et flexibles,
– ils peuvent supporter des conditions extrêmes telles que la poussière, l'humidité, etc,
– ils peuvent communiquer avec d'autres contrôleurs,
– ils sont faciles à programmer et à dépanner,
– ils comprennent des unités d'affichage, d'entrée et de sortie.

Cependant pour les microcontrôleurs,

– ils coûtent moins chers,
– la programmation est un peu fastidieuse à cause du langage de la machine,
– un microcontrôleur est également un contrôleur logique mais utilisé dans des systèmes dédiés qui sont programmés une fois pour toutes, programme pour lequel l'utilisateur ne pourra pas accéder et dont le programme n'aura pas besoin d'être modifié fréquemment,
– ils sont développés pour des systèmes très spécifiques,
– Le temps de développement est plus long,
– Avec un microcontrôleur, il faut concevoir sa propre interface de signal,
– Le microcontrôleur n'est pas un contrôleur logique d'entrées / sorties extensible à l'infini.

A terme, il ressort clairement que dans plusieurs applications ne nécessitant pas de contrôleurs avec des contraintes temps-réel ou environnementales poussées, les PLCs peuvent très bien être remplacés par les microcontrôleurs. En effet, on assiste à une montée fulgurante des capacités de calcul de ces derniers, avec des prix de plus en plus réduits et accessibles.

1.3 Modélisation des SCCs

Au début des années 1970, le besoin de décrire des SCCs de complexité de plus en plus croissante va se faire ressentir [19]. Ceci est dû à

TAB. 1.2: Comparaison entre PLCs et Microcontrôleurs

Criterion	**Microcontrollers**	**PLCs**
Processor	<2Ghz	>= 2Ghz
Register width	4, 8, 16, 32 bits	16, 32, 64 bits
Memory (RAM, ROM, Flash)	Low(<64 KB) & Medium (64 KB to 1 MB)	Medium (64 KB to 1 MB) & High (> 1 MB)
Software Development cost	High (>$1000): is done for just one unit -Development Time is more. -Is Developed for Dedicated equipments	Low: Is done once for thousands of units (or Send with a development environment) -Is Developed as a general equipments (as a computer) -Development Time is more but once.
Production cost	Between 2 and 200	Between 70 and >4000
Number of units	< 10 units	> 500 units
Power consumption	Low, but generally for a constraint environment	High, but generally for a well powered environment
Lifetime	Months or Years	Years and Decades
Reliability	Low (is generally not reliable and not certified), and occasionally fail	High (generally work reliably and is certified)
Robustness	Low	High
Inputs & Outputs	Few	Few, large and extensible
System controlled	Small systems	Complex systems
maturity issue	No	Yes, have been in place since the 1970s
Environmental issue for which it is designed	Not hard	Very hard
Standardization / certification issue	No	Yes
Housing issues	No, should be done from zero	Yes, ready to direct employment with an appropriate interface (pre-actuators)
Programming issues	No (the programming is somewhat tedious)	Has a well-known programming paradigm
Programming languages	Assembly language, C, C++, Python, ...	Languages of the IEC 61131-3 standard

l'utilisation massive des PLCs dans l'automatisation des procédés de production industrielle. La technologie des contrôleurs logiques utilisés dans les SCCs a alors connu des avancées fulgurantes partant de la logique câblée à la logique programmée, basée sur les contrôleurs logiques programmables.

La mise en œuvre des SCCs obéit aux différentes étapes d'élaboration du logiciel embarqué communément propres aux automates programmables (APs), que sont l'étude des besoins, la spécification, l'implémentation, la mise en service, l'exploitation et la maintenance [32]. Les étapes d'étude des besoins et de spécification de ce processus sont alors très souvent réalisées de façon informelle à travers des documents sous diverses formes, ce qui conduit à une implémentation manuelle qui est coûteuse en temps et sujette à de nombreuses erreurs, voire des incohérences qui sont elles aussi répercutées à l'implémentation. Ceci justifie l'intérêt porté sur la spécification formelle réalisée à l'aide de certains langages de modélisation [32]. Elle offre la possibilité de faire des vérifications au niveau du modèle de spécification puis d'automatiser les étapes suivantes de processus, notamment la génération du code du contrôleur. Cela fait l'objet de nombreux efforts consentis par des chercheurs du domaine.

Au fil du temps, de nombreux modèles formels vont être utilisés, partant des modèles traditionnels aux langages de modélisation (IEC 60848 [16] du Grafcet et IEC 61131-3 [1] dédiés aux PLCs). Une attention particulière est donc portée sur le langage de spécification Grafcet, ainsi qu'aux travaux portant sur la transformation du Grafcet et de son implémentation.

Le but d'un automatisme logique ou d'un système à événements discrets est de réaliser les changements demandés à l'entrée en modifiant l'état des sorties. Pour cela, il est nécessaire de procéder à une bonne modélisation. C. CHEN et J. DAI ont présenté les principales qualités d'un bon modèle de conception des systèmes à événements discrets (SED) [14]. Il en ressort que toute conception de SED doit prendre en compte un espace d'états discret et des événements qui provoquent le changement de ces états. De ce fait elle doit être capable :

- de décrire l'ordre d'évolution d'un grand nombre d'états contenus dans le système à étudier,
- de prendre en compte des situations de concurrence ou comportements parallèles de fonctionnement et de les représenter simplement et aisément,
- de ne décrire que le comportement qui a été modifié suite au changement survenu au niveau des entrées de l'automatisme, car un état du système ne peut être influencé que par un nombre relativement petit de variables d'entrée,
- D'illustrer d'une manière simple la sortie du contrôleur après un changement des entrées.

Ces modèles de conception peuvent être regroupés dans deux catégories : les automates et les modèles facilitant une implémentation, ici qualifiés

de modèles classiques. Dans la catégorie des automates, on peut citer les circuits séquentiels synchrones, les graphes de fluence, les tables d'état, les réseaux de Petri et le GRAFCET, lesquels étaient indépendants des spécifications matérielles et logicielles. Des modèles facilitant une implémentation, le plus populaire est la logique à relais qui est une représentation graphique basée sur une analogie aux systèmes physiques de relai [19]. Lorsque les systèmes de contrôle commande sont de faible complexité, les premiers modèles permettent de les spécifier. On peut distinguer: la table d'états, le graphe de fluence et la logique à relais, dont une brève description est donnée ci-après.

1.3.1 Modèles classiques de spécification

1.3.1.1 La table d'états

Elle est constituée de lignes et de colonnes. Chaque colonne est associée à une combinaison des valeurs des entrées et une ligne est associée à un état interne. Un état interne est une combinaison de valeurs des variables d'entrée définissant un état total. Le changement d'état est perceptible au niveau des lignes et d'une variation des entrées en colonne. Ce modèle est très important pour étudier en détail une petite partie d'un système; mais quand le système devient grand, le modèle devient pratiquement inutilisable [19]: pour 20 variables d'entrée, il faut plus de 20 millions de colonnes .

1.3.1.2 Le diagramme d'états-transitions

Encore appelé graphe de fluence, ce diagramme a été étudié par Kofman pour la synthèse des systèmes séquentiels. Il est équivalent à la table d'états et en constitue une représentation graphique. Ici, chaque nœud représente un état total du système et chaque arc relie deux nœuds. Chaque arc indique le changement d'état et est marqué par une combinaison des valeurs des variables booléennes qui exprime la condition de changement d'état. A la droite d'un nœud, on indique éventuellement l'action à exécuter. Mais sa principale faiblesse est l'impossibilité de représenter les états concurrents du système [19].

1.3.1.3 La logique à relais

Ce modèle[5] est un outil graphique permettant de dessiner des diagrammes à relais. Il est constitué de deux rails horizontaux et de plusieurs lignes entre ces deux rails appelées barreaux dont chacune représente une équation. Chaque barreau horizontal correspond à une équation booléenne et est associé à une variable interne ou de sotie. La variable dont dépend l'équation est représentée par un cercle. Deux barres verticales représentent un relai et permettent d'activer ou d'interrompre le barreau correspondant.

5. *Relay Ladder Logic* en anglais

Ici on peut représenter les états concurrents, mais la nature séquentielle du système n'apparaît pas toujours [19].

1.3.2 Modèles de haut-niveau

Dans les années 1970, le désir de décrire le fonctionnement des systèmes de contrôle commande (SCC) de plus en plus complexes va révéler aussitôt les limites des modèles précédents. En effet, il fallait trouver un moyen de représenter les systèmes de contrôle logiques ayant un grand nombre d'entrées. D'autres modèles de spécification des SCCs vont alors être utilisés à cet effet. C'est le cas dans un premier temps des réseaux de Petri, puis du langage Grafcet. Ceux-ci ont un pouvoir expressif plus important, comparativement aux premiers. En effet, ils ne sont pas inspirés des outils électroniques de mise en œuvre des SCCs, lesquels réduisent considérablement les possibilités et facilités de modélisation.

1.3.2.1 Les réseaux de Petri

Etudié en 1962 par Petri, c'est un outil de modélisation des systèmes d'événements discrets. Un réseau de Petri est représenté par un graphe biparti orienté composé de deux types de nœuds: les places et les transitions. Deux places ne peuvent être reliées entre elles, ni deux transitions. Un arc relie toujours deux nœuds de natures différentes. Les places peuvent contenir des jetons représentant généralement des ressources disponibles. La distribution des jetons dans les places est appelée le marquage du réseau de Petri [20].

Il permet de modéliser et de visualiser les comportements concurrentiel et synchrone, ainsi que le partage des ressources entre les systèmes [21].

Cependant, l'exécution d'un réseau de Petri n'est pas déterministe. Il peut y avoir plusieurs possibilités d'évolution à un instant donné. En effet, si quelques transitions en conflit sont simultanément franchissables, seulement l'une d'elle est franchie ; cependant il n'y a aucune règle pour choisir la transition à franchir [20].

1.3.2.2 Le Grafcet

Ce modèle vit le jour pour répondre aux besoins sus-évoqués. Il est inspiré du modèle des réseaux de Petri. Il a été défini par un groupe de francophones dont une moitié des 24 signataires du rapport final en 1977 est constituée d'académiciens et l'autre moitié d'industriels [15]. En 1988, il fut standardisé par la Commission Électronique Internationale [15] et devint la norme internationale CEI 60848 citeinternational2002iecG7.

Le Grafcet est un outil de définition graphique de tout système automatisé dont les évolutions peuvent s'exprimer séquentiellement. Conçu par l'ADEPA, le Grafcet est un formalisme clair et strict, permettant de traduire un fonctionnement sans ambiguïté. Sa force réside en ce qu'il permet

de spécifier le fonctionnement des SEDs aussi complexes soient-ils. À l'heure actuelle, le Grafcet est devenu plus qu'un outil de description, c'est un langage de spécification et de programmation graphique des automatismes industriels. Il est d'ailleurs devenu le formalisme standard international le plus utilisé pour la spécification et la description de haut niveau des systèmes séquentiels complexes.

La section 1.4 est consacrée à la présentation en détail du Grafcet.

1.3.3 Les langages de la norme CEI 61131-3

La troisième partie du standard CEI 61131-3 [1] décrit cinq langages de programmation des PLCs. C'est l'aboutissement d'un travail d'harmonisation des langages dédiés à la programmation des PLCs. Il résulte du premier vrai effort de normalisation des langages de programmation des APIs. Ces langages sont :

- **Ladder Diagram (LD)** ou **Langage Ladder** est aussi appelé schéma à contacts. C'est un langage graphique très populaire utilisé par les automaticiens pour programmer les APIs. Il ressemble aux schémas électriques, et est facilement compréhensible par les ingénieurs de ce domaine. Déjà présenté en 1.3.1.3, il est incontestablement le langage le plus ancien du domaine des PLC qui apparaît dans ce standard.
- Le **Sequential function chart** (SFC) est un langage graphique de programmation des Automates Programmable Industriel. Ce langage est une interprétation assez libre et plus permissive du Grafcet dont il est inspiré. Bien que le Grafcet et le SFC soient identiques dans la pratique, le Grafcet est destiné à la base à la spécification, alors que SFC est plus penché vers la programmation.
- Le **Function Block Diagram** (FBD) ou **diagramme à boîtes fonctionnelles** est aussi un langage graphique de la norme CEI 61131-3. Un diagramme en FBD est constitué de blocs rectangulaires qui décrivent chacun une fonction, avec à gauche des entrées et à droite des sorties. Chaque bloc est constitué à l'intérieur de blocs plus simples, liés les uns aux autres. Chaque sortie d'un bloc peut être reliée à une entrée d'un autre bloc. Il réalise les fonctions logiques plus ou moins complexes.
- **Instruction List** (IL) ou **liste d'instructions** est un langage de bas niveau comparable à l'assembleur.
- Le **Structured Text** (ST) ou **texte structuré** est un langage de programmation textuel de haut niveau dont la structure se rapproche des langages Ada et Pascal. Des instructions plus ou moins complexes sont supportées, telles que les structures de contrôle et des boucles. Il inclut aussi des fonctions mathématiques. Son niveau d'abstraction plus élevé par rapport à IL offre une meilleure lisibilité et une

programmation plus intuitive aux utilisateurs d'aujourd'hui qui sont généralement familiarisés avec au moins un langage de type C.

Un programme PLC peut être écrit dans un ou plusieurs de ces langages à la fois. Le langage SFC est très souvent utilisé pour donner une structure générale aux programmes.

Ces langages sont intégrés dans les plateformes industrielles de développement des PLCs. Les programmes obtenu par écriture ou par modélisation sont compilés puis le code est généré dans les langages cibles. Ils disposent aussi d'outils pour la vérification/validation des programmes/modèles. Entre autres environnements de développement, on peut citer:

- *CoDeSys*, très utilisé et édité par la société allemande **3S-Smart Software**.
- *SoMachine*, *UNITY_Pro*, *UNITY*, *PL7_Pro* et *PL7_2* (sotie vers 1992) de **Schneider Electric**,
- *S7-GRAPH*, *STEP7* et *GRAPH7* de **Siemens** et
- *CX_PROGRAMMER* de **Omron**.

Le format de sérialisation[6] des programmes écrits dans les langages de la norme CEI 61131-3 a été standardisé en XML par PLCopen [48] pour faciliter l'interopérabilité des programmes dans différentes plateformes industrielles.

1.4 Le langage de spécification Grafcet

En 1977, l'Association française pour la cybernétique économique et technique a conçu le Grafcet, qui a été adoptée en 1988 par la CEI sous le standard CEI 848. Une deuxième version a été écrite en 2002 sous le nom de CEI 60848 Ed.2 [16] : Langage de spécification Grafcet pour diagrammes fonctionnels en séquence.

Le Grafcet est un langage graphique permettant de modéliser des systèmes d'automatisation. Cette norme internationale est utilisée pour la description détaillée du comportement de systèmes logiques séquentiels et s'inspire directement du langage des réseau de Petri [19, 12, 49]. Le langage Grafcet est largement utilisé dans les domaines industriels, tels que le transport ferroviaire, la production d'énergie électrique, l'industrie manufacturière, l'environnement, etc. Il permet de spécifier le comportement attendu d'un SCC connecté à un système physique avec lequel il interagit, recevant des signaux logiques qui décrivent l'environnement physique et envoyant en réponse d'autres signaux sous forme d'ordres pouvant servir d'actions à cet environnement. Il décrit les états d'un système et les actions associées permettant de prendre en compte les entrées nécessaires et de générer les

6. (serialization en anglais) est le codage d'une information sous la forme d'une suite d'informations plus petites (dites atomiques) pour, par exemple, sa sauvegarde (persistance) ou son transport sur le réseau

sorties voulues. Le SFC, l'un des cinq langages de la norme CEI 61131-3 [1], introduit peu de modifications pour renforcer son aspect pratique. Parmi les langages de spécification des contrôleurs logiques, le Grafcet présente de nombreux avantages car il est générique, complet et lisible. Sa force réside dans le fait qu'il spécifie graphiquement et facilement le fonctionnement des systèmes à événements discrets complexes. De nos jours, le langage Grafcet est devenu plus qu'un outil de description, mais un langage de programmation graphique des systèmes d'automatisation industrielle. Le langage Grafcet est bien défini statiquement par sa syntaxe et dynamiquement par ses règles d'évolution.

Nous présentons dans un premier temps la notion d'événements, qui est le fondement de base du langage Grafcet. Ensuite, nous donnons la description statique (section 1.4.2) et dynamique (section 1.4.3) du Grafcet, avec un exemple de modèle Grafcet (section 1.4.4). La démonstration des propriétés énoncées sur les événements est donnée dans les travaux de R. David [19].

1.4.1 Les fondements de base du langage

Le langage Grafcet et son interprétation est fondé sur les notions de conditions et d'événements. Les conditions/variables temporelles en sont une particularité.

1.4.1.1 Les conditions

Elles sont reliées à l'état de l'environnement de l'automatisme logique. On les représente à l'aide des variables booléennes ou bien par des prédicats ou propositions qui sont une combinaison de plusieurs variables Booléennes à l'aide d'opérateurs logiques pour former des fonction Booléennes (Exemple : $a \bullet (init + b)$, où a, $init$ et b sont des variables).

Ici, $\bullet$ est l'opérateur *and* et $+$ est l'opérateur *or*. De même, 0 représente $false$ et 1 représente $true$.

1.4.1.2 Les événements

Un événement matérialise le changement de la valeur d'une variable booléenne ou d'une combinaison de variables booléennes (fonction booléenne) de 0 à 1 ou de 1 à 0 (Tel qu'illustré en section 1.2.1.2). Ce sont des informations relatives au changement de l'état de l'environnement du système. Un événement n'a pas de durée, contrairement à la valeur d'une variable booléenne qui nécessite une durée de temps pour être observable.

L'événement *toujours occurrent* : Lorsqu'aucun événement ne survient dans l'environnement de l'automatisme, on considère un événement que nous notons e, appelé événement *toujours occurrent*. Il peut être

assimilé au temps qui passe ou bien être considéré comme l'événement généré lorsque le temps passe. Il est utilisé dans le formalisme de l'algèbre des événements.

1.4.1.3 Hypothèses et propriétés sur les événements

Soient a et b deux variables booléennes ou bien deux fonctions booléennes.

Définition 1.1. *Le front montant et le front descendant*

Soit $f(a_1,a_2,\ldots,a_m)(t)$ une fonction booléenne de m variables dont la valeur est définie à l'instant initial 0, et telle que pour les instants $t_1 < t_2 < t_3 < \ldots < t_{n-1} < t_n < t_{n+1} < \ldots$, on ait $f = 0$ dans les intervalles $[0,t_1),[t_2,t_3),\ldots,[t_{2k-1},t_{2k}),\ldots$, et $f = 1$ dans les intervalles $[t_1,t_2),[t_3,t_4),\ldots,[t_{2k+1},t_{2k+2})$:

- Si $t_1 > 0$ alors l'événement $\uparrow f = \uparrow f(a_1,a_2,\ldots,a_m)$ survient aux instants $t_1,t_3,t_5,\ldots,t_n,t_{n+2}$ et l'événement $\uparrow f' = \downarrow f(a_1,a_2,\ldots,a_m)$ survient aux instants $t_2,t_4,\ldots,t_{n-1},t_{n+1},\ldots$ (figure 1.6)
- Si $t_1 = 0$, i.e. si la valeur initiale de f est 1, alors l'événement $\uparrow f = \uparrow f(a_1,a_2,\ldots,a_m)$ survient aux instants $t_3,t_5,\ldots,t_n,t_{n+2}$ et l'événement $\uparrow f' = \downarrow f(a_1,a_2,\ldots,a_m)$ survient aux instants $t_2,t_4,\ldots,t_{n-1},t_{n+1},\ldots$ (figure 1.6)

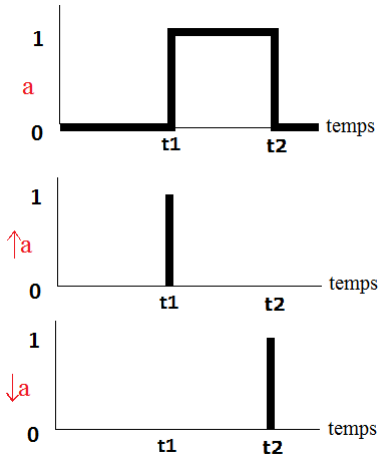


FIG. 1.6: Événements: front montant et front descendant de a

En résumé, le passage d'une variable booléenne a (resp. une fonction booléenne $f(x)$) de 0 à 1 est appelé front montant de a (resp. front montant de $f(x)$) et noté $\uparrow a$ (resp. $\uparrow f(x)$). Lorsqu'elle passe de 1 à 0, c'est son front descendant, et il est noté $\downarrow a$ (resp. $\downarrow f(x)$).

Propriété 1.1. *Ordre de priorité de l'opérateur '$\uparrow$' par rapport à '$+$' et '$\bullet$'*

(i) $\uparrow a \bullet b = (\uparrow a) \bullet b$

(ii) $\uparrow a + b = (\uparrow a) + b$.

Définition 1.2. *Le moment de survenue des événements*

(i) $\uparrow a \bullet b$ est un événement qui a lieu au même moment que l'événement $\uparrow a$, chaque fois que $b = 1$ à l'instant correspondant.

(ii) $\uparrow a\bullet \uparrow b$ est un événement qui survient au même instant quand $\uparrow a$ et $\uparrow b$ surviennent simultanément (ceci n'est possible que si a et b ne sont pas indépendants)

(iii) $\uparrow a+ \uparrow b$ est un événement qui survient au même instant que survient soit l'événement $\uparrow a$, soit l'événement $\uparrow b$.

(iv) Soit S un système dont le comportement dépend de l'ensemble des événements E. Soit $E(t)$ le signal de l'événement associé à S à l'instant t : $E(t) \in E \bigcup \{e\}$ où e est l'absence d'un événement de E à l'instant t.

Propriété 1.2. *Indépendance de deux événements*

Deux événements E_1 et E_2 sont indépendants s'il n'y a aucun événement E_i tel que E_1 et E_2 s'écrivent $E_1 = X + A \bullet E_i$ et $E_2 = Y + B \bullet E_i$; où X et Y sont deux événements, A et B sont deux variables booléennes.

Hypothèse 1.1. *Deux événements indépendants ne peuvent avoir une occurrence simultanée. Si a et b sont indépendants, alors $\uparrow a\bullet \uparrow b = 0$.*

De cette hypothèse découle la propriété suivante :

Propriété 1.3. *Combinaison d'événements*

(i) Le produit de deux événements est un événement.

(ii) La somme de deux événements est un événement.

(iii) Le produit d'un événement avec une condition est un événement.

Propriété 1.4. *Autres propriétés*

(i) $\uparrow a =\downarrow a'$

(ii) $\uparrow a \bullet a =\uparrow a, \uparrow a \bullet a' = 0, \downarrow a \bullet a' =\downarrow a, \downarrow a \bullet a = 0$

(iii) $\uparrow a\bullet \uparrow a =\uparrow a, \uparrow a\bullet \uparrow a' = 0, \uparrow a \bullet e =\uparrow a$

(iv) Si a et b sont deux variables indépendantes, alors

- $\uparrow (a \bullet b) =\uparrow a \bullet b + a\bullet \uparrow b$,
- $\uparrow (a + b) =\uparrow a \bullet b'+ \uparrow b \bullet b'$

(v) Si a, b et c sont trois variables indépendantes, alors $\uparrow (a.b)\bullet \uparrow (a.c) =\uparrow a \bullet b \bullet c$

1.4.1.4 Contraintes ou variables temporelles

Les travaux réalisés par F. Schumacher et al. [56] sur la transformation du Grafcet en RdP ont contrbué à une sémantique formelle des contrainte temporelle dans le Grafcet.

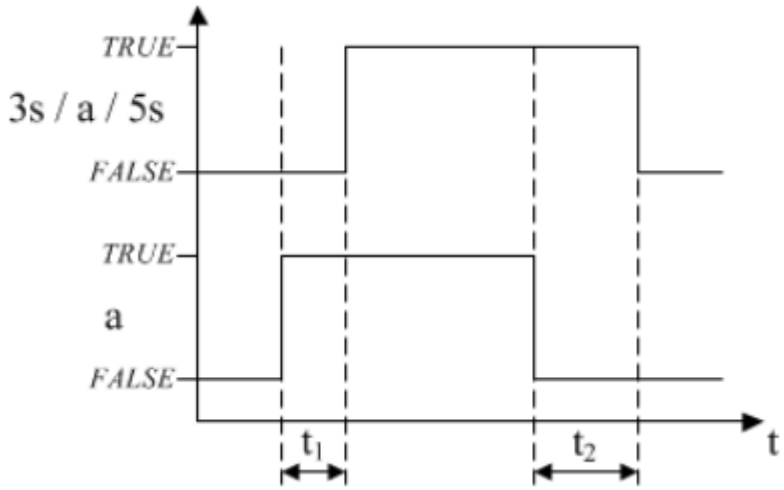


FIG. 1.7: La condition temporelle de retard $[3s/a/5s]$

Selon la norme CEI 60848, les contraintes de temps peuvent être modélisées en termes de conditions, appelées contraintes temporelles. Le comportement dynamique du Grafcet étant caractérisé par sa nature événementielle, les contraintes temporelles jouent un rôle important sur son évolution. La forme générale de notation d'une contrainte temporelles dans le Grafcet est définie par $t1/*/t2$ (que nous appelons *Delayed 2*), où l'astérisque * est un espace réservé pour la variable qui dépend du temps.

La norme CEI 60848 autorise l'utilisation de variables booléennes d'entrée et de sortie ainsi que de variables d'étape comme variables dépendantes du temps. Outre la variable dépendante du temps, les durées de retard $t1$ et $t2$ ($t1 < t2$; $t1$,$t2 \geq 0$) doivent être définies.

Comme le montre la figure 1.7, une contrainte de temps pourrait par exemple être spécifiée comme $3s/a/5s$. Cela signifie que la variable booléenne temporelle $[3s/a/5s]$ change sa valeur de $false$ à $true$, ce qui équivaut à l'événement $\uparrow [3s/a/5s]$, trois secondes après que $\uparrow a$ se soit produit. $[3s/a/5s]$ change sa valeur de $true$ à $false$ cinq secondes après l'occurrence de l'événement $\downarrow a$.

Cet exemple illustre la relation entre les retards de temps spécifiques ($t1$ et $t2$) et les deux événements d'entrée ($\uparrow$ et $\downarrow a$), définissant ainsi un intervalle de temps discret pendant lequel la variable logique de la contrainte temporelle $[3s/a/5s]$ a la valeur $true$:

$[t1..t2]$ = ["$3s$ après l'apparition de $\uparrow a$" .. "$5s$ après l'apparition de $\downarrow a$"].

Il convient alors de déterminer l'état de la variable booléenne $[3s/a/5s]$ par un chronomètre qui observe l'intervalle de temps discret $[t1..t2]$. Chaque fois qu'un événement se produit, tel que $\uparrow *$" ou $\downarrow *$, le chronomètre est réinitialisé à partir de zéro. L'exemple de la figure 1.7 est le cas le plus général d'une contrainte de temps dans le Grafcet. La norme CEI 60848 autorise également deux notations abrégées pouvant être considérées comme

des cas particuliers de $t1/*/t2$:

- La première abréviation fait référence au cas où $t2 = 0$. Au lieu d'écrire $t1/*/0s$, la notation adoptée est $t1/*$. Ce type de contrainte de temps spécifie un délai simple (que nous appelons *Delayed 1*).
- Contrairement à un simple délai, une simple limite de temps est spécifié en Grafcet par l'expression $\neg(t1/*)$, $\neg$ étant le symbole utilisé pour la négation d'une expression booléenne (que nous appelons *Limited*). La valeur de la variable booléenne de la limite de temps $[\neg(t1/*)]$ passe de $false$ à $true$ lorsque $\uparrow *$ survient et reste vraie pendant la période $t1$. À l'apparition de l'événement $\uparrow [t1/*]$, la variable logique $[\neg(t1/*)]$ change sa valeur de $true$ à $false$.

Du point de vue des variables dépendantes du temps, la contrainte temporelle peut être vue comme une sorte de chronomètre observant le changement d'état de la variable qui dépend du temps. Il fournit des informations sur la durée écoulée depuis la dernière occurrence du front montant ou descendant de la variable qui dépend du temps. Lorsque la durée écoulée est supérieure ou égale au temps de retard de la variable dépendante du temps, comme spécifié par $t1 = 3s$ et $t2 = 5s$ sur la 1.7, la contrainte temporelle correspondante devient $true$.

Ces contraintes de temps sont très importantes pour les évolutions en fonction du temps. En conséquence, il convient pour chaque étape d'être caractérisée par son état (actif ou inactif) et son âge, l'âge étant le temps écoulé depuis son activation. Il en est de même des autres variables du Grafcet. Selon la norme CEI 60848, les contraintes temporelles sont considérées comme des conditions booléennes qui sous-tendent une syntaxe spécifique et doivent donc être interprétées de manière particulière. Leur application dans le Grafcet se fait au niveau des réceptivités et des actions continues.

1.4.2 Description statique du Grafcet

Comme le langage Grafcet est inspiré du langage des réseaux de Petri, ces deux langages ont à la base la même syntaxe [36, 20]. Un modèle Grafcet (tel que présenté en Figure 1.9) est un graphe orienté avec deux types de nœuds: les étapes et les transitions. Les étapes sont représentées par des carrés tandis que les transitions sont représentées par des lignes horizontales. Les étapes initiales sont représentées par des carrés ayant des lignes doubles. Les étapes peuvent être numérotées ou nommées, mais il n'est pas nécessaire de numéroter les transitions. Les étapes et les transitions sont interconnectées par des arcs dirigés appelés connexions. Ces arcs relient nécessairement les étapes aux transitions et les transitions aux étapes. Les convergences et les divergences sont des nœuds spéciaux qui peuvent être utilisés dans un modèle dans des cas particuliers pour séparer des étapes des transitions. Leur objectif est de structurer ces nœuds de base, afin de

modéliser les alternatives, les états concurrents et la synchronisation. Lorsqu'il y a de nombreuses étapes en amont d'une transition, une *convergence en ET* représentée par un double trait est placée entre ces étapes et cette transition. Une *divergence en ET*, représentée par un double trait, est utilisée pour séparer une transition comportant de nombreuses étapes en aval. Lorsqu'il y a de nombreuses transitions en amont d'une étape, une *convergence en OU* représentée par une simple ligne horizontale est utilisée. Une *divergence en OU* également représentée par une simple ligne horizontale est utilisé pour séparer une étape de nombreuses transitions situées en aval de celle-ci.

Les actions peuvent être associées à une étape pour agir sur la partie opérative du système, ce via une variable de sortie. Il existe deux types d'actions: les actions stockées et les actions continues. Une action stockées associée à une étape est réalisée chaque fois que cette étape est active. Les actions continues ne sont réalisées que lorsqu'une situation stable est atteinte. En revanche, les actions stockées sont connectées logiquement au comportement de l'étape (changement d'état de l'étape), ce qui est exprimé au moyen d'une flèche dessinée vers le haut ou vers le bas dans le graphe Grafcet [56]. En cas de flèche vers le haut, l'action enregistrée est exécutée dès que l'étape est activée, tandis qu'une flèche dessinée vers le bas symbolise l'exécution lors de la désactivation de l'étape, respectivement.

Une condition de transition appelée réceptivité doit être associée à chaque transition. Cette condition est une expression booléenne comportant des expressions logiques (des constantes, des comparaisons, des signaux d'entrée et de sortie, des variables d'activité des étapes, des variables temporelles, etc.) composées d'opérateurs et d'événements logiques (front montant et front descendant). Cette condition exprime la contrainte de passage des étapes en amont d'une transition aux étapes en aval de cette transition. Des structures hiérarchiques (étapes englobantes, macro actions et ordres de forçage) peuvent également être introduites dans un grafcet à des fins de structuration de la modélisation [32].

1.4.3 Description dynamique du Grafcet

Le comportement dynamique du Grafcet peut être comparé à une machine séquentielle qui fournit une conversion événementielle d'une séquence d'entrée en un ensemble de sorties, en tenant compte de l'état interne du contrôleur.

1.4.3.1 Règles d'évolution

L'évolution du Grafcet est possible grâce au franchissement des transitions selon cinq règles d'évolution [16] visant à assurer un comportement déterministe. Elles sont:

– **Règle 1:** initialement, toutes les étapes initiales sont actives; toutes

les autres étapes sont inactives.

- **Règle 2:** une transition est activée lorsque toutes les étapes qui précèdent immédiatement cette transition sont actives. Une transition est franchissable lorsqu'elle est activée et que la condition associée à cette transition est vraie. Une transition franchissable doit être immédiatement franchie.
- **Règle 3:** le franchissement d'une transition provoque simultanément l'activation de toutes les étapes immédiatement suivantes et la désactivation de toutes les étapes immédiatement précédentes.
- **Règle 4:** Lorsque plusieurs transitions sont franchissables simultanément, elles sont franchies simultanément.
- **Règle 5:** lorsqu'une étape doit être à la fois activée et désactivée, en appliquant les règles d'évolution précédentes, elle est activée si elle était inactive ou reste active si elle était précédemment active.

Une étape définit un état partiel du système et peut être active ou inactive. Par conséquent, une variable booléenne X_i, appelée variable d'activité d'étape, est définie pour chaque étape. La variable X_i a pour valeur $vrai$ (1) si l'étape i est active et $faux$ (0) sinon. L'état général d'un Grafcet, appelé sa situation, est caractérisé par l'ensemble de toutes les étapes actives à un moment donné. Il peut être représenté par un vecteur $X = (X_i)$. Les étapes initiales sont initialement activées (Règle 1). Dès que le temps passe et que des événements se produisent, le changement de situation du grafcet caractérise l'évolution du système qu'il modélise. L'évolution de la situation se fait par franchissement des transitions (Règles 2 et 3).

Si plusieurs transitions sont franchissables lors de la survenue d'un événement, elles sont toutes franchies simultanément. La situation suivante est appelée situation stable et est donnée par l'ensemble des étapes actives après le franchissement de la dernière transition. Toutes les étapes intermédiaires font partie d'une situation instable, et n'ont été activées et désactivées que de manière transitoire. Ce comportement est appelé évolution transitoire. Au cours de l'évolution transitoire, les actions continues appartenant aux étapes d'une situation instable ne sont pas exécutées tandis que toutes les actions stockées sont exécutées, quelle que soit la nature de la situation (stable ou instable).

1.4.3.2 Hypothèses et algorithme d'interprétation

Pour éviter les ambiguïtés présentées par les règles d'évolution de Grafcet, de nombreuses recherches ont été menées pour formaliser l'évolution de Grafcet. R. David a proposé dans [19] un algorithme d'interprétation basé sur ces règles. Il est basé sur deux hypothèses, correspondants au fonctionnement en mode fondamental qui est un mode classique en théorie des systèmes séquentiels asynchrones :

Hypothèse 1.2. *Deux événements externes non corrélés ou indépendants*

ne peuvent pas avoir une occurrence simultanée.

Hypothèse 1.3. *Un grafcet a le temps d'atteindre un état stable entre deux occurrences distinctes d'événements externes. En effet, le passage d'une situation stable à une autre a une durée nulle, même s'il y a souvent quelques situations instables intermédiaires.*

L'étude de cet algorithme pour une meilleure compréhension nous a poussé à produire le graphe de la figure 1.8.

Quelques cycles remarquables de ce graphe :

Le cycle ($Pas5 \rightarrow Pas3 \rightarrow Pas4 \rightarrow Pas5$) décrit l'évolution du système lorsqu'une situation instable est rencontrée. Tant que la situation est instable, l'automatisme suit ce cycle d'exécution. Pendant celui-ci, on ne peut exécuter que des actions impulsionnelles. Ce cycle s'arrêtera quand une situation stable sera atteinte.

L'automatisme suit le cycle d'exécution ($Pas6 \rightarrow Pas2 \rightarrow Pas3 \rightarrow Pas6$) dans la mesure où la situation reste la même malgré l'incidence des événements externes. Il reste donc dans cette situation jusqu'à ce que la situation change sur l'occurrence d'un autre événement.

Généralement, le cycle d'exécution sera ($Pas5 \rightarrow Pas6 \rightarrow Pas2 \rightarrow Pas3 \rightarrow Pas4 \rightarrow Pas5$) dans la mesure où l'occurrence d'événements externes provoque un changement de situation de l'automatisme. Ce cycle est exécuté jusqu'à ce que le système entre dans l'un des deux cas précédents, quand il en sort, il revient continuer ce cycle.

1.4.4 Exemple de modèle Grafcet

Le grafcet présenté en figure 1.9 est inspiré d'un grafcet proposé dans [43] pour spécifier le fonctionnement du contrôleur d'un système autonome multi-énergies d'approvisionnement en eau domestique. Les autres éléments tels que les actions stockées et les actions à niveau conditionnelles ont été ajoutés pour avoir un modèle Grafcet qui capture l'ensemble de ses éléments de base du langage. Nous utiliserons ce grafcet pour nos illustrations.

La description structurelle de ce modèle est donnée comme suit: il possède huit (08) étapes numérotées de 1 à 8 dont l'étape 1 initiale, neuf (09) transitions numérotées de (1) à (9), et de nombreuses actions. Voici la description de quelques actions associées à ce modèle:

- $C := 0$ et $N := 10$ sont des actions stockées reliées à l'étape 1 et réalisées au moment de sa désactivation (à cause du symbole $\downarrow$ qui leur est associé).
- $VR1$, AV et REC sont des actions continues associées respectivement aux étapes 3, 4 et 8.
- L'action *A if (bWD OR ppM1)* est une action à niveau conditionnelle. Elle peut être activée si l'étape 6 est active et que la condition *bWD OR ppM1* est *true*.

La réceptivité de la transition (2) est *hT2 AND rain*, obtenue par la composition des variables $hT2$ et $rain$ par l'opérateur logique AND. Elle exprime le fait que lorsque l'étape 2 est active, la transition (2) est activée, et si la valeur de l'expression booléenne *hT2 AND rain* est $true$, alors la transition (2) est franchissable et doit être franchie; son franchissement provoque l'activation de l'étape 3 et la désactivation de l'étape 2. Cette dynamique a été bien expliquée dans la section 1.4.3. Dans ce grafcet bWD, $ppM1$, $hT2$, $rain$, $bT1$, $mT2$, ... sont des variables booléennes qui modélisent des signaux binaires en entrée.

1.5 Transformations et mise en œuvre du Grafcet

D'emblée, le Grafcet est présenté par R. David comme un puissant outil de spécification des SCCs dans la mesure où son fonctionnement est déterministe et qu'il permet de représenter les comportements concurrentiel et synchrone des SCC [19]. Sa validation a été rendue possible par les travaux de M. Blanchard qui a montré que tout grafcet possède un grafcet équivalent construit à partir de son graphe des situations accessibles, lequel n'est en fait qu'une machine d'état rudimentaire sur laquelle de nombreuses techniques de validation éprouvées depuis plusieurs années sont applicables [7].

Après l'adoption du langage Grafcet pour la spécification du fonctionnement des contrôleurs programmables, de nombreux travaux ont été réalisés pour sa transformation. Certains s'intéressent à la transformation du Grafcet dans un formalisme qui facilite des tests, vérifications et validation, tandis que d'autres s'intéressent à la transformation du Grafcet en code d'un contrôleur programmable.

1.5.1 Transformations du Grafcet pour analyses

L'analyse du Grafcet concerne généralement la vérification et la validation. La vérification du Grafcet permet de se rassurer qu'un modèle Grafcet réalisé est conforme au standard CEI 60848, i.e. de prouver la bonne utilisation du langage Grafcet. Cette tâche est très souvent intégrée aux compilateurs dans les phases d'analyse syntaxiques ou sémantiques. Quant à la validation, elle permet de vérifier que le modèle Grafcet représente effectivement le fonctionnement souhaité. Des règles de gestion peuvent être énoncées à cet effet par les concepteurs du cahier de charge. Elle est donc du ressort du designer qui propose un Grafcet de spécification pour le système décrit dans le cahier de charges. Il doit alors démontrer que les règles de validation énoncées sont satisfaites par son modèle.

Les principaux travaux qui traitent de la transformation du Grafcet pour des besoins de vérification ou de validation sont présentées comme suit:

Dans [12], **F. Charbonnier et al.** prouvent la contrôlabilité d'un système spécifié par Grafcet en construisant l'automate fini correspondant à l'aide duquel peut être justifié aisément l'accessibilité de tous les états.

Jean-Marc Roussel et al. [53] présentent une méthode de validation et de vérification des grafcets. Cette méthode repose sur la génération automatique d'un graphe de situations accessibles (GSA) qui est un automate à états finis, modélisé par un ensemble de situations accessibles et un ensemble d'évolutions possibles, une situation étant un état du système. En plus du Grafcet, l'utilisateur doit énoncer des règles fonctionnelles que doit respecter le modèle vis-à-vis du système modélisé. La vérification permet alors au designer de prouver la propriété de stabilité du modèle, l'absence de blocage (dead-lock), la bonne utilisation des actions stockées, ... Pour la validation, elle consiste à vérifier un certain nombre de règles fonctionnelles (propriétés) énoncées au départ par rapport au système contrôlé. Une formalisation mathématique du Grafcet est faite en termes de machine de Mealy à 6-uplets. La validation automatique est alors réalisée grâce à un langage développé et permettant de formaliser les propriétés fonctionnelles à vérifier. Lorsqu'une propriété n'est pas prouvée, la connaissance des situations ou des évolutions à problème permet de corriger le Grafcet initial. En dehors du fait que la traduction d'une règle fonctionnelle du système modélisé dans le formalisme proposé est une tâche difficile, une liste exhaustive des règles de cohérence du modèle par rapport au langage n'est pas donnée. Ces travaux ne prennent pas non plus en compte les aspects structurels et temporels du Grafcet.

Pour des soucis de normalisation, **P. Le Parc et al.** [36] ont proposé une modélisation du Grafcet par le langage de flux de données synchrone, ce qui fournit à la fois une sémantique explicite et permet d'en réaliser des simulations sur des architectures matérielles. De plus, l'utilisation des outils de *preuve de Signal* permet la vérification des propriétés du Grafcet, ouvrant le champ à des applications critiques décrites dans ce langage très expressif.

Julien Provost et al. [49] proposent une sémantique formelle du Grafcet, en se servant d'une machine à états finis . Cette machine peut être utilisée pour des besoins de vérification, de validation et pour la construction de séquences de tests de conformité. Toutefois, cette solution est théorique, car il n'a pas été proposé un outil de transformation automatique du Grafcet en machine à états finis pour les fins sus-évoquées. De même, les aspects temporels et hiérarchiques du Grafcet ne sont pas considérées.

Après une étude des exigences et des obstacles à la transformation de Grafcet en code PLC CEI 61131-3 [1], **F. Schumacher et al.** travaillent pour compléter la formalisation du Grafcet à travers le formalisme des réseaux de Petri (RdP). Alors que ces chercheurs réalisent la transformation des concepts hiérarchiques du Grafcet en RdP CIPN[7] [59], ils s'attellent ensuite à la transformation de certaines contraintes temporelles du Grafcet

7. Control Interpreted Petri Net

en RdP TPN[8] [56]. L'intérêt des RdP est de tirer profit des acquis sur les formalismes et calculs dans ce domaine pour décrire la structure et le comportement dynamique du Grafcet. Ces travaux ont alors permis de lever certaines ambiguïtés sur l'interprétation du langage Grafcet et avancer vers sa formalisation exhaustive.

Ces travaux sur la formalisation du Grafcet ont conduit à une sémantique formelle pour chacune de ses constructions, permettant ainsi sa transformation en code de contrôle sûr.

1.5.2 Transformations du Grafcet pour génération de code

En tant que langage de spécification, le Grafcet est utilisé à la base pour décrire le fonctionnement du contrôleur puis le passage à sa mise en œuvre (programmation) se faisait manuellement par les ingénieurs de contrôle [58, 32]. Du fait de sa simplicité et de sa généricité, de nombreux travaux ont été réalisés pour en faire un langage de programmation des SCCs.

Carla Ferreira et al. [25] présentent une méthode de génération automatique de code C ou Palasm (circuit PLD) pour des contrôleurs spécifiés en langage Grafcet. Les processeurs ciblés pour exécuter le code C sont des systèmes à base de microcontrôleurs. Ils requièrent une extension pour prendre en compte des aspects temporels et un accès aux périphériques externes. Le framework proposé à cet effet accepte des modèles conçus dans IsaGRAPH ou des spécifications comportementales sous forme textuelle du système de contrôle à implémenter. Toutefois, ce travail ne décrit pas les transformations réalisées, et ne prend pas en compte les caractéristiques spécifiques des cibles dédiées. Le passage d'une cible à une autre nécessiterait alors la réalisation d'un autre processus de synthèse.

Du fait de la similarité du Grafcet avec le langage SFC, des travaux similaires à [58] ont été réalisés par **De Tommasi et al.** [22] pour réaliser un outil éducatif open source pour la conception de logiciels d'automatisation conforme à la norme IEC 61131-3. Ici, la validation est faite avec la méthode *hardware-in-the-loop*[9]. A titre pédagogique, la génération de code est faite en langage assembleur. Cet outil (UniSim) offre cependant des langages de la norme CEI 61131-3 (dont le SFC) pour l'édition des modèles, et propose aussi l'utilisation du format de sérialisation des modèles PLCopenXML. UniSim peut alors être personnalisé et servir d'outil de design des modèles Grafcet.

Quant à **F. Schumacher et al.**, faisant suite à leurs travaux sur la description du Grafcet, ils proposent une représentation formelle du Grafcet permettant de générer automatiquement du code de contrôle PLC en

8. Timed Petri Nets

9. Le contrôleur interagit avec un ordinateur qui simule un système par le biais d'un logiciel, auquel il envoie des ordres et reçoit l'état du système simulé.

langage SFC (CEI 61131-3) à partir d'une spécification Grafcet [58, 57]. Le grafcet pris en entrée est normalisé i.e. dépourvu d'aspects structurels de modélisation que sont les étapes englobantes, les Grafcets partiels et les ordre de forçage. Parce que le code généré en SFC est difficilement lisible et maintenable du fait de l'altération des structures hiérarchiques dans le Grafcet normalisé, une amélioration sera ensuite proposée [32] où le code est généré en ST (CEI 61131-3) pour un Grafcet généraliste tout en préservant les structures hiérarchiques. Le code généré est sérialisé au format PLCopenXML et utilisable dans les environnements de développement PLC (section 1.3.3). En outre, dans la méthode adoptée, une volonté manifeste est exprimée pour l'approche dirigée par les modèles (IDM) sans toutefois se servir des éléments du fondement de base l'IDM.

Cependant, la transformation directe du Grafcet en code pour des contrôleurs à base de microprocesseurs sans passer par les langages du standard CEI 61131-3 de programmation des PLCs restera une question actuelle:

Dans [4], Bayó-Puxan et al. présentent une méthodologie de compilation du Grafcet pour les microcontrôleurs programmés en C. Ils mettent alors un accent sur la transformation de la perspective événementielle (qui est une caractéristique) du Grafcet en perspective séquentielle qui est le propre des des contrôleurs à base de microprocesseurs. Pour cela, un enchainement d'étapes (algorithme) est proposé, partant des initialisations au cycle de scrutation. Dans ce cycle, les actions à niveau ne sont évaluées et exécutées que si une situation stable est atteinte. Certains aspects hiérarchiques du Grafcet tels que les ordre de forçage sont pris en compte. Cependant, une méthode de génération de code C est décrite sans qu'une automatisation ne soit proposée. Il doit en plus être modifié manuellement pour être adapté à une cible microcontrôleur précise. De même, les règles de transformation de sont pas explicitées.

Dans le soucis de proposer une représentation formelle du Grafcet au niveau du code, J. Machado & al. ont présenté dans [37] une méthodologie de conception de contrôleurs sûrs permettant de générer facilement un code de contrôle pour les contrôleurs logiques spécifiés en Grafcet. Leur proposition utilise des équations algébriques du Grafcet comme représentation formelle de sa description de Grafcet. Ces équations algébriques sont d'une importance capitale pour la représentation sûre du Grafcet. Afin de garantir la sûreté de fonctionnement du contrôleur conçu, ce modèle intermédiaire a servi pour la simulation et pour la vérification de modèles Grafcet, et ce de façon complémentaire. Ils ne proposent cependant pas d'outil pour la mise en œuvre finale des contrôleurs spécifiés. Ces équations algébriques proposées [37] serviront ensuite dans [9] pour traiter d'une approche formelle concernant le contrôle des systèmes aérospatiaux, partant de la spécification SFC et proposant une génération de code en langage C.

De toutes les solutions existantes sur la transformation du Grafcet en code, il ressort que la synthèse multi-cibles sur microcontrôleurs n'a pas été abordée. Alors que la transformation manuelle du Grafcet en code est

dépassée, il s'agit pour certaines de générer du code PLC sans se soucier de la cible PLC [58, 32, 32], puisque les environnements industriels propriétaires disposent de la définition des cibles utilisables, alors que pour d'autres c'est la génération d'un code C généraliste pour microcontrôleurs [25, 4, 37].
Les caractéristiques des cibles microcontrôleurs ne sont donc pas prises en compte, et les règles de transformation en code ne sont pas présentées. On remarque aussi que les dernières méthodes négligent plusieurs aspects du Grafcet normalisé tels que les contraintes temporelles, les événements front montant/descendant, la multiplicité et la typologie des actions associées aux étapes, et ne vérifient pas les modèles conçus avant leur transformation en code.

1.5.3 Synthèse et enjeux

Le tableau 1.3 similaire à celui proposé dans [57] permet de faire une synthèse des travaux relatifs à la transformation du Grafcet en code de contrôle. Les critères sont étendus et décrits en ligne, tandis que les propositions sont décrites en colonne.

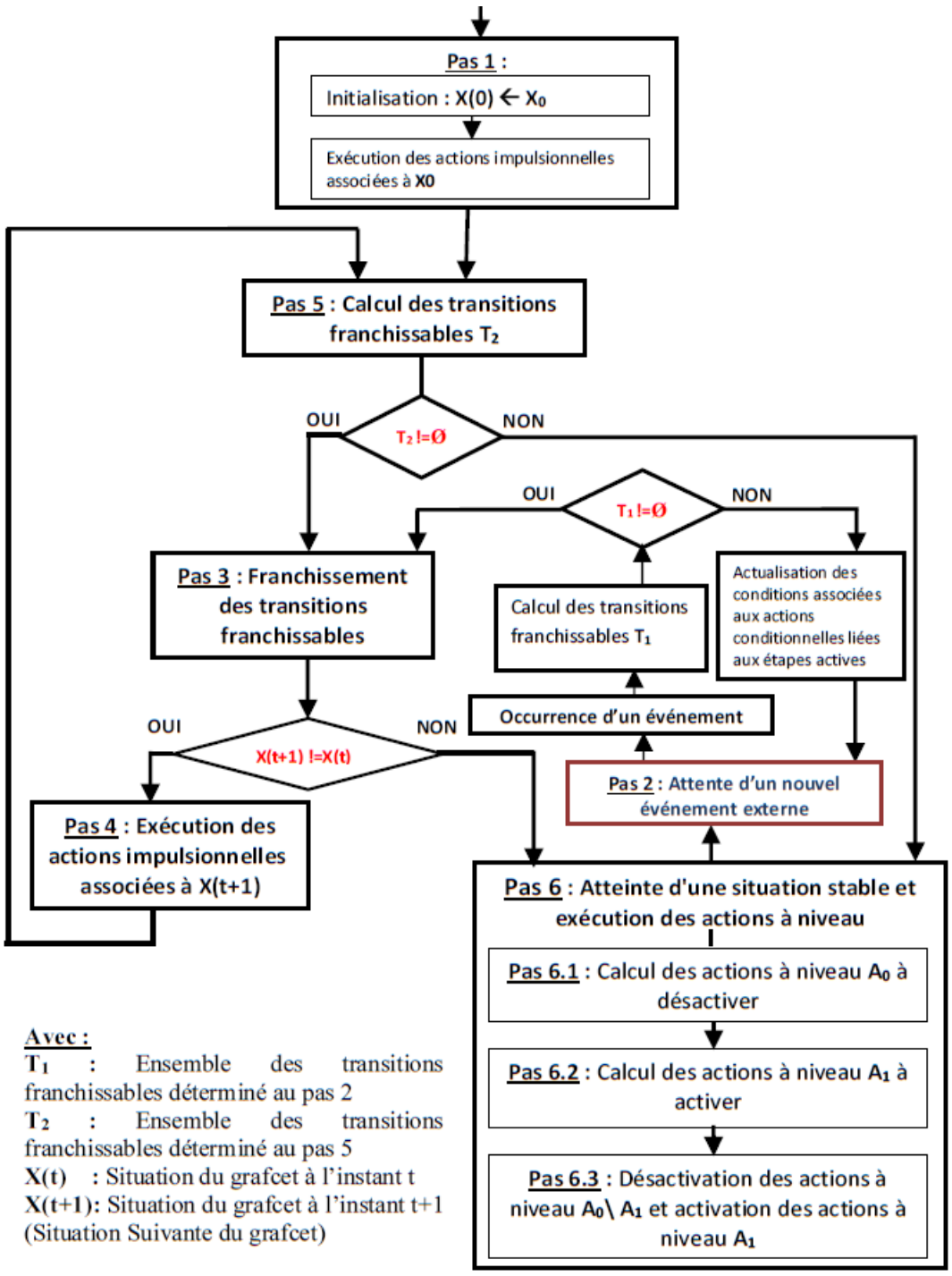


FIG. 1.8: Algorithme d'interprétation du Grafcet (Forme graphique)

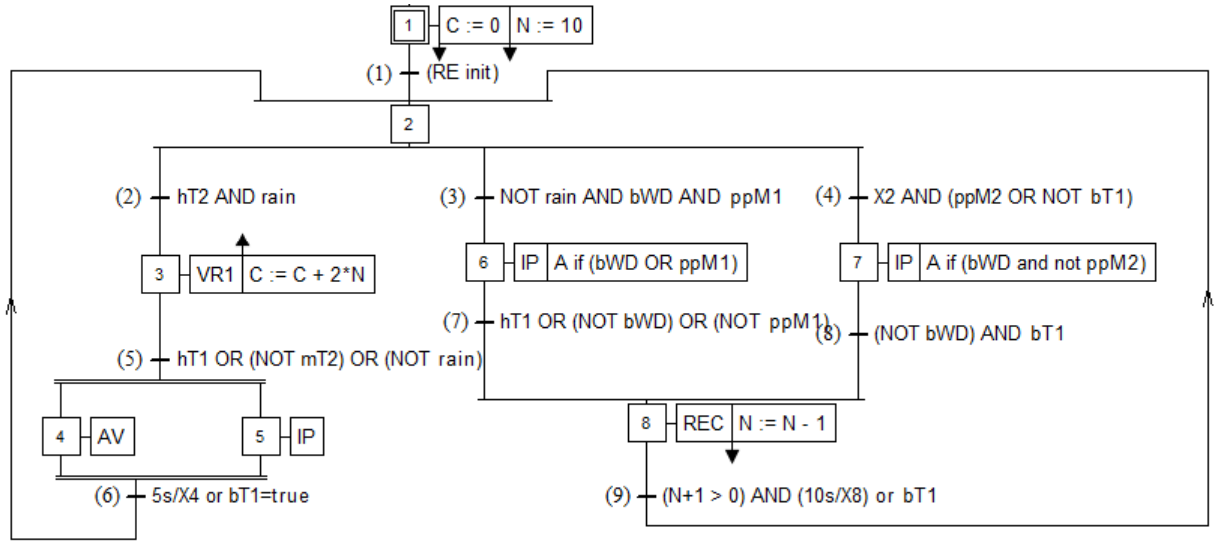


FIG. 1.9: Example de modèle Grafcet

TAB. 1.3: Synthèse des travaux sur la transformation du Grafcet en code

	Eléments du langage Grafcet								Critères objectifs						
	Editeur Grafcet	Grafcet de base (Normalisé)	Actions à niveau (Continues)	Actions stockées	Structures hiérarchiques	Contraintes temporelles	Spécification formelle	Vérification de modèles	Simulation	Génération de code PLC (CEI 61131-3)	Génération de code	Génération automatique de code	Code multi-cibles	Modèle Multi-cibles	Règles de transformation	Approche IDM
Carla Ferreira et al.																
[25] C. Ferreira, S. Monteiro, J. Monteiro	X	X	X					X			X	X	X			
De Tommasi et al.																
[22] G De Tommasi, A Pironti	X	X	X		X	X		X	X	X		X	X			
Bayó-Puxan et al.																
[4] O. Bayó-Puxan, J. Rafecas-Sabaté, J. Bergas-Jané		X	X	X	X	X					X					
J. Machado & al.																
[37] J. Machado, E. Seabra, J. Campos, F. Soares, C. Leão		X	X				X	X	X							
Borges et al.																
[9] P. Borges, J. M. Machado, E. Seabra, L. F. Silva		X	X			X		X			X					
F. Schumacher et al.																
[58] F. Schumacher, S. Schröck, A. Fay	X	X	X	X		X		X		X		X	X			
F. Schumacher et al.																
[57] F. Schumacher, A. Fay		X	X	X	X	X	X				X					
Robert Julius et al.																
[32] R. Julius, M. Schürenberg, F. Schumacher, A. Fay	X	X	X	X	X	X				X		X	X		X	
Objectif visé	X	X	X	X		X	X	X	X		X	X	X	X	X	X

Toutes les propriétés de ce tableau caractérisant les approches proposées pour la transformation du Grafcet sont intuitives. Pour les propriétés *Génération de code* et *Génération automatique de code*, il existe des travaux qui étudient la transformation de modèles Grafcet sans en proposer un outil de synthèse automatique [4, 9, 57].

De même, une fois que le code PLC est généré, l'aspect multi-cibles est pris en compte, car les environnements propriétaires de développement des PLCs contiennent la description de leurs cibles. Tous les travaux ayant permis de générer du code PLC voient alors la case **multi-cibles** cochée, mais pas celle du **modèle multi-cibles**, car aucun modèle de description des cibles n'est présenté. Il n'y a pas d'indication permettant de savoir si ces plateformes de synthèse multi-cibles proposent un modèle permettant d'ajouter dynamiquement une nouvelle cible.

1.6 Conclusion

Généralement, les SCCs sont mis en œuvre sur les PLCs. Des standards internationaux ont adopté des langages de programmation de ceux-ci, dont le SFC ou Grafcet. Ce langage est alors largement utilisé pour la spécification et la programmation des PLCs. Du fait de la montée en puissance des capacités de calcul des microcontrôleurs et du fait qu'ils soient moins onéreux comparativement aux PLCs, ils peuvent être utilisés comme solution de contrôle dans des environnements moins exigeants en contraintes. Face à la difficulté de production du logiciel de contrôle, il est question de réaliser un environnement de programmation des microcontrôleurs qui exploite le langage Grafcet pour spécifier le fonctionnement du système à réaliser. La diversité des microcontrôleurs exige de prendre en compte la synthèse multi-cibles. Les approches existantes dans le domaine de la transformation du Grafcet en code de contrôle ne prennent pas en compte cet aspect. Il sera question pour nous de proposer des modèles qui prennent en compte tous les aspects du Grafcet, l'architecture des cibles, ainsi que les transformations nécessaires à la génération de code pour la cible choisie; ce qui constitue la synthèse multi-cibles.

De toutes les solutions existantes, aucune n'utilise l'approche par IDM, bien que celle-ci soit prometteuse et fortement recommandée pour la synthèse automatique de codes à partir des modèles. Notre objectif est donc de tirer profit des avancées en IDM pour traiter de la synthèse multi-cibles Grafcet sur microcontrôleurs. Dans un premier temps, nous proposons un modèle mathématique de représentation du Grafcet, lequel est indépendant des plateformes cibles et susceptible de faciliter la synthèse pour une cible quelconque.

CHAPITRE DEUX

Génération de code Grafcet multi-cibles par matrices de codage

Sommaire

2.1 Introduction

Le Grafcet est un langage de représentation graphique. Il est par conséquent de haut niveau, facilement lisible et compréhensible. Il n'est pas inspiré des outils sous-jacents matériels existants pour offrir à ses instances une directe implémentation sur différentes cibles. C'est l'un des aspects qui fait sa force par rapport aux autres modèles de spécification tels les schémas de relais (RLL[1]) qui peuvent être mis en œvre directement à l'aide des relais. Il permet donc de spécifier aisément et facilement les systèmes plus complexes. Pour passer à son implémentation , il est nécessaire de trouver une représentation du Grafcet, laquelle devra nous rapprocher de son implémentation. C'est le but de la représentation matricielle du Grafcet. Cette représentation fournira des structures de données appropriées permettant de représenter fidèlement un grafcet. Une fois cette représentation obtenue, il sera possible de réaliser une implémentation dans une plateforme d'édition et d'extraction des matrices de codage du Grafcet. En considérant une cible microcontrôleur particulière, il sera donc possible de générer le code pour celle-ci. Finalement, l'approche proposée devra être validée.

2.2 Le codage matriciel du Grafcet

2.2.1 Idée de la représentation

En revisitant le langage Grafcet sous l'angle de l'IDM, tant du point de vue de sa structure que de son évolution, il est possible de réaliser qu'un Grafcet est en soi un programme qui est un modèle [69] et peut être considéré comme une donnée [63] pour un autre programme qui n'est rien d'autre qu'un algorithme d'interprétation du Grafcet, tel que celui proposé par R. David [19].

Concrètement, la définition du langage Grafcet (standard CEI 60848) fait allusion à un vecteur X représentant l'état de l'automatisme modélisé. Cette définition peut être complétée pour caractériser outre l'état courant, les différents éléments du modèle. C'est dans cette optique que F. Schumacher et al. [57] proposent une représentation formelle du Grafcet. Mais, celle-ci est complexe et difficilement exploitable pour automatiser la génération du code de contrôle.

Un réseau de Petri (RdP), comme n'importe quel graphe, peut être représenté à l'aide d'une matrice d'incidence D [18]. La matrice D est

1. Relay Ladder Logic

construite en utilisant deux matrices élémentaires $D-$ et $D+$ telles que $D = (D+) - (D-)$ soit décrite comme suit: en considérant un RdP avec n transitions et m places, $D+$ et $D-$ sont des matrices $n \times m$; $D + [i,j] = 1$ si la transition i a un lien de sortie vers la place j et $D + [i,j] = 0$ sinon; et $D - [i,j] = 1$ si la transition i a un lien d'entrée en provenance de la place j et $D - [i,j] = 0$ sinon [71]. L'utilisation d'une matrice unique $n \times m$ pour représenter un RdP permet d'optimiser l'espace de stockage [18]; mais cette représentation est limitée car elle ne conserve pas les propriétés réflexives (c'est-à-dire la présence d'auto-boucles) des RdP. De même, D contient trois valeurs possibles (-1, 0 et 1), et ne peut donc être une matrice booléenne. Dans [57], la définition de D correspond à *Cbasic*.

Un Grafcet est aussi un RdP et peut bénéficier de cette représentation pour d'éventuelles formalisations. Pour nous, au lieu d'utiliser une matrice d'adjacence D comme pour les RdP, nous privilégions l'utilisation de matrices booléennes nommées E (équivalent à $D+$) et S (équivalentes à $D-$), toutes de dimensions $n \times m$. Ils ne garantissent aucune perte d'information, quel que soit le type de Grafcet. De plus, l'espace de stockage est optimisé. La mémoire nécessaire pour stocker E et S dans une architecture matérielle est toujours inférieure à celle nécessaire pour stocker D.

2.2.2 Le codage de la partie statique

Un grafcet donné a un nombre fini de transitions et d'étapes que nous notons respectivement n et m. Les transitions peuvent être numérotées de 1 à n, tandis que les étapes de 1 à m.

Nous présentons un modèle semblable à celui des matrices d'adjacence des graphes, à la différence que les matrices obtenues ne seront pas nécessairement carrées du fait de la nature hétérogène des nœuds du Grafcet(étapes et transitions).

Deux possibilités de codage se présentent suivant les nœuds de référence choisis : on peut se placer au niveau des transitions et décrire les positions des étapes par rapport à ces transitions, ou bien se placer au niveau des étapes et décrire les transitions par rapport aux étapes. Dans l'une ou l'autre possibilité, on obtient une matrice d'entrée E et une matrice de sortie S. Toutefois, le référentiel constitué par les transitions est approprié, car selon le standard, le Grafcet évolue par franchissement des transitions. Quant au vecteur $INIT$ décrit les étapes initiales du graphe.

2.2.2.1 Représentation de la structure de base du Grafcet

Le vecteur $INIT$ (de dimensions $1 \times m$) et les matrices booléennes E et S (de dimensions $n \times m$) sont décrits comme suit:

Définition 2.1. ***Vecteur*** $INIT$ ***de la situation initiale :***
$INIT$ est un vecteur binaire ayant à chaque entrée la valeur 1 *ou* 0*, suivant que l'étape correspondante est une étape initiale ou pas.*

$$INIT_{(1,j)} = \begin{cases} 1 & \textit{si l'étape } j \textit{ est une étape initiale} \\ 0 & \textit{sinon} \end{cases} \tag{2.1}$$

Définition 2.2. ***Matrice*** E ***des étapes d'entrée des transitions :*** *Étant donnée une transition* (i)*, on peut avoir* 0, 1*, plusieurs voire toutes les étapes du graphe en entrée de cette transition (figure 2.1 (a)). L'entrée* (i,j) *de* E *vaut* 1 *ou* 0 *selon que l'étape* j *se trouve ou pas en entrée de la transition* i*.*

$$E_{(i,j)} = \begin{cases} 1 & \textit{si l'étape } j \textit{ est directement en entrée de la transition } (i) \\ 0 & \textit{sinon} \end{cases} \tag{2.2}$$

E est une matrice binaire qui code les étapes en entrée des transitions du grafcet. Chaque ligne de E code une transition du grafcet : elle renseigne sur l'ensemble des étapes qui se trouvent en entrée de celle-ci. Ainsi, chaque colonne de E code une étape du grafcet : elle renseigne sur l'ensemble des transitions pour lesquelles cette étape est en entrée.

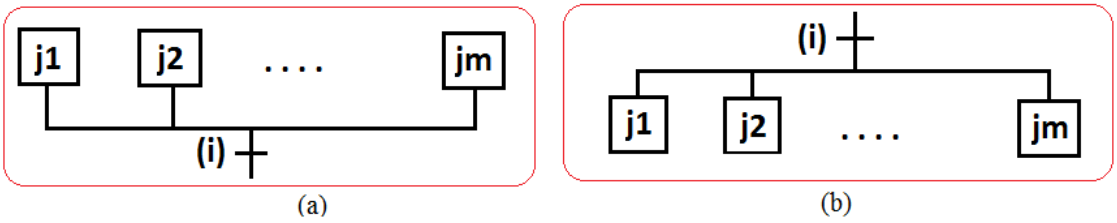


FIG. 2.1: Étapes en amont et en aval d'une transition (i)

Définition 2.3. ***Matrice*** S ***des étapes de sortie des transitions :*** *Étant donnée une transition* (i)*, on peut avoir* 0, 1*, plusieurs voire toutes les étapes du graphe en sortie de cette transition (figure 2.1 (b)). Comme la matrice* E*, chaque entrée* (i,j) *de* S *vaut* 1 *ou* 0 *selon que l'étape* j *se trouve ou non en sortie de la transition* i*.*

$$S_{(i,j)} = \begin{cases} 1 & \textit{si l'étape } j \textit{ est directement en sortie de la transition } (i) \\ 0 & \textit{sinon} \end{cases} \tag{2.3}$$

S est une matrice binaire qui code les étapes en sortie des transitions du grafcet. Chaque ligne de S code une transition du grafcet : elle renseigne sur l'ensemble des étapes en sortie de cette transition, tandis que chaque colonne de S code une étape du grafcet: elle renseigne sur l'ensemble des transitions pour lesquelles cette étape est en aval.

En général, E , S et $INIT$ sont des matrices creuses. En dehors de la structure de base, il y a aussi les actions associées aux étapes.

2.2.2.2 Matrice de description des actions

Nous nous intéressons principalement aux actions à niveau inconditionnelles, c'est-à-dire celle dont la condition d'activation est *true*. Ces actions correspondent aux sorties du système de contrôle. Pour calculer leur valeur, il est nécessaire de situer chaque action relativement aux étapes du grafcet. Soit nbA le nombre d'actions contenues dans le grafcet. Nous les numérotons de 1 à nbA, et on a :

Définition 2.4. ***Matrice*** MA ***des actions*** *:*
Chaque entrée (k,j) *de* MA *vaut* 1 *si l'action numéro* k *est associée à l'étape* j*, et* 0 *sinon:*

$$MA_{(j,k)} = \begin{cases} 1 & si\ l'action\ k\ est\ associée\ à\ l'étape\ j \\ 0 & sinon \end{cases} \tag{2.4}$$

La matrice MA des actions est donc de dimensions $nbA \times m$. Chaque ligne k de MA indique toutes les étapes du graphe auxquelles l'action k est associée. Ainsi, chaque colonne j de MA donne l'ensemble des actions associées à l'étape j.

2.2.2.3 Codage des réceptivités des transitions

A chaque transition est associée une réceptivité qui est une condition. Il s'agit d'une expression algébrique utilisant des variables logiques composées par des opérateurs logiques. L'évolution dynamique du Grafcet a besoin de la valeur de chaque réceptivité de transition, qui est une valeur booléenne. Pour cela, le vecteur booléen colonne R (Méthode d'évaluation décrite ci-après) permet de caractériser ces valeurs.

Cependant, il s'agit du résultat d'une opération. Notre objectif étant de numériser tout le Grafcet, il est donc nécessaire de coder l'ensemble de l'expression en utilisant des nombres. Ce codage (par la fonction $RPNp$) doit permettre la représentation de l'expression logique de façon à faciliter son évaluation (par l'algorithme 2.7).

Nous notons RS le tableau[2] contenant le codage des réceptivités des transitions. RS contient des entiers et comporte autant de lignes qu'il y a de transitions, ce qui n'est pas le cas des colonnes, puisque toutes les réceptivités n'ont pas la même taille. La réceptivité i est codée par la fonction $RPNp$ et son résultat est transféré à la ligne i de RS :

$$RS_i = RPNp(condition(transition_i)) \tag{2.5}$$

Une présentation en détail de la représentation et de l'évaluation des expression du Grafcet est donnée à la section 2.2.5.

2. pas nécessairement une matrice

2.2.2.4 Illustration sur un exemple

Le codage matriciel a pour but de coder tout grafcet sous forme numérique pour en faire une donnée. La finalité est de soumettre cette donnée à un interpréteur qui exécute l'algorithme d'interprétation du Grafcet (sur une cible ou bien par simulation).

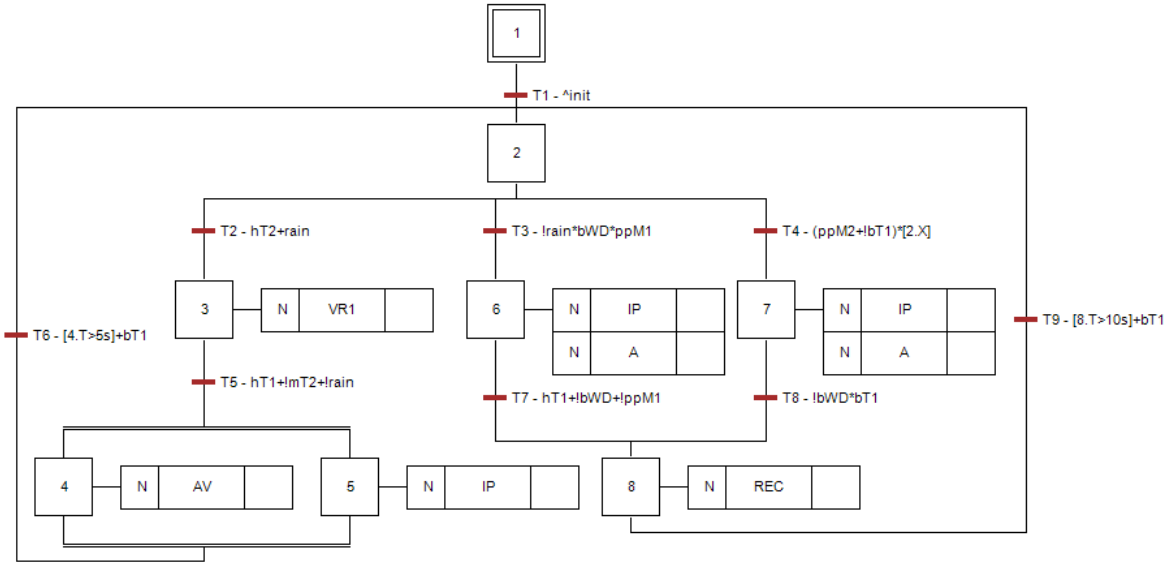


FIG. 2.2: Grafcet d'illustration du codage matriciel

En appliquant le codage matriciel du Grafcet sur le grafcet de l'exemple de la figure 2.2 (reproduit du grafcet de la figure1.9) [3], on obtient les matrices suivantes :

$$INIT = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.6}$$

$$E = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.7}$$

3. Le grafcet en figure 2.2 est une simplification de celui de la figure 1.9, où les actions stockées ne sont pas représentées, ni les actions conditionnelles. Ces éléments ne sont pas pris en compte dans cette modélisation

$$S = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.8}$$

$$MA = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.9}$$

$$RS = \begin{array}{cccccccc} 3 & 11 & 3 & 0 & & & & \\ 3 & 13 & 15 & 0 & & & & \\ 6 & 15 & 2 & 16 & 1 & 17 & 1 & \\ 6 & 18 & 10 & 2 & 0 & 25 & 1 & \\ 7 & 12 & 14 & 2 & 0 & 15 & 2 & 0 \\ 3 & 32 & 10 & 0 & & & & \\ 7 & 12 & 16 & 2 & 0 & 17 & 2 & 0 \\ 4 & 16 & 2 & 10 & 1 & & & \\ 3 & 33 & 10 & 0 & & & & \end{array}$$

Ce grafcet possède donc 09 transitions, 08 étapes, 09 signaux (variables) d'entrée et 05 actions (signaux/variables) en sortie.
La codification présentée dans les équations 2.9 et 2.10 tient compte de l'ordre suivant d'apparition des variables dans le modèle:

- Les variables d'entrées: bT_1, $init$, hT_1, hT_2, mT_2, $rain$, bWD, ppM_1 et ppM_2 (09 au total)
- Les variables de sorties ou des actions: A, VR_1, IP, REC et AV (05 au total)

2.2.2.5 Théorème et propriétés du codage du grafcet

Nous caractérisons le codage de la structure de base du Grafcet avec un théorème et des propriétés. Le résultat suivant est obtenu, nonobstant la numérotation accordée aux étapes et aux transitions. En effet, la permutation des numéros de deux étapes par exemple entraine la permutation de leurs rôles, ce qui se matérialise par la permutation des colonnes correspondantes dans les matrices de codage. Il importe donc que la permutation de

deux colonnes quelconques de toutes ces matrices ne change pas le comportement du contrôleur modélisé[4].

Théorème 2.1. ***Unicité (canonicité) des matrices de codage du grafcet:***

Soit un grafcet qui a n transitions et m étapes. Les matrices E, S et $INIT$ de codage de sa statique sont uniques.

En d'autres termes, partant d'un même grafcet et en appliquant le codage matriciel de la statique du grafcet ci-haut défini, on obtient la même matrice E, S et MA (modulo la numérotation des éléments).

Démonstration: Soit un grafcet ayant n transitions et m étapes. Ce grafcet a deux types de nœuds qui sont les étapes et les transitions.

Unicité de E et de S(de dimensions $n \times m$)

Nous allons décomposer notre grafcet en deux sous graphes : un sous graphe G_1 dont les arcs relient les étapes aux transitions et un autre sous graphe G_2 dont les arcs relient les transitions aux étapes.

Soit $N = ((1),(2),\ldots,(n),1,2,\ldots,(m))$ l'ensemble des nœuds de notre grafcet qui a $n+m$ éléments, (i) avec $i \in \{1,\ldots,n\}$ est une transition et $j \in \{1,\ldots,m\}$ est une étape. N décrit un arrangement des nœuds de nos deux graphes dans un ordre tel que les transitions viennent avant les étapes. Avec N, nous avons la matrice d'incidence I_1 de G_1 et I_2 de G_2 définies tel qu'il suit :

$$I_{1_{(1,j)}} = \begin{cases} 1 & \text{s'il existe un arc de } G_1 \text{ qui relie l' étape } j \text{ à la transition } (i) \\ 0 & \text{sinon} \end{cases}$$

$$I_{2_{(1,j)}} = \begin{cases} 1 & \text{s'il existe un arc de } G_2 \text{ qui relie l' étape } j \text{ à la transition } (i) \\ 0 & \text{sinon} \end{cases}$$

avec $(i,j) \in \{1,2,\ldots,n\} \times \{1,2,\ldots,m\}$

Cependant, du fait que deux nœuds de même nature ne peuvent être reliés entre eux, les sous-matrices suivantes sont nulles[5] :

$I_{1_{(1:n,1:n)}}$[6] $= I_{2_{(1:n,1:n)}} = O_{(n,n)}$ et $I_{1_{(n+1:n+m,n+1:n+m)}} = I_{2_{(n+1:n+m,n+1:n+m)}} = O_{(m,m)}$

De même, par définition, $I_{1_{(n+1:n+m,1:n)}} = I_{2_{(1:n,n+1:n+m)}} = O_{(n,m)}$, et on a:

$$I_1 = \begin{bmatrix} 0 & 0 \\ E^t & 0 \end{bmatrix} \; I_2 = \begin{bmatrix} 0 & S \\ 0 & 0 \end{bmatrix}$$

Puisque les matrices I_1 et I_2 sont uniques, du fait que les nœuds du graphe sont numérotés, E et S sont aussi uniques parce que E^t et S sont respectivement les sous-matrices des matrices uniques I_1 et I_2.

Unicité du vecteur $INIT$ de taille m

Les étapes du grafcet sont numérotées de 1 à m. Soient $INIT_1$ et $INIT_2$ deux vecteurs décrivant la situation initiale du grafcet considéré, tel

4. La matrice RS est aussi unique, modulo la numérotation des variables utilisées dans le Graphe

5. $O_{(n,m)}$ est la matrice nulle de taille $n \times m$

6. $M_{(i:j,k:l)}$ est la sous matrice de M partant de la i^e à la j^e ligne et de la k^e à la l^e colonne

que $INIT_1 \neq INIT_2$.

Ainsi, $\exists j \in \{1,\ldots,m\} \setminus INIT_1(j) \neq INIT_2(j)$.

$INIT_1(j) \neq INIT_2(j) \Rightarrow INIT_1(j) \bullet INIT_2(j) = 0 \Rightarrow j$ est une étape initiale et j n'est pas une étape initiale, ce qui est absurde; la supposition est donc fausse.

On a donc: $\forall j \in \{1,\ldots,m\}, INIT_1(j) = INIT_2(j) \Rightarrow INIT_1 = INIT_2$; d'où l'unicité du vecteur situation initiale $INIT$. ■

Propriétés des matrices de codage du Grafcet Au regard du standard, les propriétés suivantes peuvent être identifiées dans les matrices de codage.

Propriété 2.1. ***La convergence «en ET» dans les matrices de codage*** *: La présence de plusieurs « 1 » sur une ligne de E représente une convergence « en ET »*

Propriété 2.2. ***La divergence « en ET » dans les matrices de codage*** *: La présence de plusieurs « 1 » sur une ligne de S représente une divergence « en ET »*

Propriété 2.3. ***La convergence « en OU » dans les matrices de codage*** *: la présence de plusieurs « 1 » sur une colonne de S représente une convergence « en OU »*

Propriété 2.4. ***La divergence « en OU » dans les matrices de codage*** *: la présence de plusieurs « 1 » sur une colonne de E représente une divergence « en OU »*

2.2.2.6 Reconstruction du grafcet à partir des matrices de codage de sa statique

Le codage matriciel du Grafcet garantit la possibilité de reconstruire le grafcet à partir des matrices de codage de la partie statique du Grafcet. Ceci permet de se rassurer qu'il n'y a pas de perte d'information après le codage. Les propriétés 2.1, 2.2, 2.3 et 2.4 sont importantes à cet effet. La figure 2.3 présente l'algorithme de la reconstruction du Grafcet à partir des matrices de codage de sa statique.

L'expression présentée en figure 2.4(a)) représente la conversion en hexadécimal du codage matriciel d'un grafcet. En la décodant selon la codification présentée en 2.4.3.1, on obtient les matrices de codage correspondantes (figure 2.4(b)). L'exécution de l'algorithme de reconstruction de la figure 2.3 permet d'obtenir le grafcet de la figure 2.4(c). Ce grafcet contient des nœuds structurels (une *convergence en ET* et une *divergence en ET*) qui ont été identifiés (respectivement) grâce aux propriétés (2.1 et 2.2) du codage matriciel.

2.2.3 Le codage de la partie dynamique du Grafcet

La dynamique du grafcet est la description de son évolution dans le temps, en tenant compte des événements externes et de sa situation interne.

1- Draw **m** steps carefully separated
2- Mark initial steps according to the vector INIT
3- For every transition t_i,
   - Create the set EE_i={$Steps_j$ / E_{ij}=1}
   - Create the set SS_i={$Steps_j$ / S_{ij}=1}
   - Draw transition t_i that links steps EE_i to steps SS_i with the receptivity RS[i].
     If **|EE_i|>1** then transition **i** is a simultaneous convergence.
     If **|SS_i|>1** then transition **i** is a simultaneous divergence.
     The selection convergences and divergences are automatically created when adding a transition that links steps to a particular steps S_j, this step was already in output stream of an existing transition
4- Add actions on steps with respect to the MA matrix

FIG. 2.3: Algorithme de reconstruction du Grafcet

Une fois que la structure statique du Grafcet est bien codée, le reste consiste à fournir un algorithme qui utilise les structures de données résultant de la phase précédente pour exprimer formellement l'évolution du Grafcet, conformément aux règles d'évolution énoncées par l'algorithme d'interprétation (section 1.4.3.1 et figure 1.8). Pour dériver un algorithme de l'évolution dynamique correspondant aux matrices de codage du Grafcet, nous avons étudié les règles de Grafcet et l'algorithme d'interprétation (section 1.4.3.2) afin de trouver des instructions capturant chaque règle. De la nature événementielle du Grafcet, nous avons conçu l'algorithme d'interprétation séquentielle (SIA) présenté à la figure 2.5.

Voici quelques principaux vecteurs utilisés dans cet algorithme. Nous considérons un grafcet ayant n transitions, m étapes et p actions:

Définition 2.5. ***Vecteur*** X ***de la situation du grafcet****: À l'initialisation, sa valeur est* $INIT$ *(équation 2.1). À un instant quelconque, on a:*

$$X_j(t) = \begin{cases} 1 & \textit{si la l'étape } j \textit{ est active} \\ 0 & \textit{sinon} \end{cases} \tag{2.11}$$

Définition 2.6. ***Vecteur*** VT ***des transitions validées****: Le vecteur colonne* VT *de dimensions* $n \times 1$ *est décrit comme suit:*

$$VT_{(i,1)} = \begin{cases} 1 & \textit{si la transition } (i) \textit{ est validée} \\ 0 & \textit{sinon} \end{cases} \Rightarrow VT_{(i,1)} = \begin{cases} 1 & \textit{si } X \times E_i = E_i \\ 0 & \textit{sinon} \end{cases} \tag{2.12}$$

En considérant une transition (i), si toutes les étapes en entrées de cette transition sont actives, alors la transition (i) est validée (Règle 2). Dans ce cas l'entrée correspondante de VT vaut 1; elle vaut 0 dans le cas contraire.

Définition 2.7. ***Vecteur*** R ***des réceptivités des transitions****: Étant*

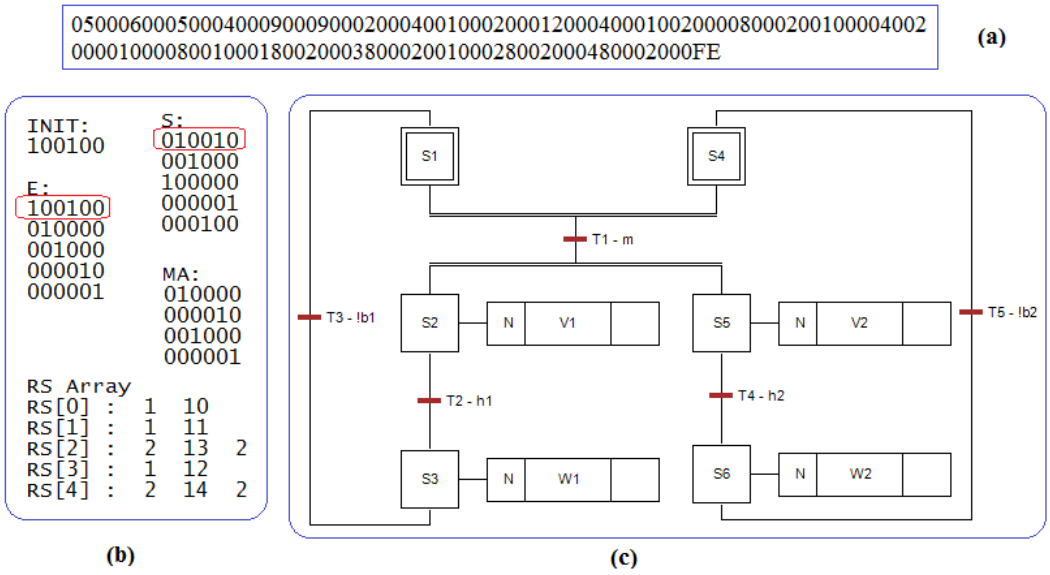


FIG. 2.4: Un grafcet obtenu par reconstruction du Grafcet

données les valeurs des entrées $I(t)$ *et la situation courante* $X(t)$, R *est évalué comme suit :*

$$R(t) = f(X(t),I(t))R_{(i,1)} = RPN\_Eval(RS_i) \Rightarrow R = RPN\_Eval(RS) \tag{2.13}$$

On utilise la méthode d'évaluation RPN_Eval présentée en section 2.2.5.2.

Définition 2.8. ***Vecteur des transitions franchissables*** FT*:*

$$FT_{(i,1)} = \begin{cases} 1 & \textit{si la transition } (i) \textit{ est franchissable} \\ 0 & \textit{sinon} \end{cases} \tag{2.14}$$

Une transition (i) sera franchissable si elle est validée et que sa réceptivité est 1. Donc, $FT_{(i,1)} = VT_{(i,1)} \bullet R_{(i,1)} \forall i \in \{1,2,\ldots,n\} \Rightarrow FT = VT \times R$

Définition 2.9. ***Vecteur d'activation et vecteur de désactivation*** *: Pour toute entrée i non nulle du vecteur* FT*, il faut récupérer dans le vecteur* VA *l'ensemble des étapes à activer se trouvant en sortie de la transition* (i) *et dans* VD *l'ensemble des étapes à désactiver situées en entrée de la transition* (i)*. Si* $FT_{(i,1)} \neq 0$ *alors, faire*

$$\begin{cases} \textit{Étapes à activer:} & EV = EV + S_i \\ \textit{Étapes à désactiver:} & DV = DV + E_i \end{cases} \tag{2.15}$$

Définition 2.10.

Vecteur ED *: Ce vecteur nous permet de déterminer toutes les étapes qui étaient actives dans la situation précédente et qui vont rester actives après*

le franchissement des transitions. Il est donc défini par:

$$ED = EV \bullet ED \tag{2.16}$$

Définition 2.11.
Vecteur VX ***de validation des étapes déjà actives*** *: Il faut déterminer les étapes qu'il ne faut pas désactiver avec les étapes qu'il faut activer. Ceci permet d'éviter d'omettre une étape quelconque, quelque soit sa position dans le graphe. Il permettra de trouver l'ensemble des étapes qui étaient actives et qui doivent rester actives.*

$$VX = \overline{DV} + ED \tag{2.17}$$

Définition 2.12. ***Calcul du vecteur*** O ***des sorties à activer*** *Il s'agit de voir si parmi les étapes actives, il y en a une à laquelle est associée une action. Ainsi,*

$$O_{(k,1)} = \begin{cases} 1 & si\ si\ X \bullet MA_k \neq 0 \\ 0 & sinon \end{cases} \tag{2.18}$$

2.2.4 Algorithme de mise en œuvre dynamique

L'algorithme de mise en œuvre est donné à la figure 2.5. Il est décrit comme suit :

Selon la règle 1 de l'évolution du Grafcet, initialement, la situation X est définie par la valeur du vecteur $INIT$.

EV, DV, ED, VX, E_i et S_i sont des vecteurs ligne $1 \times m$. Pour chaque cycle de scrutation, les entrées sont lues et les vecteurs R, VT et FT sont évalués conformément à la règle 2 de l'évolution du Grafcet. Les transitions sont franchies par l'évaluation et la mise à jour de la nouvelle situation constituée de nouvelles étapes activées et d'anciennes étapes qui doivent rester actives. Chaque transition franchissable (i) est franchie selon ce qui suit:
Conformément à la règle 3 de l'évolution du Grafcet, les étapes en amont de la transition franchissable (i) doivent être désactivées et ses étapes en aval doivent être activées: pour cela, deux vecteurs de ligne, DV et EV, sont utilisés. E_i est la ième ligne de la matrice E et contient des informations sur les étapes en amont de la transition (i); E_i est alors ajouté au vecteur de désactivation DV. De la même manière, S_i est la ième ligne de la matrice S et capture les informations sur les étapes en aval de la transition (i); c'est pourquoi le vecteur de ligne S_i est ajouté à EV. Cela est fait pour toutes les transitions franchissables (i).

Conformément à la règle 5 de l'évolution de Grafcet, les étapes qui étaient actives et qui doivent être activées doivent rester actives: pour cela, nous utilisons un vecteur de ligne VX pour la validation de la situation actuelle X. Il caractérise toutes les étapes qui doivent rester actives. Ces étapes sont obtenues dans deux cas:

- **Désactivation des étapes d'entrée de la transition** (i): les étapes d'entrée de la transition franchissable (i) doivent être désactivées. La

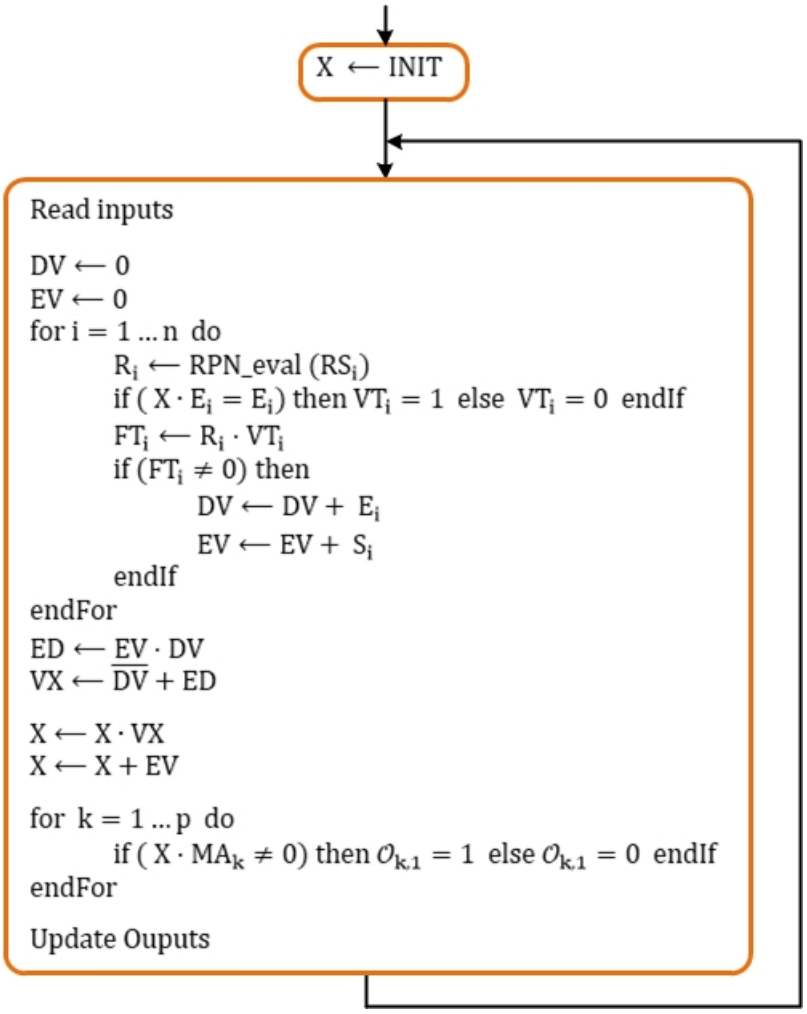


FIG. 2.5: Algorithme d'interprétation SIA du codage dynamique du Grafcet

désactivation est obtenue par la négation du vecteur DV ($\overline{DV}$). La négation de DV permet également de garder actives les étapes qui étaient actives et qui ne sont pas en entrée de la transition (i), ou de toute autre transition. De telles étapes appelées étape de puits [16] ne doivent pas être désactivées. Elles ne peuvent être désactivées que si un ordre de forçage est envoyé depuis un grafcet partiel de niveau supérieur ou lors de la désactivation d'une étape englobante englobant cette étape. Mais ici, nous ne traitons pas les ordres de forçage, ni des étapes englobantes, elles devraient donc rester actives.

– **Étapes devant être activées et désactivées**: Le deuxième cas concerne les étapes qui doivent être activées et désactivées simultanément ($ED \leftarrow EV \bullet DV$).

VX est évalué à l'aide de $VX \leftarrow \overline{DV} + ED$, pour caractériser toutes les étapes devant rester actives. Finalement, les transitions sont franchies

simultanément avec les instructions suivantes:

- Capture des étapes qui doivent rester actives: $X \leftarrow X \bullet VX$.
- Ajouter à X le vecteur d'activation EV des nouvelles étapes à activer: $X \leftarrow X + EV$.

Complexité d'exécution d'un cycle de scrutation de cet algorithme: Nous l'exprimons par C comme suit:
$C = n \times (max|R_i|, i = 1...n) + n \times m + n \times p = n \times (max|R_i|, i = 1...n+m+p)$.
Le premier terme correspond à l'évaluation de R, le second à l'évaluation de la nouvelle situation X et le troisième à l'évaluation des sorties O.

2.2.5 Modélisation des expressions Grafcet

La modélisation des expressions Grafcet dans le codage matriciel concerne en même temps leur représentation/codage et la méthode d'évaluation.

2.2.5.1 Méthode de codage: la fonction $RPNp$

La notation polonaise inversée (RPN [7]) est une forme de la notation polonaise. C'est une notation très ancienne (Lukasiewicz, 1929) utilisée pour faciliter l'évaluation automatique des expressions algébriques qui peuvent être arithmétiques ou logiques [30]. Très utilisée dans les calculatrices modernes, elle fournit une représentation d'une expression algébrique de manière rendre linéaire son évaluation. L'utilisation de la notation RPN suit deux phases :
La première consiste à analyser l'expression booléenne (exprimée dans un ordre infixé des opérateurs) pour en produire une représentation dans la notation RPN (exprimée dans l'ordre postfixé des opérateurs, et représentée par une pile). La deuxième phase est l'évaluation de l'expression après le remplacement des variables par leurs valeurs.

L'idée de l'utilisation de la notation RPN est de faire réaliser la première phase par un programme qui tourne sur l'ordinateur [8] et la deuxième phase par une fonction intégrée au programme de contrôle [9].

Par conséquent, l'ensemble des données résultant représentant le Grafcet pourrait être exécuté par un interpréteur (ou machine virtuelle) qui implémente l'algorithme d'interprétation du Grafcet avec toutes les autres fonctions utiles.

Une fois que les données sur les conditions sont bien structurées et codées, elles pourront être évaluées chaque fois que nécessaire par un algorithme très rapide de complexité linéaire et utilisant deux piles.

Les variables utilisée peuvent être des variables d'activité des étapes (X_j), des variables d'entrée ou de sortie, des variables de synchronisation ou des comparaisons construites avec des variables numériques ou logiques.

7. Reverse Polish Notation
8. un module de la plateforme de synthèse du Grafcet
9. s'exécutant sur la cible embarquée

Nous utilisons la notation RPN pour représenter toutes les conditions, en considérant que:

- Les opérateurs sont numérotés: 0 pour l'opérateur + (OR), 1 pour l'opérateur $*$ (AND), 2 pour l'opérateur de négation ! (NOT), 3 pour opérateur de front montant, 4 pour opérateur de front descendant.
- Les variables sont numérotées à partir du numéro 5.

La conversion en notation $RPNp$ peut être mise en œuvre comme suit:

- Analyser l'expression et construire l'arbre syntaxique (AST [10]) correspondante
- Parcourir en profondeur l'AST produite avec la méthode postfixée

Pour illustrer, considérons trois variables d'entrée a, b et c et la condition $!c * (a + b)$ pour la transition (i). L'arbre correspondant est présenté à la figure 2.6. Le parcourt en profondeur dans un ordre postfixé produit la liste d'éléments suivants (classés à partir de la tête de pile): $a\ b + c\ ! *$

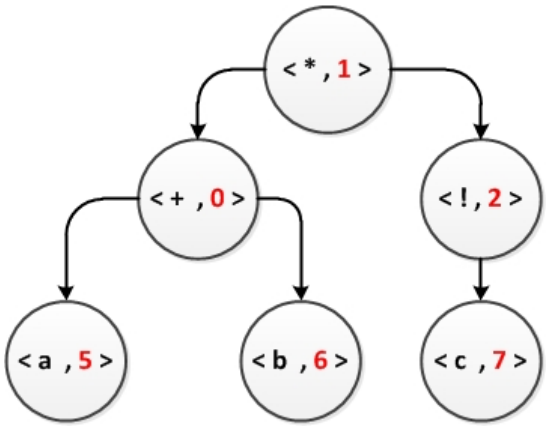


FIG. 2.6: L'AST pour l'expression algébrique booléenne $(a + b) \cdot \neg c$

NB: Il est aussi possible d'utiliser le parcours préfixé, auquel cas chaque élément visité est empilé. La liste est donc obtenue en dépilant complètement les éléments de la pile.

En utilisant le code associés à chaque symbole, le codage de cette expression donne : 5 6 0 7 2 1.
Puisque cette expression a six (06) éléments, cette taille peut être ajoutée au début de la liste pour en faciliter l'exploitation sous forme de tableau au sein d'un code de contrôle. On obtient donc:

$RS[i] = 6\ 5\ 6\ 0\ 7\ 2\ 1$

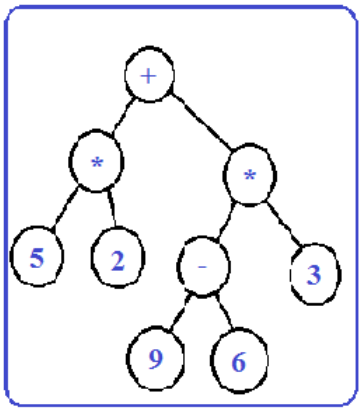
$RS[i]$ est donc une pile, avec $RS[i][0]$ qui est le nombre d'éléments de cette pile.

Pour accéder à la valeur d'une variable, une correspondance est établie entre les codes utilisés dans la matrice RS et un tableau *input* qui code les

10. Arbre de Syntaxe Abstraite

- **Construite l'AST** : Parser l'expression et construire un arbre de syntaxe abstraite
- **Construire la forme RPN de E** : Parcourir l'AST de façon postfixée et générer une séquence S (**liste**) de tokens
- **Evaluation de l'expression RPN** : on utilise deux piles P1 et P2.
  - On empile dans P1 tous les éléments de la séquence S de la fin au début.
  - On dépile les éléments de P1 au fur et à mesure jusqu'à ce qu'il soit vide. A chaque étape, on fait :
    - Si **opérande** Alors on empile dans P2
    - Si **opérateur** Alors
      - V2 ← dépiler P2
      - [v1 ← dépiler P2]
      - effectuer le calcul v ← opérateur (v1*[,v2]*)
      - empiler v dans P2

FIG. 2.7: Algorithme d'évaluation d'une expression codée en RPN



(a)

Pas	P1 [tête ... queue]	P2 [tête ... queue]	Opération
0 (init)	5 2 * 9 6 – 3 * +		
1	2 * 9 6 – 3 * +	5	
2	* 9 6 – 3 * +	2 5	
3	9 6 – 3 * +	10	**2*5**
4	6 – 3 * +	9 10	
5	– 3 * +	6 9 10	
6	3 * +	3 10	**9 – 6**
7	* +	3 3 10	
8	+	9 10	**3*3**
9		**19**	**10+9**

(b)

FIG. 2.8: Évaluation RPN de l'expression $5 * 2 + (9 - 6) * 3$

valeurs des entrées du système: Si c est le code d'une variable d'entrée quelconque du système, alors $input[c - 5]$ est la valeur de cette variable.

2.2.5.2 Méthode d'évaluation

Cette méthode est clairement explicitée dans [30]. Le principe est présenté sur la figure 2.7.

Pour illustration de cet algorithme, l'expression $5 * 2 + (9 - 6) * 3$ dont l'AST est donnée sur la figure 2.8(a), le déroulement de l'évaluation est donné dans le tableau de la figure 2.8(b). On s'arrête dès lors que $P1$ est vide et le résultat se trouve dans $P2$, i.e. 19. On voit très bien que ce calcul est linéaire en 9 étapes, correspondant au nombre d'éléments dans l'expression RPN.

```
boolean RE_table[RisingEdgeMax]; //Set initially at 0's
boolean actualRE;
int actualREId;
V ← Dépiler(P1)
Si V = Operator ^ (rising edge) Alors
        actualREId ← Dépiler(P1)
        value ← Dépiler(P2)
        boolean tmp ← RE_table [actualREId];
        actualREId ← tmp;
        retourner (b==0)&&(tmp==1);
FinSi
```

FIG. 2.9: Algorithme d'évaluation d'une expression Front Montant en RPN

2.2.5.3 Évaluation des expressions avec front montant

Parce qu'on numérote simplement les variables dans les expressions de réceptivité, il est aussi possible de donner le numéro du front montant. Cependant, il faut considérer en plus de la valeur actuelle la valeur précédente de l'expression considérée. On peut se servir d'un tableau *RE_table* de booléens (d'une taille limite $RE\_Max = 16$ par exemple) pour sauvegarder leurs valeurs. Après chaque phase d'évaluation, il faudra sauvegarder dans chaque entrée i de *RE_table* l'ancienne valeur correspondante. L'algorithme de traitement d'un nœud front montant est donné sur la figure 2.9

Il en est de même de l'évaluation d'une expression avec Front Descendant[11].

2.3 Plateforme de synthèse du Grafcet

L'approche de synthèse du Grafcet par matrices de codage a été mise en œuvre à travers une plateforme que nous avons développée à cet effet sous *Microsoft Visual Studio* (MVS), baptisée $GrafcetConverter$.[12]. Dans cet environnement, la plateforme de synthèse est codée en langage C#, qui est un langage orienté objet.

Son architecture générale est présentée sur la figure 2.10. Cette plateforme est constituée de trois principaux modules décrits comme suit :

- **L'outil d'édition UniSim** : *UniSim* [65] est un outil libre développé par des chercheurs de l'université de *Naples-Frédéric-II*, Faculty of Engineering. Déjà évoqué en section 1.5.2, cet outil permet d'éditer

11. Falling edge

12. MVS est une suite de logiciels de développement pour Windows et mac OS conçue par $Microsoft$. Ses principaux langages de programmation sont C# et VB.NET

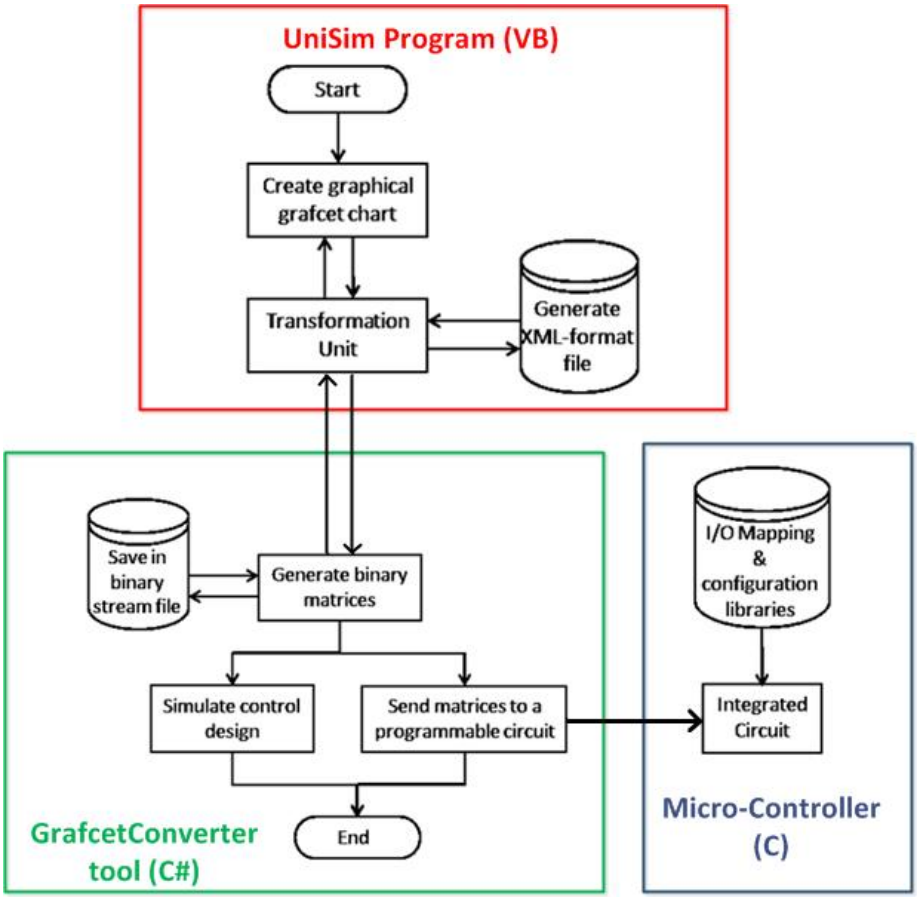


FIG. 2.10: Architecture générale de *GrafcetConverter*

des modèles SFC ou Grafcet. Cependant, il présente des limites, en l'occurrence de l'absence de prise en compte des événements (fronts montants/descendants) et de certaines contraintes temporelles. Face à ces limites, nous en avons réalisé une extension de cet outils.

C'est une interface utilisateur graphique qui offre une galerie d'objets graphiques spécifiques pour les éléments du Grafcet (les étapes, les transitions, les conditions de transition, les actions, ...). L'utilisateur peut facilement sélectionner ces objets graphiques, les placer sur une page de dessin et les relier afin de créer des instances du Grafcet. UniSim offre une sérialisation des données au format XML[48, 38] dont les langages offrent des fonctionnalités pour son analyse (parsing) [22].

- **L'outil GrafcetConverter** : ce module que nous avons construit fait référence au premier pour l'édition des modèles Grafcet. Il nous permet d'extraire directement les éléments Grafcet d'un fichier au format XML et de les stocker dans un modèle objet interne à partir duquel nous calculons les matrices de codage qui en découlent. Cet outil fournit un moteur de simulation Grafcet pour la vérification du modèle, avant de transmettre les données Grafcet à un microcontrôleur. De plus, nous fournissons également un module de génération de code selon les deux approches évoquées (*équations algébriques* et *matrices de codage du Grafcet*). Nous avons ensuite généré du code C pour deux microcontrôleurs: $dsPIC30F4013$ et $ATmega328P$ qu'équipent respectivement les cartes d'expérimentation *EasyDsPIC4A* et *Arduino UNO* (tableau 4.1).

- **Circuit intégré**: la carte est programmée en compilant et en transférant le code dans sa mémoire EEPROM via un câble USB ou un port COM RS-232. Le microcontrôleur est alors configuré et prêt à recevoir un ensemble binaire de données issues du codage matriciel du Grafcet, que l'on transfère à travers le port port série (ou port COM RS-232). Pour le code généré par équations algébriques de codage du Grafcet (section 4.4.1), la structure du Grafcet est noyée dans ses instructions. Il n'a besoin que d'être compilé et transféré en mémoire EEPROM pour contrôler le système modélisé par le grafcet de départ. Une fois terminé et la carte réinitialisée, elle fonctionne et sert de contrôle permanent du système en interagissant avec son environnement.

Dans les sections suivantes, nous donnons une description des fonctionnalités et des algorithmes conçus pour réaliser cette plateforme de synthèse.

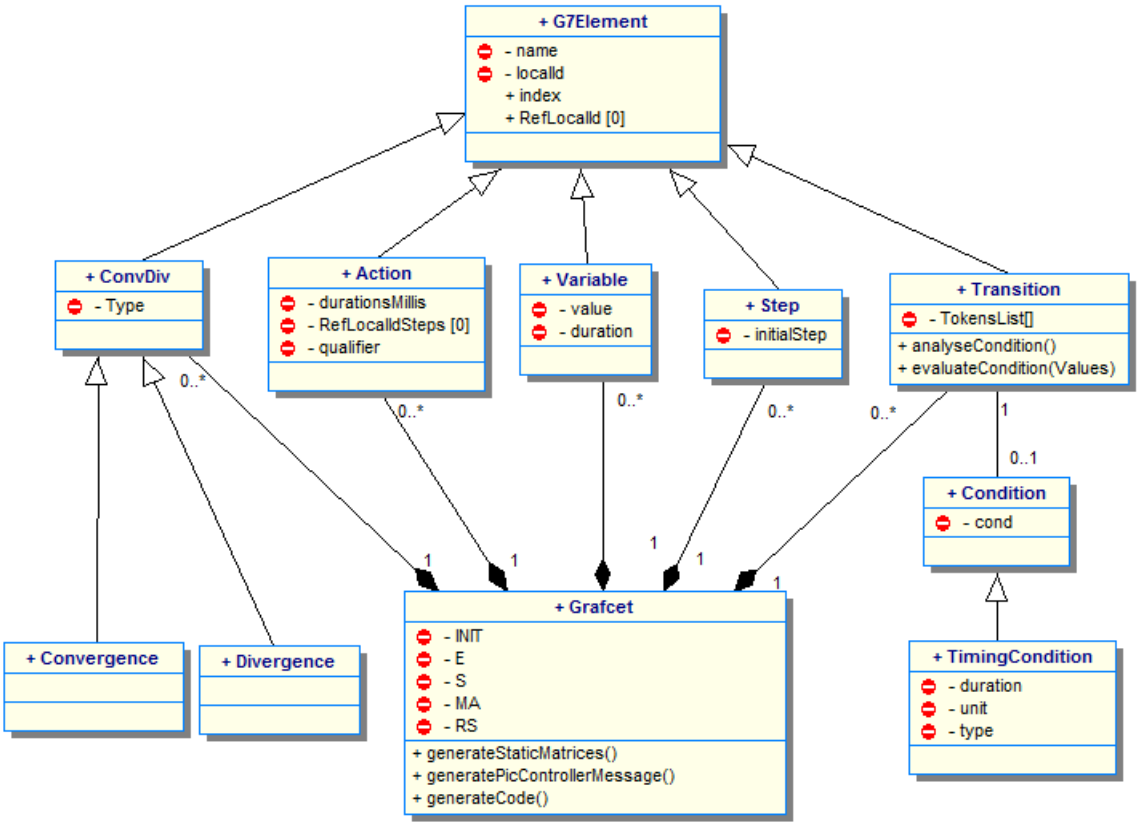


FIG. 2.11: Diagramme de classes des éléments du Grafcet issus de UniSim

2.3.1 Représentation des éléments du Grafcet

L'outil UniSim d'édition des modèles Grafcet offre une représentation sérialisée des modèles sous le format PLCOpen XML [13]. L'étude de la structure de ces fichiers PLCOpen XML permet de proposer une structuration de classes utilisées pour développer un parseur. Celui-ci a pour but d'analyser le fichier XML du modèle et d'en extraire toutes les informations pertinentes pour la représentation du modèle.

Les classes retenues dans ce modèle sont présentées sur le diagramme de classes de la figure 2.11. Les actions sont des variables de sortie associées aux étapes par des blocs d'actions. Ces blocs ne sont pas représentés, mais chaque variable référence des étapes auxquelles l'action correspondante est associée en utilisant l'attribut $RefLocalIdSteps$. Les informations liées aux positions (x et y) des éléments du modèle sur le graphique ne nous intéressent pas, car ne capture pas sa fonctionnalité. La réalisation de ce diagramme est donc guidée par la structure du fichier XML d'un grafcet, tel que présentée à la figure 2.12 pour le grafcet de la figure 2.4(c).

Dans ce diagramme de classes, les positions relatives entre étapes et transitions ne sont pas représentées et ne peuvent être obtenues directement.

13. Extensible Markup Language

```
<body>
  <SFC>
    <step name="S1" height="44" width="44" localId="1" initialStep="true" negated="false">
    <step name="S2" height="44" width="44" localId="2" initialStep="false" negated="false">
    <actionBlock>
    <step name="S3" height="44" width="44" localId="3" initialStep="false" negated="false">
    <actionBlock>
    <step name="S4" height="44" width="44" localId="4" initialStep="true" negated="false">
    <step name="S5" height="44" width="44" localId="5" initialStep="false" negated="false">
      <position x="386" y="186" />
      <connectionPointIn>
        <connection refLocalId="13" />
      </connectionPointIn>
    </step>
    <actionBlock>
    <step name="S6" height="44" width="44" localId="6" initialStep="false" negated="false">
    <actionBlock>
    <transition height="44" width="44" localId="7">
      <position x="292" y="144" />
      <connectionPointIn>
        <connection refLocalId="12" />
      </connectionPointIn>
      <condition>
        <reference name="m" />
      </condition>
    </transition>
    <transition height="44" width="44" localId="8">
    <transition height="44" width="44" localId="9">
    <transition height="44" width="44" localId="10">
    <transition height="44" width="44" localId="11">
    <simultaneousConvergence localId="12">
    <simultaneousDivergence localId="13">
      <position x="0" y="0" />
      <connectionPointIn>
        <connection refLocalId="7" />
      </connectionPointIn>
    </simultaneousDivergence>
  </SFC>
</body>
```

FIG. 2.12: Le «body» du fichier Xml du grafcet en Fig. 2.4(c)

En effet, les modèles édités dans *UniSim* sont sérialisés dans le format XML *PLCOpenXml*. Cette sérialisation utilise des identifiants (*id* et *localId*) pour retrouver les positions relatives entre les différents objets manipulés.

Sur la figure 2.12, on peut remarquer que chaque étape, chaque transition et chaque convergence/divergence a un identifiant (*localId*). Par exemple, l'étape $S1$ a pour *localId* 1, il vaut 7 pour la transition $T1$ et 13 pour la **divergence en ET** (*simultaneousDivergence*). La transition $T1$ ($localId = 7$) a pour réceptivité m. On remarque aussi que dans ce fichier, chaque objet décrit ses données et sa position relativement à l'objet en entrée (avec la balise $< connectionPointIn >$).

Sur la figure 2.12, on peut remarquer que chaque étape, chaque transition et chaque convergence/divergence a un identifiant (*localId*).

Nous avons alors réalisé un parseur de fichiers XML basé sur le DOM [14]

14. Document Object Model

[70]. Il s'agit d'un standard du W3C[15] qui définit une technologie grâce à laquelle il est possible de lire un document XML et d'en extraire différentes informations (éléments, attributs, commentaires, etc) afin de les exploiter. Au travers de DOM, le W3C fournit une recommandation, c'est-à-dire une manière d'exploiter les documents XML. Cette recommandation est complètement indépendante de toute plate-forme et langage de programmation. La plupart des langages de programmation en proposent une implémentation. Le DOM représente un document sous forme d'arbre et propose des opérations pour facilement l'interroger.

Ce parseur lit un fichier XML (de modèle Grafcet) et construit tous les objets identifiés à l'intérieur de celui-ci pour les représenter dans une instance de la classe $Grafcet$. Cette tâche est faite de façon linéaire tout en interrogeant chaque fois la structure d'arbre (DOM) sur les éléments d'une certaine typologie (étapes, transitions, action d'une étape, convergences, divergences, etc).

2.3.2 Algorithme de construction des matrices

Lorsque l'inventaire des objets du Grafcet est réalisé à l'intérieur d'une instance $Grafcet$, une analyse de ces objets doit être faite à dessein de construire les matrices de codage du Grafcet. Il est question d'identifier dynamiquement les positions relatives des objets pour aboutir aux valeurs des matrices $INIT$, E, S et MA.

L'identification des éléments est rendue difficile à travers la présence des éléments du Grafcet tels que (convergence/divergence en ET/OU), lesquels séparent structurellement les étapes et les transitions pour offrir des alternatives et des synchronisations. En effet, la figure 2.12 met en exergue l'étape $S5(localId = 5)$, la transition $T1(localId = 7)$ et la *la divergence en ET* ($simultaneousDivergence$) dont $localId = 13$.

L'étape $S5$ indique dans $connectionPointIn$ la référence à la structure en entrée avec pour $refLocalId = 13$. Il s'agit de l'unique *divergence en ET* de ce modèle: **la *divergence en ET* est donc en entrée de l'étape** $S5$. De même, dans la *la divergence en ET* ($simultaneousDivergence$), $connectionPointIn$ indique la référence à l'objet dont l'identifiant $refLocalId = 7$. Cet identifiant correspond à la transition $T1$. Il en découle que **la transition** $T1$ **est en amont de cette *divergence en ET***.

Ces deux éléments permettent de réaliser que la transition $T1$ est en entrée de l'étape $S5$, ou encore que **l'étape** $S5$ **est en aval de la transition** $T1$, ce qui permet e positionner 1 à l'entrée correspondante de la matrice S (qui décrit les étapes en aval/sortie des transitions).

Le tableau 2.1 décrit la typologie de tous les éléments manipulés dans le fichier Xml de sérialisation des modèles Grafcet.

15. W3C: *World Wide Web Consortium* est un organisme de standardisation à but non lucratif. De même que DOM, il est à l'origine de XPath qui en est une alternative

TAB. 2.1: Typologie des objets du Grafcet dans le fichier Xml

step	description d'une étape
transition	description d'une transition
simultaneousConvergence	description d'une convergence en ET
simultaneousDivergence	description d'une divergence en ET
selectionConvergence	description d'une convergence en OU
selectionDivergence	description d'une divergence en OU
actionBlock	description d'un bloc d'actions associées à une étape
variable	description d'une variable (locale ou globale)

```
For Each step in Step_list do
        INIT[step.index] ←0
        If (step.IsInitialStep)
                INIT[step.index]←1
        End If
End For Each
```

(a) Calcul de *INIT*

```
MA[i, j] ← 0; (0 <= i < nb of actions and 0 <= j < nb of steps)
For Each variable in Variable_list do
        If (variable. IsAnAction)
                For each localId in variable. RefLocalIdSteps do
                        Id ← getStepIndexById(localId)
                        MA[variable.index, Id] ← 1
                End For
        End If
End For Each
```

(b) Calcul de *MA*

FIG. 2.13: Algorithme du calcul des matrices $INIT$ et MA

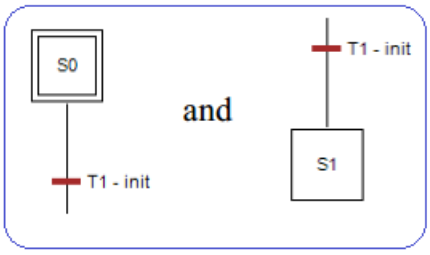
Pour automatiser le calcul des matrices de codage du Grafcet, nous avons réalisé des algorithmes de construction qui utilisent les listes suivantes :

- *Variable_list* : liste des variables, construite à partir des *actionBlock* des étapes,
- *Step_list* : liste des étapes identifiées dans le modèle,
- *Transition_list* : liste des transitions identifiées dans le modèle,
- *Simultaneous_Conv_list*: liste des convergences en ET,
- *Simultaneous_Div_list*: liste des divergences en ET,
- *Selection_Conv_list* : liste des convergences en OU,
- *Selection_Div_list* : liste des divergences en OU,

L'algorithme de la figure 2.13 décrit la construction des vecteurs $INIT$ et MA.

Pour calculer les matrices E et S, trois cas sont identifiés. L'un concerne le cas de simple connection entre étapes et transitions, et les deux autres concernent les étapes et transition séparées par des convergences/divergences en ET et en OU:

- **Cas 1** : Le cas des connections simples entre étapes et transitions. L'algorithme correspondant est donné en figure 2.14.
- **Cas 2** : concerne les connections entre étapes et transitions à travers les convergences en ET et les divergences en OU. Il est traité par



(a) Illustration

```
For Each transition in Transition_list do
    For Each step in Step_list do
        E[transition.index, step.index] ← 0;
        // first case
        If(transition.refLocalId = step.localId) do
            E[transition.index, step.index] ← 1
        End If
        // second case
        If (step.refLocalId = transition.localId)
            S[transition.index, step.index] ← 1
        End If
    End for each
End for each
```

(a) Algorithme (E et S)

FIG. 2.14: Algorithme du calcul de E et S (cas simple)

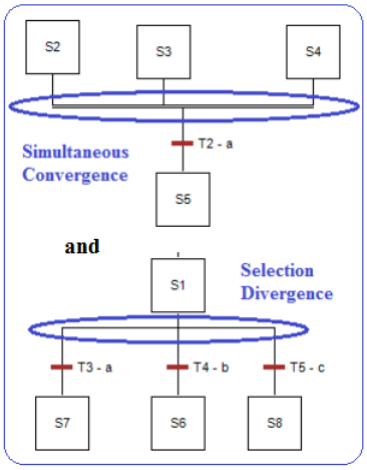
l'algorithme présenté à la figure 2.15 qui a pour rôle de compléter le remplissage de la matrice E.

– **Cas 3** : concerne les connections entre étapes et transitions à travers les divergences en ET et les convergences en OU. Il est traité par l'algorithme présenté à la figure 2.16 qui a pour rôle de compléter le remplissage de la matrice S.

Pour ce qui concerne la matrice MA, après avoir donné un *index* à chaque variable utilisée, son calcul est réalisé en analysant l'expression de la réceptivité puis en la représentant sous le format RPN décrit en 2.2.2.3.

2.3.3 Simulation du modèle Grafcet

Lorsque les matrices de codage sont chargées, des simulations peuvent être faites dans l'environnement pour scruter pas-à-pas l'évolution dynamique du grafcet. La figure 2.17 est une capture de l'interface de simulation, contenant les données sur le modèle grafcet présenté à la figure 2.4, avec ses matrices de codage de la statique. Il est alors possible de mettre une entrée à 1 en cochant le box de la variable correspondante. Par défaut, le résultat des vecteurs de la situation X du grafcet et des sorties (O) est affiché. En cochant le box *display all dynamic vectors*, on affiche automatiquement tous les autres principaux vecteurs de la dynamique du Grafcet : le vecteur R des réceptivités, celui VT des transitions validées et celui FT des transitions franchissables. Ce simulateur exécute de façon continue ou *step-by-step* l'algorithme d'interprétation du codage matriciel du Grafcet détaillé en section 2.2.4. Il est aussi possible de choisir la périodicité du *timer* (ou horloge) qui calcule les durées d'activité des variables. Ces durées sont utiles au cas où le grafcet en présence manipule les conditions temporelles.

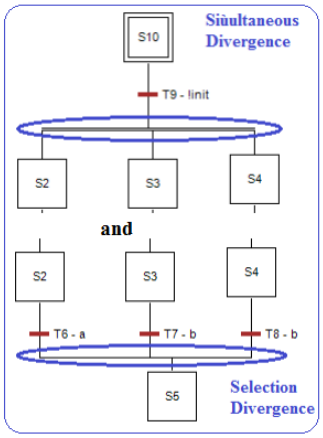


(a) Illustration

```
For each conv_div in Simultaneous_Conv_list UNION Selection_Div_list do
    For each StepLocalId in conv_div. RefLocalIdList do
        For each transition in transition_list do
            If( transition.refLocalId = conv_div.localId ) do
                // look for the steps in input of conv_div
                For Each step in Step_list do
                    If( conv_div.refLocalId = step.localId)
                        E[transition.index, step.index] ← 1
                    End If
                End For Each
            End If
        End For Each
    End For Each
End For Each
```

(b) Algorithme de complétion de E

FIG. 2.15: Algorithme de la complétion de E (avec conv. en ET et div. en OU)



(a) Illustration

```
For Each conv_div in Simultaneous_Div_list UNION Selection_Conv_list do
    For each TransLocalId in conv_div. RefLocalIdList do
        For Each step in Step_list do
            If( step.refLocalId = conv_div.localId ) do
                // look for the steps in input of conv_div
                For Each transition in Transition_list do
                    If( conv_div.refLocalId = transition.localId)
                        S[transition.index, step.index] ← 1
                    End if
                End for each
            End If
        End For Each
    End For Each
End For Each
```

(a) Algorithme de la complétion de S

FIG. 2.16: Algorithme de la complétion de S (avec div. en ET et conv. en OU)

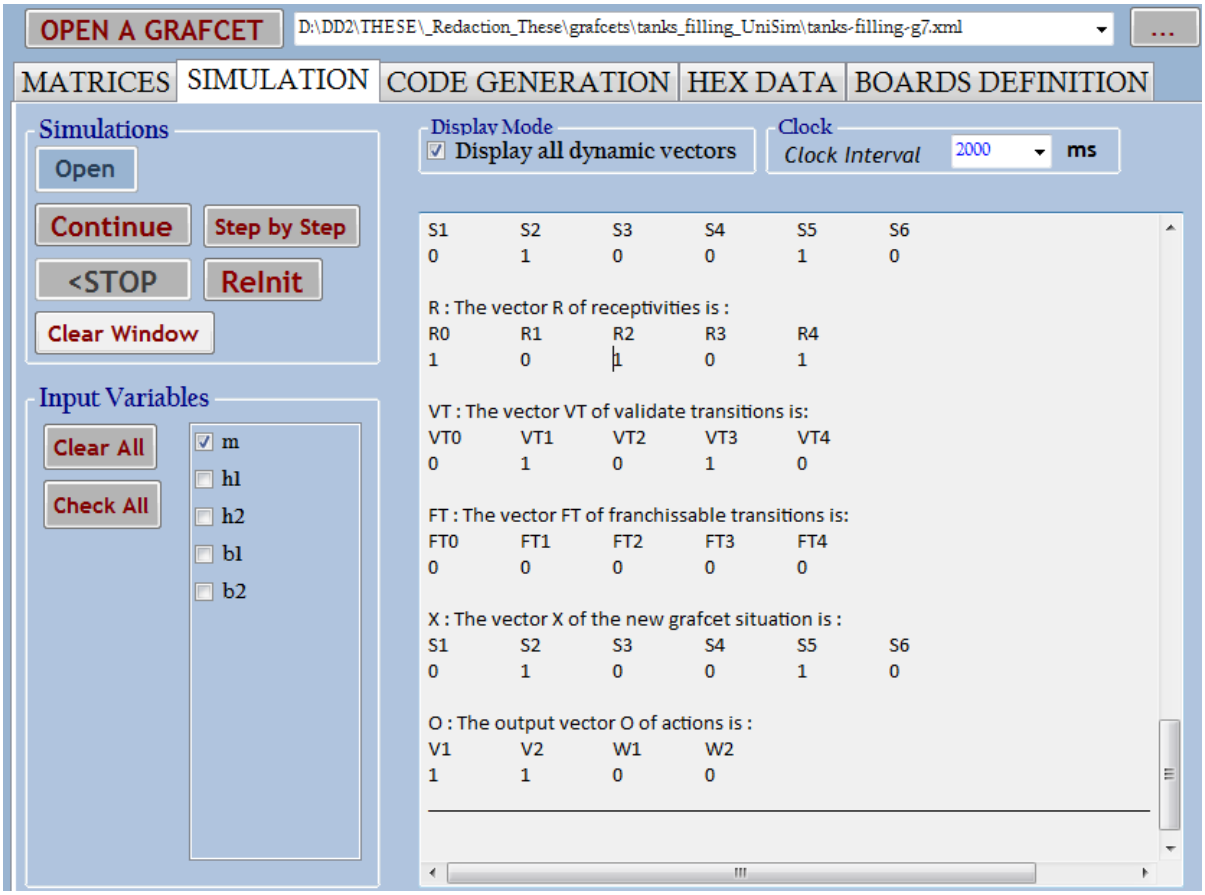


FIG. 2.17: Simulation de l'algorithme d'évolution dynamique sur un modèle grafcet

2.4 Vérification des modèles et génération du code

Après avoir procédé au calcul des matrices de codage dans l'environnement, on peut passer à la génération du code de contrôle sur des cibles microcontrôleur.

2.4.1 Modélisation des cibles

Pour chaque cible, on a représente des caractéristiques matérielles (telles que la) et des caractéristiques fonctionnelles que sont les broches et leur mode de fonctionnement (entrée/sortie), la méthode de configuration des pins en entrée ou en sortie, un timer, des librairies nécessaires, la méthode de lecture des entrées et d'écriture des sorties, les méthodes de lecture et d'écriture des données en mémoire (Flash), ... Certaines de ces caractéristiques sont réalisées à l'intérieur des fichiers spécifiques. Cette description des modèles a été réalisée en XML. Par exemple, la figure 2.18 présente un aperçu de ce fichier pour le microcontrôleur $ATmega328P$ intégré à la plateforme de synthèse.

2.4.2 Vérification des modèles en entrée

Une fois le modèle Grafcet construit, l'environnement offre la possibilité de choisir une cible pour ensuite procéder au mapping des entrées/sorties du modèle avec celles de la cible choisie.

Le mapping consiste à vérifier que le modèle peut être réalisé à l'aide de la cible choisie. Précisément, il est question de voir si la cible microcontrôleur dispose d'un nombre nécessaire de broches pour la connection de toutes les entrées et sorties du modèle. En supposant que le modèle a $nbIn$ entrées et $nbOut$ sorties, et que la carte possède en tout N pins accessibles et configurables en entrée et en sortie, on a la condition donnée par l'inéquation 2.19, qui doit toujours être respectée.

$$nbIn + nbOut \leq N \tag{2.19}$$

On vérifie aussi que la cible choisie dispose d'au moins un timer configuré pour la mesure du temps, au cas où le modèle Grafcet contient des temporisations dans ses conditions. Avant la génération de code, on choisit pour chaque variable d'entrée/sortie du modèle une pin (ou broche) pour son mapping sur la carte. Un exemple est présenté à la figure 2.19. Ce mapping est en fait une application injective de l'ensemble des entrées/sorties vers l'ensemble des pins disponibles. On vérifie enfin que deux pins différentes ne sont pas choisies pour la même variable.

```
<microcontroller name="ATmega328P" family = "Atmel AVR-Arduino Uno" manufacturer="Microchip Tech.">
    <architecture size = "8" unit = "bits"/>
    <processor speed="16" unit ="MHz"/>
    <RamMemory type = "SRAM" size = "2" unit = "Ko"/>
    <ProgramMemory type = "flash" size = "32" unit = "Ko"/>
    <DataMemory type = "EEPROM" size = "1" unit = "Ko"/>
    <DigitalPINS>
        <PIN address = "0" type = "IO"/>
        <PIN address = "1" type = "IO"/>
        <PIN address = "2" type = "IO"/>
        <PIN address = "3" type = "IO"/>
        <PIN address = "4" type = "IO"/>
        <PIN address = "5" type = "IO"/>
        <PIN address = "6" type = "IO"/>
        <PIN address = "7" type = "IO"/>
        <PIN address = "8" type = "IO"/>
        <PIN address = "9" type = "IO"/>
        <PIN address = "10" type = "IO"/>
        <PIN address = "11" type = "IO"/>
        <PIN address = "12" type = "IO"/>
        <PIN address = "13" type = "IO"/>
    </DigitalPINS>
    <AnalogPINS>
        <PIN address = "A0" type = "In"/>
        <PIN address = "A1" type = "In"/>
        <PIN address = "A2" type = "In"/>
        <PIN address = "A3" type = "In"/>
        <PIN address = "A4" type = "In"/>
        <PIN address = "A5" type = "In"/>
    </AnalogPINS>
    <usefulFunctions>
        <SourceCode_Header path="../boards_settings/ATmega328P/sc_header.c"/>
        <SourceCode_ReadPins path="../boards_settings/ATmega328P/sc_read_pins.c"/>
        <SourceCode_WritePins path="../boards_settings/ATmega328P/sc_write_pins.c"/>
        <SourceCode_PINModeConfig path="../boards_settings/ATmega328P/sc_pinMode_config.c"/>
        <SourceCode_TimerDefinition path="../boards_settings/ATmega328P/sc_timer_definition.c"/>
        <SourceCode_ReadInMemory path="../boards_settings/ATmega328P/sc_readInMemory.c"/>
        <SourceCode_WriteInMemory path="../boards_settings/ATmega328P/sc_writeInMemory.c"/>
    </usefulFunctions>
</microcontroller>
```

FIG. 2.18: Caractéristiques de configuration du microcontrôleur ATmega328P

2.4.3 Génération du code avec interpréteur

Il s'agit dans une première phase de générer du code qui implémente l'algorithme d'interprétation du Grafcet. La deuxième phase consistera à lire un Grafcet et à générer les données via le codage matriciel pour ensuite envoyer à la carte. Le grafcet correspondant sera alors exécuté par le code interpréteur préalablement chargé pour contrôler le système. Nous présentons la préparation des données à envoyer à la carte et la génération du code de l'interpréteur.

2.4.3.1 Préparation des données par codification en hexadécimal

Cette approche fait ressortir en tout quatre phases pour passer du Grafcet aux données exploitables par une cible pour l'exécution du Grafcet de départ. Ces étapes sont illustrées sur la figure 2.20.

Le formatage des données en hexadécimal est présenté dans tableau de la figure 2.21. Lorsque les données ainsi formatées sont récupérée par

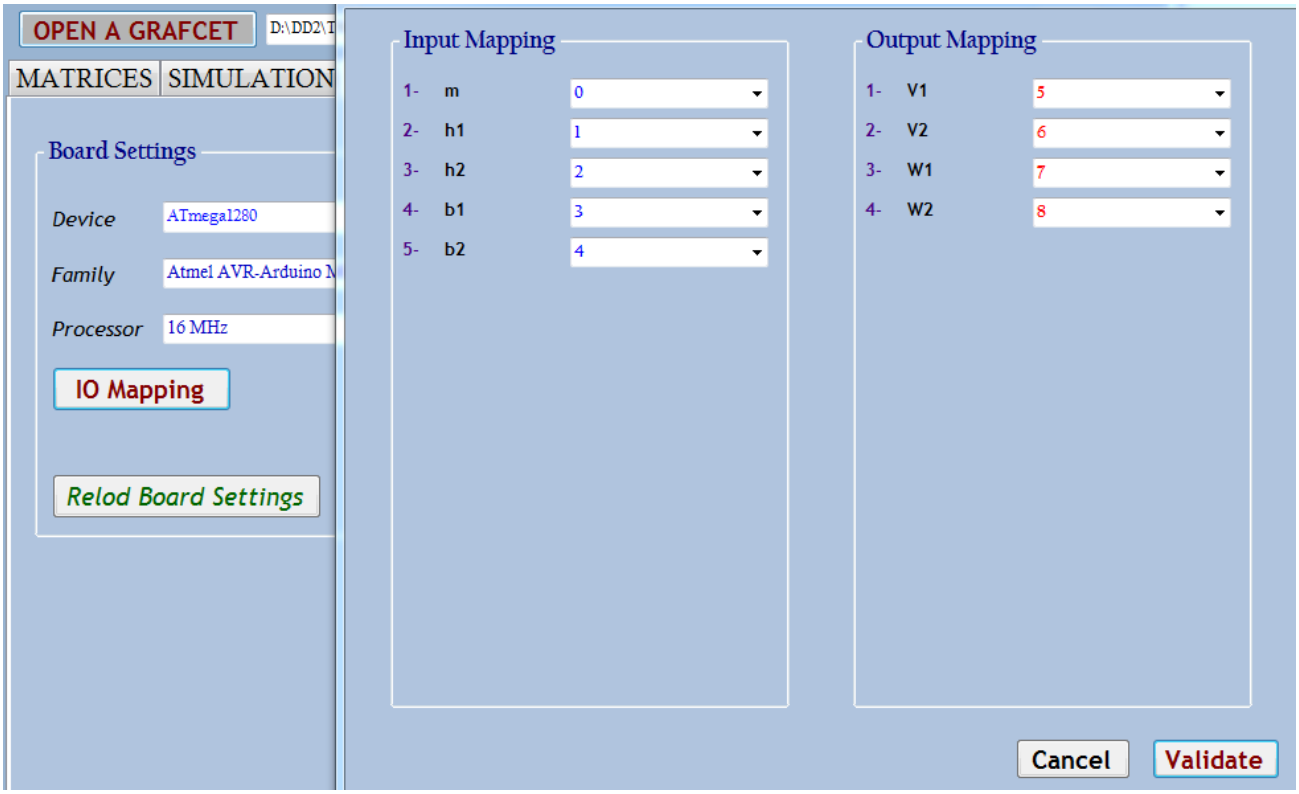


FIG. 2.19: Mapping des variables d'entrée/sortie du modèle grafcet

le programme de contrôle exécuté par la cible microcontrôleur, elles sont formatées selon la structure de données présentée dans le listing 2.1 :

Listing 2.1: Structure du tableau DatainEE des données sur le Grafcet

```
unsigned int DatainEE [] = {
        Number_of_transitions,
        Number_of_steps,
        Number_of_action,
        Number_of_inputs,
        E,
        S,
        X_INIT,
        MA,
        RS
}
```

Dans ce tableau en figure 2.21, les données sont récupérées sous forme d'entier non signé[16], i.e. 16 bits en tout[17], ou encore 2 octets. C'est pourquoi chaque élément est codé sur 2 octets. Il s'agit des opérateurs, des variables en général, des variables d'entrée, de sortie, d'étape et des variables tempo-

16. `unsigned int`
17. au lieu de 15 bits pour les *int*

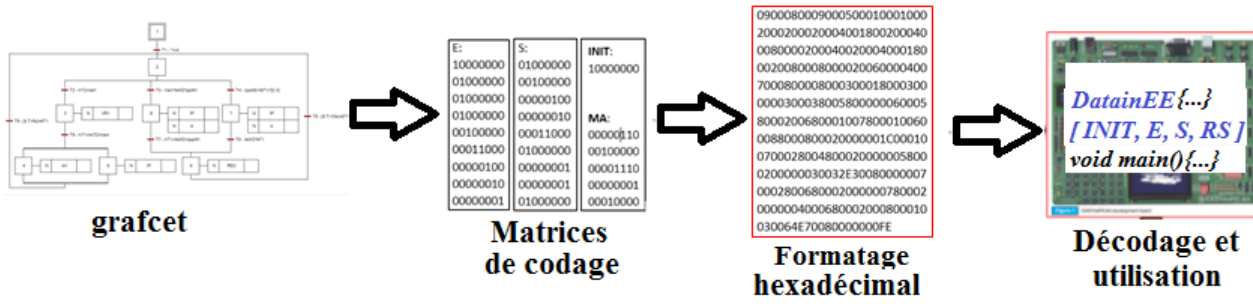


FIG. 2.20: Etapes d'obtention des données

relles. La deuxième colonne présente les données en binaire. À travers un codage en hexadécimal (colonne 3), on ressort la formule qu'il faut utiliser pour détecter la nature d'une donnée. Ceci décrit un protocole de communication entre l'environnement de synthèse et la le contrôleur embarqué qui exécute le code de contrôle. C'est ce formatage qui facilite la modification du programme exécuté par la carte en n'envoyant que les données de codage du Grafcet (à travers un port série de communication RS3xx).

2.4.3.2 Génération du code de l'interpréteur et architecture

Le code de l'interpréteur est généré en fonction des caractéristiques du microcontrôleur choisi en entrée. Un flux de sortie est créé et dans lequel sont ajoutée les informations nécessaire du code de l'interpréteur, au fur et à mesure que l'on parcoure les objets du Grafcet lu. Le schéma général du fonctionnement de la génération de code Grafcet pour interprétation par l'interpréteur est donné en figure 2.22. Dans ce schéma, on fait ressortir le fait que les données sur le grafcet peuvent être chargées après avoir programmé la carte par le code de l'interpréteur. Ces données résident très souvent en mémoire $EEPROM$ et peuvent être lues par l'interpréteur pour son exécution.
T_{11} est la transformation du grafcet de spécification en matrices de codage et T_{12} est l'opération inverse. Ces matrices peuvent être simulées en exécutant une implémentation de l'algorithme d'interprétation. Elles peuvent aussi être chargée directement dans un microcontrôleur exécutant une implémentation de cet algorithme d'interprétation du Grafcet.
T_2 est la transformation que se sert de l'algorithme d'interprétation du Grafcet et de la description d'un microcontrôleur pour générer du code pour ce microcontrôleur.

Pour le microcontrôleur $ATmega328P$, une présentation sommaire du code de l'interpréteur généré pour le grafcet d'exemple est présentée à l'annexe 5.7. T_{11}, T_{12} et T_2 sont toutes implémentées dans la plateforme de synthèse du Grafcet.

Element	What to find	CODE	Formulae
Operators 0b 0PPP Y Y=XXXX XXXX XXXX CODE & 0x8000 = 0x0000	OR(+)	0b 0000 0000 0000 0000 = 0x0000	CODE & 0x0000
	AND (*)	0b 0001 0000 0000 0000 = 0x1000	CODE & 0x1000
	NOT(!)	0b 0010 0000 0000 0000 = 0x2000	CODE & 0x2000
	RISING EDGE(^)	0b 0011 0000 0000 0000 = 0x3000	CODE & 0x3000
	True	0b 0100 0000 0000 0000 = 0x4000	CODE & 0x4000
Variables 0b 1VVX Y Y=XXXX XXXX XXXX CODE & 0x8000 = 0x8000	Input	0b 100X XXXX XXXX XXXX	CODE & 0xE000 = 0x 8000
	Output	0b 101X XXXX XXXX XXXX	CODE & 0xE000 = 0x A000
	Step	0b 110X XXXX XXXX XXXX	CODE & 0xE000 = 0x C000
	Timing	0b 111X XXXX XXXX XXXX	CODE & 0xE000 = 0x E000
Input variable *Max = 255*	Input number	0b 1000 0000 NNNN NNNN	N°= CODE & 0x00FF
Output variable *Max = 255*	Output number	0b 1010 0000 NNNN NNNN	N°= CODE & 0x00FF
Step variable *Max = 32*	Step number	0b 1100 0000 000N NNNN	N°= CODE & 0x00FF
Timing variable ***(Unit = seconds)***	Step Number Max = 2^5=32	0b 111S SSSS DDDD DDDD	N°= CODE & 0x1F00
	Duration of TV Max Duration = 2^8=255 s	0b 111S SSSS DDDD DDDD	N°= CODE & 0x00FF

FIG. 2.21: Tableau de formatage des données codées

2.4.4 Génération du code Grafcet avec équations algébriques

Cette approche est une alternative pour la génération du code de l'interpréteur Grafcet (section 2.4.3). En effet, lorsque le système devient grand, on observe une grande lenteur lors de l'exécution de l'interpréteur, qui doit analyser un volume de données relativement grand pour exécuter le grafcet correspondant. Les équations algébriques du Grafcet sont une solution de représentation formelle du Grafcet que nous avons présentée en 4.4.1.

Le code Grafcet généré ici est aussi obtenu en fonction des caractéristiques du microcontrôleur choisi en entrée. L'approche adoptée est la génération du code par un visiteur. Il est implémenté pour chaque objet utile une méthode prenant en entrée un flux en écriture pour y ajouter le code nécessaire associé à cet objet. Le schéma général du fonctionnement de cette génération de code est donné en figure 2.23. Dans ce schéma, on fait ressortir le fait que l'obtention des équations algébriques se fait aussi

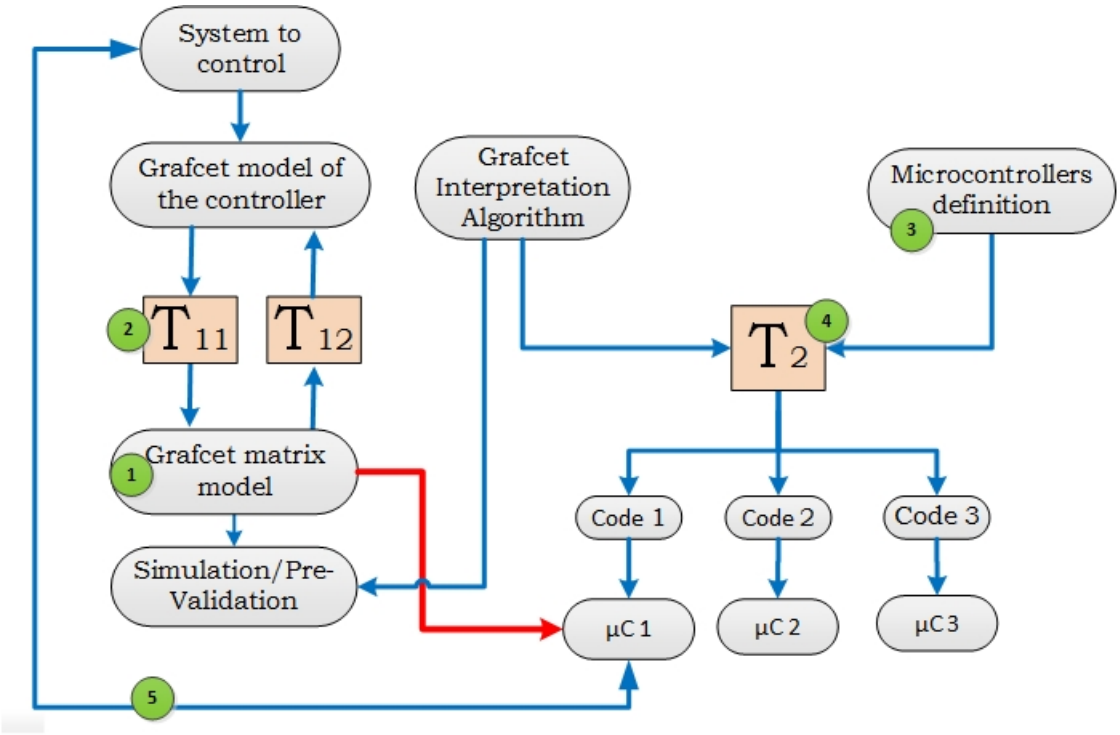


FIG. 2.22: Schéma général de la génération de code pour interpréteur

à travers les matrices de codage. Par exemple, elles sont particulièrement utiles pour retrouver les étapes en entrée et en sortie des transitions et les transitions en entrée et en sortie des étapes. La différence avec la génération du code de l'interpréteur (section 2.4.3.2) est que la structure du grafcet est noyée directement dans le code généré, à travers les équations algébriques. La transformation T_2 de ce schéma exploite les matrices de codage et la description d'un microcontrôleur pour générer du code de contrôle pour cette cible.

Pour le microcontrôleur $ATmega328P$, une présentation sommaire du code Grafcet intégrant les matrices de codage est donnée à l'annexe 5.7 pour le grafcet de l'exemple (figure 2.2).

2.4.5 Processus de synthèse

Tout le processus de synthèse offert par cette plateforme peut être représenté par la figure 2.24. Il présente quatre principales phases de mise sur pied d'une solution de contrôle décrites comme suit :

- **Étape 1. Spécifications Fonctionnelle du Système (SFS)** :
  La conception et le développement de tout système commencent par la description des exigences. Cette étape est la plus importante dans la conception de tout système de contrôle. Il fournit des détails sur la solution proposée à mettre en œuvre pour répondre aux besoins des

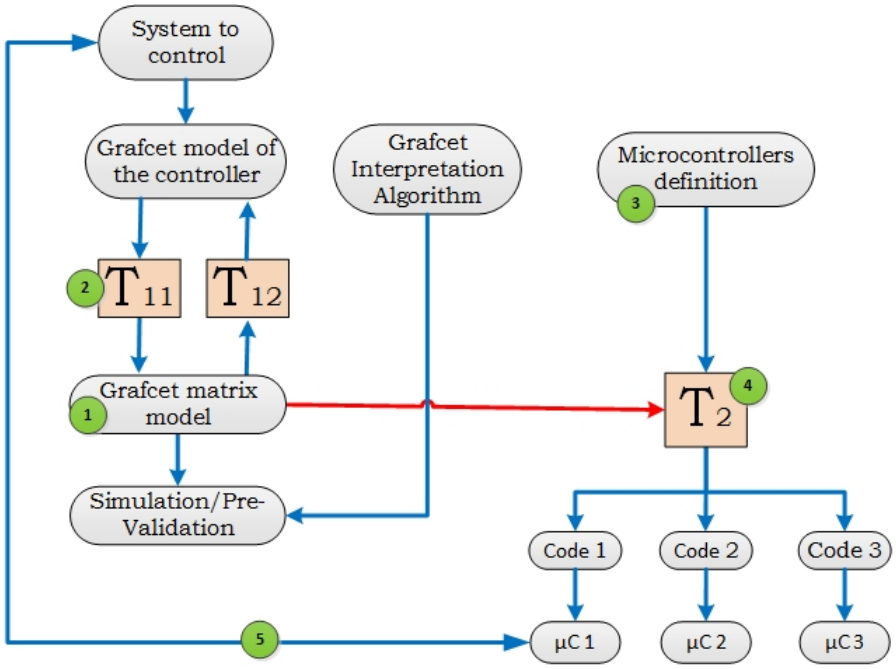


FIG. 2.23: Schéma de la génération de code par équations algébriques

utilisateurs. Généralement, il s'agit d'une description textuelle de que le système doit faire et quelles fonctions et facilités doivent être offertes par ce système pour satisfaire aux exigences. Après avoir décrit la spécification du fonctionnement de toutes les fonctions (par exemple, pompage, commutation, prise de mesure ...), des objets (capteurs, pompes, commutateurs ...) et des interactions séquentielles associées au système, le contrôleur est globalement spécifié avec un modèle d'*entres/sorties* présentant tous les signaux en entrée et toutes les actions en sortie.

- **Étape 2. Le modèle Grafcet** :
  L'étape suivante consiste à représenter techniquement les spécifications fonctionnelles du système par un modèle Grafcet. Nous avons proposé l'outil logiciel *UniSim* pour l'édition des modèles Grafcet. Une description de cet outil est donnée en début de cette section.
- **Étape 3. L'outil** $GrafcetConverter$:
  C'est la plateforme de synthèse que nous avons conçu pour l'extraction, le calcul des matrices du Grafcet et la génération de code pour une cible choisie.
- **Étape 4. Circuit intégré** :
  C'est l'étape du passage à la mise en œuvre effective sur la cible choisie à l'avance. Dans un premier temps, nous avons utilisé le microcontrôleur $dsPIC30F4013$ (qu'équipe la carte $EasyDsPIC4A$) pour lequel le code a été généré à travers la plateforme $GrafcetConverter$. Ce code en langage C qui implémente l'algorithme d'interprétation

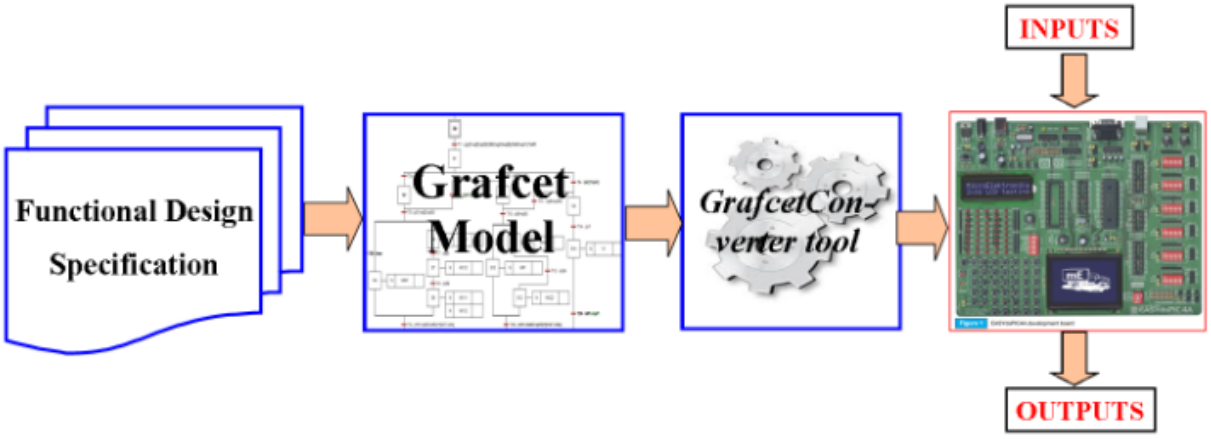


FIG. 2.24: Processus général de synthèse de la solution

du Grafcet est utilisé dans l'IDE MPLAB pour programmer la carte *EasyDsPIC4A*. Après avoir étudié l'extension à d'autres cibles, nous avons décrit le microcontrôleur $ATmega328P$ (qu'équipe la carte *Arduino UNO*) pour lequel une autre fonctionnalité de la plateforme de synthèse est réalisée pour générer du code.

2.5 Validation de l'approche par codage matriciel

En plus des simulations réalisables dans la plateforme de synthèse ($GrafcetConverter$) du Grafcet par matrices de codage, nous avons réalisé une application réelle partant de la spécification à la mise en œuvre sur microcontrôleur en passant par la génération du code. Cette application traite du contrôle des feux de signalisation à un carrefour situé à la rencontre de deux voix de circulation. Les feux concernent aussi bien les voitures que les piétons qui traversent les voies.

L'installation de tous les équipements nécessaires à un tel carrefour étant une tâche difficile et onéreuse, nous avons acquis une carte d'extension qui simule complètement le fonctionnement des feux de signalisation à un carrefour. Cette carte d'extension est alors commandée par un microcontrôleur exécutant le programme de contrôle généré par la plateforme $GrafcetConverter$.

Présentée sur la figure 2.25, cette carte TP comporte deux voies de circulation (voie 1 ou $V1$ et voie 2 ou $V2$), ainsi qu'un ensemble de diodes électroluminescentes (ou LEDs) dont 10 par voie (avec 5 LEDs de chaque côté). Il s'agit donc d'un système électronique ayant une entrée d'alimentation 5V et constitué de sorties (les LEDs) que l'on peut commander à l'aide d'un microcontrôleur.

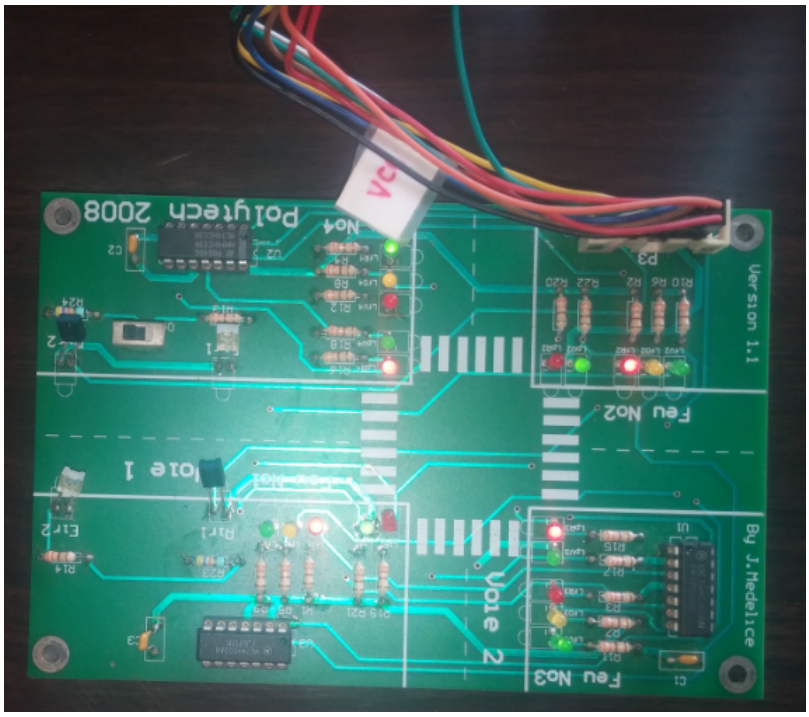


FIG. 2.25: Carte d'extension simulant les feux de carrefour

2.5.1 Description du fonctionnement des feux de carrefour

Dans un carrefour à deux voies, les véhicules circulent dans les deux sens sur chaque voie. De chaque côté il y a un passage piéton. La figure 2.26 donne une illustration de ce système. Les piétons ne peuvent circuler en même temps que les voitures. Sur une voie, lorsque les voitures sont à l'arrêt à cause du feu rouge, les piétons peuvent traverser cette voie d'un côté comme de l'autre. Sur la même voie, le scénario qui se produit d'un côté est identique à celui de l'autre côté ; par exemple, lorsque la voie est ouverte d'un côté pour la circulation des piétons, elle l'est aussi de l'autre côté. Le comportement des feux doit être par conséquent le même des deux côtés, i.e. le signal d'allumage des feux rouges est le même pour les deux côtés opposés sur la même voie de circulation: un même signal agira sur deux sorties à la fois. Au lieu d'avoir un même feu pour les voiture (rouge, orange et vert) et pour les piétons (rouge et vert), cette carte présente un feu pour les voitures à part et un autre pour les piétons à part.
La signification habituelle des feux ne change pas : le rouge interdit le passage, le vert ouvre le passage et l'orange marque un temps de transition du rouge au vert pour les voitures.

Sur la figure 2.26(b), on observe aussi la description des différents feux. On distingue les feux piéton ($LpR3$,$LpR4$ et $LpR1$,$LpR2$) et des feux de voitures ($LfR1$,$LfR2$, $LfO1$,$LfO2$, $LfV1$,$LfV2$, $LfR3$,$LfR4$, $LfO3$,$LfO4$ et $LfV3$,$LfV4$). Ces feux sont identifiés par leurs noms et décrits en paires

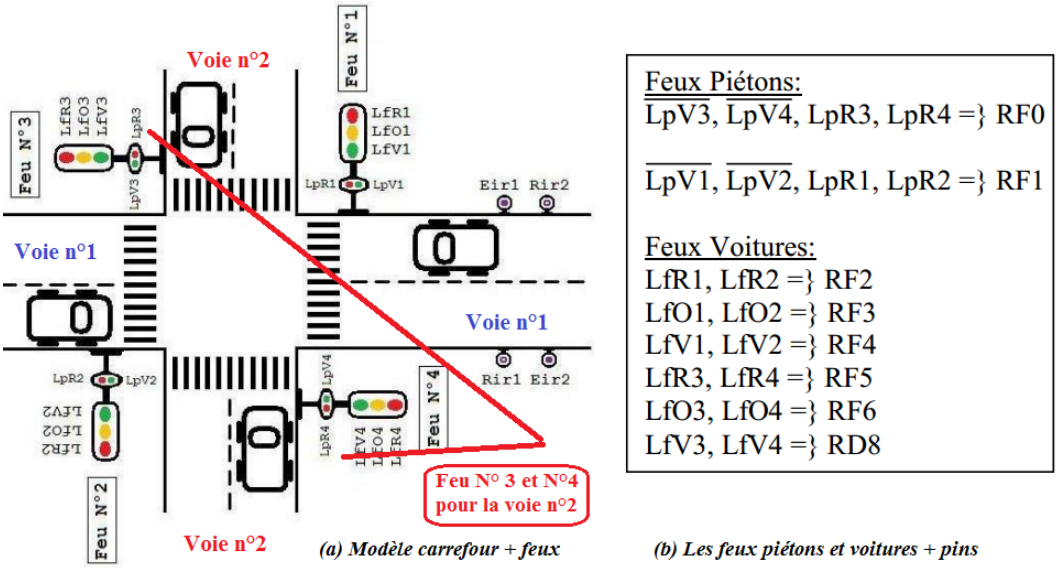


FIG. 2.26: Illustration des feux de carrefour (piétons et voitures)

à cause du fonctionnement similaire des deux côtés d'une même voie. Sur ce schéma, une un mapping entre les feux et les broches (ou pins) associées du microcontrôleur sont aussi décrites. Par exemple, la broche $RF2$ du microcontrôleur sera associée aux à la sortie correspondant aux feux $LfR1$,$LfR2$.

2.5.2 Matériel expérimental

Pour cette expérimentation, nous avons choisi les outils matériels suivants:

- **La carte EASYdsPIC4A** : Il s'agit d'une carte à microcontrôleur disposant de nombreuses entrées (des boutons) et de nombreuses sorties (LEDs). Elle accepte plusieurs microcontrôleurs différents tels que $dsPIC30F2010$, $dsPIC30F3011$, $dsPIC30F3013$ et $dsPIC30F3014$. Comme présenté en figure 2.27, c'est le microcontrôleur $dsPIC30F4013$ qui est enfichée sur l'entrée $DIP40B$. Un certain nombre de ports ($PORTB$, $PORTE$, $PORTF$, ...) sont disponibles pour brancher d'autres ports externes (ou d'extension). Ces ports externes conduisent vers des systèmes à contrôler.
- **La carte d'extension Feu** : Elle émule le fonctionnement d'un carrefour à deux voix. Cette carte a été réalisée par J. Medelice. Nous utilisons sa version 1.1 réalisée à l'aide de semi-conducteurs tels que $MC74HC03AN$ et $MM74HC03N$ pour réaliser ses entrées/sorties avec des portes logiques. Son modèle présenté en figure 2.26 est associé à l'énumération des feux utilisés.

  Quant aux outils logiciels, nous avons utilisé ceux décrits comme suit:
- l'outil *UniSim* pour le dessins et la sauvegarde des grafcets de spéci-

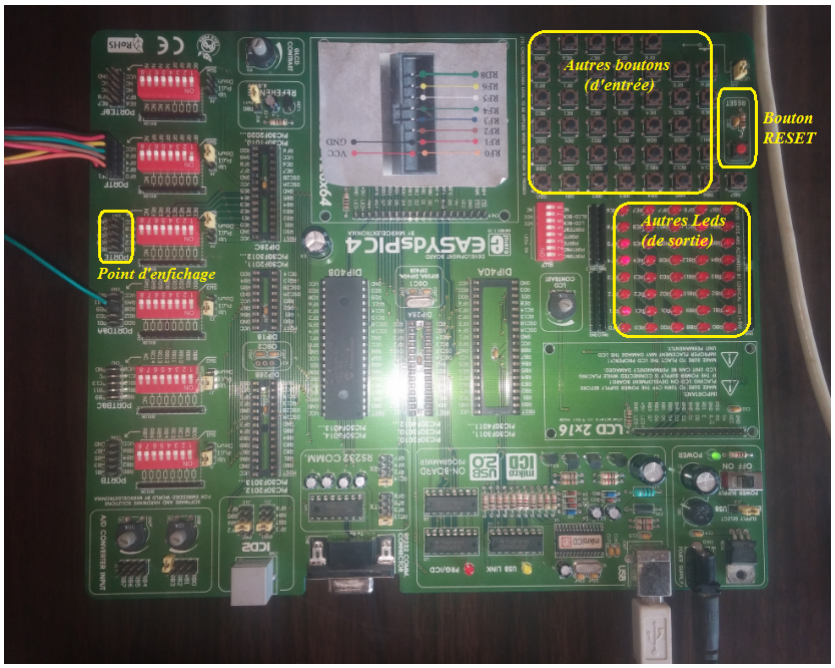


FIG. 2.27: Carte $EASYdsPIC4A$ équipée du μC $dsPIC30F4013$

fication.

- la plateforme de synthèse $GrafcetConverter$ pour la lecture des modèles Grafcet et la génération du code et des données utiles pour le contrôle du système.
- *l'IDE MPLAB* pour compiler et générer le code de contrôle (en hexadécimal) destiné à être chargé sur la carte.
- l'outil *SerialPort Terminal* pour le chargement du code hexadécimal sur la carte via un port série liant la carte microcontrôleur à un ordinateur.

2.5.3 Spécification Grafcet et réalisation

Cette carte d'extension possède des broches qui conduisent vers des sorties précises. Il faut donc brancher ses entrées sur la carte à l'aide de la fiche conçue à cet effet de sorte que les signaux envoyés soient acheminés soigneusement vers les LEDs.

Pour le microcontrôleur, les sorties du système sont décrites dans le tableau 2.2, avec la correspondance avec les sorties du microcontrôleur et les LEDs associées.

Chaque code utilisé pour les LEDs ou lampes (tableau 2.2) a une signification. L signifie lampe, p signifie piéton, f (feu) est relatif aux voitures, R c'est le rouge, V c'est le vert et O c'est l'orange.

Les feux de signalisation peuvent fonctionner sous deux modes : le mode continu et le mode discret.
Pour le mode discret, on peut faire avancer le système étape par étape en

TAB. 2.2: Sorties du système des feux de carrefour

Code Sortie Grafcet	Sorties Microcontrôleur	Leds sur Carte des Feux	Usager	Voie
O0	RF0	*{LpR3, LpR4}*	Piéton	V2
O1	RF1	*{LpR1, LpR2}*	Piéton	V1
O2	RF2	*{LfR1, LfR2}*	Voiture	V1
O3	RF3	*{LfO1, LfO2}*	Voiture	V1
O4	RF4	*{LfV1, LfV2}*	Voiture	V1
O5	RF5	*{LfR3, LfR4}*	Voiture	V2
O6	RF6	*{LfO3, LfO4}*	Voiture	V2
O7	RD8	*{LfV3, LfV4}*	Voiture	V2

appuyant sur des boutons. Pour cela, certaines entrées sont utilisées. Il s'agit précisément qu niveau du modèle des entrées a, b et c décrites comme suit: l'entrée a est associée à la broche $RB0$, l'entrée b à la broche $RB1$ et l'entrée c à la broche $RB2$. Pour une application nécessitant plus d'entrées, d'autres broches (de $RB3$ à $RB11$) sont disponibles sur le microcontrôleur.
Le mode continu est celui dans lequel le système évolue automatiquement sans attendre la survenue d'événements exeternes. Les sonditions d'évolution sont relatives à la mesure de la durée passée sur les étapes du modèle.

Pour chaque mode de fonctionnement, un grafcet a été réalisé, lesquels sont présentés en figure 2.28. A un moment donné, le système ne pourra exécuter qu'un seul de ces grafcets.
Tous expriment un cycle de fonctionnement au cours duquel les LEDs sont activées et/ou désactivées successivement.
Le premier grafcet (en figure 2.28(a)) correspond à celui du mode discret. Il faut appuyer de façon successive (b, c, a, b, c, a) sur les boutons (réceptivités respectives des transitions $T1$, $T2$, $T3$, $T4$, $T5$ et $T6$) pour faire un cycle complet. L'entrée a (resp. b et c) correspond au bouton de la carte $EASYdsPIC4A$ matérialisée par $RB0$ (resp. $RB1$ et $RB2$).
Le second grafcet (en figure 2.28(b)) correspond à celui du mode continue. Ici, le système évolue de façon continue. C'est pourquoi les conditions associées aux transitions sont des contraintes temporelles représentant des temporisations ou délais. Par exemple, la condition pour faire passer de l'étape $S0$ à l'étape $S1$ est $[S0.T > 5s]$, ce qui signifie que l'étape $S0$ doit avoir été active pendant une durée au moins égale à $5s$ pour que cette condition soit vraie. Auquel cas, la transition $T1$ est franchie, entrainant la désactivation de l'étape $S0$ (avec les actions à niveau associées $O1$, $O4$ et $O5$) et l'activation de l'étape $S1$ (avec les actions à niveau $O1$, $O3$ et $O5$).

Le contrôle des feux de signalisation
Lorsque le code généré pour un grafcet est compilé et chargé sur le microcontrôleur, ce dernier s'exécute selon la spécification donnée par le grafcet. Chacun des grafcets de la figure 2.28 contient six étapes auxquelles sont associées des actions. En considérant le grafcet (b) du mode continu, voici

la description des étapes avec les conditions d'évolution associées:

- **Pas 1 (étape** $S0$**)** : Initialement, l'étape $S0$ est activée. Sa durée d'activité sera de $5s$, car la transition $T1$ a pour réceptivité $[S0.T > 5s]$. Pendant son activité, les sorties $O1$, $O4$ et $O5$ sont activées, ce qui correspond respectivement aux feux (LEDs) *LpR3, LpR4*, *LfV1, LfV2* et *LfR3, LfR4*. À partir du tableau 2.2, on déduit que sur la voie $V1$, les feux piéton sont au rouge tandis que ceux des voitures sont au vert; alors que pour la voie $V2$ ils sont au rouge. Après $5s$ passées dans $S0$, on passe à $S1$ à cause de la condition $[S0.T > 5s]$ associée à la transition $T1$.
- **Pas 2 (étape** $S1$**)** : Une fois l'étape $S1$ active, les sorties $O1$, $O3$ et $O5$ sont activées, ce qui signifie que $O1$ et $O5$ sont restées actives alors que $O4$ est désactivée avec l'activation de $O3$. Il s'agit de l'allumage des feux orange de la voie $V1$, signalant la transition vers l'ouverture de la voie $V2$ pour les véhicules. Comme la condition de la transition $T2$ est $[S1.T > 1s]$, l'activité de $S1$ ne durera pas plus de $1s$, et on passe au **Pas 3**.
- **Pas 3 (étape** $S2$**)** : L'étape $S1$ est désactivée et celle $S2$ est activée. Les actions $O1$ et $O5$ restent activées alors que l'action $O3$ est désactivée au profit des actions $O0$ et $O2$ qui sont activées. Du feu orange sur la voie $V1$, on passe alors au feu rouge pour voitures. C'est aussi le feu rouge ($O5$) qui est allumé pour les voitures de la voie $V2$. Cette situation ne dure pas, car elle est transitoire. Après $1s$ passée à l'étape $S2$, l'automatisme passe au **Pas 4**.
- **Pas 4 (étape** $S3$**)** : À l'activation de l'étape $S3$, les sorties $O0$ et $O2$ restent actives alors que $O1$ et $O5$ sont désactivée. De la situation transitoire où les feux de voiture sur les deux voies sont au rouge, les piétons peuvent circuler sur la voie $V2$ sans que les voitures puissent circuler sur la voie $V1$. La circulation des voitures est alors ouverte sur la voie $V2$ avec l'activation de $O7$. De même que les voiture ont circulé sur la voie $V1$ (Pas 1) pendant $5s$, elles vont aussi circuler sur la voie $V2$ pendant $5s$ (La condition de $T4$ est $[S3.T > 5s]$), après quoi on passe au **Pas 5**.
- **Pas 5 (étape** $S4$**)** : Ici, on active l'étape $S4$. L'action $O0$ reste active, i.e. les piétons continuent à passer sur la voie $V2$, alors que le feu voitures passe à l'orange. Sur la voie $V1$, $O2$ reste active, i.e. les voitures ne circulent pas encore. C'est une situation transitoire de la voie $V2$ à la voie $V1$ pour la circulation des voitures. Après $1s$ passé sur cette étape, on passe au **Pas 6**.
- **Pas 6 (étape** $S5$**)** : L'étape $S5$ devient active et l'action $O6$ se désactive, alors que les autres ($O0$ et $O2$) restent actives. Cette étape est transitoire et identique au **Pas 3** où les voitures sont interdites de circuler sur les 2 voies alors que les piétons y continuent à circuler. Après $1s$ de situation transitoire, l'automatisme reprend son fonctionnement

au **Pas 1**.

Au vu de ce qui précède, les étapes sont équivalentes deux à deux : $S0$ et $S3$ pour la circulation des voiture sur l'une et sur l'autre des deux voies, $S1$ et $S4$ pour un état transitoire d'une voie à l'autre et relative à la circulation des voitures, et $S2$ et $S5$ (identiques) qui représentent le dernier moment où il faut fermer la circulation des voitures sur une voie et l'ouvrir sur une autre. Cette dernière situation est importante pour éviter qu'il y ait collision de voitures.

Mise en route du système

Initialement, on branche la carte à microcontrôleur $EASYdsPIC4A$ au secteur, puis on enfiche la carte d'extension des feux de signalisation sur le port $PORTF$ de la carte à microcontrôleur. Une fois le programme de contrôle compilé à travers l'IDE $MPLAB$ et chargé sur le microcontrôleur à l'aide du logiciel *SerialPort Terminal* au travers du port $RS232$, le microcontrôleur est réinitialisé au travers de son bouton $RESET$ et l'exécution du programme commence. Sur la figure 2.29, on peut observer l'exécution du microcontrôleur avec la carte d'extension Feu qui réagit aux commandes du premier en changeant d'état au fur et à mesure.

Le contrôle des feux de signalisation simulés par la carte d'extension à partir du microcontrôleur $EASYdsPIC4A$ peut aussi être réalisé à l'aide de la carte *Arduino UNO* équipée du microcontrôleur $ATmega328P$.

2.6 Profilage des codes générés

L'objectif est de caractériser l'exécution des codes générés sur les plateformes microcontrôleurs. Il s'agit particulièrement de calculer la durée du cycle de scrutation ou bien sa fréquence, qui est le nombre de cycles exécutés par seconde. Nous nous intéressons aussi à la taille du programme généré et son pourcentage par rapport à la mémoire disponible sur le microcontrôleur. Le profilage est la mesure des performances d'un logiciel ou d'une application[47].

2.6.1 Méthode de profilage

Il existe de nombreuses méthodes pour profiler un code, lesquelles peuvent être logicielles ou matérielles. La technique que nous utilisons ici est *l'insertion d'un code d'instrumentation dans le code de l'application* [47], qui est une méthode logicielle. Cette méthode favorise l'enregistrement du temps d'exécution pour différentes fonctions en utilisant des compteurs de logiciels. Il est considéré comme le meilleur dès lors que le code instrumenté n'est pas exécuté par un simulateur, mais par une cible embarquée.

Dans notre cas, pour évaluer la durée du cycle de scrutation, nous procédons comme suit:

le programme de contrôle est exécuté pendant 5 secondes (5000 ms) au cours

desquelles nous comptons le nombre de cycles à l'aide d'une variable compteur N. N est incrémenté à la fin de chaque cycle de scrutation. Aussitôt les 5 secondes écoulées, nous affichons le résultat. Cette opération est reprise vingt (20) fois en tout pour chaque programme Grafcet. On peut alors calculer la moyenne $\overline{N}$ est calculée. L'opération suivante est alors réalisée pour estimer la durée moyenne d'un cycle de scrutation (équation 2.20):

$$ST = \frac{5}{\overline{N}}(seconds) = \frac{5}{\overline{N}} \times 1000(milliseconds) \tag{2.20}$$

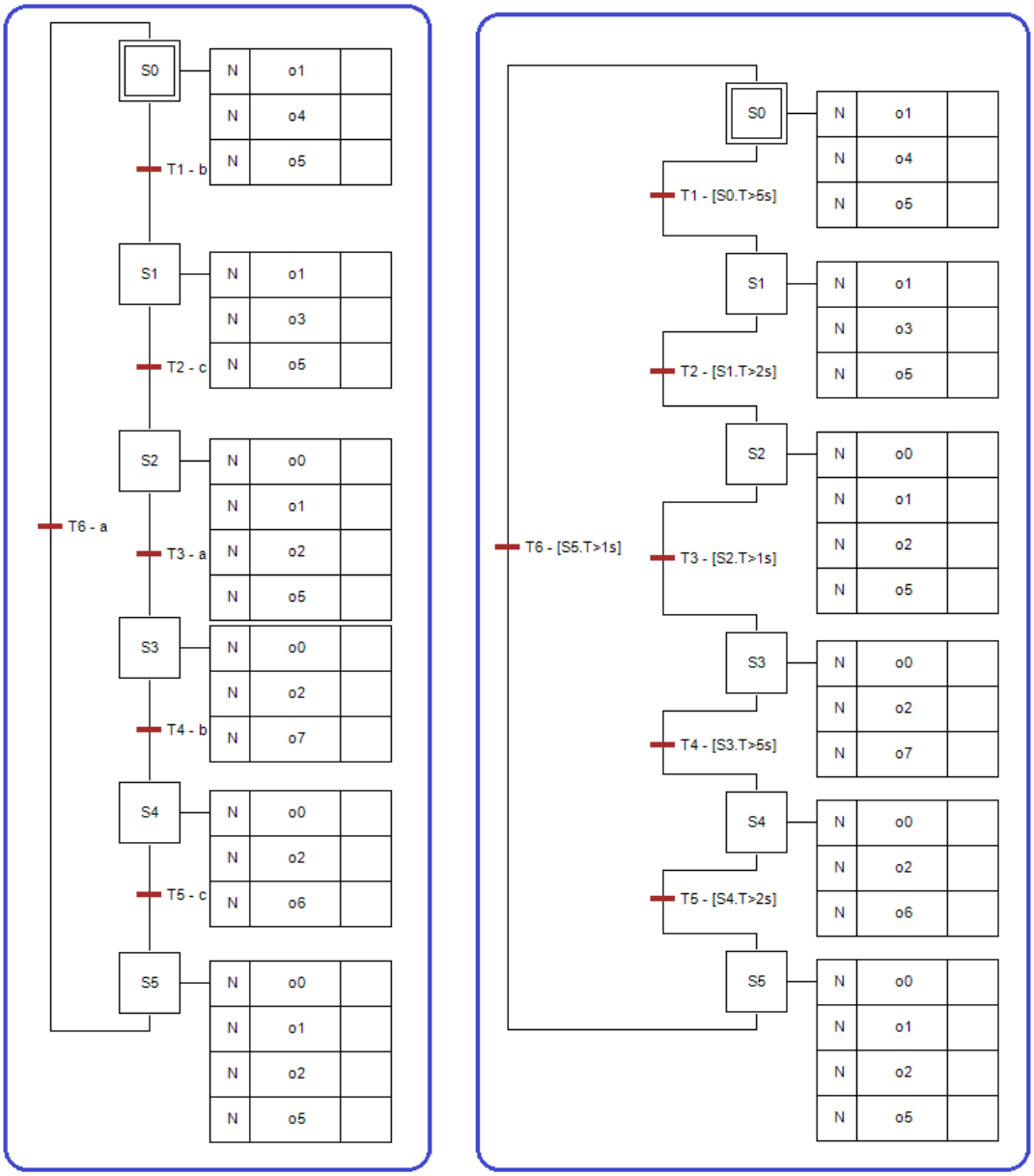
Pour cela, le microcontrôleur ATmega328P est utilisé, car son environnement de développement (IDE Arduino) offre une interface série pour la communication avec les cartes Arduino à travers un port COM. Le code Arduino instrumenté a la forme de celui du listing 2.2 :

Listing 2.2: Forme du code Arduino instrumenté

```
...
//Fonction d'initialisation du programme
void setup(){
        ...
        Serial.begin(9600); //pour afficher la valuer de N
        time_val = millis(); //récup. de l'instant initial
        N = 0;
}
//Fonction exécutée en boucle par le programme
void loop(){
        ...
        N++;
        if(millis() - time_val > 5000){//calcul du temps écoulé
                //et comparaison à 5000 ms
                Serial.println(N); //Affiche le nbre de cycles
                N = 0; //Réinitialisation du compteur
                //pour un prochain décompte
                time_val = millis(); //récup. de l'instant courant
        }
}
```

2.6.2 Caractéristiques structurelles des grafcet choisis et mesures d'exécution

Ne disposant pas d'un benchmark de modèles Grafcet, nous avons choisi au total dix (10) grafcets avec des caractéristiques différentes pour lesquels nous avons généré du code instrumentalisé et soumis à l'exécution sur la carte Arduino UNO.



(a) Grafcet 1 (avec entrées a, b et c) **(a) Grafcet 2 (sans entrées : évolue avec le temps)**

FIG. 2.28: Grafcets de spécification

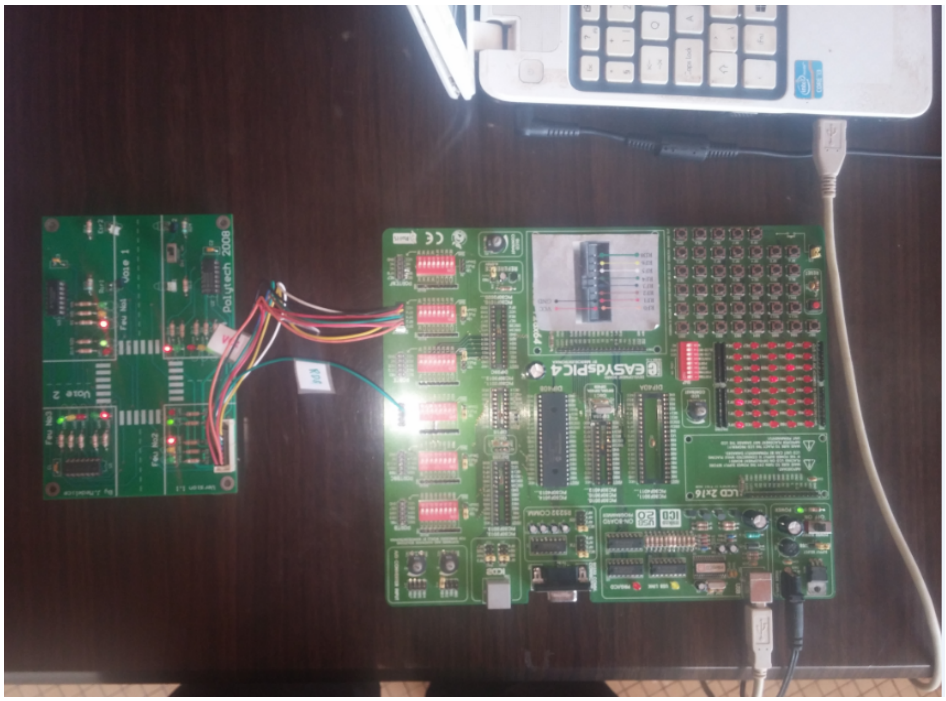


FIG. 2.29: Exécution du système des feux de signalisation

TAB. 2.3: Données sur les grafcets pour le profilage du code avec matrices de codage

	g7-1	**g7-2**	**g7-3**	**g7-4**	**g7-5**	**g7-6**	**g7-7**	**g7-8**	**g7-9**	**g17-0**
NbS	2	5	6	8	11	11	10	15	14	16
NbT	2	5	6	8	11	11	11	16	16	16
NbIn	2	3	0	3	5	4	3	6	12	11
NbOut	1	3	7	10	7	8	8	8	10	11
NbTC	2	0	3	3	3	4	7	8	2	4
SizeP (bytes)	5783	5832	5790	5858	5892	6170	5876	5920	6348	6350
%Prog	17,93	18,08	17,95	18,16	18,27	19,13	18,22	18,35	19,68	19,69
SizeGV (bytes)	1140	1188	1146	1214	1248	1316	1232	1276	1494	1496
%GB	55,66	58,01	55,96	59,28	60,94	64,26	60,16	62,30	72,95	73,05

N°	**g7-1**	**g7-2**	**g7-3**	**g7-4**	**g7-5**	**g7-6**	**g7-7**	**g7-8**	**g7-9**	**g17-0**
Exec. 1	2432	2211	2313	2169	2076	2076	2474	2160	1601	2133
Exec. 2	2431	2209	2311	2168	2076	2075	2473	2158	1601	2199
Exec. 3	2432	2209	2317	2168	2079	2076	2473	2158	1600	2194
Exec. 4	2431	2209	2317	2168	2080	2075	2473	2159	1601	2187
Exec. 5	2431	2208	2317	2168	2073	2076	2472	2158	1601	2152
Exec. 6	2431	2209	2317	2168	2073	2076	2473	2158	1601	2198
Exec. 7	2431	2209	2317	2168	2073	2076	2472	2158	1859	2198
Exec. 8	2432	2209	2317	2168	2073	2023	2473	2158	2023	2199
Exec. 9	2432	2209	2317	2168	2073	2076	2473	2159	2023	2095
Exec. 10	2431	2209	2317	2168	2073	2075	2473	2158	2023	2209
Moy $\overline{N}$	2432	2209	2316	2168	2076	2073	2473	2158	1955	2186
ST (ms)	2,0563	2,2631	2,1586	2,3060	2,4087	2,4118	2,0220	2,3166	2,5573	2,2876

TAB. 2.4: Données sur les grafcets pour le profilage (avec équations algébriques de codage)

	g7-1	**g7-2**	**g7-3**	**g7-4**	**g7-5**	**g7-6**	**g7-7**	**g7-8**	**g7-9**	**g17-0**
NbS	2	5	6	8	11	11	10	15	14	16
NbT	2	5	6	8	11	11	11	16	16	16
NbIn	2	3	0	3	5	4	3	6	12	11
NbOut	1	3	7	10	7	8	8	8	10	11
NbTC	2	0	3	3	3	4	7	8	2	4
SizeP (bytes)	3498	3300	3932	4474	4744	4816	4702	5392	5636	5828
%Prog	10,84	10,23	12,19	13,87	14,71	14,93	14,58	16,72	17,47	18,07
SizeGV (bytes)	228	240	266	289	307	307	302	340	346	355
%GB	11,13	11,72	12,99	14,11	14,99	14,99	14,75	16,60	16,89	17,33

	g7-1	**g7-2**	**g7-3**	**g7-4**	**g7-5**	**g7-6**	**g7-7**	**g7-8**	**g7-9**	**g17-0**
Exec. 1	17290	12427	13331	11705	12292	12065	10105	7665	5485	4755
Exec. 2	17276	12422	13326	11700	12287	12060	10101	7659	5483	4753
Exec. 3	17283	12417	13326	11700	12287	12060	10097	7659	5483	4753
Exec. 4	17283	12422	13326	11700	12287	12065	10097	7659	5483	4753
Exec. 5	17276	12417	13321	11700	12287	12060	10093	7656	5483	4751
Exec. 6	17283	12422	13321	11700	12287	12065	10097	7659	5483	4753
Exec. 7	17283	12422	13321	11700	12287	12065	10101	7659	5483	4753
Exec. 8	17276	12422	13321	11700	12287	12065	10101	7659	5483	4753
Exec. 9	17276	12417	13321	11695	12282	12060	10093	7656	5483	4751
Exec. 10	17290	12422	13321	11700	12292	12065	10097	7659	5483	4753
Moy N	17282	12421	13324	11700	12288	12063	10098	7659	5483	4753
ST (ms)	0,2893	0,4025	0,3753	0,4274	0,4069	0,4145	0,4951	0,6528	0,9119	1,0520

Le microcontrôleur ATmega328P qui équipe la carte Arduino UNO possède 32Mo (32256 octets) de mémoire programme et 2 Mo (2048 octets) de mémoire RAM. Sa description est aussi donnée dans le tableau 4.2.

Les caractéristiques de ces grafcets avec le code généré sont données dans le tableau 2.3. En colonne, il y a les modèles Grafcet alors qu'en ligne, il y a les caractéristiques de ces modèles : le nombre d'étapes (NbS), le nombre de transitions (NbT), le nombre d'entrées($NbIn$), le nombre de sorties($NbOut$) et le nombre de contraintes temporelles ($NbTC$).

La deuxième partie de ce tableau donne la taille totale ($SizeP$) après compilation du programme généré par l'approche des matrices de codage du Grafcet, ainsi que son pourcentage par rapport à la mémoire programme totale($\%Prog$) qui est de $32Mo$. Cette taille est d'au maximum 20% de la mémoire programme. Ce tableau donne aussi la taille totale des variables globales ($SizeGV$) utilisées, ainsi que leur pourcentage ($\%GV$) par rapport à la taille totale de la mémoire RAM. En général, ces variables occupent moins de 75% de la RAM.

Après exécution de ces codes sur la plateforme microcontrôleur choisie, nous avons obtenu les mesures données dans la troisième partie du tableau 2.3. La durée d'exécution moyenne du cycle de scrutation (ST) est donc de l'ordre de 2,3 millisecondes pour les codes générés par l'approche de codage matriciel du Grafcet.

2.6.3 Comparaison avec la génération de codes par équations algébriques du Grafcet

En générant du code pour les mêmes grafcets avec la méthode par équations algébriques de codage (décrites en section 4.4.1), les données sur le profilage de l'exécution de ces programmes sont présentées dans le tableau 2.4.

La comparaison des tailles des programmes générés pour ces deux méthodes est donnée en figure 2.30(a), alors qu'en figure 2.30(b) il s'agit des durées moyennes de cycles de scrutation. On observe que la taille des programmes des interpréteurs varie peu, du fait de la faible taille des données sur les matrices de codage du Grafcet, comparativement à la taille de l'interpréteur lui-même. En plus, la taille du code de l'interpréteur est supérieur dans chaque cas à celle du code non interprété (par équations algébriques), ce qui est normal car l'interpréteur est générique et contient généralement des fonctionnalités supplémentaires au regard d'une application spécifique.

De la figure 2.30(b), il ressort que la durée d'exécution moyenne de chaque cycle de codage n'excède pas 1,1 ms pour les codes générés et contenant des équations algébriques du Grafcet, ce qui est de 2 à 4 fois moins inférieur que ceux générés pour l'interpréteur.

Il en ressort donc qu'en général, l'une ou l'autre approche de génération de code en passant par les matrices de codage produit une durée du cycle de scrutation inférieure à $10ms$. Selon [47], ces approches peuvent être

utilisées pour réaliser la synthèse des SCCs ayant des contraintes temps réel dures. Ainsi, quand bien même d'autres actions telles que la vérification des entrées du contrôleur et l'intégrité du code de contrôle seraient ajoutées au cycle de scrutation, il serait difficile que la durée de scrutation (ST) excède $100ms$. En effet selon [47], la durée du cycle de scrutation pour des applications temps réel dur varie de l'ordre de quelques millisecondes à quelques dizaines de millisecondes, tandis que pour les application temps réels mou, il est de l'ordre de quelques centaines de millisecondes. Nous pouvons donc dire qu'à l'aide de ces approches, il est possible à de réaliser des applications de contrôle commande temps réel dur basées sur des microcontrôleurs bas coût.

2.7 Conclusion

Dans ce chapitre, nous avons présenté la synthèse du Grafcet par matrices de codage. Pour cela, nous avons proposé une représentation du Grafcet à l'aide de cinq matrices de codage ($INIT$, E, S, MA et RS) associées à un algorithme d'exécution séquentielle, lesquels respectent les règles d'évolution du Grafcet. Ce modèle garantit le critère d'indépendance vis-à-vis des plateformes cibles. Cette solution permet de générer automatiquement des interpréteurs de modèles Grafcet à implanter sur les cartes à microcontrôleur. Le Grafcet devenant une simple donnée, il est facile de modifier le comportement du contrôleur sans toutefois modifier son code exécuté. Une plateforme de synthèse du Grafcet a aussi été développée. Elle offre des fonctionnalités d'édition de modèles Grafcet (*UniSim*), de production automatique des matrices de codage, de simulation et de génération du code pour certaines cibles microcontrôleurs. En plus de la génération du code de l'interpréteur directement calqué de l'algorithme d'interprétation du Grafcet, il est aussi possible de générer du code au travers des équations algébriques du Grafcet (décrites en section 4.4.1) qui en constitue aussi une représentation formelle. Ces deux méthodes de génération de code sont implémentées chacune à l'aide d'une fonction particulière écrite par cible.

Outre la simulation dans la plateforme de synthèse, la validation de l'approche de synthèse par codage matriciel du Grafcet a été réalisée par l'utilisation d'une carte d'extension qui simule parfaitement les feux de signalisation à un carrefour. Deux modes d'exécution ont été proposés: le mode discret (avec entrées/sorties) et le mode continu (avec sorties et contraintes temporelles) pour chacun desquels un grafcet de spécification est réalisé. A partir de ce grafcet, du code est généré, compilé et exécuté sur la plateforme $EASYdsPIC4A$[18] qui est connectée à la carte d'extension des feux. Lors de l'exécution, le microcontrôleur envoie des ordres appropriés pour allumer certains feux et éteindre d'autres.

Le profilage des codes générés pour interprétation a été réalisé à l'aide

18. équipée du microcontrôleur $dsPIC30F4013$

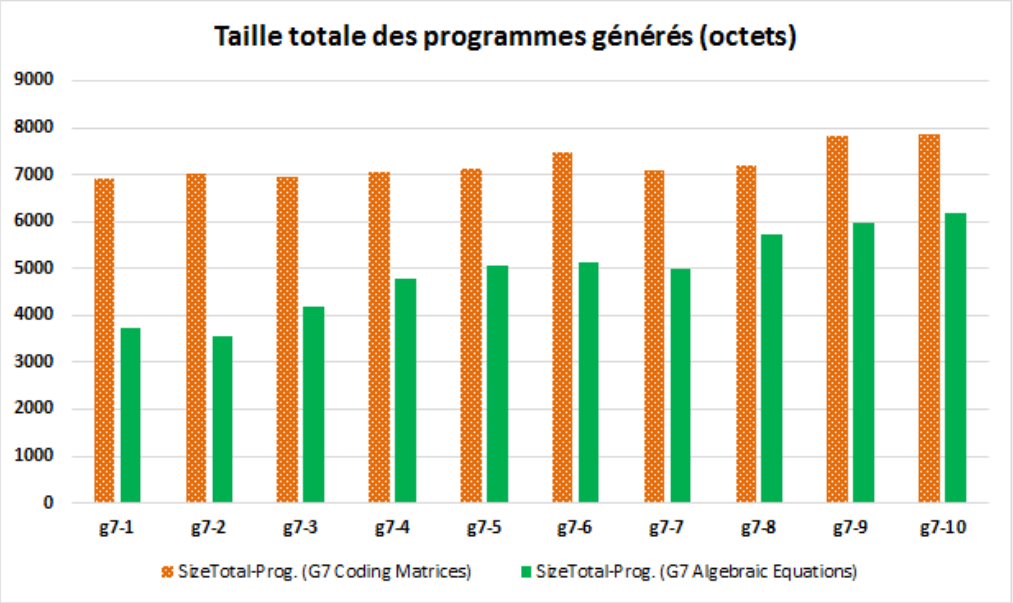
de la carte *Arduino UNO*[19]. Pour plusieurs grafcets exécutés, nous avons mesuré la durée du cycle de scrutation des programmes de contrôle. Le résultat montre que l'algorithme exécuté par l'interpréteur Grafcet est compact et aussi rapide qu'un code spécifique généré par équations algébriques Grafcet.

Malgré les avantages de la synthèse du Grafcet par matrices de codage, certaines difficultés rencontrées lors de la mise en œuvre de cette approche ont prouvé ses limites. Elles concernent principalement la prise en compte des autres aspects du Grafcet et la facilité de caractériser des cibles microcontrôleurs pour la synthèse multi-cibles. Ces limites sont tributaires à l'environnement généraliste de programmation utilisé. Pour les éléments du Grafcet non encore représentés, la prise en compte des actions à niveau conditionnelles et des actions stockées complexifierait beaucoup l'algorithme d'interprétation proposé. Il en est de même des actions à niveau conditionnelles et des actions stockées. .

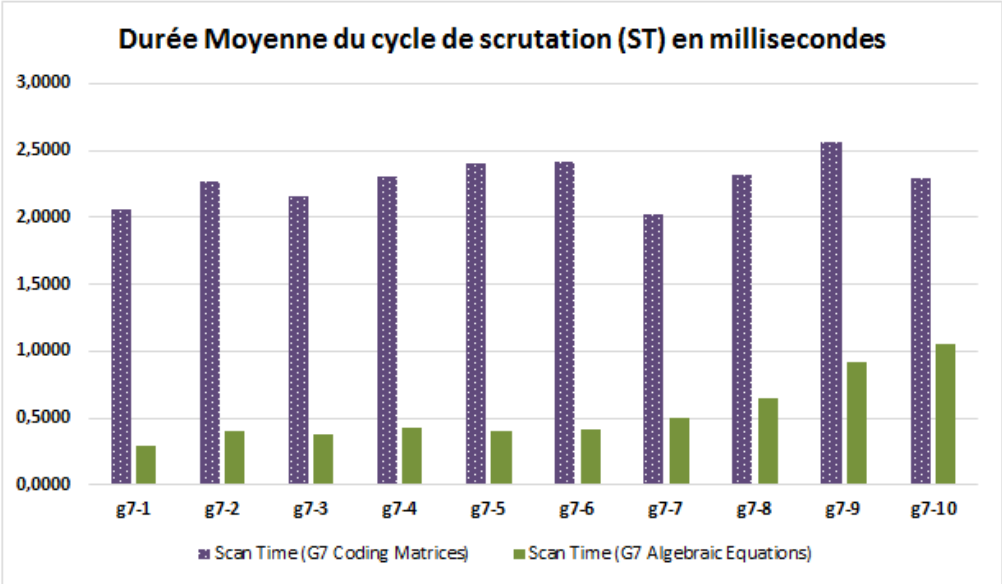
Outre ces limites, il y a le fait que nous avons utilisé un environnement libre (*UniSim*) d'édition des modèles Grafcet. L'extension de cet éditeur pour prendre en compte d'autres aspects du langage non encore prévus (tel que les événements de front montant/descendant et certaines conditions temporelles) est une tâche houleuse et délicate, car n'a pas été prévue lors de la conception d'$UniSim$.

Nous avons donc suivi une philosophie IDM pour mettre en œuvre l'approche par codage matriciel dans un environnement de développement généraliste, à travers le développement d'un modèle indépendant des plateformes cibles (PIM) et la synthèse du code pour des cibles spécifiques partant de ce modèle intermédiaire, ce qui correspond à un développement logiciel en Y (promu par le MDA). Une mise en œuvre dans un environnement dédié à l'IDM s'avère très prometteuse au regard des limites de cette approche.

19. équipée d'un microcontrôleur $ATmega328P$



(a) Taille des programmes générés



(b) ST pour chaque programme grafcet

FIG. 2.30: Taille du code hexadécimal et *ST* des deux méthodes de génération de code

Chapitre Trois

Métamodelisation Grafcet et prise en compte des expressions

Sommaire

3.1 Introduction

Les difficultés rencontrées dans la mise en œuvre de la synthèse multicibles du Grafcet par matrices de codage sont principalement tributaires au type généraliste de plateforme utilisée. En effet, dans une telle plateforme, il est difficile de formaliser la notion de modèle et de transformation de modèles comme prévue par les standards, de façon à rendre souple le processus d'ingénierie. Comme le Grafcet est un langage dédié à un domaine précis[1] et essentiellement consacré à la modélisation, il convient d'utiliser des outils de l'Ingénierie Dirigée par les Modèles(IDM) [55, 31] pour proposer un langage de domaine spécifique ou dédié ($DSML$) au développement des SCCs par Grafcet. Pour cela, il convient aussi d'utiliser une plateforme IDM pour tirer profit de nombreuses fonctionnalités offertes pour la définition et la transformation de modèles. L'usage d'une telle plateforme vise le développement d'une solution logicielle permettant de construire la spécification Grafcet du contrôleur d'un système, de procéder à sa vérification, puis de transformer automatiquement la spécification en du code dédié à une cible microcontrôleur choisie en entrée. Le développement de ce $DSML$ passe alors par la métamodélisation du Grafcet, avec une emphase sur les expressions manipulées au sein du langage.

3.2 L'ingénierie dirigée par les modèles

L'ingénierie dirigée par les modèles (IDM)[2] est le domaine de l'ingénierie logicielle qui utilise des modèles et des transformations de modèles pour produire des artefacts[3] logiciels tels que du code, de la documentation

1. développement des SCCs
2. ou *Model Driven Engineering (MDE)* en anglais
3. *Artefact:* terme général désignant toute sorte d'information créée, produite, modifiée ou utilisée par les travailleurs dans la mise au point du système

et même des plans de test. L'adoption de l'IDM permet d'industrialiser la production de logiciels grâce à de nombreuses technologies émergentes. Le principe de base de l'IDM est le suivant: «Tout est modèle» [6]. C'est l'aboutissement d'une évolution importante du principe de la technologie Object, «Tout est un objet», qui a révolutionné la construction de logiciels, mais qui a atteint ses limites en raison des systèmes de plus en plus complexes et des évolutions technologiques très rapides.

3.2.1 Concepts de modèles, métamodèles et métamodélisation

L'ingénierie dirigée par les modèles est une généralisation de l'architecture MDA (Model Driven Architecture) proposée par l'OMG en 2001 [29]. L'IDM n'est pas destiné à utiliser les modèles uniquement comme une simple documentation mais comme des entrées/sorties formelles pour des outils informatiques mettant en œuvre des opérations précises. Dans cette optique, **la métamodélisation** joue un rôle clé. C'est une technique courante pour la construction d'une collection de «concepts» dans un certain domaine. Cela permet de définir la syntaxe abstraite des modèles et les relations entre les éléments du modèle. **Un modèle** est une représentation d'un système sous étude. Les principes de l'IDM stipulent qu'une vue particulière d'un système peut être capturée par un modèle et que chaque modèle est écrit dans le langage de son métamodèle. En outre, «un métamodèle est un modèle de modèles» [29]. Un métamodèle est donc un modèle qui définit la structure d'un langage de modélisation [17]. Ainsi, tout modèle de ce langage de modélisation devrait satisfaire la structure définie au niveau de son métamodèle.

Il en découle deux relations entre les modèles: *representedBy* (représenté par) et *conformsTo* (conforme à). Un modèle est dit conforme à son métamodèle au même titre qu'un programme est conforme à la grammaire du langage de programmation dans lequel il est écrit [6]. À cet égard, l'OMG a introduit l'architecture à quatre niveaux illustrée à la figure 3.1 [23].

La couche $M0$ est le vrai système. Au niveau $M1$, on décrit un modèle représentant ce système. Ce modèle est conforme à son métamodèle qui est défini au niveau $M2$ et le métamodèle lui-même est conforme au métamétamodèle qui se trouve au niveau $M3$, tandis que le métamétamodèle est conforme à lui-même. L'OMG a aussi proposé le format MOF[4] comme norme pour la spécification des métamodèles. MOF est une couche unique de méta-métamodèle, car elle est instanciée à partir de son propre modèle; c'est pourquoi MOF est défini en MOF [17]. De cette façon, même le métamodèle UML est défini en termes de MOF. Un standard supporté par MOF est XMI, qui définit un format de sérialisation et d'échange basé sur XML pour les modèles au niveau des couches $M3$, $M2$ et $M1$. Dans le

4. Meta Object Facility

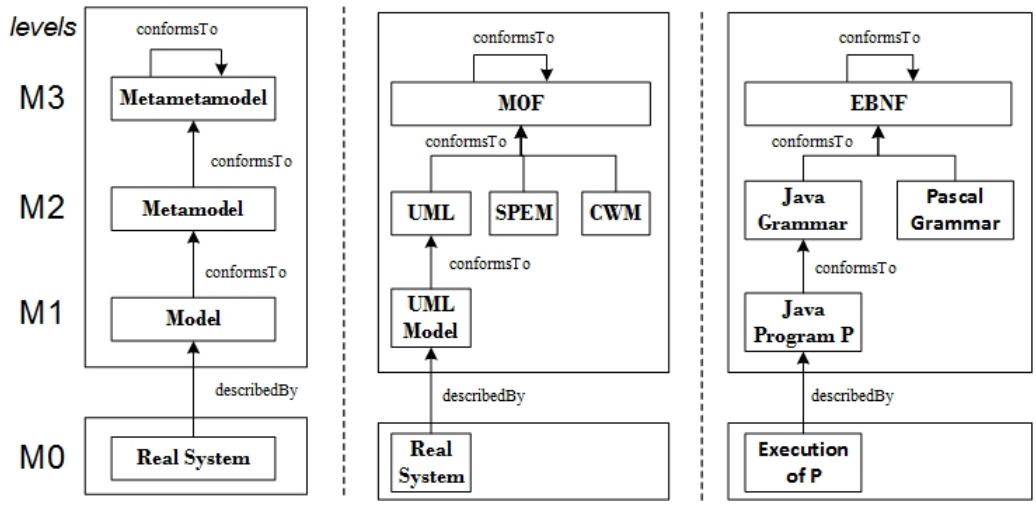


FIG. 3.1: Les quatre niveaux de l'architecture de métamodélisation

standard EMF, *Ecore* est le langage fourni pour spécifier les métamodèles [6]. MOF est comparable à la forme de Backus-Naur étendue (EBNF[5]) qui est le métalangage décrivant les grammaires des langages de programmation (C, $Java$, $Pascal$, etc.). De nos jours, plusieurs métamodèles sont définis et utilisés, chacun correspondant à un besoin, un objectif ou une situation spécifique.

3.2.2 L'approche MDA : PIM, PDM et PSM

Dans l'approche MDA traditionnelle (illustrée sur la figure 3.2), l'objectif était de pouvoir générer des modèles spécifiques à une plateforme (PSM) à partir de modèles indépendants de cette plateforme (PIM) [35]. Afin de réaliser cette tâche automatiquement, des modèles précis des plates-formes cibles ont été définis tels que le Web, $CORBA$, $DotNet$, EJB, etc. Les modèles de description de plate-forme (PDM) semblent actuellement être le chaînon manquant de l'approche MDA. Son extension à travers l'IDM en apporte une solution générale. Les notions de modèle spécifique à la plate-forme (PSM) et de modèle indépendant de la plate-forme (PIM) seront ensuite utilisées.

3.2.3 Langages de modélisation, syntaxes abstraites et concrètes

Un **langage de modélisation** est un ensemble de tous les modèles possibles conformes à sa syntaxe abstraite. Il est représenté par une ou plusieurs syntaxes concrètes et satisfaisant une sémantique donnée [17, 35]. Le processus de définition d'un langage de modélisation commence géné-

5. Extended Backus-Naur Form

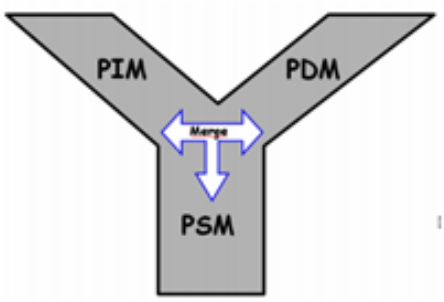


FIG. 3.2: Etapes de l'approche MDA

ralement par la capture et l'identification des concepts, des abstractions et des relations sous-jacentes au domaine d'application. Il correspond à la phase d'analyse de domaine du développement d'un langage de modélisation ($DS(M)L$[6]). Ce processus de définition est un exercice d'abstraction et de conceptualisation permettant de produire la **syntaxe abstraite** du langage de modélisation, également appelé **métamodèle** [17]. Comparés à la définition des syntaxes abstraites des langages de programmation, les métamodèles correspondent à des grammaires hors-contexte, spécifiées dans un formalisme compact basé sur la notation Backus-Naur Form (BNF). Concrètement, dans le contexte de l'OMG, la pratique courante pour définir les langages de modélisation consiste à utiliser le mécanisme du profil UML ou directement à l'aide du langage MOF.

La **syntaxe concrète** d'un langage de modélisation fait référence à sa notation, c'est-à-dire à la façon dont les utilisateurs l'apprendront et l'utiliseront, soit en lisant, soit en écrivant ou en concevant des modèles. La notation d'un langage de modélisation est très importante car elle correspond à la perspective et à l'expérience qu'auraient ses utilisateurs. Le succès d'un langage de modélisation dépendra donc du juste équilibre entre la simplicité et l'expressivité de la syntaxe concrète choisie pour les utilisateurs. En général, les notations graphiques semblent plus appropriées pour illustrer les relations entre les concepts. Cependant, comme un langage de modélisation peut avoir plusieurs syntaxes concrètes, différentes notations peuvent être fournies, à savoir des notations graphiques, textuelles, tabulaires, basées sur des formulaires, ou même une combinaison de celles-ci.

Un langage de modélisation doit être **pragmatique**. Le pragmatisme s'intéresse à la signification et à l'interprétation du langage en fonction du contexte [17]. Les langages de modélisation sont classés. Nous pouvons avoir un langage de modélisation généraliste ($GPML$[7]) ou un langage de modélisation spécifique à un domaine ($DSML$). Alors qu'un $GPML$ est caractérisé par le plus grand nombre de constructions génériques (par exemple UML, $SysML$), un $DSML$ a tendance à utiliser peu de concepts plus proches

6. Domain Specific (Modeling) Language
7. General Purpose Modeling Language

d'un domaine d'application particulier. Il est alors plus facile de lire, de comprendre et de communiquer avec un $DSML$ [17]. Les $DSLs$ peuvent améliorer largement la productivité, la fiabilité, la maintenabilité et la portabilité des logiciels [17, 23].

3.2.4 Les *MétaOutils* orientés modèle

Les approches IDM reposent sur des technologies et s'appuient généralement sur des outils complexes appelés «*MétaOutils*[8] pilotés par les modèles» et communément appelés «workbench de langages». Ces «workbenchs de langages» constituent des ateliers de travail sur les langages de modélisation. A ce titre, ces outils fournissent un ensemble de fonctionnalités pour aider les utilisateurs à définir les $DS(M)Ls$. Ils intègrent alors des outils spécifiques d'édition de modèle, de validation de modèles, de transformation de modèles, etc. Les principaux *MétaOutils* orientés modèle sont : *Eclipse EMF*, *Microsoft Software Factories* et *JetBrains MPS* [17].

Outre la métamodelisation, la transformation de modèle est également une opération centrale de l'IDM. Elle sera présentée plus tard en section 4.2.

3.2.5 La modélisation sous Éclipse

La modélisation sous Éclipse est un grand projet (ou EMP[9]) comprenant un ensemble intégré d'outils et de frameworks extensibles, incluant EMF(en son cœur), la modélisation graphique, la modélisation textuelle et des outils de modélisation prenant en charge les spécifications OMG telles que UML, OCL et $SysML$ [17, 26]. Eclipse Modeling Framework (EMF) est l'infrastructure principale de modélisation et de génération de code permettant de créer des outils et des applications basés sur les modèles définis dans le méta-métamodèle *Ecore* [41]. Plusieurs outils et espaces de travail (*frameworks*) sont également développés au-dessus d'EMF. On peu citer : GMF, $Sirius$, $GMF-Tooling$, $MoDisco$, $Papyrus$, $Acceleo$, ATL, $Epsilon$, MMT, $Xtext$, etc. En général, la plupart de ces outils sont populaires, relativement faciles à utiliser et à maintenir, ayant un soutien ouvert fort de la communauté de développement [17].

Le langage de modélisation Grafcet étant fondamentalement spécifique à la communauté des concepteurs et développeurs des systèmes de contrôle, l'approche IDM convient à suffisance pour son analyse et à sa mise en œuvre multi-cibles.

3.2.6 Programmation vs. modélisation

L'approche IDM promeut le développement logiciel via la modélisation, alors que l'approche usuelle consiste en la programmation. C'est pour-

8. Issu de $MetaTools$ en anglais
9. Eclipse Modeling Project [41]

quoi il est nécessaire de situer la modélisation au regard de la programmation. Pour cela, nous utilisons les travaux de M. Voelter [67] y relatifs.

D'un point de vue général, cet auteur parle de modèles descriptifs et de modèles normatif ou prescriptifs. Un *modèle descriptif* décrit un système existant dans le but de comprendre son fonctionnement, de communiquer cette compréhension aux parties prenantes et de prédire comment le comportement du système modélisé évoluera avec le temps ou en réaction à des stimuli. Contrairement, un *modèle normatif* représente un plan pour un système qui n'existe pas encore. Son objectif principal est de guider la construction de ce système. Dans notre cas sur la mise en œuvre du Grafcet, nous proposons un modèle normatif. Ainsi, M. Voelter voit l'orienté modèle comme désignant «*un moyen de développer un système logiciel S dans lequel les utilisateurs modifient un ensemble de modèles normatifs M_i représentant les préoccupations C_i à un niveau d'abstraction approprié afin d'influer sur le comportement de S*». Un processus automatique construit une implémentation complète de S reposant sur une plateforme P. Au regard de cette définition, il est évident qu'elle décrit (aussi) la programmation, la partie construction étant le compilateur. Toutefois, on pourrait soutenir que la différence entre modélisation et programmation est :

- que les compilateurs produisent du code machine (ce qui n'est pas vrai pour le compilateur Java qui produit du bytecode, lequel est un autre «modèle»),
- que l'écart sémantique entre l'entrée et la sortie est plus grand dans le cas de l'orienté modèle,
- que le contraire est le cas (également pas tout à fait vrai, considérons un compilateur Modelica qui crée un code C efficace à partir d'équations différentielles),
- ou que nous n'utilisions la séparation des problèmes que dans le cadre de modèles (ce n'est pas toujours vrai lorsqu'on considère la combinaison SQL + Java + HTML + JavaScript + CSS).

Il en ressort qu'il est impossible de tracer une ligne de démarcation entre la programmation et la modélisation (au sens de l'orienté modèle). Cependant, la programmation et les modèles ont clairement des histoires, des traditions et des communautés distinctes. En conséquence, bien que les deux soient fondamentalement identiques, ils mettent en avant différents aspects d'une approche commune. La figure 3.3 met en évidence cet accent. Sur ce schéma, plus l'élément est foncé, plus il a du poids pour cette propriété, ce qui est contraire lorsqu'il est moins foncé.

Les éléments de la figure 3.3 sont décrits comme suit:

- **A**: Les modèles sont souvent ciblés sur des domaines non logiciels, intégrant ainsi des concepts du domaine directement dans le langage. La programmation, par contre, met l'accent sur les concepts orthogonaux et composables (fonctions, classes, modules, etc).
- **B**: En conséquence, la programmation (et ses langages) est optimisée

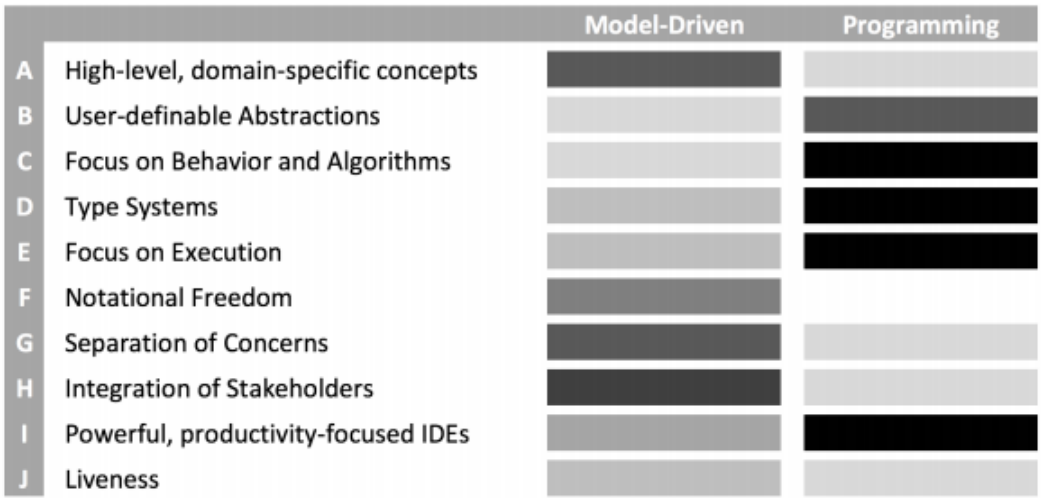


FIG. 3.3: Comparaison entre programmation et modélisation [67]

pour permettre aux utilisateurs de créer leurs propres nouvelles abstractions. En modélisation, les abstractions sont souvent prédéfinies et ne peuvent être assemblées dans des programmes que de manière très particulière.

- **C**: La programmation inclut presque toujours la spécification du comportement et la notion d'algorithme est fondamentale. Dans l'orienté modèle par contre, il y a beaucoup de langages qui ne spécifient que la structure (diagrammes de classes UML par exemple) ou bien spécifient le modèle comportemental au lieu de formuler le comportement de manière algorithmique.
- **D**: En conséquence de l'accent mis sur le comportement, les algorithmes et les abstractions définies par l'utilisateur, les langages de programmation reposent généralement sur des systèmes sophistiqués. En revanche, les modèles peuvent souvent se contenter de contrôles beaucoup plus simples.
- **E**: La programmation est clairement axée sur l'exécution: si elle ne s'exécute pas, ce n'est pas un programme. Dans le modèle, si l'exécution est toujours un facteur selon sa définition, d'autres aspects tels que l'analyse ou la communication avec les parties prenantes peuvent être la principale raison d'utiliser des modèles.
- **F**: À toutes fins pratiques, la programmation repose sur des notations textuelles, alors que l'orienté modèle est plus flexible dans son utilisation d'autres notations telles que des notations mathématiques, des tableaux ou des diagrammes. On pourrait même dire que l'orienté modèle met beaucoup l'accent sur les notations graphiques.
- **G**: Traditionnellement, un programme particulier était écrit dans un langage de programmation et les préoccupations étaient généralement partagées. Plus récemment, cela a changé pour la programmation, par

exemple avec la combinaison *SQL + Java + HTML + CSS + Javascript*. La modélisation a toujours eu tendance à décrire les différentes préoccupations du système séparément, avec des modèles et des langages différents.

- **H**: En partie à cause de l'accent mis sur la séparation des préoccupations, mais aussi en raison d'abstractions spécifiques à un domaine et de notations plus diverses, le modèle est mieux adapté à l'intégration de non-programmeurs ou d'experts de domaine dans le processus de développement logiciel. Cependant, les tentatives faites par des non-programmeurs pour lire le code source n'ont toujours pas été fructueuses.
- **I**: Quiconque a déjà utilisé les outils «typiques» pour l'orienté modèle conviendra probablement que les EDIs de programmation sont plus puissants et faciles d'usage: de la navigation dans le code à la refactorisation en passant par l'exécution de tests intégrés, il est difficile pour les outils de modélisation de les concurrencer.
- **J**: La Vivacité (ou *Liveness*) se réfère généralement à la disparition de la distinction entre un programme et son exécution. Comme il s'agit d'une tendance relativement nouvelle, les deux domaines étudiés sont relativement faibles. Nous suggérons que la modélisation est légèrement meilleure, car dans certains cas, les modèles sont créés spécifiquement pour l'analyse et le retour rapide, et parce que la vivacité peut être supportée plus facilement pour des domaines plus étroits (c'est-à-dire plus difficile pour les langages généraux).

Il en ressort que la modélisation et la programmation adressent des préoccupations différentes, l'aspect fondamental de l'orienté modèle est le fait qu'il est propre à un domaine métier et vise la facilité d'usage. Pratiquement, on note aussi que les outils modernes combinent à la fois les deux philosophies, selon leurs besoins.

La littérature présente de nombreux DSL développés pour résoudre des problèmes spécifiques. C'est le cas de V. Monthe et al [40] qui propose un DSL (RsaML) pour la description de l'architecture logicielle des robots avec intégration des propriétés temps réelles. C'est aussi le cas de T Messi Nguele [45] dont les travaux de thèse ont consisté à proposer un DSL pour la fouille des réseaux sociaux sur des architectures Multi-cœurs.

3.3 Métamodélisation Grafcet

Après avoir procédé à une analyse des métamodèles Grafcet existant, le langage Grafcet est analysé pour en identifier les concepts et leurs relations. Ils peuvent ensuite être formalisés via un métamodèle, lequel devra être validé à travers une plateforme de modélisation Grafcet.

3.3.1 Analyse des métamodèles Grafcet existant

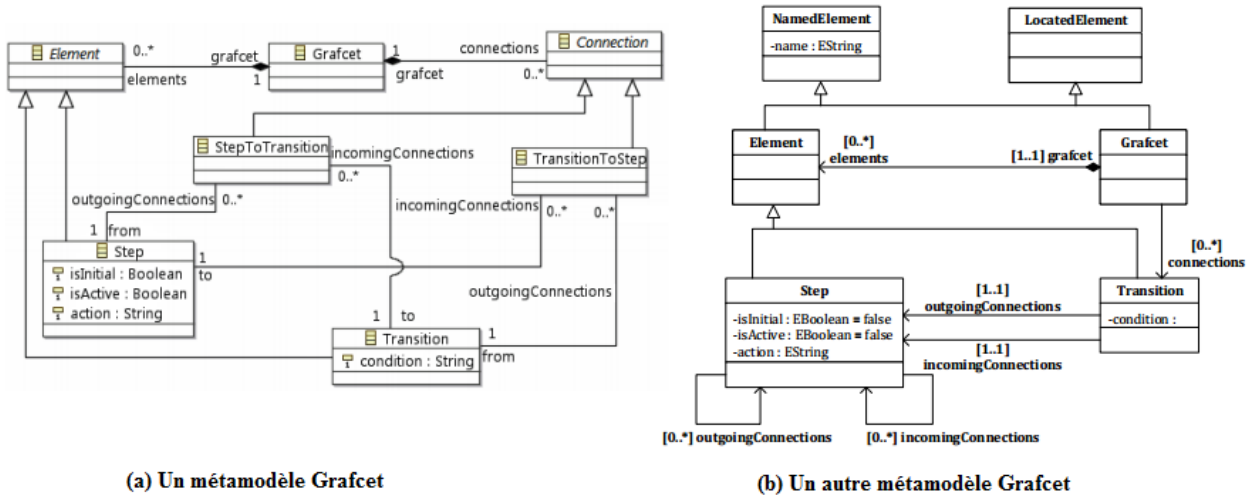
F. Schumacher et al. affirment dans [57] que la génération automatique de code de contrôle à partir de modèles de conception formels joue un rôle déterminant dans la promotion de l'IDM dans le domaine de l'ingénierie de l'automatisation. De même, ils clament que l'utilisation du paradigme IDM pour l'étude du langage Grafcet faciliterait certainement sa réalisation tout en résolvant les problèmes inhérents. Cependant, il apparaît clairement au regard de la littérature que très peu de travaux se sont penchés sur l'usage du Grafcet dans un processus IDM de façon à en proposer une syntaxe abstraite (ou métamodèle Grafcet) viable.

Dans [32], Robert Julius et al. étudient la transformation du Grafcet en code PLC, tout en incluant des structures hiérarchiques. Ils font allusion à un métamodèle Grafcet lors de la présentation du support logiciel pour le processus de transformation, mais aucun métamodèle Grafcet n'a été présenté. On remarque tout de même que ces auteurs font une confusion entre un modèle Grafcet UML et un métamodèle Grafcet: un métamodèle décrit la syntaxe abstraite d'un langage tout en mettant en exergue l'aspect pragmatique qui est à la base de son utilisation dans des processus automatiques de transformation. Voici quelques propositions de métamodèles de Grafcet identifiés dans la littérature.

Dans [51], Y. Qamsane et al. ont proposé un métamodèle Grafcet pour la transformation de contrôleurs distribués en modèles Grafcet afin de faciliter sa mise en œuvre. Comparativement au langage Grafcet, le métamodèle proposé (figure 3.4(b)) représente la structure de base du Grafcet, mais considère les étapes et les transitions comme des éléments Grafcet du même ordre. Il est alors possible qu'une étape soit précédée ou suivie d'une autre étape, ce qui ne correspond pas à la norme. De plus, une seule action (sous forme textuelle) peut être associée à une étape, sans prendre en compte son type (continu ou stocké); il en est de même des expressions Grafcet au niveau des réceptivités des transitions qui sont considérée comme un simple texte, ouvrant la voie à d'énormes incohérences.

De même dans [3], F. Basciani et al. présentent un métamodèle Grafcet (figure 3.4(a)) pour ressortir les transformations de modèle entre métamodèles incompatibles, avec une illustration basée sur les transformations entre Grafcet et réseaux de Petri. Ce modèle est conforme à la norme Grafcet et permet de représenter les concepts de la structure basique du langage, tout en restant superficiel. Bien qu'il ne puisse être utilisé pour représenter de manière exhaustive un Grafcet normalisé, ce métamodèle peut être étendu pour prendre en compte les aspects manquants: plusieurs actions associées à une étape, les types d'actions (actions de niveau et actions stockées), la notion de variables, les événements, les contraintes de temporelles, etc.

Il convient donc de noter qu'il existe très peu de travaux visant à réaliser la mise en œuvre du Grafcet par approche IDM. Bien que l'IDM



(a) Un métamodèle Grafcet **(b) Un autre métamodèle Grafcet**

FIG. 3.4: Métamodèles Grafcet de la littérature

soit recommandé comme approche, il n'est pas adopté dans la pratique [32]. Dans [32], les auteurs rappellent la formalisation des éléments de modélisation du Grafcet pour servir de base au développement piloté par les modèles [59, 56, 60, 28]. Bien que ces travaux étudient le Grafcet à des fins de formalisation, de validation et de test, aucun n'a abouti à un véritable métamodèle Grafcet, base de l'approche IDM.

Pour cela, certaines raisons sont données pour expliquer la non-adoption de l'approche IDM dans les pratiques des ingénieurs. En effet, bien que l'IDM offre de grandes opportunités et soit largement accepté par les chercheurs, les praticiens de terrain ne sont généralement pas familiarisés avec cette approche et ses outils [32]. Ils clament alors que les méthodes proposées dans la littérature sont basées sur des langages de modélisation qui ne leur sont pas familiers. Cependant, un langage tel que le Grafcet est déjà un langage de modélisation dans un domaine spécifique et ses utilisateurs ne sont pas nécessairement ceux qui le programment manuellement, mais aussi ceux qui utilisent des outils développés pour l'éditer et le traduire automatiquement en code. Les outils MDE utilisable à cette fin doivent être basés sur des modèles proposés par les chercheurs. Une fois les modèles de base mis en œuvre dans un environnement MDE, il sera possible de développer facilement tous les autres outils subséquents relatifs aux aspects tels que la vérification, la validation et la génération de code.

La deuxième raison est leur conviction que la plupart des approches dirigées par des modèles sont irréalistes dans la pratique car elles autorisent les modifications et les révisions sur les modèles afin de conserver la cohérence des codes et des modèles, tandis que dans la pratique les programmeurs d'API implémentent les modifications requises directement au niveau du code de l'API [32] . Mais on peut remarquer que les codes générés automatiquement peuvent être modifiés manuellement si nécessaire. De plus, la recherche sur le *roundtrip* a pour objectif de trouver des solutions

afin que ces changements subis par le code généré puissent être automatiquement répercutés sur les modèles utilisés en entrée [61]. Il faut cependant noter que de plus en plus, les industriels intègrent l'approche IDM dans leurs solutions logicielles [61].
De même, la conception et la spécification formelle des contrôleurs sont négligées dans la pratique industrielle pour d'autres raisons telles que: la pression du temps, la nécessité de transformer manuellement le modèle de spécification en code de contrôle et le manque de support approprié pour les outils IDM existant [57]. Les ingénieurs préfèrent généralement utiliser UML pour modéliser, puis implémenter le modèle UML dans un langage orienté objet ou non.
Nous pouvons donc remarquer le manque d'enthousiasme à utiliser l'approche IDM et ses outils. En effet, l'appropriation de ces approches nécessite des efforts considérables [61]. Néanmoins, l'IDM préconise l'utilisation de modèles pour le génie logiciel, ainsi que leur implémentation.
En résumé, bien que le langage Grafcet soit fondamentalement spécifique à un domaine, il n'existe pas de travaux approfondies qui aient été menées pour définir les bases de modélisation, permettant ainsi le développement par approche IDM des plateformes destinées aux ingénieurs de contrôle.
Notre souci ici est d'analyser, de conceptualiser le langage Grafcet et d'en proposer un métamodèle qui servira de base à la production d'artefacts logiciels pour le Grafcet. Cela pourrait également convaincre la communauté des praticiens d'adopter facilement le paradigme IDM dont les avantages sont bien établis, à savoir l'amélioration de la productivité et de la qualité, la validation et la vérification des modèles, etc [66]. Nous nous intéressons alors principalement à la modélisation des éléments Grafcet sans tenir compte des structures hiérarchiques.

3.3.2 Analyse et identification des concepts Grafcet

Nous analysons le langage Grafcet pour identifier les concepts de sa structure base, les autres concepts y compris les expressions Grafcet et les variables temporelles.

3.3.2.1 Identification des concepts de la structure base du Grafcet

Selon la norme IEC 60848 2nd Ed. [16], il apparaît qu'un modèle Grafcet regroupe plusieurs étapes et transitions. Ils sont reliés entre eux par des liens orientés, également appelés connexions, ainsi que des variables. Les étapes (le concept *Etape*), les transitions ($Transition$), les liens orientés ($Connexion$) et les variables ($Variable$) sont des éléments Grafcet ($G7Element$), chacun ayant un nom. Les connexions Grafcet peuvent être positionnées en entrée ou sortie en sortie des étapes et de transitions. Deux types de connexions sont identifiés facilement : les connexions de transition

vers étape (*TransitionToStep*) et les connexions de transition vers étape (*StepToTransition*). Chaque instance de *TransitionToStep* sort d'une transition et se dirige vers une étape, tandis que chaque instance de *StepToTransition* sort d'une étape et se dirige vers une transition. Cette description est bien formalisée par le métamodèle Grafcet de base proposé par F. Basciani et al. dans [3] et présenté en figure 3.4(b). Ce modèle est loin d'être exhaustif car plusieurs autres concepts de Grafcet existent.

3.3.2.2 Autres concepts du Grafcet

Une variable d'étape est associée à une étape pour représenter son activité. Il s'agit d'une variable booléenne (*BooleanVariable*) interne au Grafcet. Toute variable (*Variable*) est d'entrée, de sortie ou interne. Nous avons besoin de représenter la durée d'activité de toute variable d'étape, ainsi que de toute autre variable booléenne, car elles peuvent intervenir dans les conditions de temporelles décrites plus loin. Plusieurs actions (*Action*) peuvent être associées à une étape. Une action est aussi un *G7Element*. Toute action est représentée et réalisée par sa variable. Cette manière de structurer les actions et variables offre la possibilité d'avoir la même action associée à plusieurs étapes différentes comme indiqué dans la norme. Une action peut uniquement être stockée (*StoredAction*) ou à niveau (*LevelAction*). Une instance de *LevelAction* a une valeur booléenne et une condition associée pour modéliser les actions à niveau conditionnelles. En l'absence de cette condition, elle est considérée comme vraie et correspond à une action à niveau continue. Chaque instance de *StoredAction* a une expression stockée qui est réalisée lors de l'activation ou de la désactivation de l'étape à laquelle elle est associée. Une instance de *StoredAction* symbolise une instruction d'affectation pour une variable booléenne ou numérique. Cet instance peut également faire référence à un ensemble d'instructions à exécuter instantanément aux instants sus-évoqués, auquel cas il s'agit d'une fonction (*Function*) ayant un nom et un corps.

3.3.2.3 Concepts relatifs aux expressions Grafcet

Les concepts discutés ici n'apparaissent pas explicitement dans le standard Grafcet, mais ils y existent implicitement et leur considération permet de résoudre certains problèmes liés à la vérification de modèle et l'obtention d'une sémantique appropriée. Les expressions arithmétiques et/ou logiques sont utilisées à plusieurs niveaux du langage. Ces expressions sont principalement liées aux conditions Grafcet et aux expressions arithmétiques. Ils sont présents à trois niveaux: les réceptivités des transitions, les conditions associées aux actions à niveau et les expressions des actions stockées. Ces expressions utilisent des opérateurs arithmétiques et logiques, qui peuvent être enrichis par des opérateurs temporels. Une expression (*Expression*) a une valeur et peut être simple ou composée à l'aide d'opérateurs. Les expressions composées peuvent être structurées niveau par niveau, tout en tenant

compte de l'ordre de priorité des opérateurs impliqués. Une telle expression peut alors avoir un opérateur (*Operator*) et deux sous-expressions représentant ses opérandes. Pour les opérateurs unaires, il s'agira d'une seule sous-expression. Les opérateurs peuvent être subdivisés en trois catégories en fonction de leur utilisation: les *opérateurs logiques* (ET, OU, NON, RE, FE, <=, <,>=,>, <>), les **opérateurs arithmétiques** (+, −, ∗, /, l'opérateur unaire −) et les **opérateurs temporels** (retardé ou *Delayed*, limité de type 1 ou *Limited*1 et limité de type 2 ou *Limited*2) tels que présentés en section 1.4.1.4. Cela permet l'identification des concepts suivants: *LogicalOperator*, *ArithmeticOperator* et *TimingOperator*.

3.3.2.4 Modélisation des variables temporelles

La norme Grafcet parle de variable temporelle, telle que $X1/3s$ (notée *Delayed*1), et $\overline{X1/3s}$ (notée *Limited*) et $2s/X1/3s$ (qui est une notation retardée définis dans la norme *CEI 617-12* et également utilisé dans le standard Grafcet, que nous notons *Delayed*2). Toutes ces expressions utilisent une variable et leur interprétation est un calcul. La condition de retard $X1/3s$ est vraie si le temps écoulé depuis que la variable $X1$ est devenue *true* est au moins égal à 3 secondes. Il ne s'agit donc pas d'une simple variable comme une variable booléenne dont la valeur peut être consultée immédiatement, mais d'une opération à effectuer. L'interprétation des expressions temporelles *Limited* et *Delayed*2 entraine aussi un calcul, tel que explicité en section 1.7.

Le récapitulatif de tous les concepts identifiés dans le domaine du langage Grafcet est donné dans le tableau 3.1.

3.3.3 Le métamodèle Grafcet

La formalisation des concepts Grafcet et les relations entre eux aboutit au métamodèle Grafcet de la figure 3.5.

TAB. 3.1: Concepts (Classes) identifiés dans le domaine Grafcet

Concepts Grafcet	**Description**	**Concept du métamodèle**
Grafcet	Un schéma dessiné et conforme au langage Grafcet	*Grafcet*
Element	Un élément du modèle Grafcet	*G7Element*
Step	Un noeud du Grafcet	*Step*
Transition	Un noeud du Grafcet	*Transition*
Oriented links	Lien orienté reliant les étapes aux transitions et les transitions aux étapes	*Connection*
Oriented link from transition to step	Lien orienté reliant les transitions aux étapes	*TransitionToStep*
Oriented link from step to transition	Lien orienté reliant les étapes aux transitions	*StepToTransition*
Variable	Pour représenter les signaux en entrée/sortie du système, ou bien l'activité des étapes	*Variable*
Boolean Variable	Toute variable ayant une valeur true/false	*BooleanVariable*
Action	Associée à une étape, elle représente une sortie du système	*Action*
Level action	Action exécutable en situation stable du Grafcet	*LevelAction*
Stored action	Action exécutable instantanément lors de l'activation/désactivation d'une étape	*StoredAction*
Réceptivité ou condition	Expression logique associée à une transition ou à une action conditionnelle	*Expression*
Expression d'une action stockée	Partie droite d'une affectation de variable	*Expression*
Logical operator	Un opérateur qui a des opérandes logiques et retourne une valeur logique	*LogicalOperator*
Arithmetic operator	Un opérateur qui a des opérandes arithmétiques et retourne une valeur arithmétiques	*ArithmeticOperator*
Timing variable	Concerne les conditions liées au temps (retardée ou limitée)	*TimingOperator*

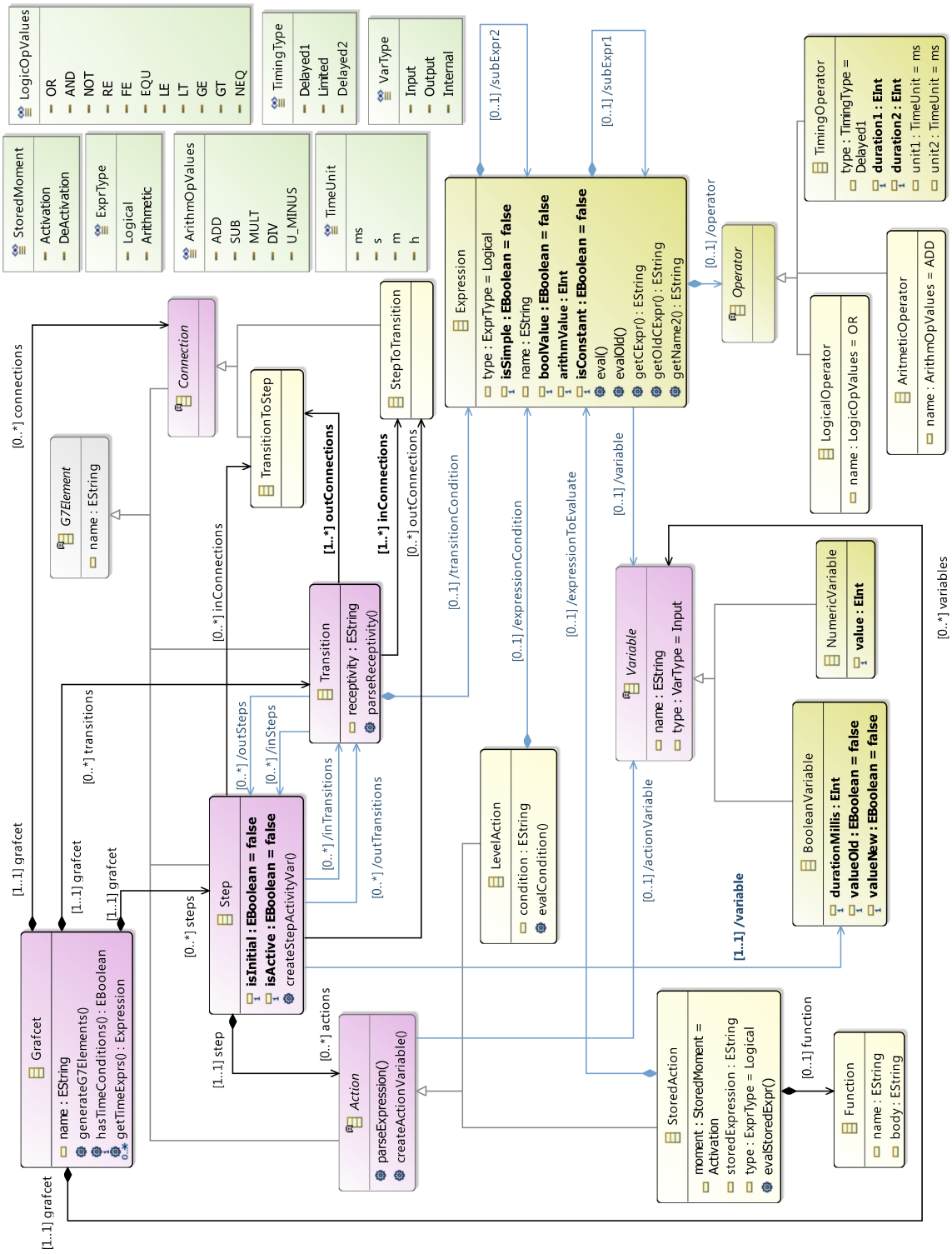


FIG. 3.5: Métamodèle Grafcet proposé

Compte tenu de l'importance des concepts tels que *Step*, $Transition$ et $Variable$, les liens bidirectionnels sont utilisés pour référencer le Grafcet qui les contient. Certaines classes sont abstraites, comme $G7Element$, *Connection*, *Action*, $Variable$ et *Operator*. En fait, toute instance de *Step*, $Transition$, *Connection* ou *Action* est un $G7Element$; une instance de *Connection* est soit un $StepToTransition$, soit un $TransitionToStep$; toute variable dans ce contexte est numérique ou booléenne; et une action est soit à niveau, soit stockée. Pour matérialiser la durée d'activité d'une instance de $BooleanVariable$, nous utilisons l'attribut $durationMillis$, qui correspond à la durée de son activité en millisecondes. Outre la valeur d'une variable booléenne, nous devons également considérer sa valeur à l'instant précédent, ce qui est utile pour fournir une sémantique aux événements front montant(RE: passage de $false$ à $true$) et au front descendant (FE : passage de $true$ à $false$), largement utilisés dans l'algorithme d'interprétation du Grafcet. La classe énumérée $LogicOpValues$ donne une énumération des opérateurs logiques, tandis que $ArithmOpValues$ donne une énumération des opérateurs arithmétiques. Un certain nombre de relations doivent être automatiquement dérivées. Il s'agit des références vers les instances de certains concepts du métamodèle. Par exemple, le nom de la variable d'activité associée à une étape est obtenu à partir du nom de cette étape (par exemple: X + **step.name**). Dans la mesure où toute instance de *Step* fait référence au Grafcet le contenant, la variable qui lie *Step* à $BooleanVariable$ peut être créée et référencée automatiquement. Il en est de même pour la variable qui lie *Action* à $Variable$. Les autres expressions dont les valeurs sont dérivées automatiquement sont les expressions Grafcet. Ils sont décrits plus loin à la section 3.3.6.

3.3.4 Concept *Expression* et raisons de sa modélisation

De nombreuses sources d'erreurs dans le Grafcet proviennent des expressions mal formées ou incohérentes. Elles doivent être représentées de manière à faciliter leur vérification et leur validation. De même, une telle représentation donne la possibilité de trouver facilement la sémantique appropriée pour toute expression, quelle que soit sa complexité.

Une expression a de nombreux attributs et une relation avec d'autres concepts. Outre le type d'expression qui peut être logique ou arithmétique (décrit par le type énuméré $ExprType$), une expression a un nom et une valeur (booléenne ou arithmétique, qui ne peut être utilisée que de manière exclusive). Pour les simulations, l'attribut $boolValue$ est utilisé lorsque l'expression est logique et l'attribut $arithmValue$ est utilisé lorsqu'elle est de type arithmétique. La méthode $eval()$ est utilisée pour évaluer sa valeur courante, tandis que $evalOld()$ permet de déterminer sa valeur à l'instant précédent. Lors de la génération de code, la méthode $getCExpr()$ sera appelée pour obtenir l'expression courante en langage C, tandis que $getOldCExpr()$

sera appelée pour obtenir l'expression C correspondante pour l'obtention de sa valeur à l'instant précédent. Ces deux dernières méthodes sont utilisées pour donner une sémantique en langage C aux expressions représentant les événements de front montant (RE) et de front descendant (FE). La méthode $getName2()$ est essentielle pour renvoyer une proposition d'identifiant pour une expression. Elle s'est avérée utile lors de la création de la variable associée utilisée pour compter le temps.

Une instance d'expression peut avoir deux sous-expressions ($subExpr1$ et $subExpr2$) combinées avec un opérateur. Lorsque cette expression est simple ($isSimple = true$), il peut s'agir d'une variable ($isConstant = false$) ou d'une constante ($isConstant = true$). Dans le cas où il s'agit d'une variable, une référence à cette variable est inférée automatiquement après une recherche parmi les variables du modèle Grafcet.

3.3.5 Les opérateurs utilisés dans les expressions

Dans le Grafcet, on distingue les opérateurs temporels et tout autre opérateur de base connu.

Étant donnée une expression de synchronisation, une instance de $TimingOperator$ est utilisée pour sa construction. Dans ce cas, une seule sous-expression est utilisée, comme dans le cas des opérateurs unaires. Une instance de $TimingOperator$ contient un type ($Limited$, $Delayed1$ ou $Delyed2$) et deux durées ($duration1$ d'unité $unit1$ et $duration2$ d'unité $unit2$); $duration2$ n'est utilisé que s'il s'agit d'une expression temporelle de type $Delayed2$. Les unités de durées sont définies dans le type $TimeUnit$.

Les autres opérateurs sont définis dans les types énumérés. Le type énuméré $LogicOpValues$ contient les éléments suivants: OR (OU logique), AND (ET logique), NON (négation), RE (front montant), FE (front descendant), EQU (comparaison d'égalité), LE (inférieur ou égal à, <=), LT (strictement inférieur à, <), GE (supérieur ou égal,>=), GT (strictement supérieur à,>), NEQ (non égal ou différent, <>). Le type énuméré $ArithmOpValues$ contient les éléments suivants: ADD (addition, +), SUB (soustraction, −), $MULT$ (multiplication, ∗), DIV (division, /) et U_MINUS (moins unaire).

3.3.6 Dérivation automatique de certaines expressions

La construction des expressions logiques et arithmétiques Grafcet telle que modélisée est une tâche fastidieuse si elle est effectuée manuellement. Dans la section 3.4, nous présentons la méthode permettant de l'automatiser et de l'intégrer à la construction Grafcet. En effet, les expressions doivent être construites automatiquement avec leurs associations associées. Les relations unidirectionnelles suivantes sont dérivées en appelant l'analyseur syntaxique des expressions Grafcet: *transitionCondition* qui est une référence de $Transition$ vers $Expression$ (pour sa réceptivité), *expressionCondition*

qui est une référence de *LevelAction* vers *Expression* (pour sa condition) et *expressionToEvaluate* qui est une référence de *StoredAction* vers *Expression*(pour son expression à évaluer). De même, lors de l'analyse d'une expression par l'analyseur, si nous rencontrons un identifiant, celui-ci se référera nécessairement à une variable et la variable correspondante sera automatiquement référencée. Lorsque cette variable n'existe pas dans l'instance Grafcet en cours, une exception ($VariableNotFoundException$) est levée. Pour détecter ce problème dans l'éditeur dynamique Grafcet, une règle OCL sera utilisée.

3.3.7 Calcul des positions relatives entre les étapes et les transitions

Outre les relations dérivées décrites en 3.3.6, la dérivation des positions relatives entre les étapes et les transitions est discutée ici. En effet, toute utilisation des modèles Grafcet résout ce problème, car cela apparaît clairement dans les règles d'évolution du Grafcet. Dans le modèle proposé, il n'y a pas de relation directe entre les concepts *Step* et $Transition$. La modélisation proposée doit permettre de savoir si pour une transition donnée, il est possible de trouver toutes les étapes en entrée (étapes en amont) et toutes les étapes en sortie (étapes en aval). C'est également le cas de la position relative des transitions par rapport aux étapes. Nous apportons une solution en utilisant une approche IDM en lieu et place du codage matriciel du Grafcet. Au lieu d'implémenter des algorithmes complexes, les environnements IDM proposent des langages tels que OCL pour interroger les instances de métamodèle. Ici, nous créons les propriétés dérivées dans des concepts *Step* et $Transition$ comme suit:

- **L'ensemble des étapes en entrée d'une transition**:
  Selon la modélisation, une étape est en entrée d'une transition s'il existe une connexion (de type $StepToTransition$) qui est à la fois à la sortie de cette étape et en entrée de cette transition. Ces étapes sont obtenues en créant et en initialisant comme suit la propriété *inSteps* dans le contexte $Transition$:

  Listing 3.1: Dérivation des étapes en entrée d'une transition

```
property inSteps:Step[*] { derived volatile }
{
        derivation: (grafcet.steps->select(step|step.
            outConnections->exists(outCon|self.inConnections->
            includes(outCon))))->asSet();
}
```

- **L'ensemble des étapes en sortie d'une transition**
  Une étape est à la sortie d'une transition s'il existe une connexion (de

type *TransitionToStep*) qui est à la fois à l'entrée de cette étape et à la sortie de cette transition. Ils sont obtenus en créant et en initialisant la propriété *outSteps* dans le contexte *Transition* comme suit:

Listing 3.2: Dérivation des étapes en sortie d'une transition

```
property outSteps : Step[*] { derived volatile }
{
        derivation: (grafcet.steps->select(step|step.
            inConnections->exists(inCon|self.outConnections->
            includes(inCon))))->asSet();
}
```

– **L'ensemble des transitions en entrée d'une étape**
Une étape est en sortie d'une transition s'il existe une connexion (du type *TransitionToStep*) qui est à la fois en entrée de cette étape et en sortie de cette transition. Elles sont obtenues par création et initialisation de la propriété *outSteps* dans le contexte *Step* comme suit:

Listing 3.3: Dérivation des transitions en entrée d'une étape

```
property inTransitions : Transition[*] { derived volatile}
{
        derivation: (grafcet.transitions->select(trans|trans.
            outConnections->exists(outCon|self.inConnections->
            includes(outCon))))->asSet();
}
```

– **L'ensemble des transitions en sortie d'une étape**
Une transition est en entrée d'une étape s'il existe une connexion (du type *TransitionToStep*) qui est à la fois en entrée de cette étape et en sortie de cette transition. Ces transitions sont obtenues par création et initialisation de la propriété *inTransitions* dans le contexte *Step* comme suit:

Listing 3.4: Dérivation des transitions en sortie d'une étape

```
property outTransitions : Transition[*] { derived volatile }
{
        derivation: (grafcet.transitions->select(trans|trans.
            inConnections->exists(inCon|
        self.outConnections->includes(inCon))))->asSet();
}
```

L'intégration de ces propriétés dans le métamodèle rend automatique le calcul de ces références dynamiques dans les instances de la classe *Expression* à l'intérieur des modèles Grafcet.

C'est après avoir présenté la gestion des expressions Grafcet que nous présenterons tour à tour la prise en compte des contraintes sémantiques (section 3.5) et l'implémentation du métamodèle Grafcet dans Éclipse EMF (section 3.6).

3.4 Gestion automatique des expressions Grafcet

Dans la sous-section précédente, nous avons évoqué la construction de façon dynamique des expressions du Grafcet. Nous faisons ressortir ici les raisons qui nous poussent à se pencher particulièrement sur l'analyse profonde des expressions Grafcet, puis nous en proposons une solution adaptée dans un contexte IDM.

3.4.1 Pourquoi analyser et formaliser les expressions

Les conditions de transition, les actions à niveau conditionnelles et les actions stockées utilisent des expressions. C'est une grande faiblesse lorsque les expressions Grafcet sont considérées comme de simples chaînes de caractères dans la phase de modélisation, car les éléments manipulés sont généralement dérivés des autres constructions des modèles. Ainsi, leur négligence entraîne de nombreuses incohérences dans les phases ultérieures du processus IDM.

Le concept Expression du métamodèle présenté en figure X donne une représentation intéressante des expressions sous une forme arborescente. Cependant, demander à l'utilisateur de construire lui-même (manuellement) une expression composée de façon récursive est une tâche pénible. Elle nécessite le référencement des variables utilisées dans l'expression, la création des opérateurs à chaque niveau, etc. Le faire manuellement est coûteux en temps et sujet à de nombreuses erreurs.

De même, l'utilisation dans une même expression des opérateurs arithmétiques et logiques, telles que `b1 OR b2 AND (n1+n2>10)`, est présente une difficulté. En effet, dans la communauté des automaticiens et la plupart des outils d'édition de modèles Grafcet, les symboles + et $*$ dénotent respectivement les opérateurs OR et AND, ce qui crée une confusion avec les opérateurs arithmétiques ordinaires correspondants (+, $*$) qui peuvent elles-même être utilisées dans les expressions. Cela rend difficile l'analyse et la validation des expressions construites. Nous envisageons proposer une syntaxe naturelle d'écriture de ces opérateurs en utilisant les mots clés tels que AND, OR, NOT, RE, FE , ... Ceci offre ensuite une grande possibilité pour la création, l'analyse et la validation des expressions complexes. En

effet, toute expression représentant une condition doit être de nature booléenne ou logique. Il est donc possible d'analyser l'arbre abstrait construit pour vérifier et valider de telles contraintes sémantiques.

Une autre raison est la nécessité de donner une sémantique appropriée aux événements front montant(RE) et front descendant (FE) utilisables au niveau des expressions ; c'est aussi le cas pour les conditions temporelles qui tirent leur sémantique des événements précédents.

Au vu de toutes ces raisons, il importe qu'une solution automatique de construction des expressions soit mise sur pied. La vérification et la validation de la sémantique d'un modèle sera définie ensuite. Comme les expressions sont de nature textuelle, il existe des outils formels d'analyse de programmes (i.e. du texte) depuis la genèse des langages de programmation. C'est le cas principalement des Grammaires formelles, qui sont en fait par analogie avec l'IDM, le métamodèle du langage formalisé. Une grammaire formelle, au lieu d'être conforme au méta-métamodèle MOF, est plutôt conforme au métalangage BNF. Ce métalangage présente une notation formelle permettant de décrire les règles syntaxiques des langages de programmation. Le dernier souci sera l'implémentation de la Grammaire et son intégration dans l'environnement de méta-modélisation. L'outil ANTLR apparaît comme une véritable solution à toutes ces préoccupations.

3.4.2 La grammaire du langage des expressions Grafcet

La quasi-totalité des langages généraux de programmation offrent la possibilité de réaliser des opérations arithmétiques et logiques en utilisant des opérateurs [5]. En section 3.3.5, nous avons présenté les opérateurs utilisés dans les expressions Grafcet. Ces expressions peuvent être formalisées à l'aide de la grammaire donnée en figure 3.6. Dans cette grammaire, $G7Expr$ est l'axiome. Le non terminal $timeLogicG7Expr$ permet de produire toutes les expressions temporelles, tandis qu'une expression atomique ($Atomic$) est soit un identificateur (Id), soit un nombre ($Number$) ou encore une valeur booléenne ($BoolValue$).

Définition 3.1. ***Grammaire récursive à gauche***
Une grammaire hors-contexte G est dite récursive à gauche s'il existe une dérivation du genre $A \Rightarrow (*)A\alpha$, où A est un nom terminal et α est une expression quelconque.

Cette grammaire (figure 3.6) hors-contexte est récursive à gauche à cause de ses nombreuses règles ayant une récursivité gauche, à l'exemple de la première règle qui est :

$G7Expr \rightarrow G7Expr('and'|'AND'|'\&\&')G7Expr$

En analyse $LL(1)$, elle est ambiguë et donc pas directement exploitable par les outils automatiques d'analyse LL(1). Il est possible de la désa-

```
G7Expr      : G7Expr ('and'| 'AND'|'&&') G7Expr
            | G7Expr ('or'| 'OR'|'||') G7Expr
            | G7Expr ('==' |'=' | '!=' | '<>') G7Expr
            | G7Expr ('<=' |'>=' | '<' | '>') G7Expr
            | G7Expr ('+'| '-') G7Expr
            | G7Expr ('*'| '/') G7Expr
            | '(' G7Expr ')'
            | ('not'| '!') G7Expr
            | RE G7Expr
            | FE G7Expr
            | timeLogicG7Expr
            | Atomic
timeLogicG7Expr --> '[' Number u '/' G7Expr ']'
                  | '[' ('not'| '!') Number u '/' G7Expr]
                  | '[' Number U '/' G7Expr '/' Number u ']'
Atomic --> Number | Id |BoolValue
```

FIG. 3.6: Grammaire des expressions Grafcet

mbigüiser et la rendre $LL(1)$ en éliminant la récursivité gauche à l'aide de :

- l'associativité des opérateurs,
- l'ordre de priorité des opérateurs

Les opérateurs intervenant dans les expressions Grafcet sont classés comme suit, du moins prioritaire au plus prioritaire [5]:

- L'opérateur OU (OR bien $||$)
- L'opérateur ET (AND ou bien &&),
- L'égalité ($=$) et la différence ($!=$ ou $\neq$)
- Les opérateurs de comparaison ($<=$, $>=$, $<$, $>$)
- L'addition ($+$) et la soustraction ($\neg$)
- La multiplication ($*$) et la division ($/$)
- La négation (! ou NOT), le front montant (RE) et le front descendant (FE)

Cependant, il n'est pas nécessaire de déterminer une grammaire $LL(1)$ équivalente à celle de la figure 3.6. En effet, il est possible de se servir de certains outils d'analyse $LL(k)$ ayant émergés durant les récentes années. C'est le cas de l'outil $ANTLR$ qui permet de produire automatiquement l'analyseur syntaxique à partir d'une spécification de la grammaire prise en entrée. Ainsi, sa version 4 offre la possibilité d'utiliser l'ordre d'apparition des règles de production lors de la spécification de la grammaire pour gérer les conflits d'apparition des règles et la rendre $LL(k)$.

3.4.3 Mise en œuvre du langage des expressions Grafcet

La mise en œuvre du langage des expressions Grafcet s'est faite avec l'outil ANTLR.

3.4.3.1 L'outil ANTLR

ANTLR signifie *Another Tool for Language Recognition*. C'est un puissant générateur d'analyseurs syntaxiques à descente récursive de type $LL(k)$ pour la lecture, le traitement, l'exécution ou la traduction de fichiers texte ou binaires structurés. Il est largement utilisé pour créer des langages, des outils et des frameworks [46]. À partir d'une grammaire, ANTLR génère un analyseur qui peut construire et parcourir des arbres d'analyse. Il permet ensuite de construire des langages de domaine spécifiques (DSL), domaine au cœur de l'IDM. ANTLR prend en entrée un fichier *.g*4 de la spécification lexicale et des règles syntaxique de la grammaire en entrée. Son langage cible est principalement le langage Java. Le flux de données de traduction global ANTLR est présenté à la figure 3.7. L'analyseur lexical (lexer) est un

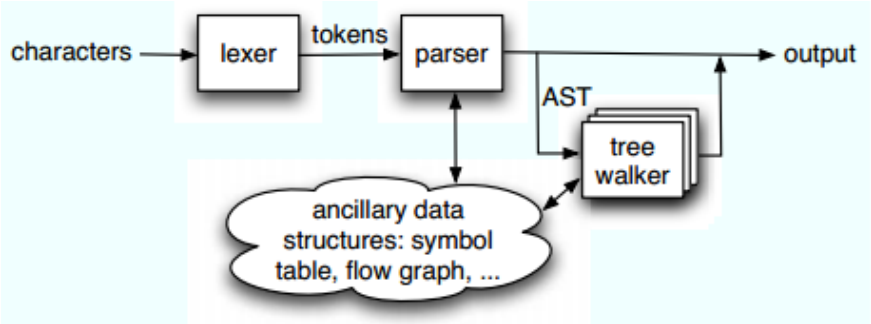


FIG. 3.7: Flux de données dans ANTLR [46]

programme qui regroupe les caractères en mots ou en symboles (*tokens*). Il peut regrouper des *tokens* associés en classes de *tokens*, ou types de *tokens* encore appelés unités lexicales, tels que INT (entiers), ID (identifiants), $FLOAT$ (nombres à virgule flottante), etc. L'analyseur syntaxique (*parser*) quant à lui est un programme qui se nourrit de ces *tokens* pour reconnaître des phrases du langage. Les analyseurs générés par ANTLR construisent l'arbre d'analyse syntaxique (AST[10]) correspondant à la phrase d'entrée.

Les *tokens* de l'analyseur lexical sont spécifiés avec des expressions régulières tandis que l'analyseur syntaxique est spécifié avec des règles de la grammaire.

3.4.3.2 Implémentation de la grammaire des expressions

La grammaire du langage des expressions Grafcet proposée en 3.4.2 a été implémentée à l'aide de ANTLR.

Le code ci-après présente le contenu du fichier *.g*4 associé à cette grammaire pour la spécification de l'analyseur lexical, ainsi que la spécification des règles de production de l'analyseur syntaxique associé.

10. Abstract Syntax Tree

Listing 3.5: Implémentation ANTLR de la grammaire du langage des expressions Grafcet (analyseur lexical et syntaxique)

```
/** Grammar for the Grafcet expressions (logical and arithmetical) */
grammar g7Expression ;
@lexer::header { package g7Expression; }
@parser::header { package g7Expression; }

//Operators
AND : 'and'| 'AND'|'&&';
OR : 'or'| 'OR'|'||';
EQUAL: '==' |'=' ;
DIFF : '!=' | '<>';
LE: '<=' ;
GE: '>=' ;
LT : '<' ;
GT : '>';
MULT: '*';
DIV : '/';
ADD: '+';
SUB : '-';
NOT : 'not' |'NOT' | '!';
RE : 'RE' | 're';
FE : 'FE' | 'fe';
BoolValue : 'true'|'false';
U : 'ms' | 's' |'m'|'h' ;

fragment Digit : [0-9];

fragment Letter : ([a-z] | [A-Z]|'_') ;
fragment IntPlus : Digit+ ;
Number : Digit+('.'Digit+)?;
Id : (Letter)(Letter|Digit)*;
WS : [ \t\r]+ -> skip ; //for spaces? Just skip

//Grammar Productions rules
myG7Expr: g7Expr ; // myG7Expr is the Axiom of the grammar (here is
    the first production)
g7Expr : op = (NOT|RE|FE) g7Expr   #UnaryLogicOp
               | left = g7Expr op = (LE|GE|LT|GT) right = g7Expr #
                  LEcmp_G7Expr
               | left = g7Expr op = (EQUAL|DIFF) right = g7Expr #
                  EqualDiff_G7Expr
               | left = g7Expr op = (AND|OR) right = g7Expr  #
                  AndOr_G7Expr

               | op = SUB g7Expr     #InfixMinus
               | left = g7Expr op = (MULT|DIV) right = g7Expr  #
                  MulDiv_G7Expr
               | left = g7Expr op = (ADD |SUB) right = g7Expr  #
                  AddSub_G7Expr

               | atomic       #primaryAtom
```

```
45                | timeLogicG7Expr     #primaryTiming
46                |'(' g7Expr ')'      #primaryParenthesis
47  ;
48  timeLogicG7Expr :
49          '[' nb1 = Number unit1 = U '/' g7Expr '/' nb2 = Number unit2 =
                U ']' #timeLogicDelayed2
50          | '[' nb = Number unit = U '/' g7Expr']'    #timeLogicDelayed1
51          | '[' op = NOT nb = Number unit = U '/' g7Expr']'  #
               timeLogicG7ExprLimited
52  ;
53  atomic : Number  #AtomNumber
54                |Id   #AtomId
55                |BoolValue #AtomBool
56  ;
```

L'axiome de cette grammaire est $myG7Expr$. Des attributs sont ajoutés aux productions afin de faciliter les processus d'analyse d'arbre. L'attribut *op* fait référence à la valeur de l'opérateur lors du traitement de certaines règles. C'est également le cas de $left$ et $right$ qui référencent les sous-expressions gauche et droite d'une expression. D'autres attributs sont également ajoutés tels que $nb1$, $unit1$, $nb2$, $unit2$ à l'intérieur des règles relatives aux conditions temporelles. ANTLR permet l'utilisation d'un caractère spécial # suivi d'un identifiant. Cet identifiant fait référence au nom de la méthode d'analyse qui implémente la règle correspondante. Les identifiants sont très utiles lorsqu'un non-terminal est dérivé plusieurs fois, ce qui permet de distinguer les dérivations les unes des autres. Par exemple, le non terminal $g7Expr$ de la grammaire a dix (10) alternatives de dérivations. Pour chaque règle de production associée, le nom de la méthode correspondante est spécifié suivant la règle après le symbole #. Chaque méthode est un contexte spécifique dans lequel tous les éléments de cette règle sont identifiés et traités.

Pour permettre la gestion des conflits qui apparaissent dans les règles de cette grammaire en analyse $LL(k)$, l'ordre de priorité des opérateurs est considéré pour réorganiser ses règles. Une règle concernant un opérateur de priorité élevée apparait avant celle qui concerne un opérateur de priorité inférieure. Par exemple, les opérateurs $MULT(*)$ ou $DIV(/)$ ont plus de priorité que $ADD(+)$ ou $SUB(-)$. Par conséquent, les règles de production qui les utilisent apparaissent dans l'ordre suivant:

Listing 3.6:

```
g7Expr : left = g7Expr op = (MULT|DIV) right = g7Expr #MulDiv_G7Expr
       | left = g7Expr op = (ADD |SUB) right = g7Expr #AddSub_G7Expr
```

Quand la grammaire est valide au regard de la syntaxe du langage d'entrée de ANTLR, le code de l'analyseur lexical et de l'analyseur syntaxique est généré par appel de la commande ci-après. À ce code est ajouté le code de base du *design pattern* $Visitor$ associé.

```
> antlr4 g7Expression.g4 -o g7Expression -visitor
```

L'intégration du parseur des expressions Grafcet à la plateforme IDM Éclipse EMF est explicitée en section 3.6.3.

3.4.3.3 L'outil ANTLRWorks

L'outil *ANTLRWorks* [10] est développé pour afficher l'arbre syntaxique obtenu par application des règles de la grammaire pour un mot fourni en entrée. Il est surtout utilisé à des fins de débogage de la grammaire en cours de développement. Les principaux composants de *ANTLRWorks* sont : un éditeur de grammaire avec des fonctions de refactoring et de navigation, un interpréteur de grammaire et un débogueur de grammaire. Par exemple, lorsque l'expression `h1 et non b et RE((a+b)<5)ou FE[25s/X1 ou X2 ]` est prise en entrée, *ANTLRWorks* produit l'arbre de dérivation de la figure 3.8.a.

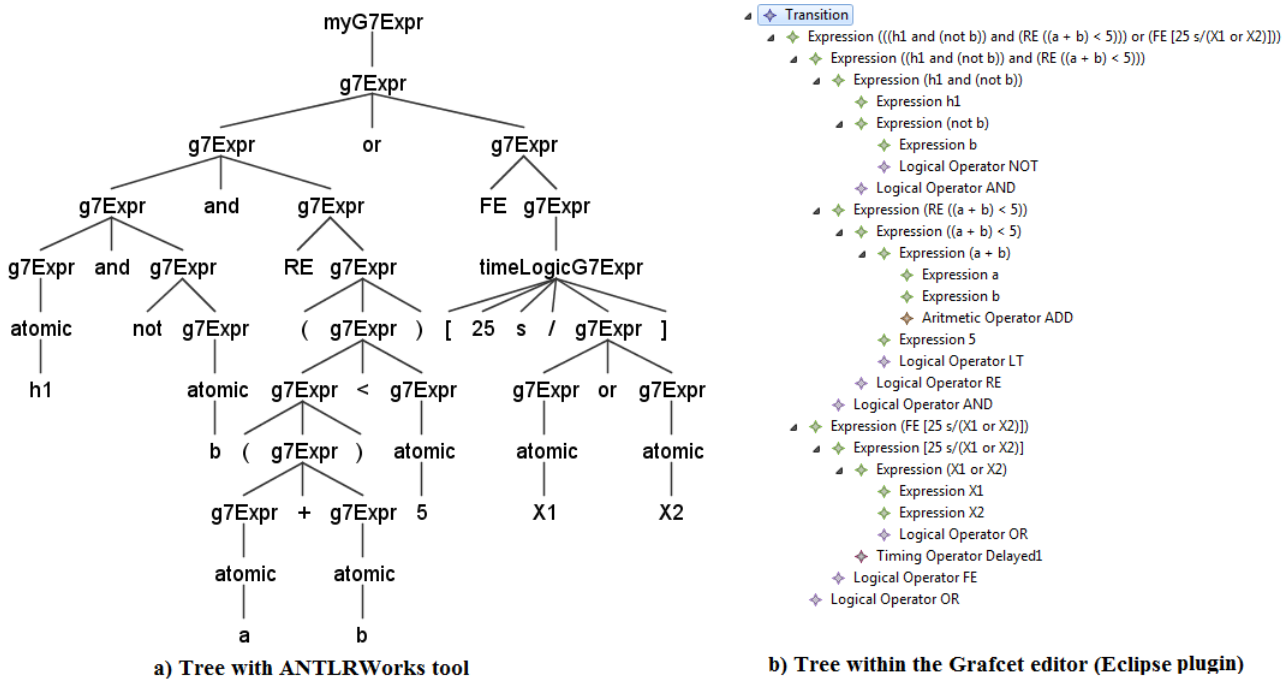


FIG. 3.8: Arbre de dérivation pour une expression logique complexe

Pour implémenter le parseur, nous avons utilisé du code généré en langage java car nous souhaitons l'intégrer à la plateforme de métamodélisation Éclipse EMF, qui utilise principalement le langage java. Cette intégration dans Eclipse EMF est présentée en 3.6.3.

3.5 prise en compte des contraintes sémantiques dans le métamodèle

Les instances Grafcet construites dans l'éditeur généré peuvent contenir des erreurs concernant la définition de la structure statique de Grafcet. Par exemple, une connexion ne doit connecter qu'une et une seule étape à une transition ou bien une et une seule transition à une étape. Lors de l'édition d'instances Grafcet, il est possible de commettre une erreur en utilisant la même connexion avec plusieurs étapes et/ou plusieurs transitions. Une instance Grafcet où cette contrainte n'est pas respectée n'est pas valide. Concernant les expressions Grafcet, on peut remarquer sur l'arbre de la figure 3.8.b (correspondant à l'expression Grafcet `h1 et non b et RE(a+b <5)ou FE[25s/X1 ou X2]`) que les variables booléennes a et b sont utilisées en tant que variables numériques dans la somme qui est opérande à la comparaison $a + b < 5$; ce qui n'est pas valide. Il en est de même des nombreuses autres vérifications à effectuer, parmi lesquelles la contrainte «*tout grafcet doit avoir au moins une étape initiale*».

Lorsque l'analyseur d'expressions a fini de construire une expression, il est donc nécessaire de la vérifier et de la valider. Parmi les vérification à faire, les expressions occupent une place prépondérante.

3.5.1 Raisons d'invalidité d'une expression Grafcet

Une expression peut être invalide pour trois raisons :

- **Alternative 1** : l'expression soumise l'analyseur d'expressions n'est pas correcte au regard de la grammaire des Expressions du Grafcet. Dans ce cas, il peut émettre des exceptions et renvoyer un résultat après avoir réalisé un retour sur erreur. La validation permet de déceler cette erreur. L'utilisateur doit alors modifier son expression pour la corriger.
- **Alternative 2** : l'expression obtenue après l'analyse est correcte mais les instances de *Expression* ne sont pas bien remplies (ou popularisées). Ce cas de figure est peu probable du fait que le parseur utilise une grammaire formelle bien implémentée. Toutefois, comme c'est lors de la visite de l'arbre abstrait obtenu après le parsing que l'on crée et remplit récursivement les champs des instances de *Expression* par les méthodes du visiteur, il est possible d'omettre le remplissage de certains champs. Cette validation permet de corriger le code du visiteur utilisé à cet effet.
- **Alternative 3** : l'expression analysée par le parseur est correcte mais les règles sémantiques des expressions ne sont pas toutes respectées. Le cas le plus ordinaire est l'utilisation d'une variable non créée à l'avance dans le modèle grafcet. En effet, lorsqu'un identificateur est rencontré dans l'expression, le programme demande au grafcet la va-

riable correspondante afin d'en faire référence. Si celle-ci n'existe pas, alors le grafcet n'est pas valide. Une exception est levée et traitée, mais la variable voulue n'est pas encore créée et référencée. Une contrainte dynamique doit être ajoutée au modèle *Expression* pour attirer l'attention de l'utilisateur à ce sujet.

Un moyen de vérification et de validation des instances Grafcet doit être mis en place. Pour cela, nous énonçons les contraintes sémantiques pour une instance entière de Grafcet et nous les formalisons en utilisant le langage OCL. Les contraintes OCL seront ensuite intégrées au métamodèle pour la validation dynamiques.

3.5.2 Définition des contraintes sémantiques

Les contraintes relatives à la construction d'instances Grafcet sont présentées dans le tableau 3.2. Ces contraintes proviennent de l'analyse de la définition statique du Grafcet et de la modélisation proposée pour prendre en compte les expressions Grafcet. Dans ce tableau, le contexte est la classe dans laquelle la contrainte intervient, alors que le nom de la contrainte est l'identificateur correspondant utilisé. Celles relatives au contexte *Expression* sont données dans le tableau 3.3.

TAB. 3.2: Contraintes générales de sémantique dynamique dans les classes Grafcet

Context	**Formalisation de la contrainte**	**Nom de la constraint**
Grafcet	Un grafcet a au moins une étape initiale	*hasAtLeastOneInitialStep*
Grafcet	Deux variables différentes ne peuvent avoir le même nom	*uniqueNamesForG7Vars*
Grafcet	Une instance de *StepToTransition* ne peut lier qu'une étape à une transition : une seule étape entrante	*validStepToTransition_StepSide*
Grafcet	Une instance de *StepToTransition* ne peut lier qu'une étape à une transition : une seule transition sortante	*validStepToTransition_TransitionSide*
Grafcet	Une instance de *TransitionToStep* ne peut lier qu'une transition à une étape : une seule étape sortante	*validTransitionToStep_StepSide*
Grafcet	Une instance de *TransitionToStep* ne peut lier qu'une transition à une étape : une seule transition entrante	*validTransitionToStep_TransitionSide*
Transition	Une transition doit avoir au moins une connexion entrante et au moins une connexion sortante	*validTransition*
Step	Toute variable associée à une étape (variable d'activité d'étape) est une variable interne	*stepVarIsInternalVar*
LevelAction	Toute variable associée à une action à niveau doit être de type BooleanVariable	*levelActionVarIsBoolVar*
storedAction	Une action stockée (numérique ou booléenne) ne peut concerner qu'une expression stockée ou bien une fonction	*storedActionIsEitherFunctionOrStore*

TAB. 3.3: Les contraintes de sémantique dynamique dans le contexte *Expression*

Expression de la de la contrainte	**Nom de la contrainte**
Quand une expression est simple et non constant, la référence de la variable concernée doit être non nulle	*variableExistsInGrafcet*
Si une expression est simple et se réduit à une variable, alors elle est du type de cette variable (*Arithmetic* ou *Logical*)	*SimpleNonConstExprIsOfVariableType*
Toute expression constant est nécessairement simple et ne reference aucune variable	*AValidConstantExpression*
Toute expression simple ne doit pas avoir de sous-expressions	*SimpleExpressionHasNoSubExpressions*
Toute expression modélisant un opérateur unaire ne doit pas avoir de sous-expression gauche ($subExpr1 = null$), mais doit avoir une sous-expression droite ($subExpr2 \neq null$)	*ValidUnaryOperationExpression*
Quand une expression modélise un opérateur, alors elle ne doit référencer aucune variable	*ValidExpressionWithOperator*
Quand une expression concerne un opérateur temporal, alors son type est *Logical*	*ValidExprWithTimeOp*
Étant donnée une expression, lorsque ses deux sous-expressions sont non nuls, alors elles sont nécessairement du même type parce qu'elle modélise une composition par un opérateur binaire	*ValidExpressionWithBinaryOperation*

3.5.3 Formalisation des contraintes avec OCL

Le langage OCL (Object Constraint Language) est le langage de requête utilisé dans QVT depuis 2002 pour décrire les exigences d'un langage standard pour la spécification de requêtes, de vues et de transformations de modèles [39]. Sa définition a commencé avec l'objectif de surmonter les limitations d'UML. Depuis lors, OCL est devenu un composant clé de toute technique d'ingénierie dirigée par modèle (MDE) en tant que langage par défaut pour l'expression de toutes sortes de requêtes sur les modèles/métamodèles, de spécifications et de manipulations [11]. OCL est également fréquemment utilisé pour exprimer des transformations de modèle, des règles bien formées[11] (dans le cadre de la définition de nouveaux DSLs) ou des modèles de génération de code (en tant que moyen pour exprimer les patterns de génération et les règles). Utilisés auparavant dans la création de propriétés dérivées des positions relatives entre les étapes et les transitions (section 3.3.7), nous utilisons également OCL pour formaliser les contraintes au sein d'instances Grafcet.

3.5.3.1 Formalisation OCL des contraintes générales du Grafcet

Les contraintes générales énoncées dans le tableau 3.2 sont formalisées et présentées comme suit :

Listing 3.7: Contrainte hasAtLeastOneInitialStep (Grafcet)

```
context Grafcet invariant hasAtLeastOneInitialStep :
       self.steps->select(s|s.isInitial)->size()>=1;
```

Listing 3.8: Contrainte uniqueNamesInVars (Grafcet)

```
context Grafcet invariant uniqueNamesInVars:
       self.variables->forAll(v1,v2| v1<>v2 implies v1.name<>v2.name)
           ;
```

Listing 3.9: Contrainte validStepToTransition_StepSide (Grafcet)

```
context Grafcet invariant validStepToTransition_StepSide :
       self.connections->select(c|c.oclIsTypeOf(StepToTransition))->
           forAll(con|self.steps->select(s|s.outConnections->includes
           (con))->size()=1);
```

Listing 3.10: Contrainte validStepToTransition_TransitionSide (Grafcet)

11. *well-formedness rules*

```
context Grafcet invariant validStepToTransition_TransitionSide :
       self.connections->select(c|c.oclIsTypeOf(StepToTransition))->
           forAll(con|self.transitions->select(t|t.inConnections->
           includes(con))->size()=1);
```

Listing 3.11: Contrainte validTransitionToStep_TransitionSide (Grafcet)

```
context Grafcet invariant validTransitionToStep_TransitionSide :
       self.connections->select(c|c.oclIsTypeOf(TransitionToStep))->
           forAll(con|self.transitions->select(t|t.outConnections->
           includes(con))->size()=1);
```

Listing 3.12: Contrainte validTransitionToStep_StepSide (Grafcet)

```
context Grafcet invariant validTransitionToStep_StepSide :
       self.connections->select(c|c.oclIsTypeOf(TransitionToStep))->
           forAll(con|self.steps->select(s|s.inConnections->includes(
           con))->size()=1);
```

Listing 3.13: Contrainte validTransition (Transition)

```
context Transition invariant validTransition :
       self.inConnections->size()>=1 and self.outConnections->size()
           >=1;
```

Listing 3.14: Contrainte stepVarIsInternalVar (Step)

```
context Step invariant stepVarIsInternalVar:
       self.stepVariable.type = VarType::Internal;
```

Listing 3.15: Contrainte levelActionVarIsBoolVar (LevelAction)

```
context LevelAction invariant levelActionVarIsBoolVar:
       self.actionVariable.oclIsTypeOf(BooleanVariable);
```

Listing 3.16: Contrainte storedActionIsEitherFunctionOrStore (StoredAction)

```
context StoredAction invariant storedActionIsEitherFunctionOrStore:
       not (self.storedExpression <> null and self.function <> null);
```

3.5.3.2 Formalisation OCL des contraintes sur les expressions

Les contraintes concernant les expressions Grafcet énoncées dans le tableau 3.3 sont formalisées en OCL comme suit :

Listing 3.17: Contrainte variableExistsInGrafcet (Expression)

```
context Expression invariant variableExistsInGrafcet:
       (self.isSimple and not self.isConstant) implies self.variable
           <>null;
```

Listing 3.18: Contrainte SimpleNonConstExprIsOfVariableType (Expression)

```
context Expression invariant SimpleNonConstExprIsOfVariableType:
       (self.isSimple and not self.isConstant and self.variable<>null
           ) implies(
       (self.variable.oclIsTypeOf(BooleanVariable) implies self.type
           = ExprType::Logical)
              or
       (self.variable.oclIsTypeOf(NumericVariable) implies self.type
           =ExprType::Arithmetic) );
```

Listing 3.19: Contrainte AValidConstantExpression (Expression)

```
context Expression invariant AValidConstantExpression:
       self.isConstant implies (self.isSimple and self.variable =
           null);
```

Listing 3.20: Contrainte SimpleExpressionHasNoSubExpressions (Expression)

```
context Expression invariant SimpleExpressionHasNoSubExpressions:
       self.isSimple implies (self.subExpr1 = null and self.subExpr2
           = null);
```

Listing 3.21: Contrainte ValidUnaryOperationExpression (Expression)

```
context Expression invariant ValidUnaryOperationExpression:
       (self.operator.oclIsTypeOf(TimingOperator)
       or (
              self.operator.oclIsTypeOf(LogicalOperator)
              and ( ((self.operator.oclAsType(LogicalOperator)).name
                  = LogicOpValues::NOT)
              or ((self.operator.oclAsType(LogicalOperator)).name =
                  LogicOpValues::RE)
```

```
                or ((self.operator.oclAsType(LogicalOperator)).name =
                    LogicOpValues::FE)
                ))
        or (self.operator.oclIsTypeOf(AritmeticOperator) and (self.
            operator.oclAsType(AritmeticOperator)).name =
            ArithmOpValues::U_MINUS)
) implies (self.subExpr1=null and self.subExpr2<>null);
```

Listing 3.22: Contrainte ValidExpressionWithOperator (Expression)

```
context Expression invariant ValidExpressionWithOperator:
       (self.operator<>null) implies (self.variable = null);
```

Listing 3.23: Contrainte ValidExprWithTimeOp (Expression)

```
context Expression invariant ValidExprWithTimeOp:
       self.operator.oclIsTypeOf(TimingOperator) implies self.type =
           ExprType::Logical;
```

Listing 3.24: Contrainte SimpleNonConstExprIsOfVariableType (Expression)

```
context Expression invariant SimpleNonConstExprIsOfVariableType:
       (self.isSimple and not self.isConstant and self.variable<>null
           ) implies
       ((self.variable.oclIsTypeOf(BooleanVariable) implies self.type
            = ExprType::Logical)
              or
       (self.variable.oclIsTypeOf(NumericVariable) implies self.type
           = ExprType::Arithmetic));
```

Listing 3.25: Contrainte ValidExpressionWithBinaryOperation (Expression)

```
context Expression invariant ValidExpressionWithBinaryOperation:
       (self.operator<>null and self.subExpr1<>null and self.subExpr2
           <>null) implies (self.subExpr1.type = self.subExpr2.type);
```

Pour faciliter la localisation d'une erreur liées aux opérateurs lors de la validation des expressions, des contraintes spécifiques sont utilisées. En Annexe 5.7, on trouvera les règles OCL relatives aux opérateurs de comparaison, aux opérateurs logiques et aux opérateurs arithmétiques.

3.6 Implémentation dans Éclipse EMF

Le métamodèle Grafcet proposé a été mis en œuvre dans *Eclipse Modeling Framework* (EMF). En effet, EMF est devenu une référence clé dans le domaine du développement de logiciels piloté par les modèles. Il s'agit d'un framework permettant de décrire les modèles de classe et de générer du code Java permettant de créer, modifier, sauvegarder et restaurer des instances de modèles. De plus, il fournit des générateurs pour supporter l'édition et la validation des modèles EMF. EMF unifie trois technologies importantes ($Java$, XML et UML) pour créer de meilleurs outils logiciels intégrés [42, 62]. EMF se compose de trois éléments fondamentaux: EMF (le noyau EMF comprenant un métamodèle *Ecore* pour décrire les modèles et le support d'exécution des modèles), $EMF.Edit$ (y compris les classes génériques réutilisables pour construire des éditeurs de modèles EMF) et $EMF.Codegen$ (décrivant la fonction de génération de code) [42] qui est une fonction de génération de code Java qui offre la possibilité à l'utilisateur de se concentrer sur le modèle lui-même et non sur les détails de son implémentation.

3.6.1 Implémentation du métamodèle dans Eclipse EMF

Nous avons installé Eclipse EMF pour implémenter le métamodèle Grafcet. Son résultat est obtenu dans un fichier *.ecore*. Le fichier *.aird* de ce métamodèle a aussi été créé pour permettre la construction graphique de ses éléments. A l'aide de l'éditeur $OCLinEcore$, une autre vue du même métamodèle est donnée textuellement. Les modifications effectuées dans cette vue impactent automatiquement le métamodèle dans les autres vues. Tous les éléments Grafcet sont des instances du méta-métamodèle Ecore. Ce sont principalement: $EPackage$, $EClass$, $EAttribute$, $EOperation$, $EEnumeration$, $EReference$ avec les cart=dinalités inférieures et supérieures. Les attributs des classes sont principalement de type $EString$, $EInt$ et $EBoolean$. Pour certaines associations, la propriété composée est vérifiée pour permettre l'instanciation de la composition et des objets; la propriété dérivée est cochée pour permettre la dérivation des fonctionnalités connexes, à l'instar des instances d'expressions. Une vue de ce métamodèle dans Éclipse EMF est données en figure 3.9.

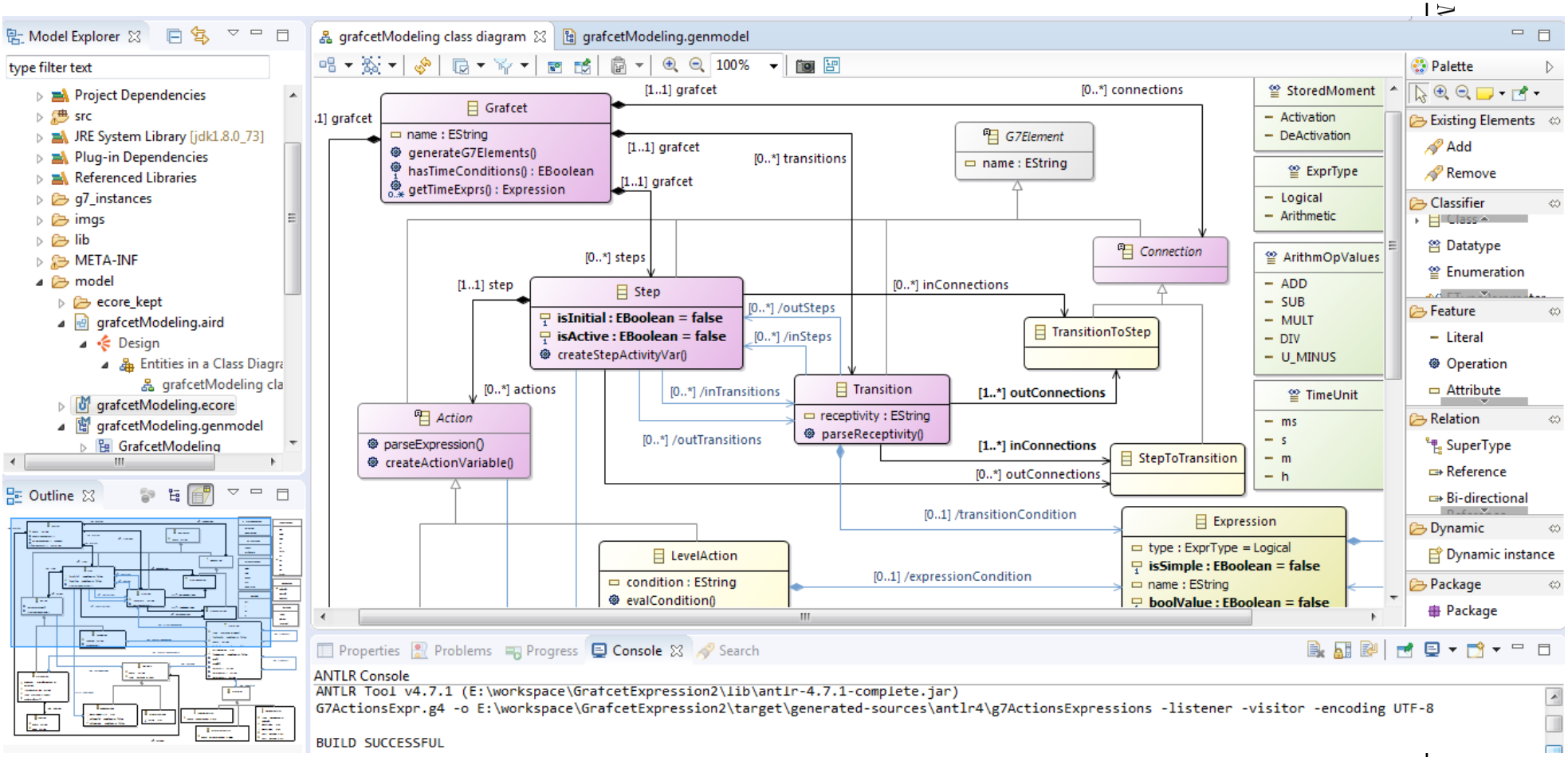


FIG. 3.9: Métamodèle Grafcet dans Eclipse EMF

3.6.2 Intégration des contraintes OCL dans le métamodèle

Les contraintes OCL dont la formalisation a été présentée en section 3.5 ont été intégrées au métamodèle dans Éclipse EMF via l'éditeur *OCLinEcore*. Nous avons utilisé Éclipse OCL, un projet axé sur la mise en œuvre du standard OCL dans Éclipse. Une étape préliminaire de cette intégration a été les vérifications effectuées sur chacune de nos contraintes OCL, afin de s'assurer qu'elles étaient syntaxiquement correctes et fournissaient le résultat attendu. Pour cela, nous avons utilisé la console interactive OCL, qui est un outil du projet Éclipse OCL.

La figure 3.10 présente une vue d'ensemble de la classe Expression après l'intégration des contraintes OCL associées, tandis que le code 3.26 illustre le contenu de la classe Grafcet dans l'éditeur *OCLinEcore*.

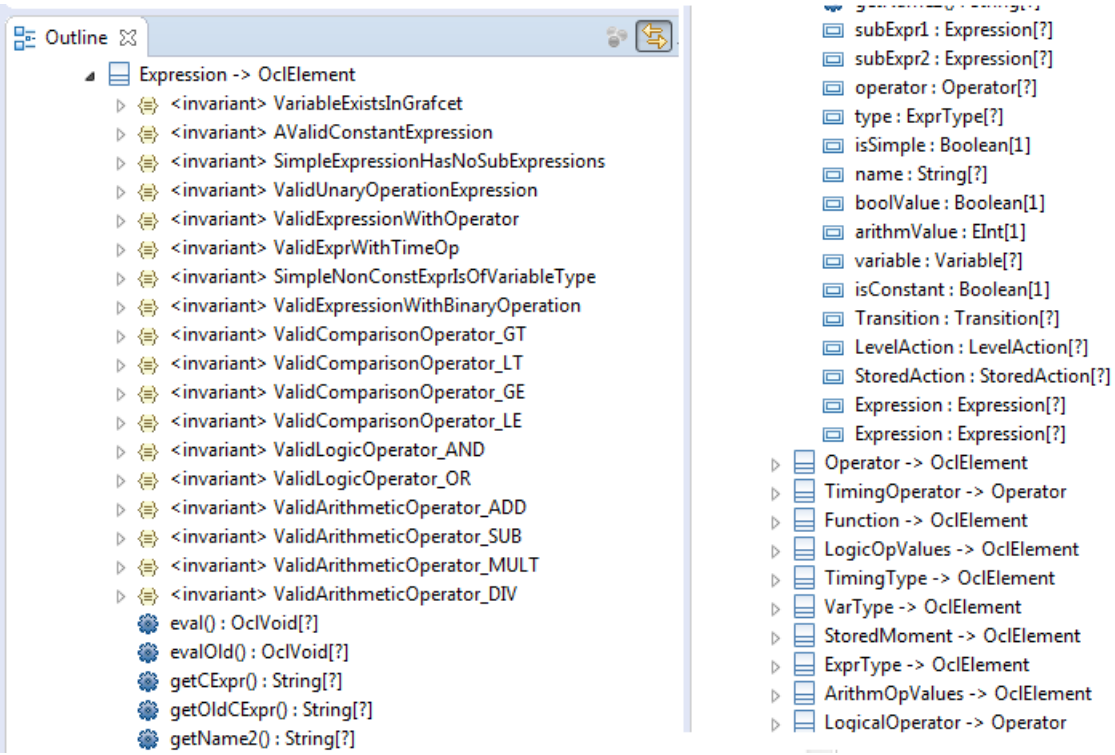


FIG. 3.10: Vue d'ensemble de la classe Expression avec contraintes OCL

Listing 3.26: Code de la classe Grafcet dans l'éditeur OCLinEcore

```
9         {
10                operation generateG7Elements();
11                operation hasTimeConditions() : Boolean[1];
12                operation getTimeExprs() : Expression[*|1] { ordered };
13                attribute name : String[?];
14                property connections#grafcet : Connection[*|1] {
                      ordered composes };
```

```
15              property transitions#grafcet : Transition[*|1] {
                    ordered composes };
16              property steps#grafcet : Step[*|1] { ordered composes
                    };
17              property variables#grafcet : Variable[*|1] { ordered
                    composes };
18              invariant uniqueNamesForG7Vars:
19              self.variables->forAll(v1,v2| v1<>v2 implies v1.name<>
                    v2.name);
20              invariant hasAtLeastOneInitialStep:
21              self.steps->select(s|s.isInitial)->size()>=1;
22              invariant
23              validStepToTransition_StepSide:
24              self.connections->select(c|c.oclIsTypeOf(
                    StepToTransition))->forAll(con|self.steps->select(s
                    |s.outConnections->includes(con))->size()=1);
25              invariant
26              validStepToTransition_TransitionSide:
27              self.connections->select(c|c.oclIsTypeOf(
                    StepToTransition))->forAll(con|self.transitions->
                    select(t|t.inConnections->includes(con))->size()=1)
                    ;
28              invariant
29              validTransitionToStep_TransitionSide:
30              self.connections->select(c|c.oclIsTypeOf(
                    TransitionToStep))->forAll(con|self.transitions->
                    select(t|t.outConnections->includes(con))->size()
                    =1);
31              invariant
32              validTransitionToStep_StepSide:
33              self.connections->select(c|c.oclIsTypeOf(
                    TransitionToStep))->forAll(con|self.steps->select(s
                    |s.inConnections->includes(con))->size()=1);
34          }
```

3.6.3 Intégration du parseur au métamodèle Grafcet

Les codes des analyseurs lexical et syntaxique des expressions Grafcet sont générés dans un package que nous avons nommé $g7Expression$.

ANTLR fournit alors un support pour l'utilisation des analyseurs générés à l'aide de deux mécanismes de parcours d'arbre dans sa bibliothèque d'exécution: une interface *Listener* d'analyse d'arborescence par écoute qui répond aux événements déclenchés lors du parcours de l'arborescence et une interface $Visitor$ réalisée suivant le modèle de conception de patterns des visiteurs.

Dans ce travail, c'est l'interface $Visitor$ qui est utilisée, et dont le code de base est obtenu lors de la compilation du fichier $.g4$ par ajout de l'option $-visitor$ à la commande de compilation. Une classe $g7ExpressionBaseVisitor$ qui implémente l'interface $Visitor$ est aussi générée. Nous personnalisons ici la classe $g7ExpressionBaseVisitor$ pour réaliser un outils d'exploration

de l'arbre de syntaxe abstraite généré par le parseur, à dessein de produire en sortie un objet du concept *Expression* du métamodèle Grafcet. Cette classe redéfinir les méthodes de $G7ExprVisitor$ tout en les faisant générer récursivement une instance du concept *Expression* à travers un parcours infixé. Sa vue d'ensemble est : *public class G7ExprVisitor extends g7ExpressionBaseVisitor<Expression> ...*

Ici, *Expression* est le paramètre de la classe $g7ExpressionBaseVisitor$. Le visiteur nous permet de réaliser le déplacement, en appelant explicitement des méthodes pour visiter les fils de l'arbre, tout en instanciant étape par étape des expressions et en les enrichissant des informations associées issues du contexte.

Par exemple, le listing 3.27 expose les instructions de la méthode $visitMulDiv\_G7Expr$ permettant de visiter le nœud de l'arbre de dérivation correspondant à la multiplication ou à la division. Dans cette méthode, l'expression *exp* est instanciée et retournée plus tard en résultat. Les sous-expressions sont obtenues en visitant les attribut $left$ et $right$ du contexte *ctx*, instance de $g7ExpressionParser.MulDiv\_G7ExprContext$. L'expression *exp* est alors enrichie à l'aide des informations utiles (*operator*, *type*, *isSimple*, *isConstant* et les noms d'attributs) fournies par le contexte *ctx*.

Listing 3.27: Méthode de visite des noeuds de multiplication et de division

```
43 @Override
44 public Expression visitMulDiv_G7Expr(g7ExpressionParser.
       MulDiv_G7ExprContext ctx) {
45     String exp_str = afficher("("+ctx.left.getText()+" "+ctx.op.
           getText()+" "+ctx.g7Expr(1).getText()+")");
46     Expression exp = g7Factory.createExpression();
47     Expression expLeft = visit(ctx.left);
48     Expression expRight = visit(ctx.right);
49     exp.setSubExpr1(expLeft);
50     exp.setSubExpr2(expRight);
51
52     AritmeticOperator op = g7Factory.createAritmeticOperator();
53     switch (ctx.op.getType()) {
54            case g7ExpressionParser.MULT:
55                   op.setName(ArithmOpValues.MULT);
56                   break;
57            case g7ExpressionParser.DIV:
58                   op.setName(ArithmOpValues.DIV);
59                   break;
60     }
61     exp.setOperator(op);
62     exp.setType(ExprType.ARITHMETIC);
63     exp.setIsSimple(false);
64     exp.setName(exp.getInfixTreeString());
65     exp.setIsConstant(false);
66     return exp;
67 }
```

L'interface par laquelle est appelé tour à tour l'analyseur lexical et syntaxique Grafcet pour analyser toute expression Grafcet et retourner une instance de la classe *Expression* est présentée sur le listing 3.28. Une chaine d'entrée et une instance *Grafcet* sont prises en paramètre de la méthode statique *parseBuildExprTree*. Dans un premier temps, l'analyse lexicale de la chaine d'entrée est faite avec production des unités lexicales (*tokens*) puis une analyse syntaxique est faite pour produire un arbre syntaxique, utile pour produire toute autre information issue d'un parcourt d'arbre, tel qu'une instance de la classe *Expression* du métamodèle Grafcet comme décrit plus haut.

Listing 3.28: Interface de l'analyseur des expressions Grafcet

```
11 public static Expression parseBuildExprTree(String expr_str, Grafcet
       g7) {
12         try {
13                 Expression exp_result = null;
14                 ANTLRInputStream input = new ANTLRInputStream(expr_str)
                       ;
15                 g7ExpressionLexer lexer = new g7ExpressionLexer(input);
16                 CommonTokenStream tokens = new CommonTokenStream(lexer)
                       ;
17                 g7ExpressionParser parser = new g7ExpressionParser(
                       tokens);
18                 //Start parsing
19                 ParseTree tree = parser.myG7Expr();
20                 G7ExprVisitor expVisitor = new G7ExprVisitor(g7);
21
22                 exp_result = expVisitor.visit(tree);
23                 return exp_result;
24         }
25         catch(Exception ex) {
26                 ex.printStackTrace();
27         }
28         return null;
29 }
```

Le package d'analyse syntaxique Grafcet résultant est intégré à la plateforme Éclipse EMF en ajoutant le plug-in ANTLR (*antlr-4.7.1-complete.jar*) à la configuration qui exécute l'éditeur Grafcet généré. Nous indiquons également le chemin de ce plugin à l'attribut *Bundle-ClassPath* du manifeste du projet. Lorsque l'éditeur Grafcet généré à partir du métamodèle Grafcet est lancé, les modèles Grafcet peuvent être édités à l'aide de l'éditeur généraliste ayant plusieurs vues (textuelle, arborescente *Sample Reflective Ecore*, etc.). L'appel de l'analyseur et du visiteur est associé aux événements dans l'éditeur, à l'instar de la validation de la saisie de la réceptivité d'une transition. Lorsqu'une expression Grafcet est saisie (dans une transition par exemple) et validée, elle est automatiquement analysée et l'instance *Expression* correspondante est générée. L'expression présentée à la figure

3.8.b (vue arborescente) est le résultat de l'analysée dynamique de l'expression dont l'arbre de dérivation est donné à la figure 3.8.a.

3.7 Validation et expérimentation du méta-modèle Grafcet

Après la génération du code de l'éditeur, on configure un plug-in Eclipse pour l'édition et la validation des modèles Grafcet.

3.7.1 Génération du code Java de l'éditeur Grafcet

Après la création du fichier de génération *.genmodel*, le code Java de l'éditeur est alors généré automatiquement. Certaines parties de ce code sont personnalisées et précédées de l'annotation @generated NOT, afin d'éviter leur destruction par les prochaines générations de code. C'est le cas des méthodes *setReceptivity*() et *parseReceptivity*() de la classe Transition, présentées dans le code du listing 3.29.

Listing 3.29: Extrait de dex méthodes de la classe Transition avec annotation @generated NOT

```
187  * @generated NOT
188  */
189  public void parseReceptivity() {
190         Expression tc = GrafcetExpressionParser.parseBuildExprTree(
                 receptivity, this.getGrafcet());
191         this.setTransitionCondition(tc);
192  }
193  ...
194  public void setReceptivity(String newReceptivity) {
195         //Debug
196         System.out.println("In setReceptivity with : " +
                 newReceptivity);
197         String oldReceptivity = receptivity;
198         receptivity = newReceptivity;
199         if (eNotificationRequired())
200                 eNotify(new ENotificationImpl(this, Notification.SET,
                         GrafcetModelingPackage.TRANSITION__RECEPTIVITY,
                         oldReceptivity, receptivity));
201         //Build the corresponding transition Condition
202         if(!(newReceptivity.isEmpty()|| newReceptivity == null)) {
203                 this.parseReceptivity();
204         }
205  }
```

Le code généré pour l'éditeur peut ensuite être exécuté sous forme de plug-in Éclipse après avoir défini une configuration.

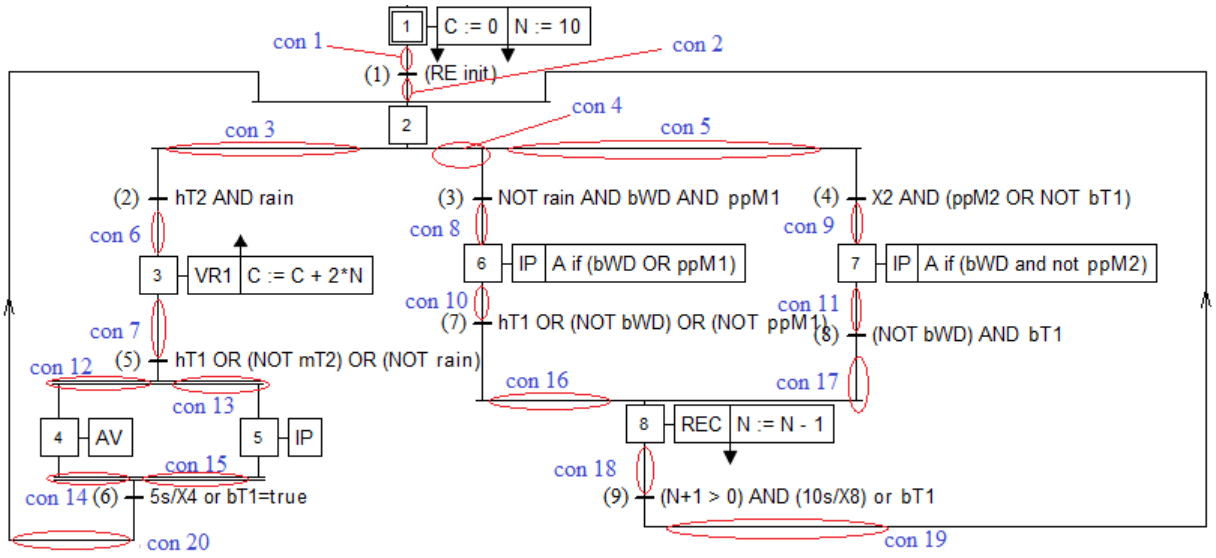


FIG. 3.11: Modèle Grafcet de l'exemple (Fig. 1.9) étiqueté

3.7.2 Édition de modèle Grafcet

Après avoir créé un projet Éclipse de modélisation à partir du métamodèle Grafcet dont la racine est $Grafcet$, le modèle encore appelé *RessourceSet* peut être édité. Plusieurs vues de l'éditeur sont présentées, parmi lesquelles une vue arborescente(*Sample Reflective Ecore Model Editor*) et une vue textuelle. Par défaut, c'est la première qui est utilisé.

En prenant l'exemple du modèle Grafcet présenté à la figure 1.9, nous fournissons un modèle Grafcet correspondant avec des liens connectés étiquetés (ou labellisés), tel qu'illustre la figure 3.11. L'étiquetage permet de distinguer les connexions les unes des autres.

Il en découle que ce grafcet exemple compte au total 20 connexions, 10 connexions de type *StepToTransition* et 10 connexions de type *TransitionToStep*. Les étiquettes peuvent servir comme noms des connexions.

Une instance du métamodèle Grafcet correspondant à l'exemple précédent est ensuite créée dans cet éditeur de modèles Grafcet, en utilisant l'éditeur spécifique *Sample Reflective Ecore Model Editor* associé au plug-in. On peut alors créer de nouveaux nœuds que sont les étapes, les transitions, les variables, les connections, les actions associées aux étapes, ... puis indiquer les différentes références entre les objets. Une vue de ce modèle dans l'éditeur arborescent est donnée en figure 3.12.

Certains aspects du modèle sont présentés en détail sur la figure 3.13. On y voit que que l'étape 6 possède deux actions à niveau IP et A. La condition associée à l'action à niveau conditionnelle IP est exprimée par l'expression *true*, laquelle est toujours satisfaite. Par contre, l'action à niveau A a pour condition *bWD ou ppM1*, pour laquelle une expression appropriée est dérivée. Les transitions et leurs réceptivités exprimées en termes d'expressions sont également présentées.

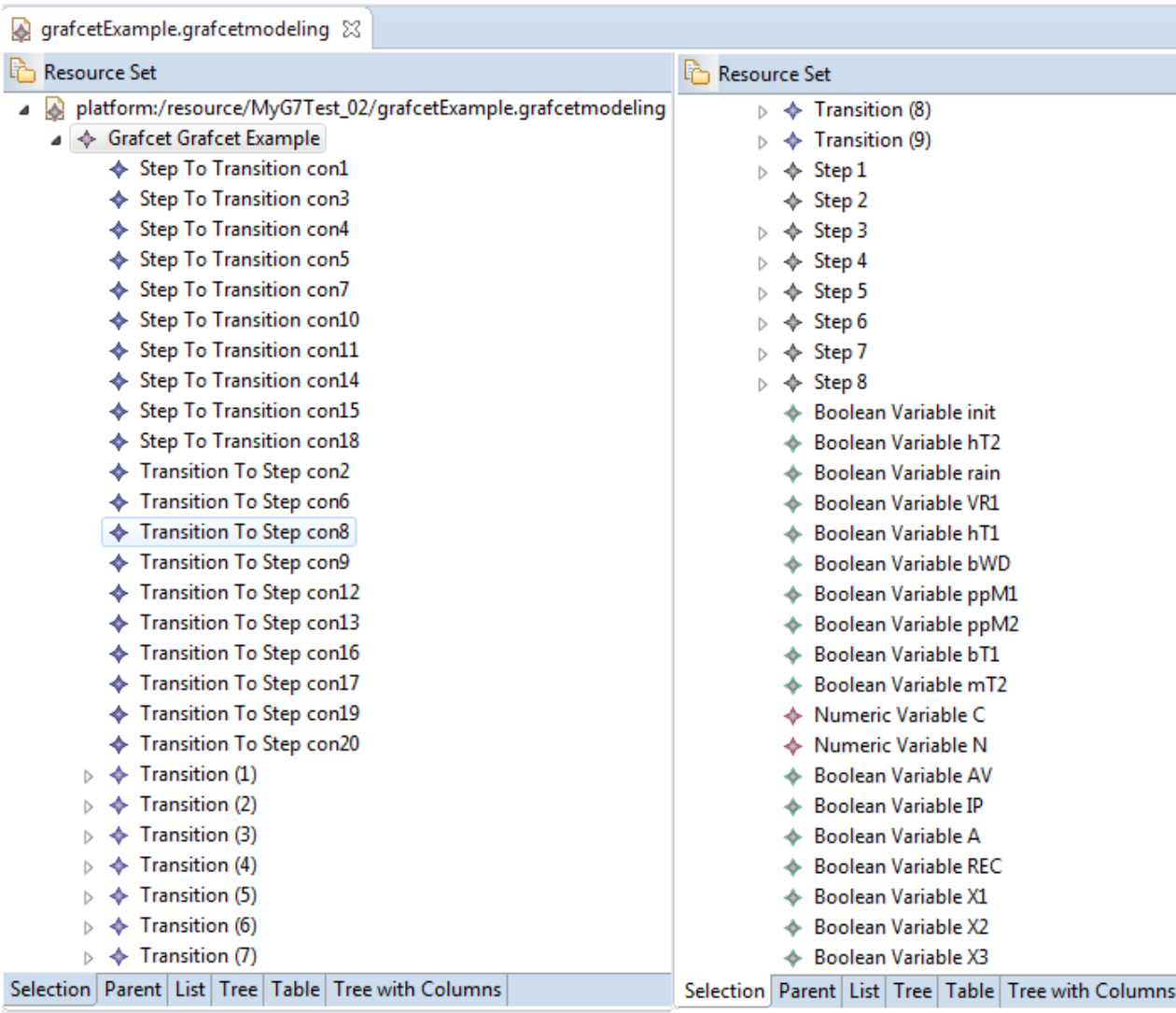


FIG. 3.12: Exemple de modèle Grafcet dans l'éditeur arborescent de modèles *Sample Reflective Ecore*

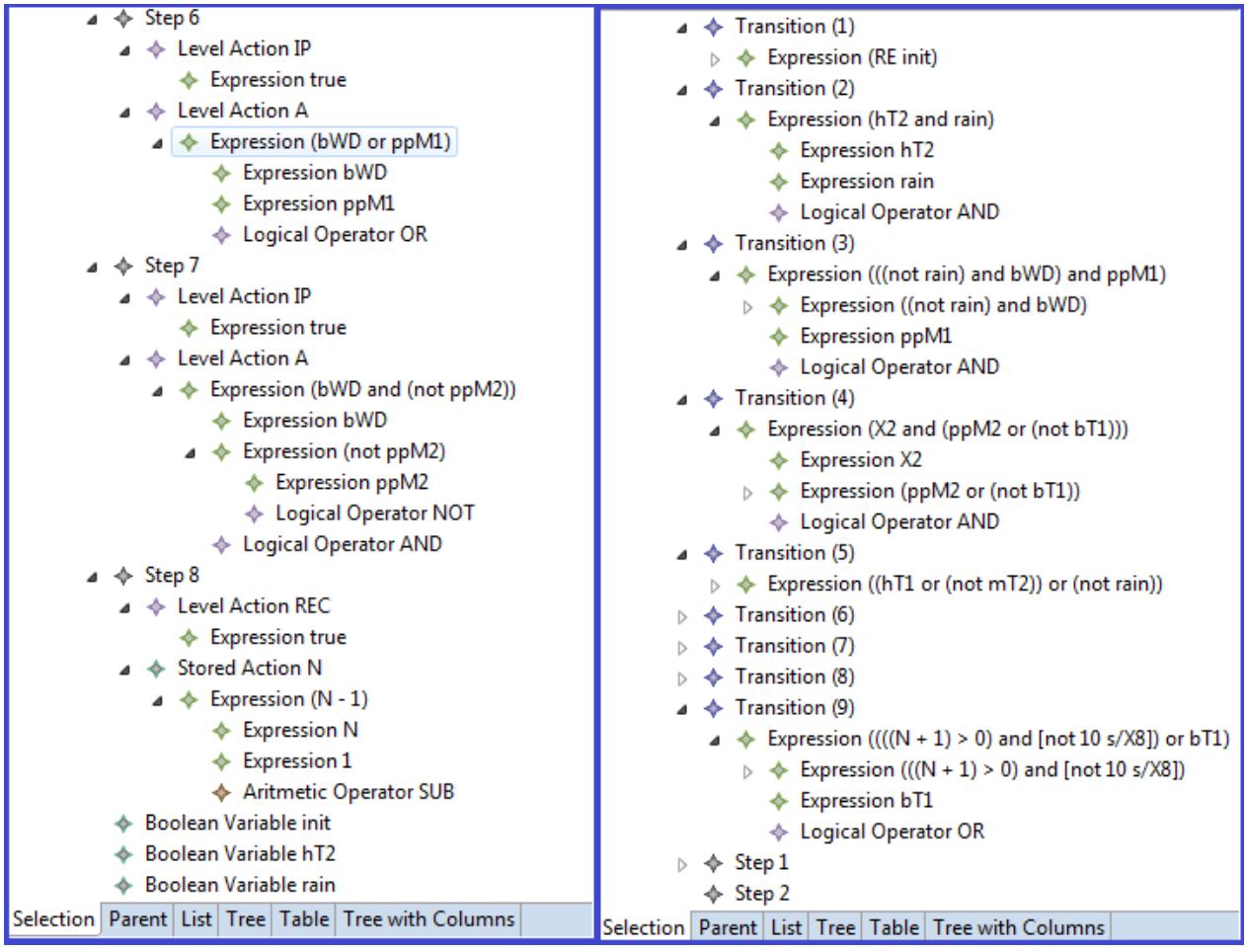


FIG. 3.13: Certains éléments détaillés du modèle *Exemple*

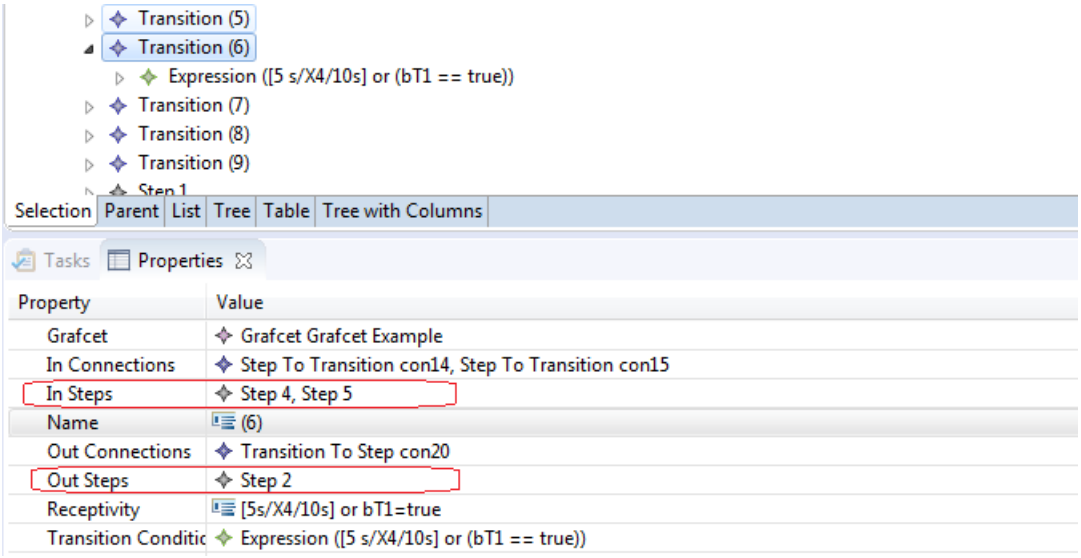


FIG. 3.14: Étapes en entrées et en sortie de la transition (6)

On remarque que de façon automatique, les caractéristiques dérivées sont automatiquement calculées. Il s'agit des expressions et des variables d'étape ($X1$, $X2$,..., $X8$), observables sur la figure 3.16. De même, les positions relatives entre étapes et transitions sont calculées automatiquement grâce aux expressions OCL intégrées au métamodèle. Ainsi, les requêtes OCL présentées en section 3.3.7 permet de calculer les position relatives comme le présentent les exemples suivants:

- **Étapes en entrée et en sortie des transitions :** la figure 3.14 présente les étapes en entrée et en sortie de la transition (6). Elle montre que les étapes 4 et 5 sont en entrée de cette transition (6), tandis que l'étape 2 est située en sortie de cette transition.
- **Transitions en entrée et en sortie des étapes :** la figure 3.15 présente l'étape 2 avec les transitions en entrée ((6), (1) et (9)) et les transitions en sortie ((3), (2) et (4)).

3.7.3 Validation de modèle Grafcet

L'éditeur de modèles utilisé dans l'environnement Éclipse EMF garantit le respect des règles statiques ou structurelles définies au niveau du métamodèle Grafcet. Quant aux règles dynamiques, elles peuvent être vérifiées en sélectionnant une instance de modèle et en demandant l'exécution du processus de validation.

En cas de sélection d'une instance de Grafcet, toutes les règles énoncées en son sein et dans tous les objets contenus sont vérifiées. Lorsqu'aucune règle n'est violée, le processus se termine avec succès comme présenté à la figure 3.16. Ceci permet de comprendre l'efficacité des règles de vérification de modèles dans un environnement IDM.

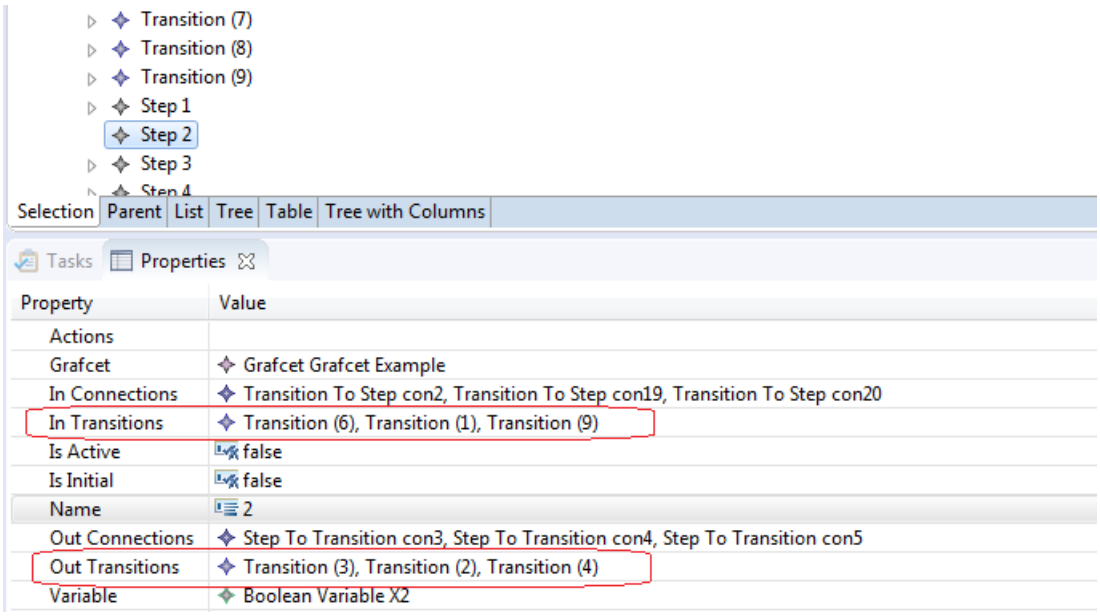


FIG. 3.15: Transitions en entrée et en sortie de l'étape 2

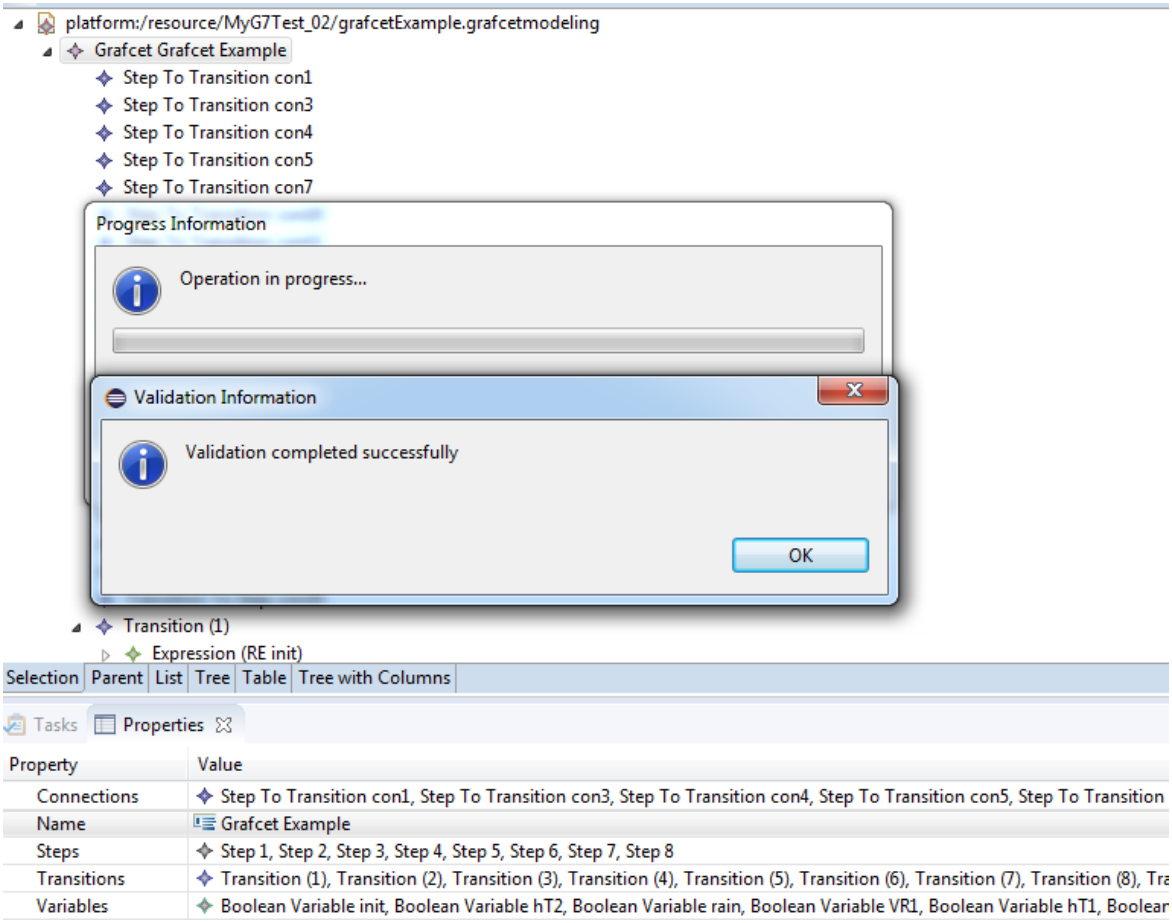


FIG. 3.16: Résultat du processus de validation de modèle (Succès)

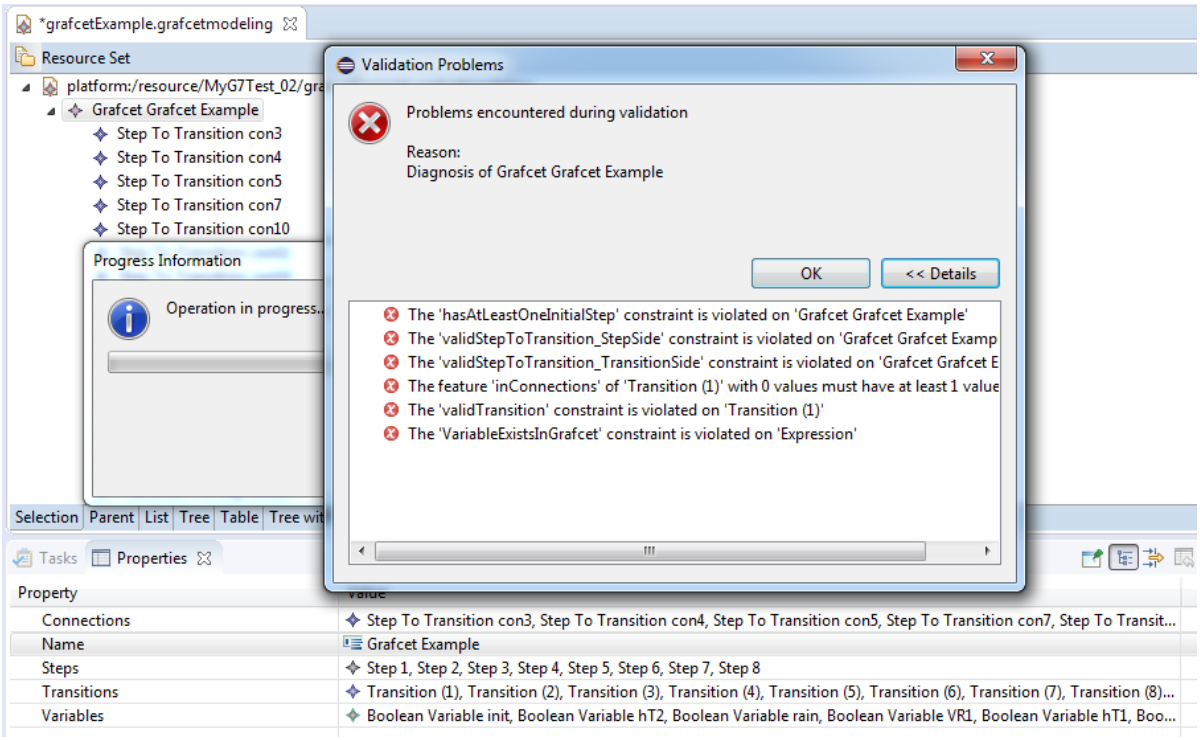


FIG. 3.17: Résultat du processus de validation de modèle (Échec)

Nous introduisons quelques erreurs dans ce modèle Grafcet comme suit tout en indiquant les règles OCL supposées être violées:

- Changement de l'étape initiale 1 en étape non initiale: ce modèle n'a plus d'étape initiale. L'objectif est de violer la règle *hasAtLeastOneInitialStep*.
- Modification de la réceptivité de la transition (9) en utilisant une variable inconnue *unknownVA* comme suit: `(N+1)>0)and [not 10s/X8 ] or unknownVA`. Il en ressort que la règle `variableExistsInGrafcet` doit être violée, puisque la variable *unknownVA* est inexistante.
- Suppression de la connexion *con*1 (de type *StepToTransition*); toutes ses références sont aussi perdues. Pour la transition (1), *con*1 est référencé dans la liste des connexions entrantes *inConnections*, tandis que pour l'étape 1, *con*1 est référencé dans la liste des connexions sortantes *outConnections*. Cela entraine la violation de quatre autres règles présentées ci-après parmi lesquelles une statique et trois dynamiques:
  - La liste *inConnections* (contenant des *StepToTransition*) est vide (contient 0 élément): **c'est ici la règle statique violée**. Une instance de la classe *StepToTransition* existe sans les deux liens d'une étape à cette connexion et de cette connexion à une transition. Cela entraine la violation des règles suivantes:
  - `validStepToTransition_StepSide` : absence d'une connexion en sortie de l'étape 1,

- ☐ `validStepToTransition_TransitionSide`: absence d'une connexion en entrée de la transition (1).
  La violation de cette règle entraine la violation de la règle suivante :
- ☐ `validTransition` : la transition (1) est dépourvue de connections entrantes.

On note que toutes ces règles sont indépendantes, mais la violation de certaines entraine la violation d'autres.

Avec ces erreurs, le résultat du processus de validation détecte des erreurs en présentant avec précision toutes les règles violées, comme illustré à la figure 3.17. Ces erreurs doivent donc être corrigées avant que le processus de validation ne puisse être exécuté à nouveau.

Il faut noter que la vérification des modèles relève de la sémantique des langages, qui est un problème NP-complet. Cependant la solution IDM est souple et extensible pour permettre la prise en compte de nouvelles règles identifiées.

Une fois ce modèle Grafcet valide au regard de son langage exprimé par le métamodèle Grafcet et les règles dynamiques OCL, il peut servir à toute autre fin, telle que la génération du code.

3.8 Conclusion

Dans ce chapitre, après avoir donné une présentation sommaire de l'IDM, nous avons étudié le langage Grafcet au regard du standard et de l'existant pour lui définir une base d'un langage de modélisation (DSML). Après avoir identifié tous les concepts véhiculés par le standard IEC 60848, nous nous sommes appesantis sur les expressions Grafcet. Leur importance pour la vérification et la validation de modèles Grafcet nous a poussé à en proposer une formalisation en termes de grammaire hors-contexte implémentée à l'aide de ANTLR [46, 10] sous forme de grammaire attribuée . L'ensemble des concepts Grafcet et les liens entre eux ont alors été formalisés dans une syntaxe abstraite du langage définie par le métamodèle Grafcet. En dehors des contraintes statiques exprimées dans ce métamodèle, des contraintes dynamiques ont été exprimées, formalisées puis intégrées au métamodèle à l'aide du langage OCL. Ce même langage a été utilisé pour interroger tout modèle Grafcet à dessein de calculer les positions relatives entre les étapes et les transitions. Les étapes en entrée et en sortie des transitions et les transitions en entrée et en sortie des étapes sont alors calculées et référencées à l'aide de propriétés dérivées.

L'ensemble de la solution (métamodèle, règles OCL et analyseur syntaxique des expressions Grafcet) a été réalisée dans l'environnement IDM *Eclipse Modeling Framework* (EMF) pour aboutir à un environnement d'édition et de validation des modèles Grafcet. A chaque étape, le modèle Grafcet

de la figure 3.11 a servi d'illustration et de validation des modèles et techniques proposées. Cet environnement IDM offre de nombreux avantages par rapport à un environnement généraliste: la génération automatique de code Java à partir du métamodèle, le langage OCL pour exprimer des règles sur les concepts du modèle ainsi que des requêtes pour l'obtention de toute information utile, et surtout la possibilité pour le designer de se focaliser sur les modèles eux-même sans se soucier de leur persistance (garantie à tous les niveaux de modélisation par le format XMI, avec des outils de sauvegarde et de chargement qui s'exécutent en arrière-plan). L'approche IDM est donc formelle et non triviale, mais surtout permet de gagner en temps et de construire des modèles sûrs.

Par rapport à notre objectif premier, en plus du fait que l'approche IDM soit complètement formelle, ces nombreux avantages font qu'il n'est pas nécessaire de passer par le codage matriciel du Grafcet (approche proposée dans le chapitre précédent) pour arriver à une mise en œuvre du modèle. Une fois un modèle Grafcet valide au regard de son langage, il peut faire l'objet de transformations pour la génération d'artefacts logiciels à l'instar du code de contrôle.

Chapitre Quatre

Génération de code Grafcet multi-cibles par approche IDM pour microcontrôleurs C-programmés

Sommaire

4.1 Introduction

Après avoir défini au chapitre précédent un langage de domaine spécifique pour la modélisation Grafcet, nous nous intéressons à présent à la génération de code. Nous focalisons notre attention sur les microcontrôleurs C-programmés pour lesquels un langage de description des caractéristiques utiles pour la génération de code est proposé. En se servant du DSML Grafcet et celui des microcontrôleurs, nous étudierons et proposerons des règles de transformation en code de contrôle dédié au microcontrôleur choisi en entrée. L'objectif est de laisser au compilateur spécifique C développé par le fabriquant la charge de la transformation du code généré en code final exécutable sur la cible. Il est donc nécessaire de commencer par une présentation de la transformation de modèle dans un contexte IDM.

4.2 La transformation de modèles

Outre la métamodelisation (déjà présentée en section 3.2), la transformation de modèle est également une opération centrale de l'IDM.

4.2.1 Guide MDA et la transformation de modèle

Le guide *MDA* définit une transformation de modèle comme étant «*le processus de conversion d'un modèle en un autre modèle du même système*». Justement, Kleppe et al. le définissent comme *la génération automatique d'un modèle cible à partir d'un modèle source, conformément à une définition de transformation* [34]. Une définition de transformation est vue comme un ensemble de règles de transformation qui décrivent ensemble la manière dont un modèle dans le langage source peut être transformé en un modèle dans le langage cible. Une **règle de transformation** étant une description de la façon dont une ou plusieurs constructions du langage source peuvent être transformées en une ou plusieurs constructions du langage cible. La figure 4.1 résume la transformation de modèle en soulignant le fait qu'un programme de transformation de modèle est conforme à des règles de transformation, et prend en entrée un modèle conforme à un métamodèle source donné pour produire en sortie un autre modèle conforme à un métamodèle cible [23].

4.2.2 Types de transformations de modèles

Deux principaux types de transformations ont tendance à être pris en compte dans l'approche IDM. D'une part, il y a **les transformations de modèle en texte (M2T)** qui génèrent ou produisent des artefacts logiciels à partir de modèles. Ces artefacts sont généralement du code source, du XML et d'autres fichiers texte. La technique la plus courante pour cette classe de transformations est connue sous le nom de génération de code.

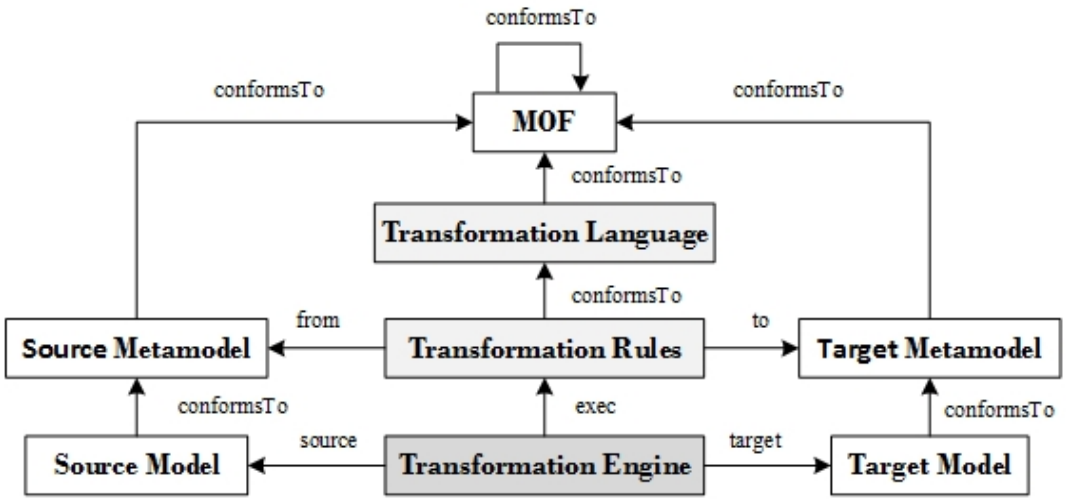


FIG. 4.1: Contexte de transformation de modèle en IDM

Pour cela, il existe de nombreuses solutions et techniques, telles que décrites dans [68, 17]. D'autre part, on distingue **les transformations de modèle à modèle (M2M)** permettent de convertir les modèles en un autre ensemble de modèles, généralement plus proches du domaine de la solution ou répondant aux besoins spécifiques des différentes parties prenantes.

4.2.3 Langages de transformation de modèles

Concernant les langages de transformation, l'OMG [1] a lancé un appel à propositions publié en 2002, la RFP QVT [2] RFP [29], décrivant les exigences d'un langage standard pour la spécification de requêtes, de vues et de transformations de modèles, conformément aux définitions suivantes:

- Une requête (***Query***) est une expression évaluée à travers un modèle. Le résultat d'une requête est une ou plusieurs instances de types définis dans le modèle source ou définis par le langage de requête. OCL (v 2.0)[11] est le langage de requête utilisé dans QVT;
- Une vue (***View***) est un modèle entièrement dérivé d'un modèle de base. Une vue ne peut pas être modifiée séparément du modèle dont elle est dérivée et des modifications apportées au modèle de base entraînent des modifications correspondantes dans la vue;
- Une ***transformation*** génère un modèle cible à partir d'un modèle source.

Même si une spécification finale QVT a été adoptée à la fin de 2005, le domaine de la transformation de modèle continue d'être un sujet de recherche

1. L'Object Management Group est un consortium américain à but non lucratif créé en 1989 dont l'objectif est de standardiser et promouvoir le modèle objet sous toutes ses formes.

2. Query View Transformation

intense [67, 24, 68]. Au cours des dernières années, parallèlement au processus OMG, un certain nombre d'approches de transformation de modèles ont été proposées par le monde universitaire et l'industrie. Les paradigmes, les constructions, les approches de modélisation et les outils qui supportent la transformation de modèle distinguent les propositions, chacune d'elles convenant à un certain ensemble de problèmes.

Les transformations sont spécifiées par le biais de langages distincts, tels que les langages de programmation classiques, mais également et surtout par des langages de transformation de modèles spécialisés, à des fins différentes et selon différents paradigmes de modélisation tels que QVT, *Acceleo*, ATL, $VIATRA$, $DSLTrans$ [17].

4.2.4 Importance des modèles dans les transformations

Les modèles sont un concept central de l'approche IDM. D'une part, un modèle peut être créé directement par des utilisateurs (concepteurs de modèles) ou peut être généré automatiquement à partir de transformations de modèle à modèle. D'autre part, les modèles peuvent être utilisés pour produire des artefacts logiciels, respectivement au moyen de transformations M2T ou de la création directe par leurs utilisateurs (c'est-à-dire les développeurs de logiciels) [68, 17]. De plus, dans certaines situations particulières, les modèles peuvent être directement interprétés et exécutés par des plateformes spécifiques intégrées à l'application logicielle. Il s'avère important de souligner que pour être utilisés efficacement dans le contexte IDM, les modèles doivent être définis de manière cohérente et rigoureuse. En général, un certain niveau de qualité est requis pour que ces modèles puissent être utilisés correctement dans des scénarios M2M ou M2T. À cette fin, les outils fournissent certaines fonctionnalités telles que **l'analyse de modèle**, **la validation de modèle** et **la simulation** [17].

4.3 Définition d'un langage de description des microcontrôleurs

Nous nous concentrons ici sur la modélisation des caractéristiques du microcontrôleur. Nous ne sommes intéressés que par ses fonctionnalités nécessaires à la génération de code. Cela permet de traiter le problème de la génération multi-cibles. Les cibles peuvent être des automates programmables ou tout autre automate. Nous considérons ici le cas des microcontrôleurs. Nous limitons le domaine aux microcontrôleurs programmables dans un langage dérivé du langage C. En effet, la plupart des fabricants de ces cibles mettent à disposition le langage C pour leur programmation. Pour la synthèse du Grafcet, le code sera généré en langage C et le compilateur

spécifique développé par le fabriquant servira pour le transformer en code exécutable sur la cible.

4.3.1 Spécificités du C pour les cibles microcontrôleurs

Le langage C est certes populaire pour les contrôleurs programmables utilisés dans les SCC, mais quelques différences se présentent. Elles portent principalement sur:

- la gestion des broches du microcontrôleur pour les entrées/sorties
- la définition du timer qui jouera le rôle de montre pour déterminer les instants de détection des événements et par conséquent calculer certaines durées.

Ces différences impliquent alors une variabilité qui pousse à considérer les spécificités de chaque cible lors de la génération du code.

4.3.2 Caractérisation des microcontrôleurs

De la diversité des microcontrôleurs découle plusieurs caractéristiques à considérer lors de la synthèse des programmes utilisateurs. Les plus essentielles sont présentées dans les tableaux ci-après. Nous les décrivons en les classant en deux sous-groupes:

- **Les caractéristiques externes ou non fonctionnelles**: Ce sont celles qui n'ont aucune influence sur les programmes synthétisés, mais beaucoup plus liées aux fabricants et au commerce. On peut citer :
  - ★ Le nom du microcontrôleur
  - ★ La famille
  - ★ Le prix d'achat
  - ★ **Taille de la mémoire programme/données** : Elle permet de garantir la persistance du programme utilisateur et des données à utiliser.
  - ★ **Taille de la RAM** : Elle est nécessaire pour se rassurer que les variables (globales ou locales) et les opérations du programme utilisateur vont résider simultanément en mémoire. En effet, face à un défaut de mémoire RAM, la carte pourrait présenter des comportements anormaux au cas où les défauts de cache ne sont plus possibles.
  - ★ **Vitesse du processeur** : Bien que les contraintes temps-réels soient négligeables dans notre cas (applications mécaniques), il est nécessaire d'avoir un minimum de vitesse processeur.

  Dans une certaine mesure, les caractéristiques telles que la taille de la mémoire programme/données, la taille de la RAM et la vitesse du processeur peuvent être des caractéristiques internes ou fonctionnelles. Cependant, nous ne les avons pas considérées ici comme telles.

– **Les caractéristiques internes ou fonctionnelles**: Ce sont celles qui imposent des choix au niveau des programmes utilisateurs synthétisés. On distingue :

★ La taille d'un mot mémoire ou des registres : Elle oriente sur la manière de lire les données en mémoire et sur l'adresse suivante où il faut lire la prochaine donnée. Éventuellement, elle oriente aussi sur le mécanisme de codage des données à transférer dans la mémoire des données du microcontrôleur à travers un port série.

★ **Le nombre des entrées/sorties accessibles** : Permet de savoir si une carte peut être utilisée pour implémenter un modèle Grafcet possédant un certain nombre d'entrées $nbIn$ et de sorties $nbOut$.

★ **La configuration des entrées et des sorties** : Il s'agit de savoir comment définir les broches en entrées ou en sorties.

★ **La lecture de l'état d'une entrée** : Pour lire la valeur du signal présent à une entrée (broche du microcontrôleur). Il s'agit d'une instruction en langage C du microcontrôleur considéré. A la base, cette lecture se fait dans un registre géré par le programme de base du microcontrôleur. L'écriture dans un tel registre est réalisée uniquement par le programme de base du microcontrôleur. En général, il est en lecture seule.

★ **L'écriture sur une sortie** : Il s'agit aussi d'une instruction du langage C du microcontrôleur considéré permettant d'écrire une valeur (logique ou numérique) sur une sortie (ou broche configurée en sortie) du microcontrôleur.

★ **La configuration des Timers** : Pour mesurer les contraintes temporelles d'évolution, on a besoin de timers dont chaque événement permet d'exécuter une interruption : on appelle un sous-programme chargé de mettre à jour les temporisateurs associés aux contraintes à vérifier. Ceci permet une attente passive en évitant une attente active : une seule horloge est donc utilisée pour gérer toutes les temporisations. L'algorithme de contrôle s'exécute alors par scrutation et non par détection d'événements (interruptions).

4.3.2.1 Les caractéristiques pour quelques microcontrôleurs

Les principaux fabricants de microcontrôleurs sont : Microchip Technologies, Motorola, XP Semiconductors, Texas Instruments et Silicon Labs. Ils fabriquent une gamme variée de microcontrôleurs et de cartes d'expérimentation à microcontrôleurs. Les tableaux 4.1 et 4.2 présentent quelques microcontrôleurs avec leurs familles, leurs fabricants, ainsi que des caractéristiques matérielles. Pour les technologies qui prévoient une mémoire de

données, cette mémoire est généralement de type EEPROM. Ainsi les données sont séparées du programme qui réside en mémoire ROM.

TAB. 4.1: Exemples de microcontrôleurs et caractéristiques (1)

N°	Microcontrôleur (MCUs)	Famille	Fabricant	Architecture	Cartes à essai	Prix $ (USD)	Processeur	RAM	Mém. Prog. (Flash)	Mém. Data	Pins Count (I/O)
1	ATmega328P [a]	Atmel AVR	Microchip [b] Tech.	8 bits	Arduino UNO / NANO/MEGA ATSTK600	2.5	20 MIPS/ 16 MHz	SRAM 2 Ko	32 Ko	1 Ko	32
2	ATmega1280 [c]	Atmel AVR	Microchip Tech.	8 bits	Arduino Mega	11	16 MHz	SRAM 8 Ko	128 Ko	4 Ko	54
3	ATTINY417 [d]	Atmel tinyAVR	Microchip Tech.	8 bits	ATTINY817-XMINI-ND	0.9	20 MIPS	256 o	4 Ko	128 o	24
4	ATxmega64A1U [e]	Atmel AVR XMEGA	Microchip Tech.	16 bits	ATmega324PB -XPRO	6	32 MIPS	4 Ko	64 Ko	2048 o	100
5	AT32UC3L016 [f]	Atmel AVR32 (L Series)	Microchip Tech.	32 bits	AT32UC3C-EK	3.5	50 Mhz	8 Ko	16 Ko		48 (36)
6	PIC12F1822 [g]	PIC12F	Microchip Tech.	8 bits	DM160228	1.2	8 MIPS	128 o	3.5 Ko	256 o	8
7	dsPIC30F4013 [h]	dsPIC30	Microchip Tech.	16 bits	EasyDsPIC4A dsPICDEM 2	6.5	30 MIPS	2 Ko	48 Ko		40 (30)
8	ATSAMD09D14 [i]	SAM D09	Microchip Tech.	32 bits	ATSAMD21E16 LMOTOR	1.1	ARM Cortex-M0 48MHz	SRAM 4 ko	16 Ko		24 (22)
9	PIC32MK05$_j$ 12GPD064	PIC32MK	Microchip Tech.	32 bits	PIC32MK GP	6.3	120 Mhz	128 Ko	512 Ko	4096 o	64 (49)

[a] http://www.microchip.com/wwwproducts/en/ATmega328P
[b] Tous les μC de Microchip : https://www.microchip.com/design-centers/microcontrollers
[c] http://www.microchip.com/wwwproducts/en/ATmega1280
[d] http://www.microchip.com/wwwproducts/en/ATtiny417
[e] http://www.microchip.com/wwwproducts/en/ATXMEGA64A1U
[f] http://www.microchip.com/wwwproducts/en/AT32UC3L016
[g] https://www.microchip.com/wwwproducts/en/PIC12F1822
[h] https://www.microchip.com/wwwproducts/en/dsPIC30F4013
[i] http://www.microchip.com/wwwproducts/en/atsamd09d14
[j] http://www.microchip.com/wwwproducts/en/PIC32MK0512GPD064

Tab. 4.2: Exemples de microcontrôleurs et caractéristiques (2)

N°	Microcontrôleur (MCUs)	Famille	Fabricant	Architecture	Cartes à essai	Prix $ (USD)	Processeur	RAM	Mém. Prog. (Flash)	Mém. Data	Pins Count (I/O)
10	MC68HC11F 1CFN3 [a]	MC68HC 11F1	Motorola	8 bits	/	6.5	3MHz	8 Ko	4 Ko		30
11	MC68HC912B32 [b]	68HC12B32	Motorola	16 bits	MC68HC912B32	11	8 Mhz	1 Ko	32 Ko	768 Octo	64
12	MCHC11F1C FNE3 [c]	HC11	NXP Semiconductors	8 bits	/	18	3 Mhz	1 Ko	512 o		30
13	MC9S12XDP5 12VAL [d]	HCS12X	NXP Semiconductors	16 bits	EVB9S12XDP5 12E-ND	27	80 Mhz	32 Ko	512 Ko	4 Ko	112 (91)
14	MSP430G255 3IN20 [e]	MSP430	Texas Instruments	16 bits	Launchpad	3	16 Mhz	SRAM (512 o)	16 Ko	/	16
15	TMS5700432BP ZQQ1R (ARM) [f]	TMS57	Texas Instruments	16 bits	LAUNCHXL-TMS57004	11	80 Mhz	32 Ko	384 Ko	16 Ko	45
16	MSP432P401M IPZR (ARM) [g]	MSP432P401	Texas Instruments	32 bits	CC3100BOOST	6,5	48 Mhz	32 Ko	128 Ko	32 Ko	84
17	EFM32TG108F4 -QFN24 (ARM) [h]	Tiny Gecko	Silicon Labs	32 bits	IC-744885	1,1	ARM Cortex-M3 32 MHz	2 Ko	4 Ko	/	17
18	EFM32JG12B50 0F1024GM48-B [i]	Jade Gecko	Silicon Labs	32 bits	SLSTK3402A	3,9	40 MHz	256 Ko	1024 Ko	/	32

[a] https://www.digikey.ca/product-detail/en/nxp-usa-inc/MC68HC11F1CFN3/MC68HC11F1CFN3-ND/410984

[b] http://www.mouser.fr/ProductDetail/NXP-Freescale/MC68HC912B32MFU8/?qs=sGAEpiMZZMsp%252bcahb6g%252bW2qjmIXKJaK%252bga5yI4JKmuQ%3d

[c] http://www.jameco.com/z/MCHC11F1CFNE3-NXP-Semiconductors-MCU-8-bit-HC11-CISC-512B-ROMLess-Automotive-68-Pin-PLCC_1042413.html

[d] https://www.digikey.com/product-detail/en/nxp-usa-inc/MC9S12XDP512VAL/MC9S12XDP512VAL-ND/1167904

[e] http://eu.mouser.com/ProductDetail/Texas-Instruments/MSP430G2553IN20/?qs=%2fha2pyFadugZAbBrd8KB%252bShBVHLFt9DLeVl2ZoeCyvZW1yswzaGx1g%2fNkXVaF

[f] http://eu.mouser.com/ProductDetail/Texas-Instruments/TMS5700432BPZQQ1R/?qs=sGAEpiMZZMuI9neUTtPr7zWYd8yNnBbm3PZ6h3lj%252bfU%252bwDA5enn6g%3d

[g] http://eu.mouser.com/ProductDetail/Texas-Instruments/MSP432P401MIPZR/?qs=sGAEpiMZZMuI9neUTtPr77PnIIPGaSzSlYMBUmKtMEpzCnXoZO2omA%3d%3d

[h] https://www.digikey.com/product-detail/en/silicon-labs/EFM32TG108F4-QFN24/914-1092-2-ND/2508066

[i] http://eu.mouser.com/ProductDetail/Silicon-Labs/EFM32JG12B500F1024GM48-B/?qs=sGAEpiMZZMuI9neUTtPr79umtpKrtfbdTf7%2f1osH3tZY0U3lHOZ%252byw%3d%3

Il faut noter que la quasi totalité de ces microcontrôleurs sont programmables en langage C.

4.3.2.2 Caractéristiques liées au langage de programmation C

Ce qui suit est la description de certaines fonctionnalités des microcontrôleurs utiles pour la génération de code:

- La spécification des broches programmables, généralement avec des nombres ou des registres. Toute broche peut être programmée sous forme digitale, analogique ou mixte.
- Certaines caractéristiques physiques telles que les mémoires (RAM, ROM), les registres et les processeurs sont considérées avec certains détails.

Plus spécifiquement, ce qui suit est la description des éléments du langage C utilisés lors de la génération du code:

- Les bibliothèques nécessaires à inclure dans le programme et contenant certaines fonctions utiles.
- Les modes de configuration des broches (`PinMode`) en entrée ou en sortie, en fonction d'un usage particulier dans une application.
- Les opérations agissant sur les pins: une fonction définissant les modes d'utilisation des pins (`pinConfigMode`), des fonctions spécifiant comment lire les valeurs des pins (`digitalPinRead`, `analogPinRead`) et des fonctions spécifiant comment écrire des valeurs sur des broches (`digitalPinWrite` , `analogPinWrite`). Chaque fonction a un type, des paramètres et des instructions. Ces informations sont utilisées pour générer le corps des fonctions.
- La configuration d'un timer (temporisateur) devant s'exécuter à une certaine périodicité définie explicitement en millisecondes. La valeur de cette périodicité est définie sur une variable spécifique nommée `TIMER_PERIOD` qui est utilisée dans les expressions temporelles Grafcet. Les instructions de la fonction définissant le timer doivent appeler la fonction de rappel(ou `callback`) `update_timingVars_callback` à chaque exécution du timer.

L'ensemble des concepts utiles du domaine de description des microcontrôleur est récapitulé dans le tableau 4.3. Une autre présentation de ces concepts obtenus par analyse descendante sont présentés sur la figure 4.2 correspondant à un diagramme de fonctionnalités[3]. Cette approche permet de comprendre et de spécifier les lignes de produits logiciels (SPL) [33].

4.3.3 Mise en œuvre sous Eclipse EMF

L'analyse des caractéristiques des microcontrôleurs présentées dans le tableau 4.3 avec les inter-relations entre eux permet d'aboutir à un mé-

3. *feature model* en anglais

TAB. 4.3: Concepts (Classes) identifiés dans le domaine du microcontrôleur

Concept	**Description**	**Codification pour le métamodèle**
Microcontrôleur	Pour décrire un microcontrôleur	*Microcontroller*
Processeur	Se réfère aux caractéristiques du processeur du microcontrôleur	*Processor*
Mémoire	mémoire avec sa capacité	*Memory*
Mémoire RAM	Mémoire vive du microcontrôleur	*RAM*
Mémoire ROM	Mémoire morte/programme du microcontrôleur	*ROM*
Mémoire Flash	Mémoire de donnée	*Flash*
Broche	Patte du microcontrôleur pour réaliser les entrées et sorties	*Pin*
Registre	Petite mémoire à accès rapide liée au processeur	*Register*
Langage C	Caractéristiques liées au langage C	*CLanguage*
Librairie	Bibliothèque de fonctions utiles à inclure dans le programme	*Library*
Mode	Pour donner l'expression du mode de fonctionnement des broches, en entrée et en sortie	*PinMode*
Opération sur les broches	Décrit les opérations sur les pins, qui sont des fonctions dont la liste est données dans les items du type énuméré *OperationName*	*PinOperation*
Fonction	Pour décrire une fonction quelconque avec paramètres et instructions	*Function*
Paramètre	Paramètre d'une fonction	*Parameter*
Instruction	Instruction à l'intérieur d'une fonction	*Instruction*
Configuration du timer	Fonction de configuration du timer	*TimerConfig*

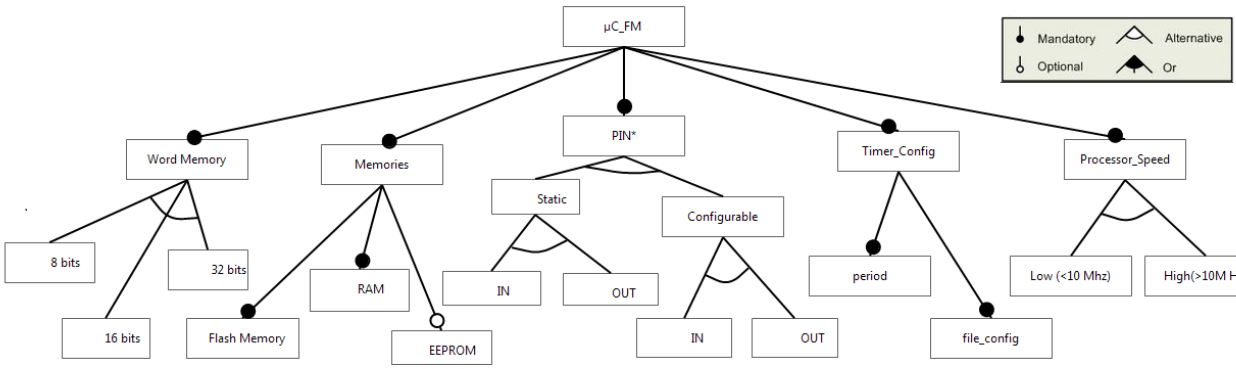


FIG. 4.2: Modèle de description des caractéristiques microcontrôleur

tamodèle microcontrôleur. Ce métamodèle décrit la syntaxe abstraite du langage des microcontrôleurs. La mise en œuvre sous Eclipse EMF de ce métamodèle donne lieu au modèle de la figure 4.3.

Son code obtenu à travers l'éditeur EMF *OCLinEcore* est présenté en Annexe 19.

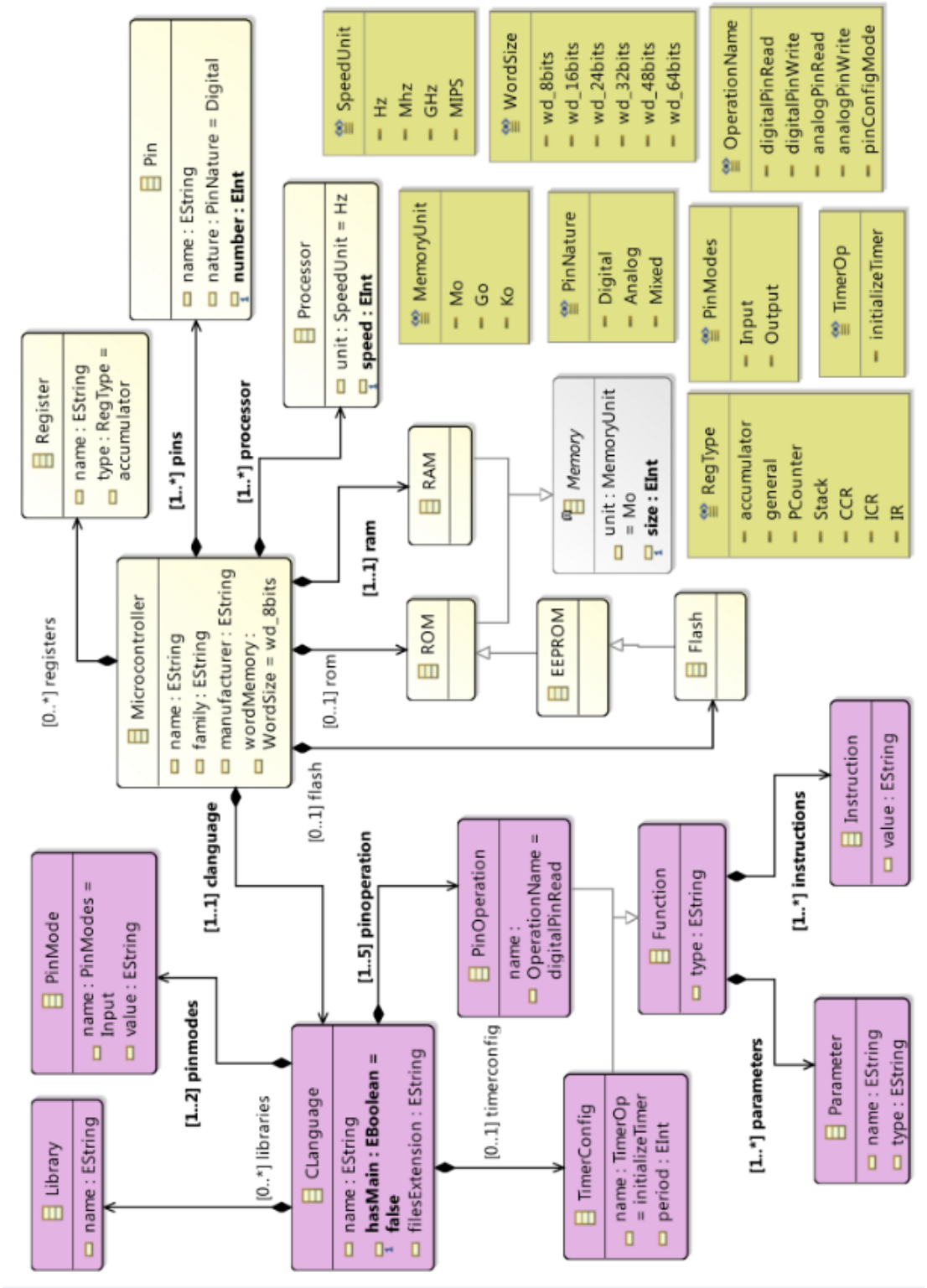


FIG. 4.3: Métamodèle de description des microcontrôleurs

4.3.4 Validation du métamodèle microcontrôleur

Comme pour le Grafcet, un éditeur de modèles microcontrôleur est également généré à partir de la plate-forme Éclipse EMF. Il permet d'éditer et de sauvegarder des modèles au format Xmi pour toute autre utilisation, telle que la génération de code.

Avant de présenter les règles de transformation $M2T$ permettant de générer du code, nous décrivons ce que devient la dynamique interne du Grafcet.

4.4 La dynamique du Grafcet dans le code

Au chapitre 2, nous avons proposé un codage matriciel dans le but d'obtenir un modèle intermédiaire indépendant de toute plateforme cible. Face aux difficultés rencontrées et énoncées au chapitre 3 relatives à sa mise en œuvre, nous avons opté pour une approche complètement IDM, avec une implémentation sous Éclipse EMF. Le métamodèle Grafcet couplé au langage OCL permet de questionner les modèles Grafcet pour obtenir toute donnée utile pour l'implémentation. Après la description de la structure statique du Grafcet, il est nécessaire de décrire sa dynamique à adopter lors de la génération du code. Pour cela, nous avons opté pour la génération du code Grafcet suivant l'approche par équations algébriques. En effet, le profilage du code réalisé en section 2.6 a prouvé les performances du code Grafcet généré suivant cette approche, comparativement à l'approche par codage matriciel. Toutefois, il est toujours possible de réaliser des transformations à partir de ces modèle pour générer du code suivant l'approche par codage matriciel du Grafcet.

Nous rappelons ici les équations algébriques du Grafcet, tout en donnant une structure personnalisée du cycle de balayage ou de scrutation des PLCs pour sa mise en œuvre sur microcontrôleurs.

4.4.1 Équations algébriques du Grafcet

Pour toutes fins utiles, nous donnons ici la description des équations algébriques du Grafcet proposée en [37]. Il s'agit d'un ensemble d'équations issues du Grafcet et conformes aux règles d'évolution (section 1.4.3.1). Elles sont décrites comme suit:

Soit $CC(tr)$ [4] une variable booléenne associée à chaque transition tr d'un modèle Grafcet. Selon les règles d'évolution du Grafcet, la transition tr est franchissable si elle est activée et que la condition/réceptivité associée $TC(tr)$ est vraie. $CC(tr)$ est alors calculée comme le présente la formule 4.1:

$$CC(tr) = (\prod_{i=1}^{m} X_i^{tr}) \times TC(tr) \tag{4.1}$$

4. Clearing Condition

où :

- X_i^{tr} est la variable booléenne de l'activité de chaque étape i située directement en amont de la transition tr.
- $TC(tr)$ est la condition associée à la transition tr et
- m est le nombre d'étapes situées directement en amont de la transition tr.

Dans cette équation, le terme $\prod_{i=1}^{m} X_i^{tr}$ exprime la condition de validation de la transition tr.

Conformément aux règles d'évolution du standard CEI 60848, chaque variable booléenne d'activité associé à chaque étape du Grafcet est calculée comme présentées aux équations 4.2 et 4.3. La première concerne l'initialisation (à $t = 0$) et la seconde concerne l'évolution de l'instant t à l'instant $t + 1$:

$$X_i(0) = \begin{cases} 1 & \text{if } X_i \text{ is an initial step} \\ 0 & \text{else} \end{cases} \tag{4.2}$$

$$X_i(t+1) = \sum_{j=1}^{p} CC(tr_j^{i-}) + X_i(t) \times \prod_{j=1}^{q} \overline{CC(tr_j^{i+})} \tag{4.3}$$

où :

- $X_i(t)$ est la variable d'activité de l'étape i à l'instant t,
- $X_i(t+1)$ est la variable d'activité de l'étape i à l'instant,
- p est le nombre de transition situées directement en amont de l'étape i,
- q est le nombre de transition situées directement en aval de l'étape i,
- $CC(tr_j^{i-})$ est la condition de franchissement (équation 4.1) de chaque transition j située directement en amont de l'étape i et
- $CC(tr_j^{i+})$ est la condition de franchissement de chaque transition j située directement en aval de l'étape i.

Pour le calcul de la valeur des actions, une variable booléenne A est associée à chaque action $\mathcal{A}$. Comme il est possible qu'une même action $\mathcal{A}$ soit associée à plusieurs étapes, la valeur de $A(t)$ est obtenue en calculant le $OU(+)$ logique des variables d'étapes $X_i^{\mathcal{A}}$,$i = 1,2,\ldots,h$; où h est le nombre d'étapes auxquelles cette action est associée (équation 4.4). On a donc :

$$A(t) = \sum_{i=1}^{h} X_i^{\mathcal{A}}(t) \tag{4.4}$$

Où $X_i^{\mathcal{A}}(t)$ est la variable d'activité d'une étape i à l'instant t à laquelle l'action $\mathcal{A}$ est associée.

L'équation 4.4 proposée dans [37] concerne uniquement les actions à niveau continues. Nous pouvons généraliser cette équation en prenant en considération les actions à niveau conditionnelles, en multipliant chaque

variable d'activité d'étape à laquelle l'action est associée par la condition d'activation de cette action. On obtient alors l'équation 4.5 :

$$A(t) = \sum_{i=1}^{h} (X_i^{\mathcal{A}}(t) \times Cond_{X_i}^{\mathcal{A}}(t)) \tag{4.5}$$

où $Cond_X^{\mathcal{A}}(t)$ est la condition à l'instant t de l'action $\mathcal{A}$ associée à l'étape ayant pour variable d'activité X_i.

Pour les actions stockées, il s'agit des expressions textuelles destinées principalement à l'affectation de variables. Elles devront faire l'objet d'une formalisation à l'aide d'outils telles que les grammaires algébriques.

Les équations ici présentées permettent de générer automatiquement du code Grafcet de contrôle.

4.4.2 Le cycle de scrutation exécuté

Dans la présentation du Grafcet en section 1.4, nous avons vu que l'interprétation du Grafcet est basée sur l'occurrence d'événements. Pour faciliter sa mise en œuvre sur des contrôleurs à microprocesseurs, certains travaux de recherche [37, 58, 32] ont proposé une modification de la perspective événementielle de l'interprétation du Grafcet en une perception séquentielle qui est un effet du mode de fonctionnement cyclique des API. Un algorithme séquentiel peut ensuite être conçu pour l'exécution de Grafcet sur des processeurs séquentiels. J. Machado et al. ont proposé dans [37] des équations algébriques (4.4.1) du Grafcet comme interprétation possible de la dynamique de Grafcet . Ils ont utilisé ces équations algébriques comme modèle commun de simulation et de vérification formelle pour obtenir une spécification de contrôleur sécurisée directement implémentée sur un automate. Nous rappelons sur la figure 4.4 le cycle de balayage de l'API adapté au Grafcet pour être traité à chaque étape de l'exécution séquentielle du contrôleur.

L'évolution du Grafcet est alors formulée en termes d'équations algébriques: après la lecture des entrées, l'exécution du programme de contrôle consiste en l'exécution d'équations relatives aux transitions franchissables, au franchissement des transitions, au calcul de la nouvelle situation, et au calcul des actions. Le cycle d'analyse se termine par la mise à jour des actions.

Après la description de la nature dynamique du Grafcet, des règles de transformation doivent être définies pour la génération du code, y compris la génération dans l'ordre des équations algébriques du cycle de scrutation.

4.5 Transformation du Grafcet en code

Ici, nous présentons la structure générale du code généré, puis les règles de transformation réalisées.

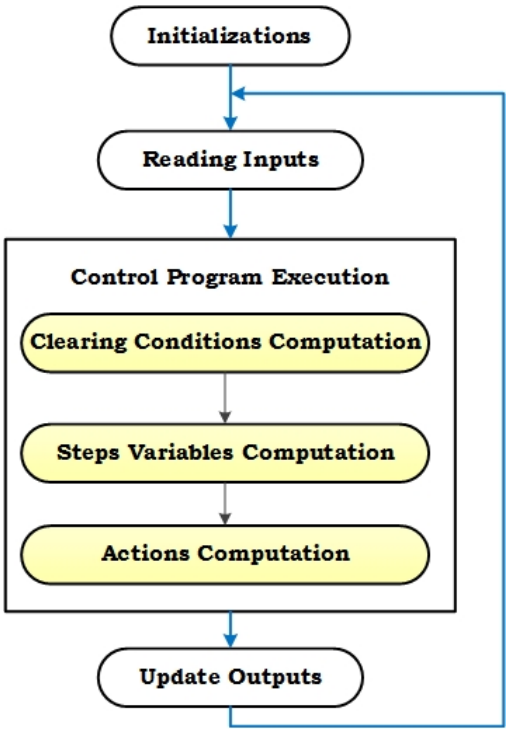


FIG. 4.4: Cycle de balayage PLC personnalisé pour le Grafcet

4.5.1 Structure générale du code généré

La structure générale du code généré est donnée en figure 4.5. Elle fait ressortir en premier lieu l'importation des bibliothèques contenant des fonctions utiles et prédéfinies dans le langage. Elle est suivie de la déclaration des différentes variables liées aux entrées/sorties du systèmes, à l'état dusystème et à la mesure des durées d'activité des étapes. Les initialisations sont réalisées dans la fonction `setup()`. Il s'agit de l'initialisation du timer dans la mesure où le modèle contient des contraintes temporelles et de la configuration des broches liées aux entrées et aux sorties. La fonction `loop()` contient alors le code exécuté en boucle par le contrôleur, tel que présenté dans le cycle de scrutation de la figure 4.4. Toutes les autres fonctions liées à la cible spécifique sont enfin générées. Elles concernent l'initialisation du timer, la fonction exécutée périodiquement par le timer, la configuration des entrées/sorties, la lecture des valeurs des entrées et l'écriture des valeurs des sorties sur les broches.

4.5.2 Les règles de transformation

Les règles de transformation décrivent comment les éléments du langage d'entrée (métamodèles Grafcet et microcontrôleur) sont transformés en éléments du langage de sortie (langage C du microcontrôleur pris en entrée). La transformation est basée sur la correspondance des éléments Grafcet avec les fragments de code C. Les fragments de code C correspondant à un élé-

ment Grafcet ne sont pas nécessairement consécutifs, car le même élément Grafcet peut être à l'origine des fragments de code C à différents endroits du code généré.

Il s'agit bien d'une transformation modèle-à-texte ($M2T$) et non d'une transformation modèle-à-modèle ($M2M$). C'est la raison pour laquelle il est difficile de présenter toutes les règles de transformation définies pour la génération de code. Toutefois, la figure 4.6 donne quelques exemples de règles de correspondance bien identifiées entre les concepts Grafcet et le code C qui en découle.

Dans la suite, nous nous attelons à expliquer ce que produit au niveau du code C les principaux éléments Grafcet.

4.5.2.1 Éléments généraux de génération du code

Variables Grafcet

Pour chaque variable Grafcet, une ou plusieurs variables correspondantes sont créées pour la gérer. Il s'agit selon le cas de:

- **Variables d'entrée/sortie**:
  - ★ La broche à laquelle cette variable est associée: une constante nommée `<pin_+ aVariable.name>`
  - ★ La variable (booléenne ou numérique) contenant son état: une variable nommée `<aVariable.name>`
  - ★ Une variable devant contenir la durée de cette variable d'activité : un entier nommé `<aVariable.name>_duration` et une autre variable contenant son ancienne valeur (valeur à l'instant précédent) nommée `<aVariable.name>_duration_Old`.
- **Variables internes** : Elles sont transformées en variables obtenues telles que présentées ci-dessus pour les variables d'entrée/sortie.

Fonction: Une fonction est créée et nommée `<aFunction.name>` avec éventuellement ses paramètres et possède un corps décrit par les instructions associées se trouvant dans la liste `<aFunction.instructions>`.

Les événements Grafcet (RE ou FE): Ces notions s'étendent des variables simples aux expressions.

`RE(anExpression)` est réalisée avec l'expression logique

`(!<anExpression.getOldCExpr()> && <anExpression.getCExpr()>)`

Similairement, `FE(anExpression)` est réalisée avec l'expression logique

`(<anExpression.getOldCExpr()> && ! <anExpression.getCExpr()>)`

Les expressions temporelles: Initialement, `<anExpression.getName2()>` renvoie récursivement un identifiant. Certaines variables sont ensuite générées pour stocker la durée des expressions dont les identifiants sont `<anExpression.getName2()>_duration` et `<anExpression.getName2()>_duration_Old` . Dans [56], F. Schumacher et al. ont analysé les variables temporelles et proposé un moyen de les évaluer en considérant que l'expression est constituée d'une variable unique. Nous l'avons étendu à n'importe quelle expression booléenne. A partir de cette représentation formelle, nous expliquons

comme suit l'interprétation de chaque type de variable temporelle à travers un calcul:

- ***Condition de délai simple (type Delayed1)***: Sa syntaxe est `[t/expression]` . Cette condition temporelle est vraie si le temps écoulé depuis l'occurrence de l'événement `RE(<expression.subExpr2>)` est supérieur à t. Alors `[t/expression]` est équivalent à:
  `<anExpression.getName2()_duration > t`
- ***Condition limitée dans le temps (type Limited)***: Sa syntaxe est $t/expression$ ou bien `[not(t/expression)]`. Cette condition temporelle est vraie si le temps écoulé depuis l'occurrence de l'événement `RE(<expression.subExpr2>)` est inférieur ou égal à t. Alors `[not(t/expression)]` est équivalent à:
  `<anExpression.getName2()>_duration <= t`
  Pour les variables temporelles de type *Delayed*1 ou *Limited*, la valeur de la variable entière `<anExpression.getName2()>_duration` est calculée comme suit à l'intérieur de la fonction callback de mise à jour des durées:

Listing 4.1: Mise à jour de la valeur de la durée d'activité d'une expression

```
if(FE(anExpression.subExpr2)){
        <expr.getName2()>_duration = 0;
} else{
        <expr.getName2()>_duration ++;
}
```

- ***Condition de délai sous forme générale (type Delayed2)***: Sa syntaxe générale est `[t1/expression/t2]` et sa valeur est définie comme suit:
  - ★ `[t1/expression/t2]` change sa valeur de $false$ à $true$ après un temps écoulé de $t1$ depuis lequel l'événement `RE(<expression.subExpr2>)` s'est produit.
  - ★ la valeur de `[t1/expression/t2]` passe de $true$ à $false$ après un temps écoulé de $t2$ depuis lequel l'événement `FE(<expression.subExpr2>)` s'est produit. Alors, `[t1/expression/t2]` est équivalent à:

Listing 4.2: Calcul de la condition t1/expression/t2

```
(<anExpression.getName2()>_duration > t1) && <anExpression.
    getCExpr()>)
||
```

```
(<anExpression.getName2()>_duration > t2) && ! <
    anExpression.getCExpr()>)
```

où la valeur de `<anExpression.getName2()>_duration` est calculée comme suit dans la fonction callback de mise à jour des durées:

Listing 4.3: Calcul de <anExpression.getName2()>_duration

```
if( (! <anExpression.subExpr2.getOldCExpr()> && <
    anExpression.subExpr2.getCExpr()>)
|| (<anExpression.subExpr2.getOldCExpr()> && ! <
    anExpression.subExpr2.getCExpr()>)){
        <expr.getName2()>_duration = 0;
} else {
        <expr.getName2()>_duration ++;
}
```

De cette façon, la même variable `<anExpression.getName2()>_duration` est utilisée pour mesurer le temps écoulé depuis l'occurrence de l'un des événements suivants: `RE(expression)` ou `FE(expression)`.

4.5.2.2 Génération des éléments de code liés au microcontrôleur

Les broches du microcontrôleur sont déclarées comme des constantes et leurs valeurs sont associées aux variables d'entrée/sortie du modèle Grafcet, selon leur mappage.

Génération des constantes associées aux pins: Pour chaque broche définie dans ce modèle, une constante est définie pour elle comme suit:

```
#define <pin.name> <pin.number>
```

Ces valeurs sont utilisées au moment de lire les valeurs des entrées dans les variables d'entrée du modèle et au moment d'écrire les valeurs des variables de sortie du modèle sur les broches configurées en sortie.

Génération des fonction de gestion des broches: Ces fonctions sont décrites dans le type énuméré *EnumerationName*. Elles concernent la configuration du mode (entrée/sortie) des broches, la lecture des valeurs sur les broches configurées en entrée et l'écriture des sortie du modèle sur les broches configurées en sortie. Comme elles sont décrites dans le modèle du microcontrôleur à l'aide de fonctions ayant leurs paramètres et des instructions, le code utilisé pour leur génération est une itération donnée comme suit:

Listing 4.4: Code Acceleo de génération des opérations sur les broches

```
[for (pinOp : PinOperation | aMicro.clanguage.pinoperation)]
[pinOp.type/] [pinOp.name/]
```

```
(
[for (param : Parameter | pinOp.parameters) separator (', ')]
[param.name/]
[/for]
)
{
[for (instr : Instruction | pinOp.instructions)]
       [instr.value/];
[/for]
}
[/for]
```

La génération de la fonction qui gère les paramètres est réalisée de la même façon.

4.5.2.3 Génération de l'initialisation du programme

Dans les initialisation, la fonction d'initialisation du timer `initializeTimer()` est appelée. Ensuite, les broches sont configurées dans un mode (Input/Output) selon l'utilisation qui est faite dans le modèle. Ainsi, pour chaque variable externe (d'entrée ou de sortie) $< aVar >$, on a : `pinConfigMode(pin_<aVar.name>, CONFIG_MODE);`

Ici, `CONFIG_MODE` a pour valeur **INPUT** ou bien **OUTPUT**, selon que la variable `<aVar.type>` est une variable d'entrée (`VarType::Input`) ou une variable de sortie (`VarType::Output`). Cependant, le contenu de la fonction `pinConfigMode(pin, mode)` dépend du microcontrôleur utilisé.

Finalement, il est nécessaire de mettre la situation du Grafcet dans son état initial. Cela est fait en mettant à $true/1$ l'activité des variables liées aux étapes initiales. Pour chaque étape `<aStep>`, si `<aStep.isInitial>` est vrai, alors

`<step.variable.name>_Old = true;`

En effet, dans le bloc d'instructions exécuté en boucle, la variable `<step.variable.name>` est obtenue par calcul à partir des valeur antérieures définies dans `<step.variable.name>_Old`.

4.5.2.4 Génération de la lecture des entrées

Pour la lecture des entrée logiques, étant donnée chaque variable $< aVar >$ instance de $BooleanVariable$, si `<aVar.type> = VarType.Input`, alors on écrit :

```
<var.name> = digitalPinRead(pin_<var.name>);
```

Pour ce qui concerne la lecture des valeurs numériques, étant donnée chaque variable `<aVar>` instance de $NumericVariable$, si `<aVar.type> = VarType.Input`, alors on écrit :

```
<var.name> = analogPinRead(pin_<var.name>);
```

Le contenu des fonctions `digitalPinRead(pin)` et `analogPinRead(pin)` dépend du contrôleur utilisé.

4.5.2.5 Génération du franchissement des transition et du calcul de la nouvelle situation

Le principe de génération de cette partie a déjà été donné en section 4.4.1. Techniquement, `<aTransition.transitionCondition.getCExpr()>` retourne l'expression C appropriée pour la réceptivité de la transition `<aTransition>`. Elle correspond à l'expression logique à calculer chaque fois pour avoir la valeur de sa réceptivité.

Comme nous avons désigné `VT_<aTransition.name>` la variable booléenne dont la valeur est *true* lorsque cette transition est validée, la variable du franchissement de cette transition est calculée comme suit:

Listing 4.5: Calcul de la variable de franchissement d'une transition

```
FT_<aTransition.name> = VT_<aTransition.name> && R_<aTransition.name
    >;
if(FT_<aTransition.name>) {transitions_fired = 1;}
```

Afin de détecter si au moins une transition est franchie, une variable nommée `transitions_fired` est initialisée à 0. C'est elle qui reçoit 1 lorsqu'une transition quelconque `<aTransition>` est franchie `FT_<aTransition.name>` est *true*).

Sa valeur est alors utilisée pour savoir si une situation stable est atteinte (`transitions_fired = 0`), auquel cas les actions à niveau sont activées. Elle peut aussi être obtenue sous une forme compacte comme donnée en équation 4.6 :

$$transitions\_fired = \sum_{<aTransition>} FT\_< aTransition.name > \qquad (4.6)$$

Ici, la somme ($\sum$) est réalisée à l'aide de l'opérateur OR ($||$).

La nouvelle situation du Grafcet est calculée selon l'équation 4.3, où `X` et `X_Old` correspondent respectivement à $X(t+1)$ et $X(t)$ de cette équation.

4.5.2.6 Génération du calcul des actions

Les équations 4.4 et 4.5 de la section 4.4.1 donnent respectivement la formule du calcul des actions à niveau continues et conditionnelles. Pour la génération du code correspondant, toutes les variables liées aux actions à niveau et stockées sont réinitialisées à la valeur par défaut (0).

A chaque cycle de scrutation, les variables liées aux actions à niveau sont mises à faux (0). Les actions à niveau ne sont calculées que si une situation stable est atteinte (`transitions_fired = 0`). Réinitialisée à 0, la valeur d'une variable d'action à niveau est mise à 1 si l'étape à laquelle elles est associée est active et que la condition associée est *true* (1):

Listing 4.6: Calcul de la valeur des sorties

```
if(!<transitions_fired>){
        for every step <aStep>
        if(<aStep.variable.name>) {
                if(<aStep.actions(LevelActions)[0]. expressionCondition
                    .getCExpr()>){
                <aStep.actions(LevelActions)[0].variable.name> = 1;
                }
        }
                ...
                //for all level actions associated to the step <aStep>
}
```

Quant aux actions impulsionnelles, il s'agit d'affectation de variable réalisée comme suit:

Listing 4.7: Action stockée à l'activation

```
if (RE (<aStep.variable>)){
        <anAction.actionVariable> = <anAction.expressionToEvaluate>;
}
```

Listing 4.8: Action stockée à la désactivation

```
if (FE (<aStep.variable>)){
        <anAction.actionVariable> = <anAction.expressionToEvaluate>;
}
```

4.5.2.7 Génération de la mise à jour des sorties

Appelées actions dans le Grafcet, ces sorties sont soit logiques, soit numériques. Pour mettre à jour les actions logiques, une situation stable doit être atteinte. Une action est alors mise à jour si sa nouvelle valeur calculée est différente de son ancienne valeur. On a :

Listing 4.9: Génération du code de mise à jour des sorties

```
if(! <transitions_fired>){
        if(<anAction.variable.name> != <anAction.variable.name> +"_Old
            "){
        digitalPinWrite(+"pin_"+ <anAction.variable.name>,
                <anAction.variable.name>);
        }
        ...
        //For every level action
}
```

Les actions stockées sont mises à jour indépendamment du type de situation (stable ou non). Leur mise à jour consiste en l'écriture de leur valeur sur la broche correspondante:

Listing 4.10: Génération de l'écriture d'une sortie binaire

```
analogPinWrite(+"pin_"+ <anAction.variable.name>, <anAction.variable.
    name>);
```

4.5.2.8 Génération du calcul de la durée d'activité des variables

La durée de l'activité d'une variable est calculée comme suit:

Listing 4.11: Calcul de la durée d'activité des variables

```
if(FE(<aVariable.name>){
        <aVariable.name>_duration = 0 ;
} else
        if(<aVariable.name>){
                <aVariable.name>_duration ++;
        }
```

Initialement, la variable `<aVariable.name>_duration` est fausse (0). Lorsque la variable devient inactive, i.e. passe de *true* à $false$ (durant l'événement `FE(<aVariable.name>)`), on réinitialise sa valeur à 0. Si elle est déjà active, alors on incrémente simplement sa valeur. Ceci permet de compter le nombre de périodicités de temps où la variable est active. Puisque la la périodicité du timer est `TIMER_PERIOD`, lorsque c'est nécessaire, la valeur de sa durée d'activité d'une variable `<aVariable.name>` est obtenue en multipliant le nombre de périodicités par la valeur de la périodicité : $< aVariable.name > \_duration \times TIMER\_PERIOD$.

Après le calcul de la durée de chaque durée d'activité de variable, son ancienne valeur est mise à jour à l'aide de :

```
<aVariable.name>_duration_Old = <aVariable.name>_duration;
```

En effet, il est difficile à l'instant t de dire avec exactitude la valeur de la durée d'activité d'une variable à l'instant $t-1$, puisqu'elle n'est pas forcément égal à sa valeur à l'instant t moins 1. Cela est dû au fait que si cette variable est passée de *true* à $false$ (avec l'occurrence de l'événement `FE(<aVariable.name>)`), sa valeur est passée à 0 et on ne saurait avoir cette valeur à l'instant $t-1$ avec `<aVariable.name>_duration -1`. De même, lorsque la variable était déjà fausse depuis quelques cycles de scrutation, `<aVariable.name>_duration = 0`. Il est donc préférable de garder dans une autre variable que nous avons nommée `<aVariable.name>_duration_Old` sa valeur à l'instant précédent. `<aVariable.name>_duration_Old` est particulièrement importante au moment d'évaluer des expressions composées associées aux évé-

nements `RE(anExpression)` ou `FE(anExpression)`, i.e. dont l'expression ne tient pas simplement à une seule variable. Dans ce cas, on a :

```
<atExpression.getName2()>_duration_Old = <atExpression.getName2()>_duration
;
```

Par exemple, `FE(var+1>3)` est évalué avec `(var_Old+1>3)&& (var+1>3)`.

4.6 Implémentation de la transformation avec *Acceleo*

La transformation IDM réalisée dans ce cadre est du type horizontale de modèle à texte ($M2T$). Comme décrite en figure 4.7, elle prend en entrée un modèle Grafcet (instance du métamodèle de la Grafcet), spécification du fonctionnement du contrôleur et un modèle du microcontrôleur à utiliser.

Le texte généré est basé sur des templates du langage *Acceleo*.

4.6.1 Le langage de transformation M2T *Acceleo*

Le langage *Acceleo*[5] est utilisé pour mettre en œuvre cette transformation. Il s'agit d'une implémentation de la spécification $MOFM2T$ définie par l'OMG [29, 24]. Elle est composée de deux types principaux de structures que sont les templates et les requêtes, toutes définies à l'intérieur des modules. Acceleo permet de générer des artefacts textuels à partir des modèles EMF. Un template est un ensemble d'instructions utilisées pour pour produire en sortie du texte, avec certaines parties du texte paramétrables et dont les valeurs sont issues des modèles en entrée, comme le montre la figure 4.8.

Dans un projet *Acceleo*, plusieurs modules peuvent être créés avec la relation d'importation entre eux. *Acceleo* peut être intégré à Éclipse via un plugin pour permettre la création de projets.

La figure 4.9 présente l'architecture générale de cette transformation.

4.6.2 Architecture des modules de transformation

Les templates de la transformation sont organisés sous forme de modules connectés de façon hiérarchique, dont la structure est données sur la figure 4.10. Chaque module contient alors un ou plusieurs templates ainsi que des requêtes OCL utilisées pour interroger et extraire des informations des modèles d'entrée manipulés. La figure 4.10.a présente la conception de ces modules alors que la figure 4.10.b décrit leur mise en œuvre.

Les modules sont organisés en trois niveaux:

– **Niveau 1** : où il y a le module principal. Il contient plusieurs templates parmi lesquels le templates en charge de la génération du programme

5. De l'entreprise française Obeo, site officiel: https://www.eclipse.org/acceleo/

principal Grafcet. Il appelle les templates des modules de niveau 2 pour générer du code en appelant les services nécessaires (comme la date ou l'heure) et d'autres fonctions de génération pour les structures Grafcet.

- **Niveau 2** : En dehors du module contenant des templates de génération des services utiles, il faut un module permettant de générer du code correspondant aux structures Grafcet. Il doit contenir des templates pour la génération de fonctions telles que celle à appeler par le temporisateur pour mettre à jour les durées d'activité des variables. Les templates de ce module doivent également intégrer le résultat de la génération de variables utiles et d'autres fonctions.
- **Niveau 3** : il concerne les 2 modules dédiés à la génération des variables utiles et d'autres fonctions à appeler dans le programme.

Chaque module contient plusieurs templates et chaque templates implémente une fonction utilisée dans un but spécifique dans le processus de génération de code. Le module principal est *generateG7MM2Code.mtl*. Il contient un template et donne la structure principale du code généré en sortie dans un fichier. Ici, chaque module est implémenté (par exemple, le module Acceleo *generate_G7_structures.mtl* implémente le module conçu pour la génération de structures de grafcet principales). Le contenu de ce fichier est donné en Annexe 6. Une fois toutes les transformations définies, elles doivent être expérimentées sur un cas; c'est ce que nous faisons dans la section suivante.

4.7 Conclusion

Après avoir décrit la transformation de modèles qui est un élément central du processus IDM, nous avons proposé un langage de description des caractéristiques des microcontrôleurs. Ce DSML contient des caractéristiques générales des microcontrôleurs ainsi que celles spécifiques aux cibles et relatives à la génération du code C pour un modèle Grafcet donné. Le DSML de description des microcontrôleurs a alors été implémenté à l'aide d'Eclipse EMF. L'éditeur généré permet alors de créer un modèle microcontrôleur décrivant les caractéristiques d'une cible microcontrôleur utiles pour la génération d'un code dédié. Avant de passer à la génération du code proprement dite, nous avons décrit les équations algébriques Grafcet qui caractérisent la dynamique du Grafcet. Ces équations déterminent la structure du code à générer pour caractériser cette dynamique d'évolution. A partir de ces équations algébriques, nous avons conçu un cycle de scrutation à exécuter par les microcontrôleurs, obtenu par adaptation du cycle de scrutation des PLCs.

Pour la transformation du Grafcet en code dédié à un microcontrôleur précis, l'architecture générale de la transformation est présentée, ainsi que les principales règles de transformation implémentées, y compris les règles

spécifiées par les équations algébriques Grafcet. C'est le langage *Acceleo* qui a servi de domaine technologique pour implémenter la transformation, à travers un plugin disponible pour Eclipse EMF. Il s'agit d'une implémentation de la spécification $MOFM2T$ définie par l'OMG [29, 24]. Ainsi, il ne suffit plus que d'avoir un modèle Grafcet et un modèle microcontrôleur EMF valides pour exécuter la transformation de modèles réalisée pour générer du code Grafcet en langage C pour la cible modélisée.

Génération des informations générales (nom du Grafcet, date et heure)
Importation des bibliothèques utiles
Génération des Déclarations
    Déclaration des constantes assignées aux broches du microcontrôleur
    Déclarer des variables pour l'état des entrées et des sorties du modèle
        (numériques et analogiques)
    Déclarer les variables d'activité d'étapes (X_i)
    Déclarer les variables sur les transitions validées (VT_i)
    Déclarer les variables sur les réceptivités des transitions(R_i)
    Déclarer les variables sur les transitions franchissables (FT_i)
    Déclarer les variables sur durées d'activité des variables (vvvvv_duration)
    Déclaration de la constante sur la durée d'une période du timer
    Déclaration de la variable sur les transitions franchies
    Signature des fonctions de manipulation des broches générées plus loin

```
void setup(){
    initializeTimer(); //Initialisation du timer
    Configuration des broches en entrées/sortie
    Définition de l'état initial du grafcet
}
void loop(){
    Lecture de l'état des entrées en appelant les fonctions dédiées
        (digitalPinRead() et analoglPinRead())
    Calcul des transitions validées
    Calcul des réceptivités des transitions validées
    Calcul des transitions franchissables
    Calcul du nouvel état (les X_i)
    Calcul des sorties
    Mise à jour des sorties avec (digitalPinWrite() et analoglPinWrite())
    Sauvegarde de la valeur des variables vvv dans les variables vvv_Old
}
int main(void)
{
  setup();
  for ( ; ; ) loop(); // repeat indefinitely the function loop()
  return 0;
}
```

Génération de la fonction d'initialisation du timer
Génération de la fonction *update_G7TimingVars_callback* appelée
        périodiquement pour mettre à jour les durées d'activité des variables
Génération de la fonction de configuration des broches
Génération des fonctions de lecture/écriture des broches

FIG. 4.5: Structure générale du code généré par transformation *M2T*

Grafcet element	**Code generated**
RE(anExpression) evaluation	*(!<anExpression.getOldCExpr()> && <anExpression.getCExpr()>)*
FE(anExpression) evaluation	*(<anExpression.getOldCExpr()> && ! <anExpression.getCExpr()>)*
Limited time condition evaluation	*<anExpression.getName2()>_duration <= t*
Limited time condition updating	*<anExpression.getName2()>_duration:* *if(FE(anExpression.subExpr2)){* *<expr.getName2()>_duration = 0;* *} else{* *<expr.getName2()>_duration ++;* *}*
Receptivity calculation	*R_<aTransition.name> = <aTransition. getCExpr()>;*
Clearing a transition computation	*FT_<aTransition.name> = VT_<aTransition.name> && R_<aTransition.name>;* *if(FT_<aTransition.name>) {transitions_fired = 1;}*
Level action computation	*if(!<transitions_fired>){* for every step <aStep> *if(<aStep.variable.name>) {* *if(<aStep.actions(LevelActions)[0].expressionCondition.getCExpr()>)* *<aStep.actions(LevelActions)[0].variable.name> = 1;* *}* … for all level actions associated to the step <*aStep*> *}*
updating outputs/actions	*if(! <transitions_fired>){* *if(<anAction.variable.name> != <anAction.variable.name> +"_Old"){* *digitalPinWrite(+"pin_"+ <anAction.variable.name>, <anAction.variable.name>);* *}* *… For every level actions* *}*
Duration of activity variables computation	*if(FE(<aVariable.name>){ <aVariable.name>_duration = 0 ;}* *else if(<aVariable.name>){ <aVariable.name>_duration ++; }*

FIG. 4.6: Exemples de règles pour la transformation *M2T*

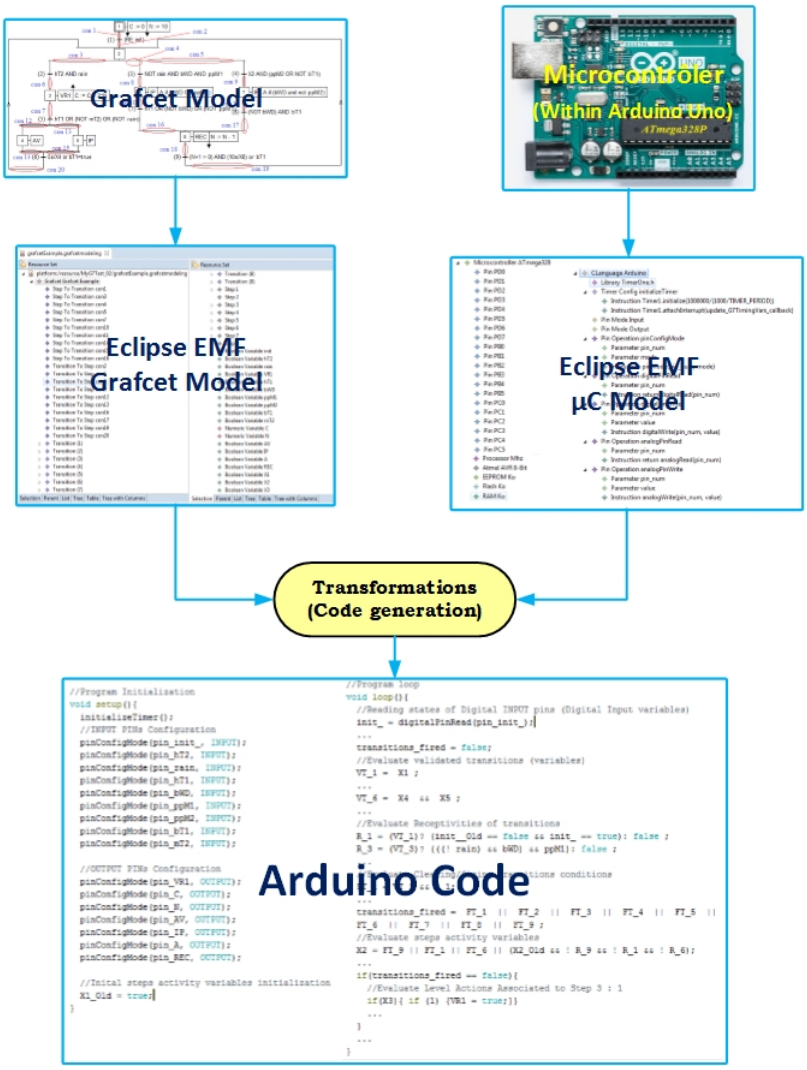


FIG. 4.7: Aperçu général du processus de transformation

FIG. 4.8: Principe de fonctionnement d'*Acceleo*

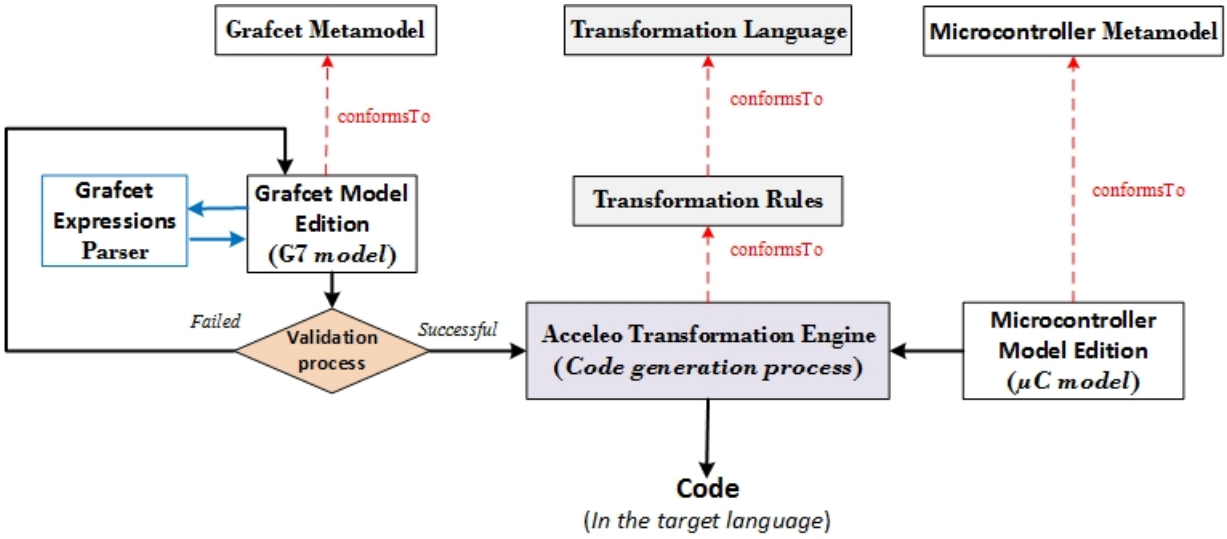


FIG. 4.9: Architecture générale IDM de la transformation

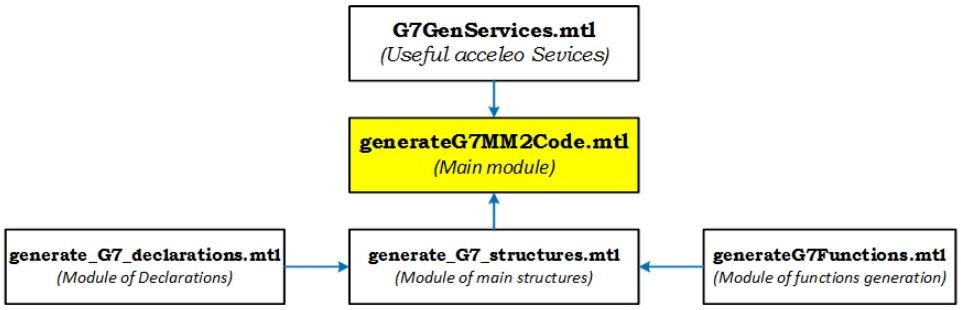


FIG. 4.10: Architecture des modules avec leurs dépendances

Chapitre Cinq

Application à la Génération par approche IDM de code Arduino pour le contrôleur d'un système multi-énergie

Sommaire

5.1 Introduction

Dans ce chapitre, nous expérimentons l'approche IDM de synthèse multi-cibles Grafcet proposée dans les chapitres précédents. L'application choisie est celle du contrôle d'un système autonome d'approvisionnement en eau domestique au centre duquel se trouve un système de commutation des sources d'énergie. Il est question de concevoir pour ce système un contrôleur basé sur un microcontrôleur, tout en considérant sa spécification fonctionnelle Grafcet traduite dans le langage DSML Grafcet étudié et présenté au chapitre 3. Après avoir réalisé un modèle des caractéristiques du microcontrôleur (*Atmega1280*) à l'aide du langage DSML défini et présenté au chapitre 4, le code de contrôle devra être généré en langage cible C (*Arduino*) par exécution de la transformation de modèle décrite au chapitre 4.

5.2 Présentation du cas d'étude

Ce cas d'étude explore la possibilité de tirer profit de la profusion de capacité de calcul à coût raisonnable pour résoudre un problème concret rencontré dans la mise en place de processus de développement durable, notamment celui lié à l'approvisionnement en eau et en énergie ... Il s'agit de l'approvisionnement en eau autonome des bâtiments à plusieurs étages, en utilisant plusieurs réservoirs (ou citernes) de stockage approvisionnées par plusieurs sources d'eau et d'énergie de pompage, basée sur plusieurs niveaux hiérarchisés de priorité d'accès à l'eau.

Plus précisément, l'objectif final est de trouver une solution combinant des solutions durables et les avantages des outils TIC afin de réaliser un système complet et autonome d'approvisionnement en eau et en énergie, en mettant l'accent sur l'économie d'énergie de pompage. Cela implique la mise en place de mécanismes de pompage appropriés et un processus efficace de commutation entre les sources d'énergie. Le système intelligent devrait garantir la fourniture continue d'eau aux ménages et basculer automatiquement entre plusieurs ressources en eau et en énergie en fonction de la disponibilité de la source et des coûts de service.

Après avoir présenté l'existant et ses limites, nous décrivons d'abord le problème de façon générale, puis nous proposons des dispositifs de pompage et un processus de commutation entre les sources d'énergie, associés à une structure architecturale garantissant une réduction significative de l'énergie de pompage. A la fin, nous donnons un cas spécifique de ce problème général qui retiendra notre attention dans les sections suivantes.

5.2.1 La gestion des systèmes multi-énergie : existant et limites

L'utilisation de plusieurs sources d'approvisionnement en eau est une solution envisageable permettant de faire face aux interruptions récurrentes dans les services d'approvisionnement en eau en milieu urbain. Il existe cependant un certain nombre de problèmes: certaines sources d'eau (puits/-forages) nécessitent de l'énergie électrique pour fonctionner. Cette énergie électrique peut elle-même être disponible sous plusieurs formes, y compris la distribution d'énergie urbaine classique ainsi que des sources locales d'énergie renouvelable. Cependant, qu'il s'agisse de sources d'eau ou d'énergie, leurs coûts diffèrent les uns des autres. Une stratégie de gestion doit donc être mise en place pour sélectionner les sources disponibles les moins coûteuses à chaque instant dans le but de garantir la présence permanente d'eau.

L'importance d'investir dans les sources d'énergie renouvelables (telles que l'énergie solaire, l'énergie éolienne, ...) est émise pour améliorer de manière significative et permanente la disponibilité de l'eau ([52]). Une méthode d'optimisation est proposée dans [64] afin de minimiser la consommation d'énergie et de réduire les fuites dans les systèmes de distribution d'eau (SDE) dans un contexte hydraulique. Cependant, l'algorithme proposé ne peut pas être appliqué dans un contexte de ménage. Un modèle d'optimisation multi-critères a été présenté dans [52]. Mais aucune des méthodes existantes ne décrit une manière opérationnelle de traiter la question de la gestion et la commutation des sources d'eau et d'énergie.

5.2.2 Présentation générale du problème

Considérons un système d'approvisionnement en eau, associé à un bâtiment ayant m étages, n sources d'approvisionnement en eau, p sources d'énergie utilisées pour pomper de l'eau et k réservoirs. Chaque étage du bâtiment est alimenté en eau par un réservoir spécifique jusqu'à ce que le réservoir soit vide. Lorsque le réservoir correspondant est vide, un mécanisme précis doit servir être déclenché pour son approvisionnement.

5.2.2.1 L'approvisionnement des réservoirs en eau

Comme présenté à la figure 5.1, chaque réservoir est alimenté en eau par la descente des eaux de pluie, par pompage de l'eau du réservoir inférieur, ou en ouvrant l'eau fournie par la distribution d'eau du réseau urbain. Chaque réservoir approvisionne k niveaux de l'immeuble (par exemple $k = 3$). Chaque réservoir T_i est équipé de trois niveaux de capteurs qui indiquent le niveau vide (bT_i), le niveau plein (hT_i) et le niveau d'eau moyen (mT_i). Il y a alors deux manière de conduire l'eau des sources d'eau au réservoir: le remplissage à énergie nulle avec de l'eau de pluie et le remplissage par pompage. L'eau urbaine peut également servir pour alimenter

les niveaux de l'immeuble lorsque les deux premières sources ne sont pas disponibles.

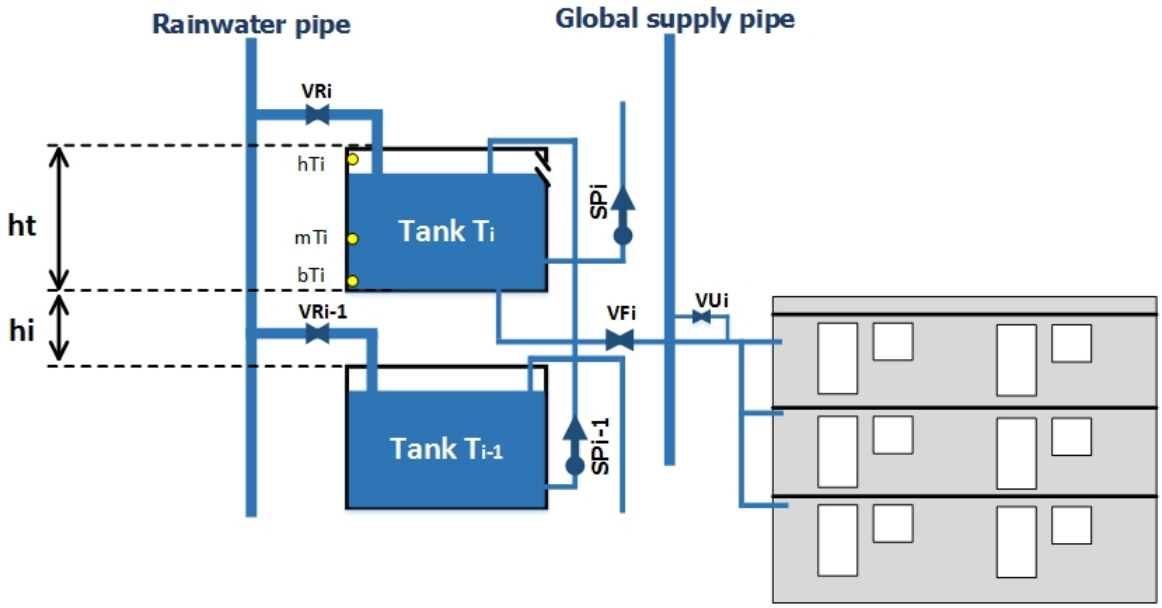


FIG. 5.1: Schéma d'un réservoir intermédiaire

Le remplissage des réservoirs à énergie nulle

Pendant les pluies, les réservoirs sont remplis par l'eau de pluie récupérée sur les toits. Le remplissage est effectué en allant du réservoir le plus haut au réservoir le plus bas. En cas de faible pluviométrie, le séquencement du remplissage des réservoirs permet de stocker l'eau recueillie en priorité dans les réservoirs ayant le potentiel d'énergie le plus élevé. Ainsi, le remplissage de T_i n'est possible que si le réservoir supérieur T_{i+1} est déjà plein($hT_{i+1} = 1$). Le remplissage du réservoir T_i consiste en l'ouverture de la vanne VR_i lorsque toutes les autres vannes VR_j $(j \neq i)$ sont closes. Le remplissage s'arrête lorsque le réservoir T_i est plein $(hT_i = 1)$, la vanne VR_i est alors close et le remplissage du réservoir T_{i-1} peut commencer.

Le remplissage des réservoirs par pompage

En absence de pluie, le pompage permet d'approvisionner en eau le réservoir T_i. Cela consiste en l'activation de la pompe SP_{i-1} qui transfère de l'eau du réservoir T_{i-1} au réservoir T_i. Selon la forme d'énergie utilisée pour faire tourner la pompe, nous pouvons distinguer trois principaux modes de pompage:

- **Le pompage par énergie solaire directe**
  Dans ce mode, le pompage de l'eau du réservoir T_{i-1} au réservoir T_i est réalisé sans coût d'énergie à l'aide des pompes à moteurs électrique dont l'énergie provient des cellules photovoltaïques situées sur des panneaux solaires capturant de l'énergie produite directement à partir du rayonnement solaire. En présence d'une luminosité solaire suffisante

($srl = 1$), ce mode de pompage à coût énergétique zéro est systématiquement activé jusqu'à ce que le réservoir T_i soit plein ($hT_i = 1$), ou bien que le niveau d'eau dans le réservoir T_{i-1} soit en deçà du niveau moyen ($mT_{i-1} = 0$), auquel cas on préfère arrêter pour ne pas pénaliser les étages associés au réservoir T_{i-1}. Ce mode de pompage est particulièrement adapté aux environnements ruraux en Afrique. En effet, la lumière du soleil est abondante (plus de six heures par jour d'ensoleillement) et est suffisante même pour le pompage de grandes quantités d'eau. De plus, l'isolement des villages ruraux rend difficile leur approvisionnement en énergie conventionnelle. Autrement, les besoins en eau dans les villages ruraux sont généralement suffisamment faibles pour pouvoir être couverts par le pompage par énergie solaire.

– **pompage par des batteries**
Pendant la nuit ou lorsque le rayonnement solaire est insuffisant, la pompe SP_{i-1} est activée à l'aide de l'alimentation par batteries lorsque le niveau d'eau dans le réservoir T_i est inférieur au niveau inférieur ($bT_i = 0$). Ce mode de pompage est activé jusqu'à ce que le réservoir T_i soit plein ($hT_i = 1$) ou que le niveau d'eau dans le réservoir T_{i-1} soit inférieur au niveau moyen ($mT_{i-1} = 0$). Les batteries accumulent l'excédent d'énergie créé par le système de panneaux photovoltaïques (PV) et le stockent pour pouvoir être utilisé la nuit lorsqu'il n'y a plus d'apport d'énergie solaire. Les batteries peuvent se décharger rapidement et conduire à un chargement direct, ...
Pour le pompage d'eau par panneaux photovoltaïques, certaines batteries (du type *peep-cycle*) permettent de fournir quelques ampères nécessaires au pompage pendant des centaines d'heures entre les charges. Ce type de batterie peut effectuer de nombreux cycles répétés de chargement-utilisation et convient mieux aux systèmes d'énergie photovoltaïque que les batteries à cycle faible, comme celles utilisées pour démarrer une voiture conçues pour fournir plusieurs centaines d'ampères pendant quelques secondes, puis l'alternateur prend le relais et la batterie se recharge rapidement. Ces deux types de batteries sont conçus pour des applications différentes et ne doivent pas être interchangés.
Idéalement, un groupe de batteries doit être dimensionné pour pouvoir stocker de l'énergie pendant 5 jours d'autonomie durant le temps nuageux. Si le groupe de batteries a une capacité de pompage inférieure à 3 jours, il fonctionnera régulièrement en profondeur et la durée de vie de la batterie sera plus courte. La taille du système, les besoins individuels et les attentes détermineront la taille de batterie la mieux adaptée pour un système donné. Le coût énergétique de ce mode de pompage est quasi nul car il se limite à la maintenance de la batterie dont l'usure est induite par la multiplicité des cycles de charge/décharge.

– **Pompage avec la source d'électricité urbaine**

Pendant les nuits où les batteries sont défectueuses, les pompes d'alimentation sont assurées par le réseau électrique urbain. Cette méthode de pompage est coûteuse car elle est facturée progressivement et payée périodiquement à l'agence d'électrification urbaine.

Pour réduire les coûts énergétiques, ce mode de fonctionnement ne sera appliqué que si le réservoir correspondant est inférieur au niveau bas ($bT_i = 0$) et s'arrête lorsque le niveau moyen est atteint ($mT_i = 0$).

Remplissage des réservoirs par le réseau de distribution urbain

La distribution d'eau du réseau urbain est réalisée par une entreprise publique ou privée. Nous supposons que ce système de réseau garantit la fourniture d'une pression suffisante pour propulser l'eau vers le réservoir supérieur T_k sans qu'aucun équipement de pompage auxiliaire ne soit nécessaire. Lorsqu'un réservoir T_i est vide ($bT_i = 0$) et qu'il n'est pas possible de le remplir d'eau de pluie ni de l'eau provenant du réservoir T_{i-1}, les étages de T_i sont approvisionnés par une conduite d'eau du réseau de distribution urbain vers les étages concernés (comme indiqué sur la figure ref fig:gen-phys-archi de l'architecture générale).

5.2.2.2 Modèle entrées/sorties du contrôleur d'un réservoir intermédiaire

En considérant la description précédente, la configuration entrée/sortie (I/O) du contrôleur d'un réservoir intermédiaire T_i est décrite par le modèle présenté dans la figure 5.2.

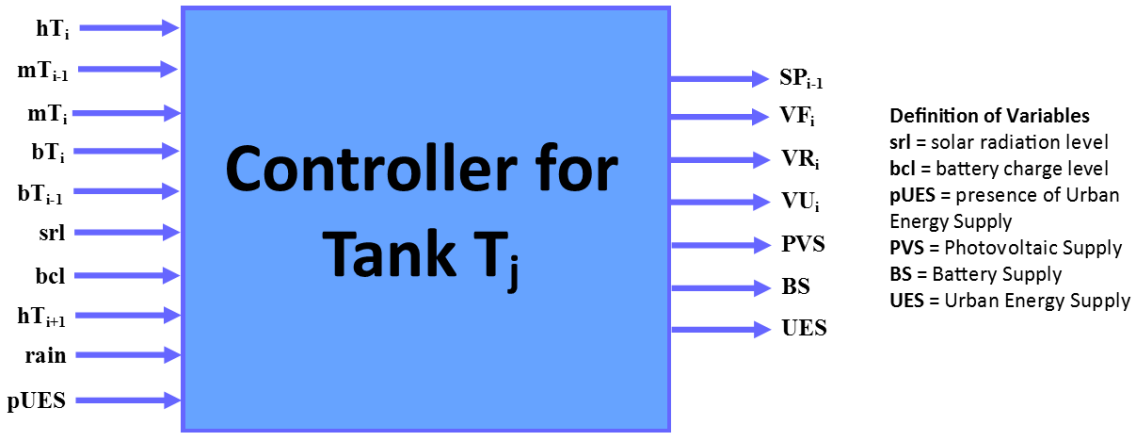


FIG. 5.2: Modèle I/O du contrôleur d'un réservoir intermédiaire

5.2.3 Modélisation et justification des choix architecturaux

Une fois présentés les concepts clés de la structure architecturale avec les dispositifs de pompage et les priorités entre les sources d'eau et les sources

d'énergie, nous proposons ici un processus de commutation garantissant une réduction significative de l'énergie de pompage. En fait, chaque processus de commutation permet d'obtenir une structure architecturale particulière.

Il existe de nombreux processus de commutation possibles associés aux structures architecturales et présentant des gains d'énergie différents. Elles dépendent de nombreux facteurs: le nombre de réservoirs k, la position des réservoirs par rapport aux étages, les mécanismes de pompage, ... Quoi qu'il en soit, le but final est de choisir une structure architecturale permettant de garantir l'approvisionnement en eau de chaque étage du bâtiment avec une réduction significative de l'énergie de pompage.

5.2.3.1 Spécification architecturale

Généralement, nous pouvons avoir n'importe quel nombre quelconque k de réservoirs. Dans le cas où il n'y a qu'un seul réservoir ($k = 1$), l'eau de pluie est collectée dans un réservoir près du toit pour permettre d'alimenter n'importe quel niveau du bâtiment avec une énergie nulle.

Quand il y a des réservoirs en grand nombre, chaque étage du bâtiment peut avoir son propre réservoir. Mais si chacun des m étages a son propre réservoir, cela nécessite réservoirs. Ceci induit un coût d'installation très élevé pour rien. Il est avantageux de regrouper les étages en blocs pour les alimenter avec le même réservoir. Soit η le nombre d'étages alimentés par réservoir (par exemple $\eta = 3$ sur la figure 5.3). $k = \frac{m}{\eta}$ est le nombre de réservoirs devant alimenter les m étages du bâtiment. Par conséquent, chaque réservoir T_i approvisionnera directement les étages $\eta \times (i-1)$, $\eta \times (i-1)+1$, $\eta \times (i-1)+2 \ldots$, $\eta \times (i-1)+\eta-1 = \eta \times i-1$. Cela évite le gaspillage d'argent et d'énergie, car l'eau présente dans ce réservoir peut probablement avoir été pompée depuis les réservoirs situés en bas.

Il serait possible (en cas de besoin) de faire passer de l'eau d'un réservoir à un autre. Si le réservoir à partir duquel l'eau est acheminée vers l'autre est au dessus, aucune énergie n'est nécessaire.

Pour résumer, nous présentons sur la figure 5.3 une architecture générique ayant m étages approvisionnées par k réservoirs avec $k-1$ pompes de surface associées aux réservoirs et une pompe immergée pour le forage. Sur cette figure, il y a $\forall i \in \{1,2,3,\ldots,k\}$:

- VR_i est la vanne qui ouvre l'eau de pluie du tuyau d'eau de pluie au réservoir T_i.
- VF_i est la vanne qui ouvre l'eau du réservoir T_i aux étages qui lui sont associés.
- VU_i est la vanne qui ouvre l'eau du tuyau du réseau urbain de distribution aux étages associés au réservoir T_i.
- SP_i est la pompe de surface qui pompe l'eau du réservoir T_i au réservoir T_{i+1} ($i \leq k-1$).

Il y a aussi les vannes suivantes:

- VR est la vanne permettant d'évacuer le surplus d'eau de pluie lorsque tous les réservoirs sont pleins.
- VU est la vanne qui conduit l'eau du réseau de distribution urbain jusqu'aux étages. Il ferme chaque fois que le niveau d'eau dans tous les k réservoirs est au dessus du niveau bas ($bT_i = 1$).

Certains capteurs sont également utilisés pour lire et envoyer des informations au contrôleur:

- Le capteur *rain* indique qu'il pleut. Quand il pleut, $rain = 1$ et tous les mécanismes de pompage s'arrêtent.
- Le capteur pU indique que le réseau d'approvisionnement urbain est alimenté en eau ($pU = 1$).
- Le capteur bWD immergé dans le puits/forage informe le contrôleur qu'il y a de l'eau disponible à l'intérieur pour être pompée par la pompe immergée IP et envoyée au réservoir T_1.

5.2.3.2 Approvisionnement des réservoirs et contraintes d'énergie

Que l'énergie de pompage soit à coût zéro ou non, il est impératif d'éviter tout gaspillage. Les réservoirs et les pompes doivent alors être positionnées de manière à pouvoir fonctionner en évitant des pertes d'énergie potentielles. Ici, nous évaluons cette énergie afin de justifier les mécanismes de pompage choisis.

Soit $E_{i,j}$ la quantité d'énergie nécessaire pour approvisionner un réservoir quelconque T_j avec de l'eau du réservoir T_i, $i,j \in \{1,2,\ldots,k\}$ et $i \neq j$ à l'aide d'un volume d'eau $V(m^3)$. En considérant que hi est la hauteur entre deux réservoirs successifs et que ht est la hauteur de chaque réservoir (comme le montrent les figures 5.1 et 5.3),

Si $i > j$, alors le volume d'eau V peut être envoyé directement du réservoir T_i au réservoir T_j sans pompage. Alors $E_{i,j} = 0$.

Si par contre $i < j$, pour déplacer le volume d'eau V du réservoir T_i au réservoir T_j directement, la hauteur totale est de $(j-i)(ht+hi)+ht$ et par conséquent $E_{i,j}(1) = \rho g V[(j-i)(ht+hi)+ht]$. Nous avons alors l'équation 5.1:

$$E_{i,j}(1) = \rho \times g \times (j-i)(ht+hi)+ht \tag{5.1}$$

Où $\rho(Kg/m^3)$ est la masse volumique de l'eau, et $g(N/Kg)$ est la force de pesanteur de la terre dans ce milieu.

Lorsque $i < j$, si le même volume d'eau V est déplacé du réservoir T_i au réservoir T_j en passant par les réservoirs T_{i+1}, T_{i+2}, $\ldots T_{j-1}$, alors $E_{i,j}(2) = E_{i,i+1} + E_{i+1,i+2} + \ldots + E_{j-1,j} = \sum_{l=i}^{j-1} E_{l,l+1}$. $E_{l,l+1} = \rho g V(2ht+hi)$,

qui est constant. Ainsi, $E_{i,j}(2) = \sum_{l=i}^{j-1} \rho g V(2ht + hi) = \rho g V(j-i)(2ht + hi)$ et on obtient l'équation 5.2:

$$E_{i,j}(2) = \rho \times g \times (j-i)(2ht + hi) \tag{5.2}$$

Le surplus d'énergie $\rho g V(j-i)ht$ est dû au fait que l'eau passe d'abord par les réservoirs intermédiaires avant d'arriver dans le réservoir de destination situé au dessus d'eux. On évalue la différence entre ces deux quantités d'énergie, ce qui donne $\rho \times g \times V[(j-i)(2ht + hi) - (j-i)(ht + hi) - ht]$. Ainsi, on a l'équation 5.3.

$$E_{i,j}(2) - E_{i,j}(1) = \rho \times g \times V(j-i-1)ht \tag{5.3}$$

En résumé, lorsque $i < j$, puisque $\rho g V(j-i-1)ht \geq 0$, $E_{i,j}(2) - E_{i,j}(1) \geq 0$. Cela montre qu'il est avantageux de déplacer l'eau directement du réservoir T_i au réservoir T_j sans passer par des réservoirs intermédiaires. Mais, cet avantage est apparent, car la réalisation d'une telle architecture où les réservoirs sont reliés deux-à-deux est très difficile pour les raisons suivantes :

- **Coûteux et encombrant**: cela nécessiterait un nombre important lignes d'eau différentes équipées chacune d'une pompe : du réservoir $T_i$au réservoir T_{i+1}, ..., T_{j-1}, T_j. Pour tout l'immeuble, il faut au total $1 + 2 + \ldots + (k-1) = \frac{(k-1)k}{2} = C_k^2$ lignes d'eau. Tout ceci est encombrant et onéreux.
- **On ne connaît pas l'étage où l'eau sera effectivement utilisée**: le volume V d'eau pourrait être utilisé dans n'importe quel étage de l'immeuble, excepté ceux approvisionnés par le réservoir T_j.

On conclut qu'il faut adopter une autre stratégie, par exemple faire pomper (en cs de besoin) de l'eau étape par étape du réservoir T_i au réservoir T_{i+1}, $i = 1,2,\ldots$

5.2.3.3 Le pompage de l'eau étape par étape

Ici, nous montrons qu'il est préférable de pomper l'eau des réservoirs vers les réservoirs à partir du forage/puits, plutôt que de la pomper vers le réservoir le plus élevé avant de la redistribuer aux étages.

Le cas de base avec deux réservoirs Ayant deux ($k = 2$) réservoirs T_1 et T_2, il y a deux processus de pompage possibles: l'eau est pompée directement du puis/forage jusqu'au réservoir T_2 (Cas 1), ou bien elle est pompée du puits/forage au réservoir T_1 avant d'être déplacée pour le réservoir T_2 (Case 2) à travers une pompe de surface.

Nous réalisons comme suit des calculs sur la quantité d'énergie requise pour pomper un volume $V(m^3)$ d'eau du puits/forage vers les étages. Nous considérons que le volume V d'eau est réparti en $V = V_1 + V_2$, où V_1 est le

volume d'eau utilisé pour approvisionner les étages inférieurs tandis que V_2 est le volume utilisé pour approvisionner les étages supérieurs.

Cas 1: l'eau est pompée directement vers T_2

L'eau est pompée directement du forage/puits vers le réservoir T_2 à partir duquel tout l'immeuble est alimentée, i.e. les étages supérieurs et ceux inférieurs. E_1 est la quantité d'énergie nécessaire pour faire monter cette eau et s'exprime comme suit : $E_1 = \rho \times g(hw + 2ht + hi)(V_1 + V_2)$, alors:

$$E_1 = \rho \times g[(hw + ht)(V_1 + V_2) + (ht + hi)(V_1 + V_2)] \tag{5.4}$$

Cas 2: l'eau est pompée vers T_1 avant d'être remonté vers T_2

En présence de deux(02) réservoirs, l'eau est pompée du puits/forage vers le réservoir bas T_1 (approvisionnant les étages à l'aide de la vanne V_1) et de T_1 au réservoir T_2 (pour approvisionner les étages supériuers à travers la vanne V_2). E_2 est la quantité d'énergie requise pour cela et donné par $E_2 = \rho \times g[(hw + ht)(V_1 + V_2) + (2ht + hi)V_2]$; alors :

$$E_2 = \rho \times g[(hw + ht)(V_1 + V_2) + (2ht + hi)V_2] \tag{5.5}$$

Où hwest la hauteur entre le puits/forage et le réservoir T_1.

Calcul du gain d'énergie

La différence d'énergie $E_1 - E_2$ est donnée par

$$E_1 - E_2 = \rho \times g[(hw + hi)V_1 - ht \times V_2] = \rho \times g[hi \times V_1 + ht(V_1 - V_2)] \tag{5.6}$$

En pratique, $ht \ll hi$. Il serait possible par exemple d'avoir $hi = 6 \times ht$.

Si $V_1 = V_2 = \frac{V}{2}$ alors $E_1 - E_2 = \rho \times g \times hi \times \frac{V}{2} > 0$, ce qui montre qu'il est avantageux lorsque l'eau est pompée en deux étapes; du puits/forage vers T_1 et ensuite de T_1 à T_2.

<u>Généralisation à k réservoirs</u>

Les calculs précédents ne prennent en considération que deux(02) réservoirs. Généralisons cela à k réservoirs. Pour cela, nous considérons l'architecture générale de la figure 5.3, et $V = V1 + \ldots + V_k$, $V_i \geq 0, \forall i \in \{1,2,\ldots,k\}$. V_i est utilisé par le réservoir T_i approvisionner les étages associés numérotés $\eta \times i - j, j \in \{0,1,\ldots,\eta - 1\}$, où η est le nombre d'étages approvisionnés par chaque réservoir T_i.

Cas 1: l'eau est pompée directement jusqu'à T_k

Lorsque le volume V d'eau est pompé du puits/forage au réservoir T_k avant d'approvisionner les autres réservoirs $(T_{k-1},\ldots,T_1)$, la quantité d'énergie correspondante est E_1 et calculée comme suit :

$E_1 = \rho g \sum_{i=1}^{k} V_i[hw + ht + (k-1)(hi + ht)] = \rho g(hw + ht) \sum_{i=1}^{k} V_i + \rho g[(k-1)(hi + ht)] \sum_{i=1}^{k} V_i$

$\frac{E_1}{\rho g} = (hw + ht) \sum_{i=1}^{k} V_i + (k-1)(hi + ht) \sum_{i=1}^{k} V_i$

Cas 2: l'eau est pompée étape par étapes de T_i à T_{i+1}

E_2 est alors la quantité d'énergie nécessaire pour pomper le volume d'eau V de réservoirs en réservoirs jusqu'à T_k. Dans ce processus, on pompe uniquement le volume d'eau nécessaire $(V_{i+1}+\ldots+V_k)$ de T_i à T_{i+1}, et ainsi de suite. On a :

$E_2 = \rho \times g \times (\sum_{i=1}^{k} V_i)(hw+ht) + \rho \times g \times (\sum_{i=2}^{k} V_i)(hi+2ht) + \rho \times g \times (\sum_{i=3}^{k} V_i)(hi+2ht) + \ldots + \rho \times g \times (V_{k-1}+V_k)(hi+2ht) + \rho \times g \times V_k(hi+2ht)$

$\frac{E_2}{\rho g} = (hw+ht)\sum_{i=1}^{k} V_i + (hi+2ht)\sum_{i=2}^{k} V_i + (hi+2ht)\sum_{i=3}^{k} V_i + \ldots + (hi+2ht)(V_{k-1}+V_k) + (hi+2ht)V_k \Rightarrow \frac{E_2}{\rho g} = (hw+ht)\sum_{i=1}^{k} V_i + (hi+2ht)(0V_1 + 1V_2 + \ldots + (k-1)V_k)$

Alors,

$$\frac{E_2}{\rho g} = (hw+ht)\sum_{i=1}^{k} V_i + (hi+2ht)\sum_{i=1}^{k}(i-1)V_i \tag{5.7}$$

Gain d'énergie résultant

La différence entre $E_1 - E_2$ est donnée par:

$\frac{E_1-E_2}{\rho g} = (k-1)(hi+ht)\sum_{i=1}^{k} V_i - (hi+2ht)\sum_{i=1}^{k}(i-1)V_i$

En réorganisant cette expression en termes de facteurs de hi et ht, on a l'équation 5.8.

$$\frac{E_1-E_2}{\rho g} = hi[(k-1)\sum_{i=1}^{k} V_i - \sum_{i=1}^{k}(i-1)V_i] + ht[(k-1)\sum_{i=1}^{k} V_i - 2\sum_{i=1}^{k}(i-1)V_i] \tag{5.8}$$

Ainsi, on obtient

$$\frac{E_1-E_2}{\rho g} = hi\sum_{i=1}^{k}(k-i)V_i + ht\sum_{i=1}^{k}(k-2i+1)V_i \tag{5.9}$$

Signe de $E_1 - E_2$: considérant que $k > 1$

$\sum_{i=1}^{k}(k-i)V_i > 0$ et $\sum_{i=1}^{k}(k-2i+1)V_i$ pourrait être négatif dans certains cas. Mais en considérant le fait que dans la réalité $ht < hi$ (avec possibilité d'avoir $ht = \frac{hi}{6}$), on peut démontrer que $|\sum_{i=1}^{k}(k-i)V_i| > |\sum_{i=1}^{k}(k-2i+1)V_i|$ et que $E_2 - E_1 > 0$.

Le cas simple est celui où le volume d'eau qui arrive dans les étages est une répartition équitable de V, i.e. $V_1 = V_2 = \ldots = V_k = \frac{V}{k} = V_0$. On a pour ce cas

$$\frac{E_1-E_2}{\rho g V_0} = hi\sum_{i=1}^{k}(k-i) + ht\sum_{i=1}^{k}(k-2i+1) \tag{5.10}$$

Mais, $\sum_{i=1}^{k}(k-i) = \frac{k(k-1)}{2}$ et $\sum_{i=1}^{k}(k-2i+1) = 0$, alors $\frac{E_1-E_2}{\rho g} = \frac{k(k-1)}{2} hi \times \frac{V}{k}$.

En d'autres termes,

$$E_1 - E_2 = \frac{(k-1)}{2}\rho \times g \times hi \times V > 0 \tag{5.11}$$

En résumé, l'usage d'un seul réservoir conduit à négliger l'eau de pluie et à gaspiller l'énergie de pompage, puisque beaucoup d'eau monte d'abord avant de descendre pour approvisionner les étages situés en bas. Il est nécessaire d'utiliser au moins $k = 2$ réservoirs. Pour gagner en énergie, l'eau doit être pompée du puits/forage au réservoir le plus bas T_1 à l'aide de la pompe immergée, et pompée de chaque réservoir T_i au réservoir T_{i+1} $(1 \leq i \leq k-1)$, en utilisant des pompes de surface. Aussi, l'ordre de collecte de l'eau de pluie dans les réservoirs doit être: d'abord T_k, ensuite T_{k-1}, $T_{k-2} \ldots T_1$ pour garantir un gain maximal en énergie potentielle. Cette eau n'est pas descendue avant d'être remontée par la suite, ce qui serait source de gaspillage.

5.3 La commutation des sources d'énergie

Dans un premier temps, nous proposons une modélisation de la commutation des sources d'énergie, puis dans un second temps une formalisation Grafcet de ce processus de commutation.

5.3.1 Modélisation

Lors de la modélisation de l'alimentation en eau des réservoirs et des étages de l'immeuble, le processus de commutation entre les sources d'eau est géré en ouvrant/fermant les vannes correspondantes. Chaque pompe pouvant fonctionner avec de nombreuses sources d'énergie, il convient également de proposer un mécanisme de commutation entre diverses sources d'énergie. Pour cela, nous proposons d'utiliser un autre composant électronique commandable/contrôlable numériquement, qui est le multiplexeur.

Avec p sources d'énergie disponibles pour une pompe, nous les rangeons de la moins coûteuse à la plus coûteuse : E_1,E_2,...,E_p. Nous associons alors une variable pE_Z (présence d'énergie à la source E_Z) pour chaque source d'énergie E_Z. $pE_Z = 1$ ($Z \in \{1,2,\ldots,p\}$) lorsque la source E_Z possède de l'énergie[1].

Un mécanisme implémenté par multiplexage peut être mis en place à cet effet. Le multiplexeur utilisé permettrait donc de sélectionner entre les p sources d'énergie disponibles celle E_Z qui est la moins coûteuse pour être connectée sur la sortie E_{out} du multiplexeur.

1. Il existe des capteurs de présence d'énergie à cet effet

TAB. 5.1: Table de vérité d'un multiplexeur

Adresses: $Z = (Z_{\lambda-1},Z_{\lambda-2} \ldots X_1,Z_0)$					$E_{out} = E_Z$
$MX_{\lambda-1}$	$MX_{\lambda-2}$	...	MX_1	MX_0	Output
0	0	...	0	0	*None*
0	0	...	0	1	E_1
...	...	...	...	...	...
1	1	...	1	1	E_p

Soit $\lambda = \lceil \log_2 p \rceil$, tel que $p < 2^\lambda$, le nombre de bits sur lesquels on code p. En effet, $p < 2^\lambda$ car il devrait y avoir parmi les sources d'énergie en entrée une qui soit vide, i.e. qui débite 0V; la sélection de celle-là signifie qu'aucune source d'énergie n'est sélectionnée.

Nous définissons les variables binaires MX_0,$MX_1 \ldots$,$MX_{\lambda-1}$ utilisées en entrée du multiplexeur MX pour la sélection entre des sources d'énergie E_1,E_2,...,E_p, tel que présenté en figure 5.4.

Pour sélectionner une source particulière d'énergie E_Z (i.e. pour connecter E_Z à E_{out}), $\forall Z \in \{1,2,\ldots,p\}$ on écrit $Z = (Z_{\lambda-1},Z_{\lambda-2},\ldots,Z_0)$, avec $(Z_{\lambda-1},Z_{\lambda-2},\ldots,Z_0)$ la représentation binaire de Z.

Posons $(MX_{\lambda-1},MX_{\lambda-2},\ldots,MX_0) = (Z_{\lambda-1},Z_{\lambda-2},\ldots,Z_0)$. Ainsi, on définit les actions du contrôleur MX_Z ($Z \in \{0,1,2,\ldots,\lambda-1\}$) comme étant les adresses d'entrée du multiplexeur.

Lorsque la combinaison des valeurs binaires en entrée $(MX_{\lambda-1},MX_{\lambda-2},\ldots,MX_0)$ correspond à une valeur décimale quelconque Z, c'est la source d'énergie E_Z qui est sélectionnée et connectée sur la sortie E_{out} du multiplexeur.

La table 5.1 est la table de vérité de ce mécanisme.

Aussi longtemps qu'il y a parmi les sources d'énergie au moins une source approvisionnée, E_{out} reste approvisionnée jusqu'à ce qu'aucune de ces source ne possède de l'énergie. De même, lorsque E_{out} n'est pas approvisionnée, elle le devient aussitôt qu'une source d'énergie devient disponible. L'énergie est fournie indépendamment du fait que E_{out} est utilisée ou pas par une pompe.

5.3.2 Spécification Grafcet de la commutation des sources d'énergie

Le mécanisme décrit précédemment peut être spécifié à l'aide d'un Grafcet, puisque le multiplexeur possède des entrées et des sorties. Le grafcet obtenu est donné en figure 5.5. Les adresses d'entrée du multiplexeur $MX_0 \ldots MX_{p-1}$ constituent les sorties du Grafcet, puisque ce sont celles-ci qui permettent de sélectionner une source bien précise. Les signaux d'entrée du modèle Grafcet sont pour leur part des signaux indiquant la présence ou pas d'énergie au niveau des sources d'énergie en présence: $pE_1 \ldots pE_p$.

Ici, nous avons considéré que toutes les pompes (de surface et immergée) sont approvisionnées avec la même source d'énergie. Toutefois, il est

possible d'avoir plusieurs types de sources d'énergie de pompage, selon la tension débitée : 24vols, 220vols, ... Ainsi, les sources peuvent être réorganisées en groupes et un mécanisme de commutation est alors développé et appliqué à chaque groupe.

5.4 Spécification Grafcet du système

En plus de la spécification Grafcet de la commutation des sources d'énergie déjà présentée en 5.3.2 (figure 5.5), nous présentons ici la spécification Grafcet du fonctionnement général du système. Nous nous focaliserons par après sur un cas spécifique où $k = 1$.

5.4.1 Grafcet général du système

Ici, nous proposons certains modèles Grafcet généraux pour la spécification des processus étudiés dans le cadre de l'approvisionnement en eau de l'immeuble. Ils constituent le modèle du contrôleur dont l'architecture générale a été présentée en figure 5.3. La spécification entrées/sorties du contrôleur spécifique aux réservoirs intermédiaires a été donnée en figure 5.2, ce qui représente une partie de la spécification I/O du contrôleur entier du système d'approvisionnement en eau domestique. Ce système gère en plus l'approvisionnement en eau des réservoirs et des étages suivant la hiérarchie multi-niveaux des priorités d'accès à l'eau et le processus de commutation entre les sources d'eau et d'énergie.

5.4.1.1 Spécification Grafcet de l'approvisionnement des réservoirs

Conformément à la description de l'approvisionnement en eau donnée en 5.2.2, il y a trois cas de réservoirs:

- le réservoir le plus bas T_1 : l'eau est envoyée dans ce réservoir à travers la pompe immergée IP.
- le réservoir le plus haut T_k : l'eau y est envoyée à travers une pompe de surface SP_{k-1}. De plus, ce réservoir ne peut recevoir d'eau de pluie directement.
- Les autres réservoirs T_i $(2 \leq i \leq k)$: l'eau est pompée dans les mêmes conditions que le réservoir T_k, et le remplissage de T_i avec l'eau de pluie est conditionnée par le fait que T_{i+1} soit d'abord plein $(hT_{i+1} = 1)$.

Sur les réceptivités des transitions du Grafcet, nous utilisons la notation "!" pour indiquer l'opérateur logique de négation.

L'approvisionnement en eau des réservoirs est bien spécifié lors de la modélisation du processus de commutation présenté en section 5.3.2. Ce processus peut être décrit à l'aide du modèle Grafcet de la figure 5.6. Les variables $ppM1$ et $ppM2$ indiquent la présence de l'énergie de pompage.

$ppM1 = srl + bcl$ concerne le premier mode inhérent au pompage avec une énergie de coût nul (issu du soleil ou du stockage de l'énergie à l'intérieur des batteries) et la variable $ppM2 = \overline{srl + bcl} * pUES = \overline{srl} * \overline{bcl} * pUES$ est liée au pompage en utilisant une source d'énergie coûteuse, où $pUES$ informe sur la présence de d'énergie électrique issue du réseau de distribution urbain.

La figure 5.6.b présente l'approvisionnement de n'importe quel réservoir intermédiaire T_i: Quand T_i a besoin d'eau ($mT_i = 0$) et qu'il pleut ($rain = 1$), la vanne VR_{i+1} est ouverte (au cas où T_{i+1} est déjà plein) et le réservoir T_i est approvisionnée jusqu'à ce qu'il soit plein ($hT_i = 1$). S'il n'y a pas d'eau de pluie ($rain = 0$), la seconde alternative est utilisée lorsque le réservoir T_{i-1} est au moins à moitié rempli ($mT_{i-1} = 1$) et qu'il y a de l'énergie disponible pour faire fonctionner la pompe de surface SP_i. L'action SP_{i-1} (pour mettre en marche la pompe de surface associée) est alors mise à 1 pour pomper de l'eau du réservoir T_{i-1} au réservoir T_i.

Il en est de même avec T_1 (Figure 5.6.a) excepté que l'eau est pompée de la pompe immergée IP, dont le fonctionnement est aussi conditionné par le signal bWD indiquant le niveau d'eau du puits/forage. Concernant T_k, lorsqu'il pleut, l'ouverture de la vanne VR_k n'est pas sujette aux conditions hT_{k+1} (le réservoir T_{k+1} n'existe pas), mais dépend uniquement de la condition $rain * !mT_k$ (Figure 5.6.c).

Pour gérer le fait que lorsqu'il pleut, les réservoirs sont remplis à partir du plus haut au plus bas, nous considérons que le réservoir T_k est rempli avec l'eau de pluie sous la condition $!mTk * rain$ (réceptivité de la transition qui va de l'étape 9 à l'étape 10); et $\forall T_i, i \in \{1, \ldots, k-1\}$ la condition pour remplir le réservoir T_i avec l'eau de pluie est $!mT_i * mT_{i+1} * rain$ (réceptivité de la transition allant de l'étape 5 à l'étape 6), puisque T_{i+1} doit être rempli en premier ($hT_{i+1} = 1$).

En plus, l'action VR qui ouvre la vanne correspondante permet l'évacuation du surplus d'eau de pluie qui coule lorsque tous les réservoirs sont déjà remplis ($hT_1 * hT_2 * \ldots * hT_k * rain = 1$). Cette vanne s'arrête lorsqu'il existe un réservoir T_j dont le niveau d'eau est en deçà du niveau moyen ($mT_j = 0$), ou bien qu'il n'y a plus de pluie ($rain = 0$ ou bien $!rain$). Dans ce cas, le processus de remplissage de T_j commence automatiquement. Pour éviter des dommages entre le moment où tous les réservoirs sont pleins et lorsqu'au moins un réservoir a un niveau d'eau inférieur au niveau moyen, des sorties d'échappement peuvent être placées sur le réservoir le plus en bas. Le processus d'ouverture et de fermeture de la vanne VR est alors spécifié à l'aide du modèle Grafcet de la figure 5.7.

5.4.1.2 Modélisation Grafcet de l'approvisionnement en eau des étages

Chaque étage est attachée à un réservoir T_i à partir duquel il est approvisionné en eau.

La vanne VF_i doit rester ouverte tant qu'il y a de l'eau à l'intérieur

du réservoir T_i ($bT_i = 1$). VF_i est fermé aussitôt que le réservoir est vide ($bT_i = 0$), auquel cas le réseau urbain de distribution prend la relève à travers l'ouverture de la vanne VU_i, lorsqu'il est approvisionné ($pU = 1$). VU_i est fermé aussitôt que le réservoir T_i devient approvisionné de nouveau ($bT_i = 1$), ou bien lorsque le réseau urbain de distribution devient non alimenté ($pU = 0$). Les conditions bT_i, $!bT_i$, $!bT_i * pU$ et $bT_i + !pU$ présentes dans le modèle Grafcet de spécification de la figure 5.8 sont alors justifiées.

5.4.2 Grafcet du cas spécifique ($k = 1$)

Dans cette sous-section, nous nous restreignons au cas spécifique du système pour $k = 1$. L'architecture physique du système obtenu est celle de la figure 5.9 où il est aussi présenté les sources d'énergie électriques dédiées aux différentes pompes.

Nous rappelons pour ce cas spécifique les principales caractéristiques, une spécification entrées/sortie et une spécification Grafcet du contrôleur à réaliser.

5.4.2.1 Principales caractéristiques du système

La figure 5.9 illustre le système de distribution d'eau pour six niveaux d'un immeuble, et constitué de: trois ($n = 3$) sources d'eau, cinq ($p = 5$) sources d'énergie de pompage, de nombreuses vannes d'accès à l'eau avec des niveaux de priorité différents, le système de contrôle, en plus de deux ($k = 2$) réservoirs d'eau (T_1 et T_2).

Comme décrit dans le cas général (section 5.2.2), nous considérons:

Trois sources d'eau:

- L'eau du réseau urbain de distribution,
- L'eau de forage/puits, et
- L'eau issue d'une source d'eau de pluie.

Au total cinq (05) sources d'énergie de pompage, organisées en deux groupes, sont utilisées pour les deux pompes, et classées du moins coûteux au plus coûteux.

Sources d'énergie pour la pompe de surface SP:

- La source E_1 ou PVS (notée E_{PVS}): de 24 volts avec énergie solaire. Sa présence est indiquée par le capteur d'énergie électrique srl.
- La source E_2 ou BS (notée E_{BS}): de 24 volts avec batteries, chargeable avec de l'énergie solaire. Le capteur d'énergie électrique bcl indique sa disponibilité.
- La source E_3 (notée E_{TRANS}): issue de la transformation de 220 volts AC à 24 volts DC, à partir de la source **E5**. Le capteur d'énergie électrique $pUES$ indique sa présence.

Sources d'énergie pour la pompe immergée IP:

- La source E_4 (notée E_{INV}): constituée d'un générateur inverseur et

débitant une tension de 220 volts. Sa disponibilité est connue grâce au capteur d'énergie électrique *bcl*.

– La source E_5 ou bien UES (notée E_{UES}): il s'agit de la source du réseau électrique public, débitant 220 volts. Sa présence est connue grâce au capteur d'énergie électrique $pUES$.

Mécanismes de pompage:

– **La pompe de surface** SP fonctionnant sous une tension de 24 volts DC; l'alimentation est fournie soit directement par des panneaux solaires E_{PVS}, ou par batterie solaire chargeable E_{BS} ou encore par la sortie du transformateur E_{TRANS}. La pompe SP pompe de l'eau du réservoir T_1 au réservoir T_2.

– **La pompe immergée** IP fonctionnant avec une tension d'entrée de 220 volts; l'alimentation est fournie soit par la sortie du générateur inverseur E_{INV} (issu des batteries de stockage) ou par le réseau de distribution urbain E_{UES}. **IP** pompe l'eau de forage/puits au réservoir T_2.

Vannes d'accès à l'eau: VU, VR, VR_1, VF_1, VF_2, VU_1 and VT_2.

L'architecture de cette étude de cas fonctionne selon la description détaillée présentée à la section 5.2.2 où on retrouve le stockage de l'eau dans les réservoirs et les moyens d'acheminer l'eau des sources aux réservoirs.

Il a été présenté le processus de commutation entre les sources d'alimentation et les vannes. Comme nous avons deux pompes avec des sources d'énergie différentes, nous concevons à cet effet deux multiplexeurs différents (comme présentés en section 5.3), les sorties des pompes SP et IP étant respectivement E_{SP} et E_{IP}.

La **pompe de surface** SP étant alimentée par les sources suivantes d'énergie E_{PVS}, E_{BS} et E_{TRANS}, la sélection de la source souhaitée parmi elles se fait au moyen du multiplexeur de commutation MX_{SP} dont les entrées sont MX_{SP_0} et MX_{SP_1}.

Quant à la **pompe immergée** IP approvisionnée par les sources d'énergie E_{INV} et E_{UES}, la sélection de la source souhaitée se fait au moyen du multiplexeur de commutation MX_{IP} dont les entrées sont MX_{IP_0} et MX_{IP_1}. Le fonctionnement de ces deux multiplexeurs (MX_{SP} et MX_{IP}) est décrit dans le tableau 5.2.

MX_{SP} (Pour la pompe SP)			MX_{IP} (Pour la pompe IP)		
MX_{SP_1}	MX_{SP_0}	$MX_{SP_{out}}$	MX_{IP_1}	MX_{IP_0}	$MX_{IP_{out}}$
0	0	**None**	0	0	**None**
1	0	E_{PVS}	1	0	E_{INV}
0	1	E_{BS}	0	1	E_{UES}
1	1	E_{TRANS}	0	0	**None**

TAB. 5.2: Table de vérité des multiplexeurs MX_{SP} et MX_{IP}

5.4.2.2 Spécification du contrôleur logique

Le système de distribution d'eau est entièrement contrôlé par un système de contrôle intelligent (un contrôleur logique). Il a sa propre alimentation en énergie qui la rend autonome.

Les points d'entrée du contrôleur sont connectés à des capteurs qui permettent de rapporter des événements et de transmettre ces informations sous forme de signaux au contrôleur. Les douze (12) capteurs en entrée et à valeurs logiques utilisés dans notre système sont résumés comme suit:

- **Les capteurs de niveau d'eau** : bT_1,mT_1,hT_1,bT_2,mT_2,hT_2, bWD.
- **Les capteur de pression**: pU,$rain$.
- **Capteurs d'énergie électrique**: srl, bcl et $pUES$ ($pE_1 = srl$,$pE_2 = bcl$,$pE_4 = bcl$, $pE_5 = pUES$).

Le contrôleur devra alors lire, analyser les signaux d'entrée et activer les actions à activer, conformément au cycle de scrutation du μC. Ces actions générées sont interprétées comme des signaux de sortie transmis à des dispositifs de sortie (tels que des actionneurs et des relais) afin de contrôler le système de circulation d'eau. Les 14 sorties numériques utilisées dans notre système sont:

- **Les inducteurs** : IP,SP.
- **Les vannes à relais** : VR, VR_1, VR_2, VF_1, VF_2, VU, VU_1, VU_2.
- **Les entrées des multiplexeurs de commutation** : MX_{SP_0} et MX_{SP_1} (pour le multiplexeur MX_1 devant sélectionner entre les sources E_{PVS}, E_{BS} et E_{TRANS} pour la pompe de surface SP), MX_{IP_0} et MX_{IP_1} (pour le multiplexeur MX_{IP} devant sélectionner entre les sources E_{INV} et E_{UES} pour la pompe immergée IP).

Le modèle entrée/sortie de ce système de contrôle logique est présenté à la figure 5.10. Dans ce modèle, les sources d'énergie sont remplacées par MX_{SP_0}, MX_{SP_1}, MX_{IP_0} et MX_{IP_1} qui permettent de commander la sélection d'une des sources d'alimentation en énergie.

5.4.2.3 Architecture de déploiement de la solution de contrôle

En figure 5.11, il est présenté l'architecture du système de contrôle final, intégrant la partie de contrôle et la partie opérative de l'automatisme logique. Cette architecture est décrite comme suit:

- La partie contrôle est centrée sur le microcontrôleur utilisé, lequel reçoit des informations de la partie opérative et envoie des ordres soit aux multiplexeurs, soit aux actionneurs de la partie opérative.
- La partie opérative est constituée de tout le système contrôlé, avec pour objectif final l'approvisionnement autonome en eau domestique. On y retrouve l'immeuble dont l'architecture physique est décrite en figure 5.9 avec tous ses constituants.

Chacun des multiplexeurs MX_{SP} et MX_{IP} reçoit en entrée des ressources constituées des sources d'énergie à mettre à disposition des pompes en sortie. La sélection d'une source d'énergie appropriée selon le mécanisme mis en place est commandée par le microcontrôleur à travers les signaux MX_{SP_0}, MX_{SP_1} (pour la pompe MX_{SP}) et MX_{IP_0}, MX_{IP_1} (pour la pompe MX_{IP}). Le microcontrôleur communique avec la partie opérative en recevant des signaux sur l'état du système et en envoyant des ordres à effectuer, selon le modèle simplifié entrées/sorties décrit en figure 5.10. La place de ces multiplexeurs dans le système de contrôle est décrite sur la figure 5.11

5.4.2.4 Modèle Grafcet de spécification du système

À partir des spécifications fonctionnelles du système d'alimentation en eau décrites dans la sous-section précédente, nous déduisons le modèle de spécification Grafcet correspondant, conformément à la solution de modélisation Grafcet décrite en 5.4 pour le système générique. La spécification Grafcet du fonctionnement de ce contrôle logique est représentée sur les figures 5.12 et 5.13, décrivant chacune le modèle d'un sous-système précis. Ces grafcets sont dessinés à l'aide de l'outil UniSim [65] pour faciliter aussi la synthèse de la solution par matrices de codage (approche décrite au chapitre 2).

Les modèles Grafcet de la figure 5.12 décrivent l'approvisionnement des réservoirs, tandis que ceux de la figure 5.13 décrivent le mécanisme de commutation entre les sources d'énergie ainsi que l'ouverture/fermeture des différentes vannes.

5.4.3 Modèle EMF de la spécification Grafcet du contrôleur

Nous reprenons les grafcets du système spécifique pour annoter les connections afin d'en construire une instance dans le plug-in Éclipse EMF proposé depuis le chapitre 3. Il en résulte le modèle Grafcet présenté dans les figures 5.14 et 5.15.

Nous obtenons alors un modèle Grafcet présenté à travers l'éditeur arborescent d'EMF. Ce modèle est constitué de 22 étapes, 23 transitions, trize (13) actions distinctes. Une vue de ce modèle est donné en figure 5.16.

La figure 5.18 présente une vue de la transition $T4$ dans l'éditeur arborescent de EMF. Toutes les étapes en entrée et en sortie des transitions sont calculées automatiquement au fur et à mesure de l'édition du modèle. Pour cette transition $T4$, la figure 5.18 fait ressortir le fait qu'une seule étape $T2\_3$ est en entrée et une seule étape ($T2\_1$) est en sortie.

L'instance du modèle Grafcet construit pour ce cas d'étude est valide. Le résultat du processus de validation est présenté à la figure 5.18.

Avant de passer à la génération du code, nous décrivons le microcontrôleur utilisé et le modèle EMF qui en découle.

5.5 Modélisation du microcontrôleur utilisé

Le microcontrôleur *Atmega*328*P* ne possède pas assez de broches pour réaliser (ou mapper) toutes les entrées/sorties du modèle de cette application. En effet, bien que ce microcontrôleur soit utilisable dans certaines applications (comme par exemple celles du profilage des programmes présentées en section 2.6), il ne possède que 23 broches généralistes configurables en entrée/sortie, alors que notre modèle dispose en tout de $12 + 14 = 26$ entrées/sorties. Il est donc nécessaire de choisir un microcontrôleur ayant au moins 26 broches généralistes configurables en entrée/sortie. Pour cela, nous avons opté pour le microcontrôleur *Atmega*1280 (de la même famille que l'*Atmega*328*P*, i.e. *Atmel AVR*) qui dispose en tout de 86 broches généralistes configurables en entrée/sortie.

5.5.1 Présentation du microcontrôleur Atmega1280

Nous en donnons une présentation générale, puis nous identifions ses caractéristiques importantes pour le modèle EMF en entrée de la génération de code.

5.5.1.1 Présentation générale du microcontrôleur Atmega1280

L'Atmega1280[2] est un microcontrôleur Microchip 8 bits CMOS hautes performances, basé sur l'architecture RISC améliorée d'AVR. Il a été fabriqué par la société Atmel et est vendu actuellement à un prix qui environne les 9,5 Euro (environ 6175 FCFA). Il est de basse consommation et combine une mémoire flash ISP de 128 Ko, une mémoire SRAM de 8 Ko, une mémoire EEPROM de 4 Ko, 86 lignes (ou broches) d'E/S à usage général, 32 registres de travail à usage général, six timers ou compteurs de temps ou minuteries (deux de 8 bits et 4 de 16 bits) flexibles, 4 USART, une interface série à 2 fils orientée octet, des convertisseurs A/N 10 bits sur 16 canaux et une interface JTAG pour le débogage sur puce. Cet appareil atteint un rendement de 16 MIPS pour 16 MHz (soit environ 1 MIPS par MHz) et fonctionne entre 2,7 et 5,5 volts.

Il est intégré à de nombreuses cartes à microcontrôleur, notamment la carte *Arduino Mega*. Ce microcontrôleur peut être programmé en langage C de base ou en langage Arduino, dérivé du C et relativement haut niveau. Programmé en Arduino, les programmes sont appelés sketchs et ont l'extension *.ino*. Ce code est exploité dans l'IDE Arduino qui dispose aussi d'un compilateur, d'un débogueur et d'un programmateur via port USB (émulant le port série *RS232*).

2. https://www.microchip.com/wwwproducts/en/ATmega1280

5.5.1.2 Identification des caractéristiques pour le modèle EMF

Selon la modélisation des microcontrôleurs faites en section 4.3, le microcontrôleur Atmega1280 est caractérisé comme suit:

- Name : ATmega1280, Manufacturer: ATMEL, taille du mot mémoire : 8 bit
- 16 MHz de processeur, 8Ko de SRAM, 128Ko of Flash memory, 4Ko of EEPROM
- **86 broches programmables** :
  Port A (PA7, PA6, PA5, PA4, PA3, PA2, PA1, PA0), soit 8 broches
  Port B (PB7, PB6, PB5, PB4, PB3, PB2, PB1, PB0), soit 8 broches
  Port C (PC7, PC6, PC5, PC4, PC3, PC2, PC1, PC0), soit 8 broches
  Port D (PD7, PD6, PD5, PD4, PD3, PD2, PD1, PD0), soit 8 broches
  Port E (PE7, PE6, PE5, PE4, PE3, PE2, PE1, PE0), soit 8 broches
  Port F (PF7, PF6, PF5, PF4, PF3, PF2, PF1, PF0), soit 8 broches
  Port G (PG5, PG4, PG3, PG2, PG1, PG0), soit 6 broches
  Port H (PH7, PH6, PH5, PH4, PH3, PH2, PH1, PH0), soit 8 broches
  Port J (PJ7, PJ6, PJ5, PJ4, PJ3, PJ2, PJ1, PJ0), soit 8 broches
  Port K (PK7, PK6, PK5, PK4, PK3, PK2, PK1, PK0), soit 8 broches
  Port L (PL7, PL6, PL5, PL4, PL3, PL2, PL1, PL0), soit 8 broches
- **Caractéristiques du langage C**: Nom: Arduino, Timer: Timer 1 de 16 bits (Nous choisissons un seul Timer parmi les six disponibles),
- Opérations sur les broches:
  pinMode(pin_num, mode); pour configurer une broche avec un mode particulier (INPUT / OUTPUT),
  digitalRead(pin_num) ; pour lire une valeur digitale d'une pin donnée,
  digitalWrite(pin_num, value) ; pour écrire une valeur digitale sur une pin donnée
  analogRead(pin_num) ; pour lire une valeur analogique(numérique) d'une pin donnée,
  analogWrite(pin_num, value) ; pour écrire une valeur analogique(numérique) sur une pin donnée.
- **Configuration du Timer 1 (temporisateur 1)**

Listing 5.1:

```
Timer1.initialize(1000000/(1000/TIMER_PERIOD));
Timer1.attachInterrupt(update_G7TimingVars_callback);
```

Ceci permet de configurer le Timer 1 (16 bits) avec une période de «TIMER_PERIOD» millisecondes, lequel appellera périodiquement la fonction *update_G7TimingVars_callback* pour la mise à jour des conditions temporelles.

5.5.2 Modèle EMF du microcontrôleur Atmega1280

A partir des caractéristiques ci-dessus décrites du microcontrôleur Atmega1280, nous avons construit une instance du métamodèle microcontrôleur à l'aide de l'éditeur arborescent d'EMF généré à partir du métamodèle. Celui-ci permet d'ajouter tous les éléments du modèle et de le sauvegarder dans le format xmi, directement exploitable par d'autres outils EMF à l'instar des transformations de modèles. La figure 5.19 présente une vue de ce modèle EMF construit.

5.6 Génération du code de contrôle en Arduino et résultat

La génération de code se fait en exécutant les règles de transformation dont la description a été faite en section 4.5.2. La compilation de ce programme se passe avec succès, comme c'est visible sur la figure 5.20. Le message produit est :

«*Le croquis utilise 4 776 octets (1%) de l'espace de stockage de programmes. Le maximum est de 253 952 octets. Les variables globales utilisent 191 octets (2%) de mémoire dynamique, ce qui laisse 8 001 octets pour les variables locales. Le maximum est de 8 192 octets.*».

Dans notre cas, *Acceleo* prend en entrée deux modèles (le modèle Grafcet et le modèle microcontrôleur) pour produire en sortie du code Arduino. Comme décrit dans le processus de synthèse en section 2.4.5, lorsque le programme de contrôle généré à partir du modèle de spécification Grafcet du système d'approvisionnement en eau est compilé et envoyé au microcontrôleur Atmega1280, cette cible sert de siège à l'unité de commande intelligente. Ainsi, à chaque cycle de scrutation, il reçoit des signaux d'entrée des capteurs (présence d'eau, de pression et d'énergie électrique), les analyse et génère des signaux de sortie sous forme de commandes ou d'actions permettant d'effectuer des tâches spécifiques (ouverture/fermeture des vannes, mettre en marche/arrêter les pompes, ou encore basculer d'une source d'énergie à une autre).

5.7 Conclusion

Dans ce chapitre, nous avons présenté un cas d'étude qui décrit la possibilité d'exploiter les microcontrôleurs existants pour résoudre les problèmes critiques en cours dans les pays en développement, en particulier l'approvisionnement autonome en eau des ménages à partir de l'énergie électrique de pompage. Pour cela, nous avons conçu un système de contrôle générique et sophistiqué qui permet l'analyse et l'étude de la disponibilité de plusieurs sources d'énergie et d'eau ayant des coûts différents; et prend automatiquement une décision sur la meilleure source à sélectionner pour

assurer un service d'eau continu dans tout le bâtiment, sans interruption et à moindre coût. Les mécanismes de pompage et le processus de commutation entre les sources d'énergie sont basés sur des calculs d'énergie garantissant une réduction significative de l'énergie de pompage. En outre, l'architecture conçue pour le système de distribution d'eau présente un concept générique d'approvisionnement en eau qui peut être intégré dans un projet de construction ainsi que dans un bâtiment existant.

Le contrôleur de ce SCC a été réalisé grâce aux outils de synthèse par approche IDM développés et présentés aux chapitres 3 et 4. Lorsque le modèle de spécification Grafcet du système de contrôle et celui de la cible microcontrôleur Atmega1280 sont lus en entrée, la transformation IDM M2T est exécutée pour générer le programme en langage Arduino qui, une fois compilé et chargé dans cette carte est exécuté. Ce contrôleur peut alors lire à chaque étape les entrées et effectuer les actions nécessaires en sortie.

Pour ce cas d'étude, il ressort clairement que la synthèse par approche IDM présente de nombreux avantages par rapport à la synthèse par matrices de codage : l'absence du calcul des matrices de codage, la possibilité de vérifier la conformité du modèle au langage Grafcet et l'offre d'une grande flexibilité quant à la représentation des expressions du modèle Grafcet.

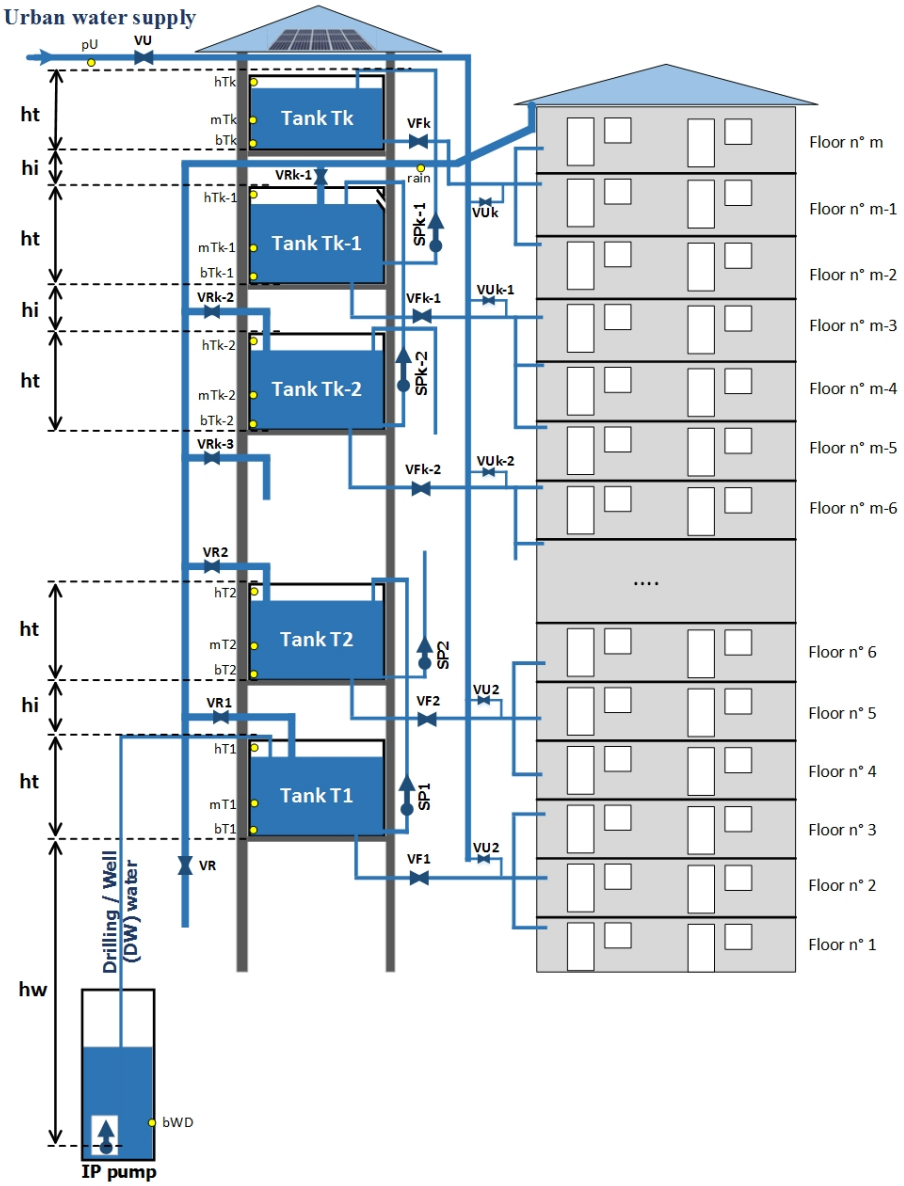


FIG. 5.3: Architecture physique du système générique

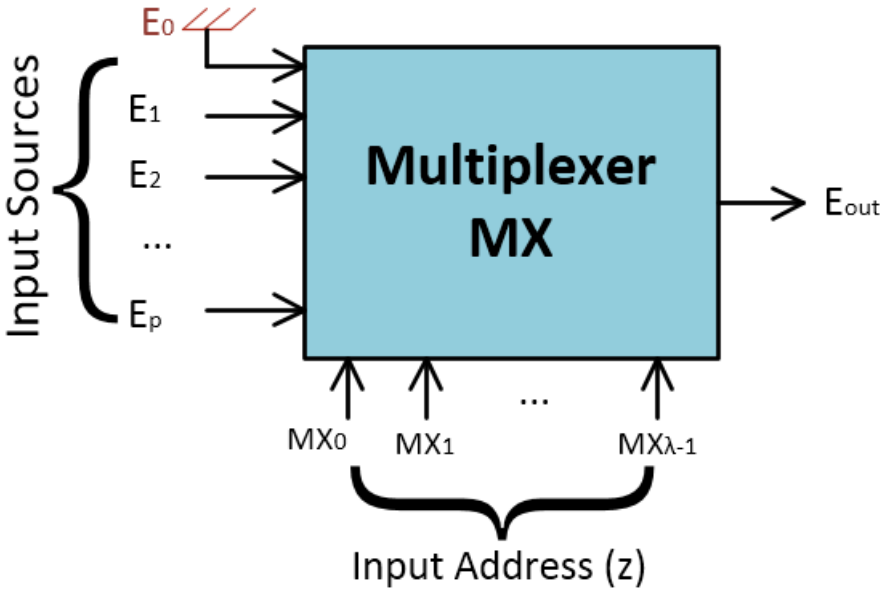


FIG. 5.4: Multiplexeur MX du processus de commutation

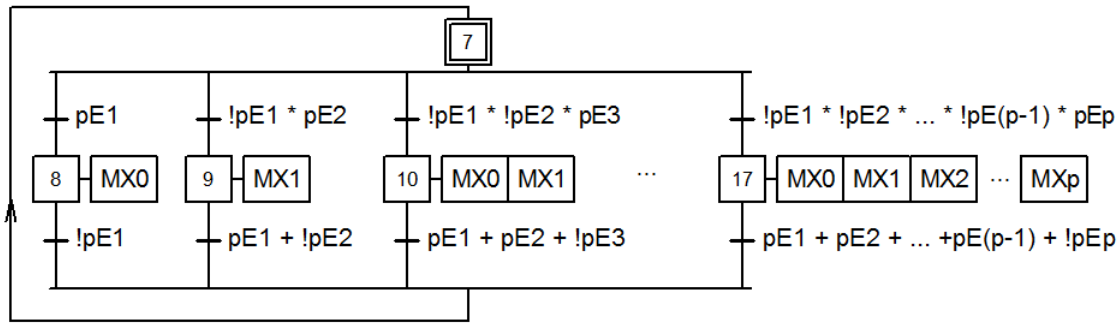
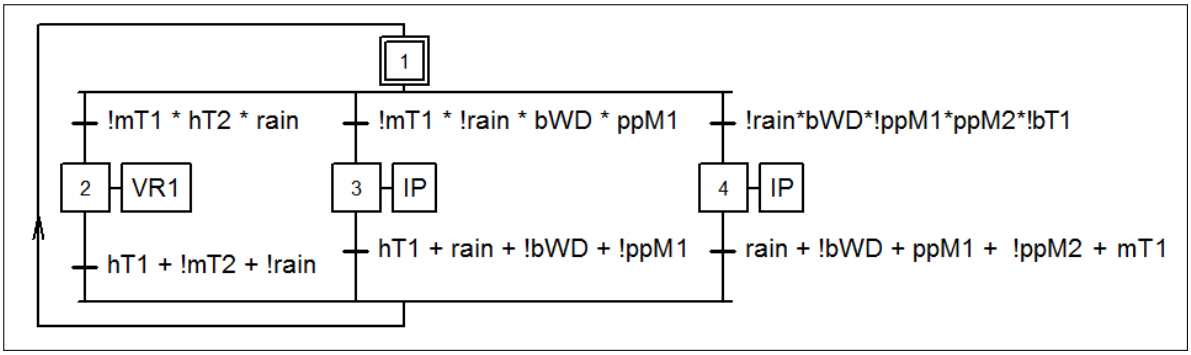
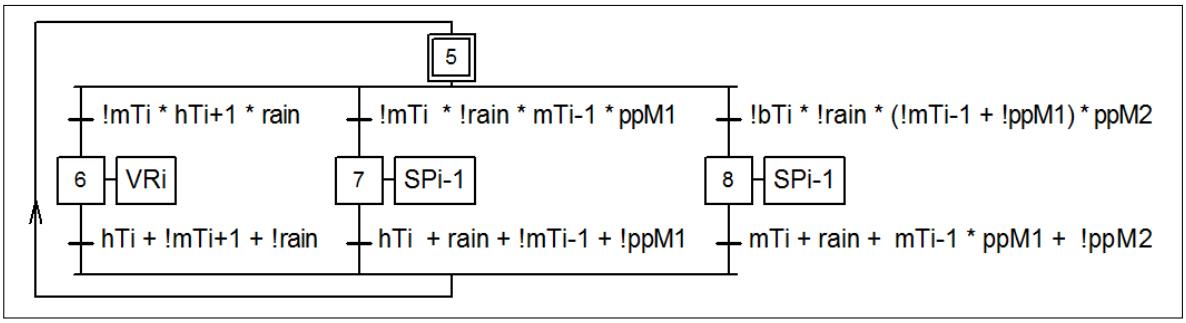


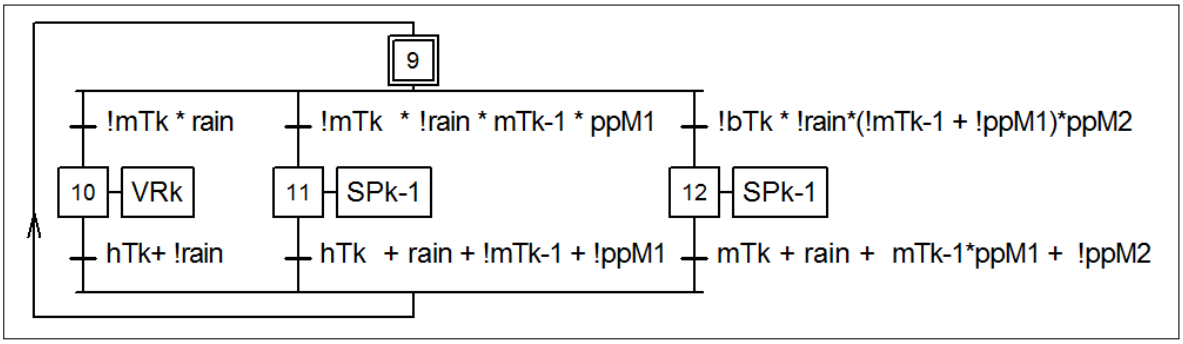
FIG. 5.5: Grafcet de la commutation des sources d'énergie



a-Grafcet pour l'approvisionnement en eau du réservoir T_1



b-Grafcet pour l'approvisionnement en eau du réservoir T_i



c-Grafcet pour l'approvisionnement en eau du réservoir T_k

FIG. 5.6: Grafcet de l'approvisionnement en eau des réservoirs

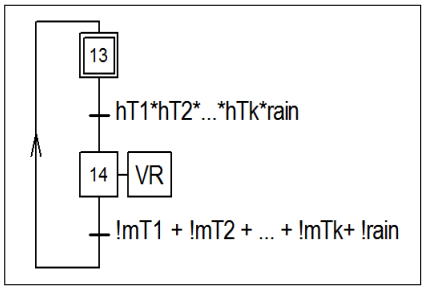


FIG. 5.7: Grafcet de spécification de l'ouverture/fermeture de la vanne d'échappement VR

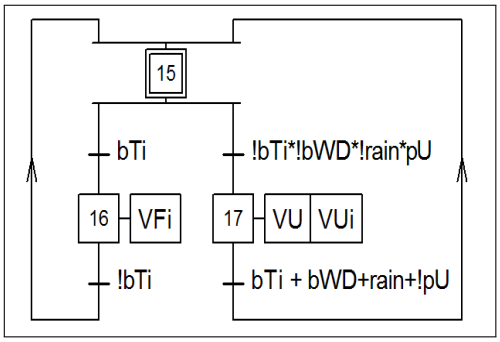


FIG. 5.8: Modèle Grafcet de l'approvisionnement en eau des étages

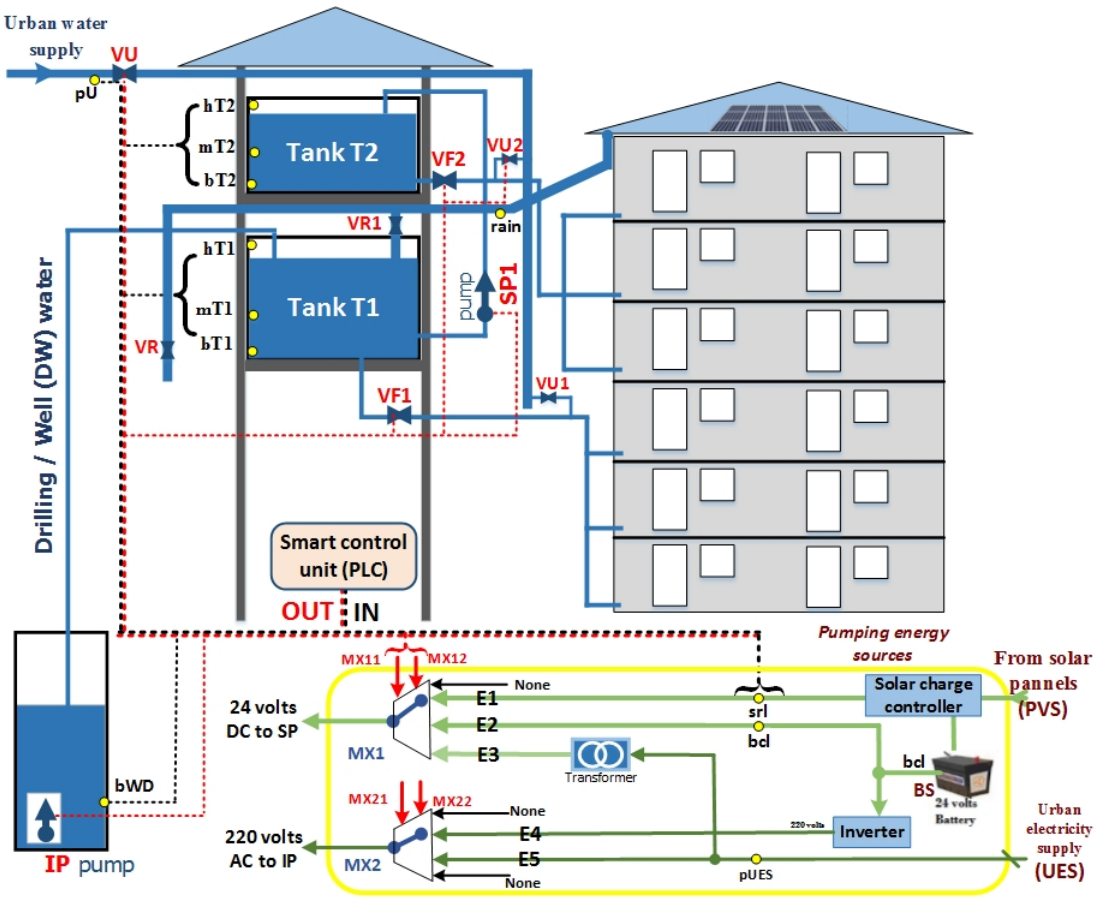


FIG. 5.9: Architecture physique du système spécifique (Cas où $k = 1$)

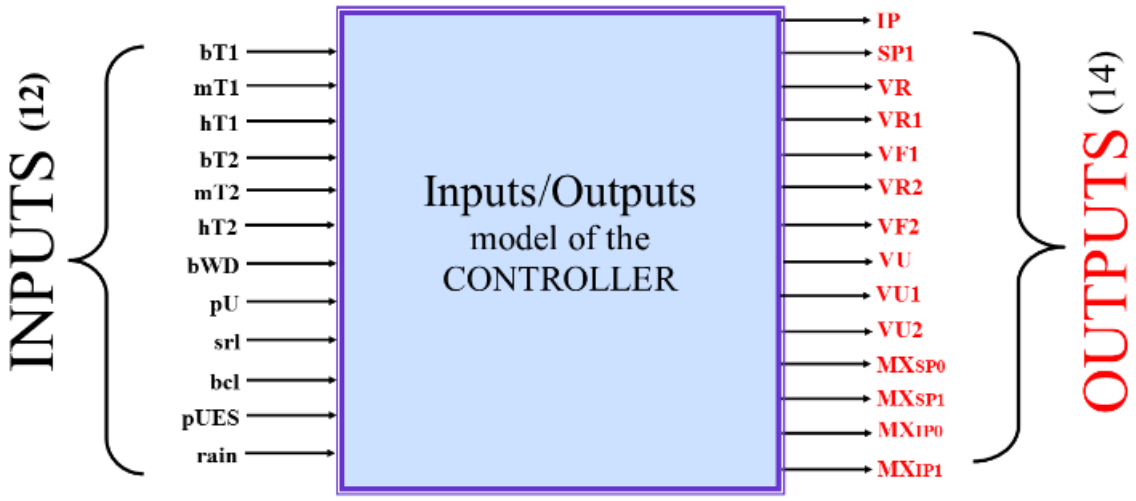


FIG. 5.10: Modèle entrée/sortie du contrôleur logique du cas d'étude

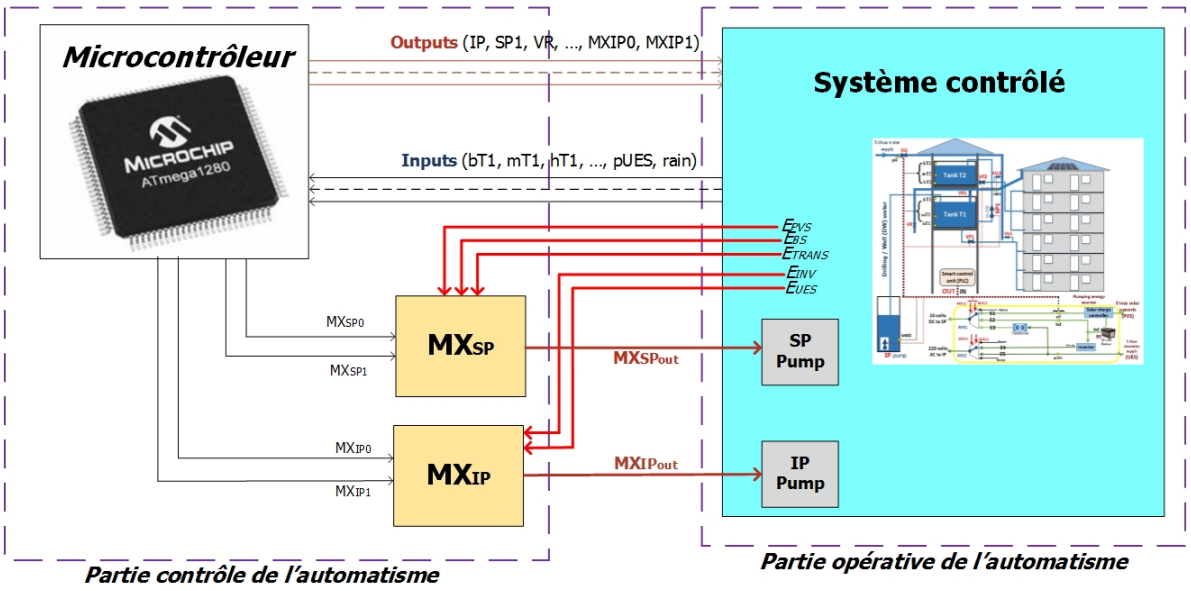
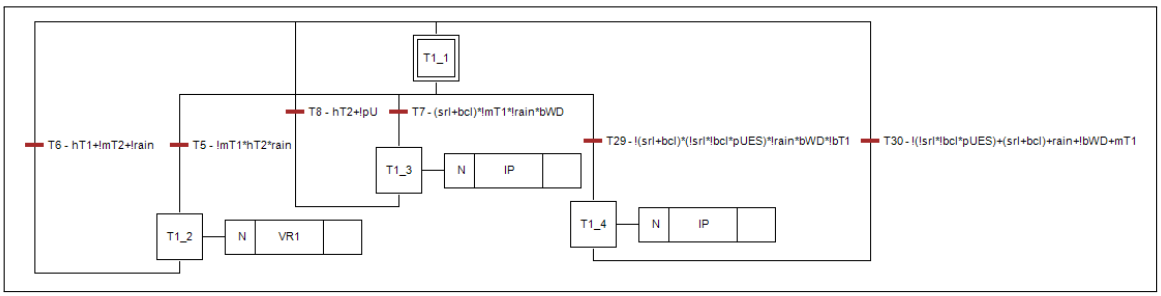


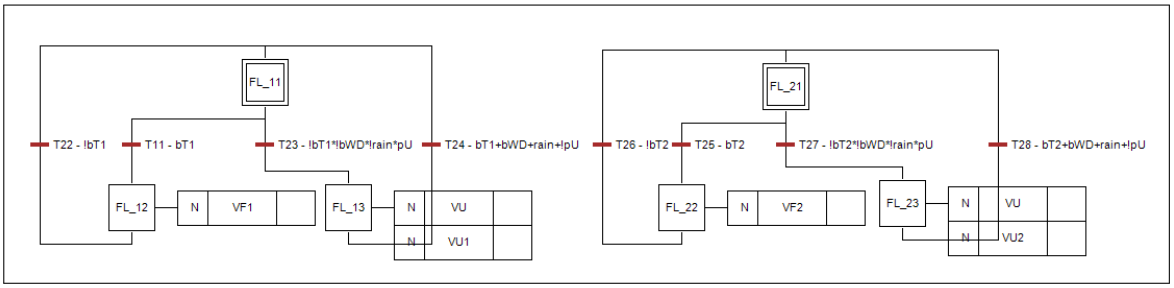
FIG. 5.11: Architecture de la solution de contrôle



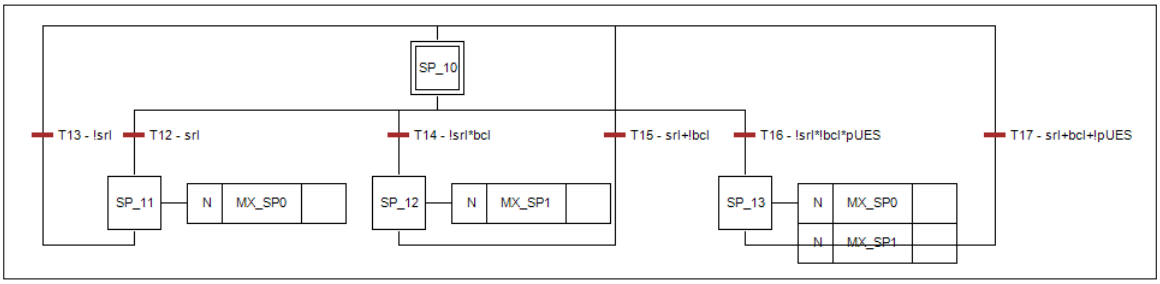
a-Spécification Grafcet de l'approvisionnement en eau du réservoir T_1

b-Spécification Grafcet de l'approvisionnement en eau du réservoir T_2

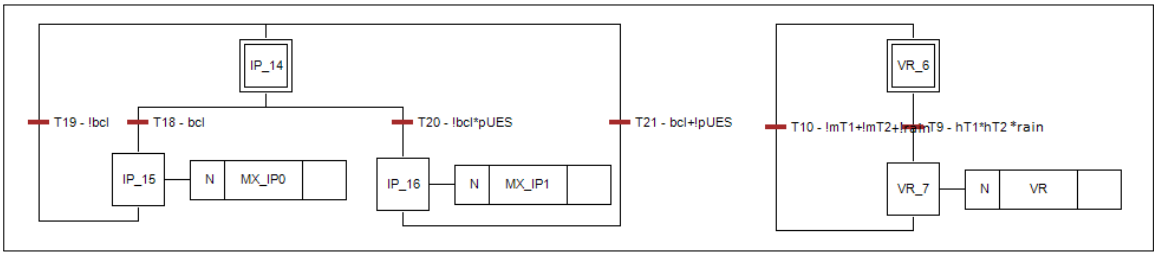
FIG. 5.12: Spécification Grafcet de l'approvisionnement en eau des réservoirs



a-Grafcet de l'ouverture/fermeture des vannes VF et VU

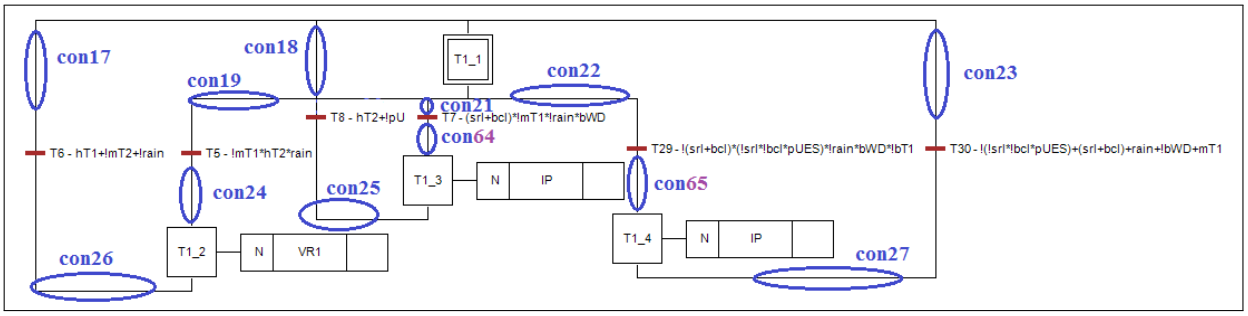


b-Grafcet de la commutation des sources d'énergie pour SP_1

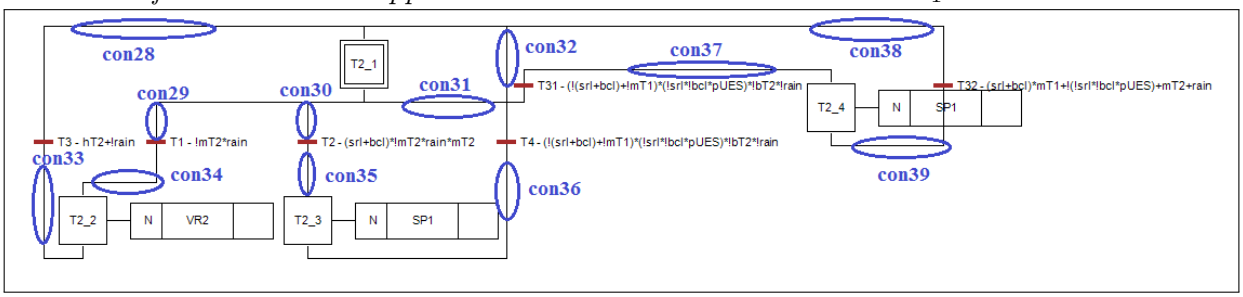


c-Grafcet de la commutation des sources d'énergie pour IP et la vanne VR

FIG. 5.13: Grafcet de la commutation des sources d'énergie et des vannes

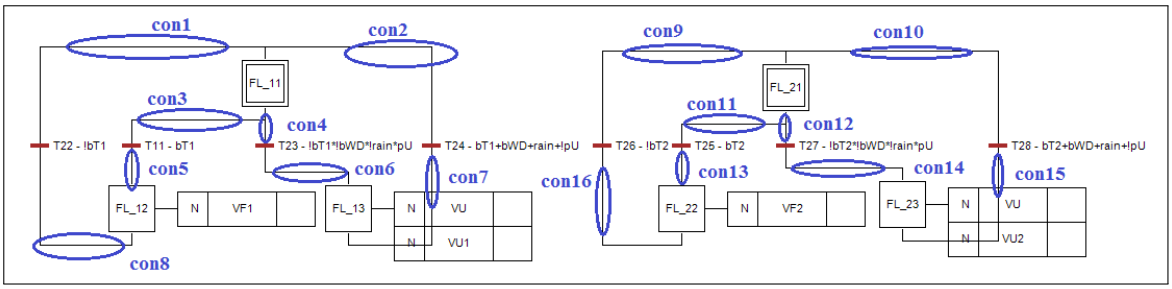


a-Grafcet annoté de l'approvisionnement en eau du réservoir T_1

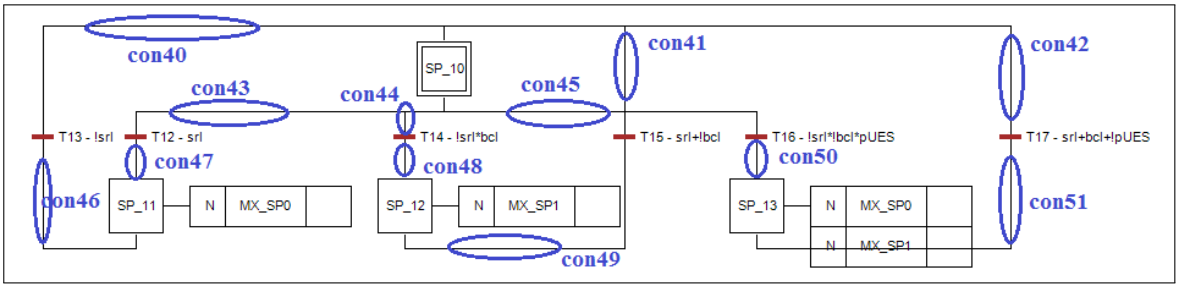


b-Grafcet annoté de l'approvisionnement en eau du réservoir T_2

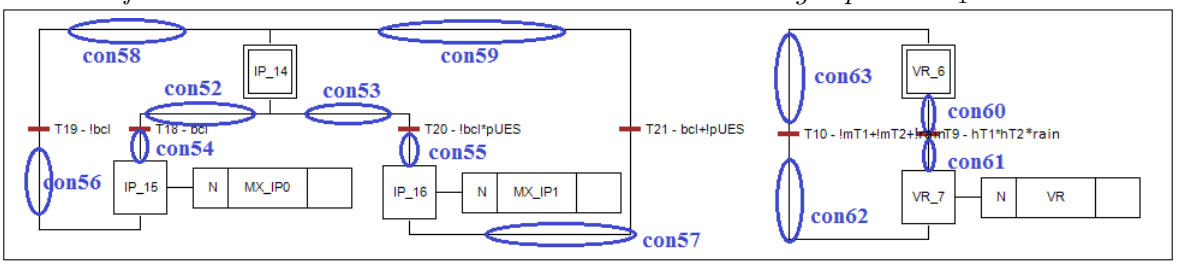
FIG. 5.14: Grafcet annoté de l'approvisionnement en eau des réservoirs



a-Grafcet annoté de l'ouverture/fermeture de VF et VU



b-Grafcet annoté de la commutation des sources d'énergie pour SP_1



c-Grafcet annoté de la commutation des sources d'énergie pour IP et la vanne VR

FIG. 5.15: Grafcet annoté de la commutation des sources d'énergie et ouverture/fermeture des vannes

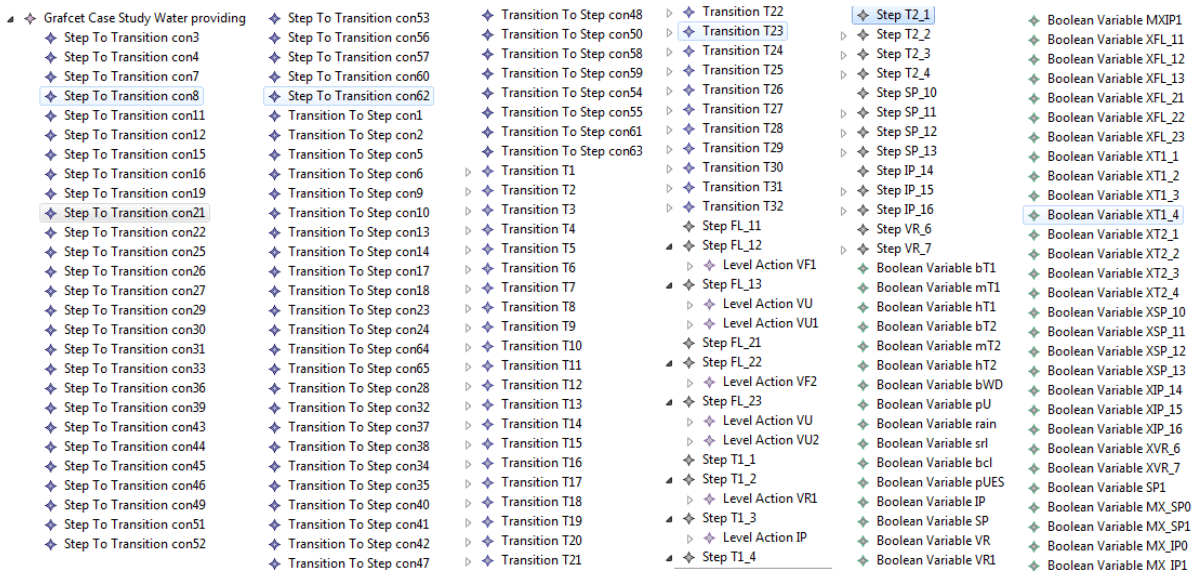


FIG. 5.16: Vue du modèle EMF du Grafcet du cas d'étude

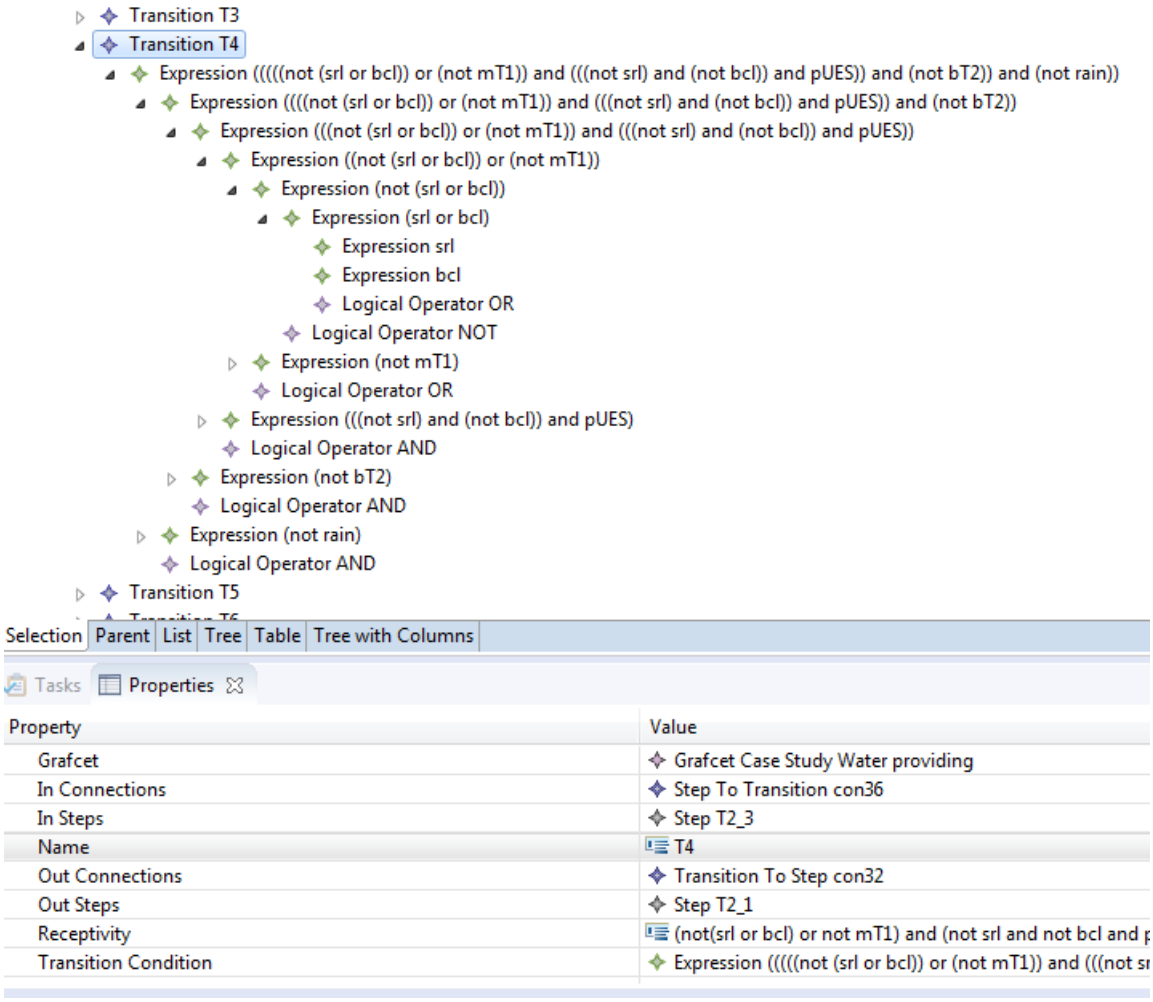


FIG. 5.17: Vue de la réceptivité de la transition T4 (dans l'éditeur arborescent EMF)

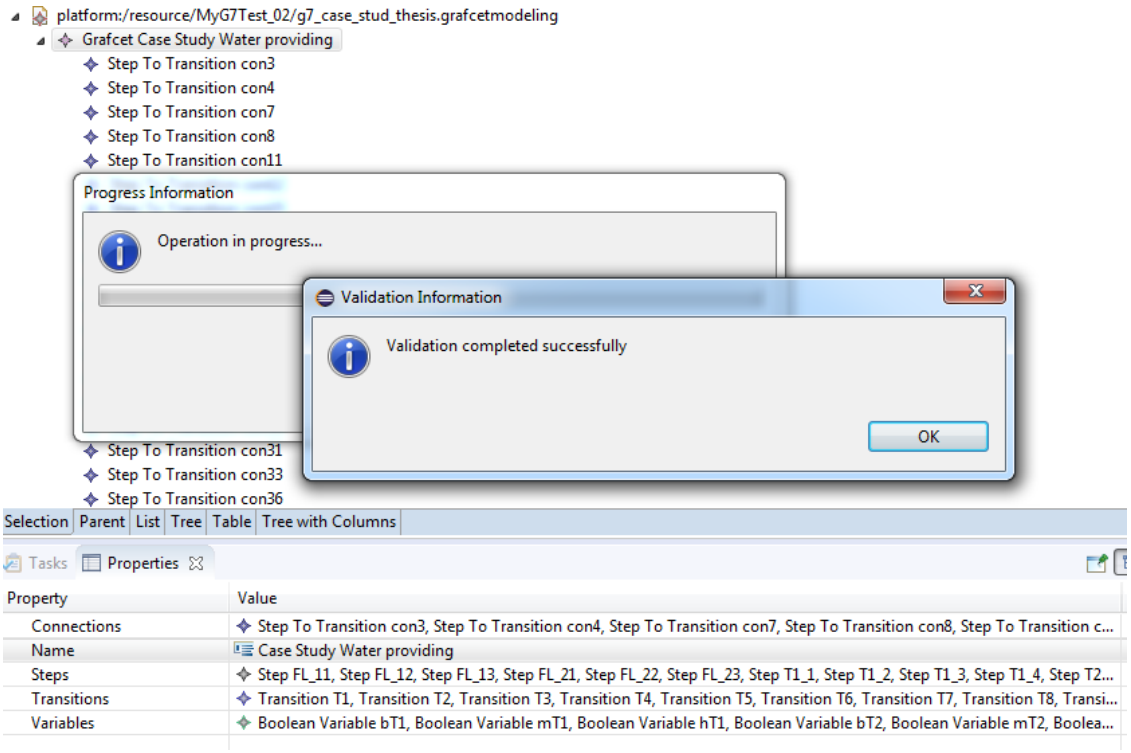


FIG. 5.18: Résultat du processus de validation du cas d'étude

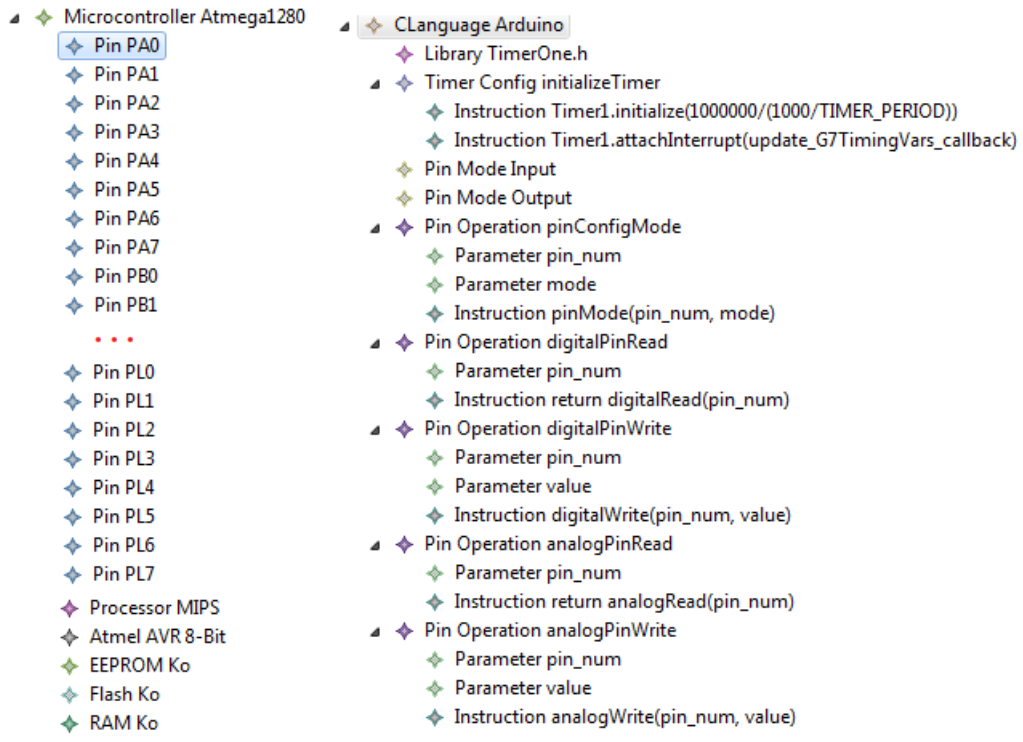


FIG. 5.19: Une vue du modèle EMF du microcontrôleur Atmega1280

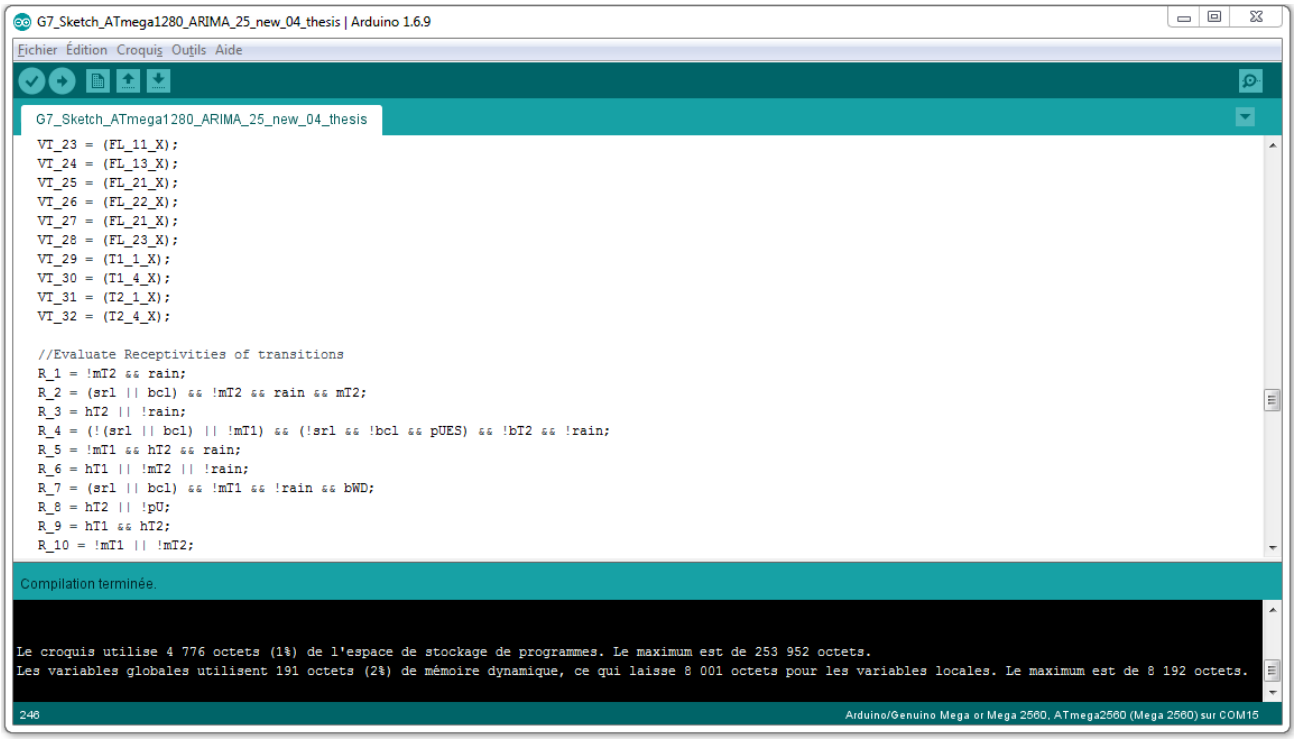


FIG. 5.20: Compilation du programme généré et résultat

Conclusion générale

En guise de conclusion, nous rappelons que le problème ayant retenu notre attention tout au long de ce travail était la conception et la réalisation d'un environnement de synthèse pour les systèmes de contrôle commande (SCCs) bas-coût. Il était question de concevoir et de développer un environnement logiciel permettant de faciliter la spécification des applications de contrôle/commande et leur synthèse sur des cibles microcontrôleurs pour le développement des systèmes embarqués bas-coût. Pour cela, le langage de description de haut-niveau choisi est le Grafcet, standard international utilisé pour la programmation des SCCs. En outre, la diversité des microcontrôleurs existant a poussé à prendre en compte la synthèse multi-cibles, qui jusqu'alors n'était pas traitée par les approches existantes. Il a donc été question pour nous de proposer des modèles qui prennent en compte tous les aspects du Grafcet, l'architecture des cibles, ainsi que les transformations nécessaires, partant d'une spécification fonctionnelle Grafcet du système et d'une spécification de la cible microcontrôleur pour générer le code de contrôle pour cette cible.

Dans un premier temps, nous avons proposé un modèle intermédiaire de représentation du Grafcet, dénommé modèle matriciel du Grafcet. C'est une représentation du Grafcet à l'aide de cinq matrices de codage ($INIT$, E, S, MA et RS) associées à un algorithme d'exécution séquentielle, lesquels respectent les règles d'évolution édictées par le standard Grafcet (CEI 60848 [16]). Ce modèle respecte le critère d'indépendance vis-à-vis des plateformes cibles. La mise en œuvre a permis la production d'une plateforme de synthèse du Grafcet dénommée *GrafcetConverter*. Celle-ci offre offre des fonctionnalités pour l'édition de modèles Grafcet (*UniSim*), la génération automatique des matrices de codage, la simulation de modèles et la génération du code pour des cibles spécifiées. Les microcontrôleurs $ATmega328P$ et $dsPIC30F4013$ ont alors servi de cibles expérimentales. Á travers cette plateforme, deux types de codes sont produits : des codes exécutés par interprétation de modèles Grafcet (calqués de l'algorithme d'interprétation du Grafcet) et ceux exprimant un modèle Grafcet sous formes d'instructions basées sur les équations algébriques du Grafcet (section 4.4.1), constituant une représentation formelle. Ces deux méthodes de génération de code sont

implémentées dans l'environnement de synthèse à l'aide de fonctions particulières écrites par cible.

Outre la simulation dans la plateforme de synthèse, la validation de l'approche de synthèse par codage matriciel du Grafcet a été réalisée grâce à une carte d'extension qui simule parfaitement les feux de signalisation à un carrefour à deux voies. A partir du modèle de spécification Grafcet, le code correspondant est généré, compilé et exécuté sur la plateforme $EASYdsPIC4A$ (équipée du microcontrôleur $dsPIC30F4013$), laquelle communique avec la carte d'extension *feux de carrefour* pour le contrôle des feux de signalisation. De même, le profilage des codes générés a été réalisé à l'aide de la carte *Arduino UNO* (équipée d'un microcontrôleur $ATmega328P$).Le calcul de la durée du cycle de scrutation est faite, laquelle n'excède pas 10 ms, caractéristique des systèmes temps-réel durs.

Toutefois, la synthèse du Grafcet par matrices de codage présente des limites liées à la prise en compte de certains aspects du Grafcet (les expressions complexes, les actions à niveau conditionnelles et stockées, et les événements) et la facilité de caractériser des cibles microcontrôleurs pour la synthèse multi-cibles. Ces limites concernent aussi la difficulté à maintenir et à faire évoluer la solution de synthèse. Après analyse, il ressort qu'il est difficile de relever ces limites dans un environnement généraliste de programmation, d'où l'intérêt d'une mise en œuvre par approche IDM.

Pour cela, nous avons étudié le langage Grafcet au regard du standard et de l'existant pour lui définir une base d'un langage de modélisation (DSML). Après identification de tous les concepts véhiculés par le standard IEC 60848, nous avons procédé à une formalisation de ces concepts avec les lien entre eux pour produire un métamodèle Grafcet. Ce métamodèle constitue la syntaxe abstraite du langage de modélisation. Ensuite, nous nous sommes appesantis sur les expressions Grafcet, vu leur importance pour la vérification et la validation de modèles Grafcet. Elles ont été formalisées sous forme de une grammaire algébrique hors-contexte. En dehors des contraintes statiques exprimées dans ce métamodèle, des contraintes dynamiques ont été exprimées, formalisées en langage OCL puis intégrées au métamodèle. Le langage OCL a été utilisé pour interroger tout modèle Grafcet à dessein de calculer les positions relatives entre les étapes et les transitions.

Après génération du parseur des expressions par l'outil ANTLR [46, 10], l'ensemble de la solution (métamodèle, règles OCL et analyseur syntaxique des expressions Grafcet) a été réalisée dans l'environnement *Eclipse Modeling Framework* (EMF) pour aboutir à un plug-in Éclipse d'édition et de validation des modèles Grafcet. Cet environnement IDM offre alors de nombreux avantages par rapport à un environnement généraliste, à savoir la génération automatique de code Java à partir du métamodèle, le langage OCL (sans effets de bord) pour l'expression des règles sur le modèle, ainsi que des requêtes pour l'obtention de toute information utile. En outre, il y a la possibilité offerte pour focaliser son attention sur les modèles eux-même,

sans se soucier des détails d'implémentation et de la gestion de leur persistance (garantie à tous les niveaux de modélisation par le format XMI). La mise en œuvre par l'approche IDM est donc formelle, permet de gagner en temps et de construire des modèles valides.

Après du DSML pour la modélisation Grafcet, il s'est agit dans un second temps de décrire les microcontrôleurs à prendre en entrée du processus de génération de code. Pour cela, nous avons identifié toutes les caractéristiques des microcontrôleurs utiles pour la synthèse du code puis nous en avons proposé un langage de description dont le modèle formel est un métamodèle microcontrôleur. Ce DSML contient des caractéristiques générales des microcontrôleurs ainsi que celles spécifiques aux cibles et relatives à la génération du code C pour un modèle Grafcet donné. Éclipse EMF a aussi servi pour implémenter le DSML, dont l'éditeur généré permet la création des modèles microcontrôleur. L'approche IDM a enfin été complétée par les transformations de modèles à travers des règles basées sur les équations algébriques Grafcet et le cycle de scrutation des PLCs adapté à l'algorithme d'interprétation du Grafcet. C'est le langage *Acceleo* qui a servi de cadre technologique pour implémenter cette transformation de type $MOFM2T$ (OMG [29, 24]).

Pour terminer, nous avons présenté un cas d'étude qui décrit la possibilité d'exploiter les microcontrôleurs existants pour résoudre le problème de l'approvisionnement autonome en eau domestique. Le contrôleur développé a pour but de réaliser une commutation entre différentes sources d'énergie ayant des coûts différents et alimentant un système de pompage d'eau, l'objectif étant de garantir la présence de l'eau à divers étages d'un bâtiment. A cet effet, une étude générale du problème a été réalisée au préalable pour proposer une architecture du système. Ensuite, des calculs sont faits pour proposer un mécanisme de pompage d'eau garantissant une réduction significative des pertes d'énergie potentielle, et par conséquent de l'énergie électrique de pompage. Il découle de cette stratégie de pompage et du modèle de commutation des sources d'énergie réalisé un modèle Grafcet du contrôleur. Pour réaliser ce contrôleur, nous avons choisi le microcontrôleur *Atmega1280* qui dispose d'assez de broches pour réaliser le modèle Grafcet obtenu. Lorsque le modèle Grafcet de spécification du système de contrôle et celui de la cible microcontrôleur *Atmega1280* sont lus en entrée et valides au regard des règles de vérification, la transformation de modèle est exécutée et le code est généré en langage Arduino. Une fois ce code compilé et chargé dans cette carte à microcontrôleur, il est exécuté et peut alors lire la valeur des entrées à chaque cycle de scrutation et envoyer des signaux en sortie, lesquels représentent des ordres sur les actions à effectuer sur la partie opérative du système.

Par rapport à l'approche de synthèse par codage matriciel du Grafcet, l'approche IDM présente de nombreux avantages: pas besoin de calculer les matrices de codage, possibilité de vérifier la conformité du modèle au langage Grafcet et grande flexibilité dans la représentation des expressions

du modèle Grafcet.

Comme perspectives, il serait intéressant de considérer dans la modélisation les structures hiérarchiques du Grafcet. De même, la solution IDM n'est pas complète au regard du langage Grafcet, du fait que nous utilisons l'éditeur arborescent généré automatiquement pour construire les modèles Grafcet. Une perspective importante pour ces travaux serait d'étudier les objets graphiques utilisés dans le domaine et de proposer un éditeur graphique des modèles Grafcet. Pour cela, l'outil *Éclipse GMF* en est une piste intéressante.

En considérant que les PLCs sont conçus matériellement de manière à fonctionner dans des environnements perturbés (ce qui n'est pas le cas des microcontrôleurs), une autre perspective intéressante serait de proposer une solution permettant de rendre robuste l'exécution du contrôleur. Pour cela, il s'avère important d'étudier et d'implémenter en plus du code principal de contrôle, des primitives de durcissement de code.

Publications

Articles de journaux

1. G. Nzebop Ndenoka, E. Simeu and R. Alhakim, Efficient controller synthesis of multi-energy systems for autonomous domestic water supply. *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées (ARIMA)*, 2017, vol. 24.

Articles de conférences

1. G. Nzebop Ndenoka, R. Alhakim and E. Simeu, Controller Synthesis of Multi-Energy System for Autonomous Domestic Water Supply, *Conférence de Recherche en Informatique Ed. 2015 (CRI'15)* Yaoundé, Dec. 2015
2. G. Nzebop Ndenoka, M. Tchuente and E. Simeu, **Langage et sémantique des expressions pour la synthèse de modèle Grafcet dans un environnement IDM**, *Conférence de Recherche en Informatique Ed. 2019 (CRI'19)* Yaoundé, Dec. 2019

Bibliographie

[1] IEC 61131-3. Programmable controllers-part 3: programming languages (3rd ed.). 2013.

[2] Michael Barr. *Programming embedded systems in C and C++.* " O'Reilly Media, Inc.", 1999.

[3] Francesco Basciani, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Automated chaining of model transformations with incompatible metamodels. In *International Conference on Model Driven Engineering Languages and Systems*, pages 602–618. Springer, 2014.

[4] Oriol Bayó-Puxan, Josep Rafecas-Sabaté, Oriol Gomis-Bellmunt, and Joan Bergas-Jané. A grafcet-compiler methodology for c-programmed microcontrollers. *Assembly Automation*, 28(1):55–60, 2008.

[5] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[6] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[7] Michel Blanchard. *Comprendre, maîtriser et appliquer le GRAFCET:[Graphe de Commande Étape-Transition].* CEPADUES éditions, 1979.

[8] William Bolton. *Automates programmables industriels-2e éd.* Dunod, 2015.

[9] Paulo Borges, José Mendes Machado, Eurico Seabra, and Luís F Silva. A formal approach for aerospace systems control considering sfc specification and c programming language. In *56th International Scientific Colloquium*, pages 1–7, 2011.

[10] Jean Bovet and Terence Parr. Antlrworks: an antlr grammar development environment. *Software: Practice and Experience*, 38(12):1305–1332, 2008.

[11] Jordi Cabot and Martin Gogolla. Object constraint language (ocl): a definitive guide. In *Formal methods for model-driven engineering*, pages 58–90. Springer, 2012.

[12] Fr Charbonnier, H Alla, and R David. The supervised control of discrete event dynamic systems: a new approach. In *Decision and Control,*

1995., Proceedings of the 34th IEEE Conference on, volume 1, pages 913–920. IEEE, 1995.

[13] François Charbonnier, Hassane Alla, and René David. Discrete-event dynamic systems. *IEEE Transactions on Control Systems Technology*, 7(2):175–187, 1999.

[14] Ching-Han Chen and Jia-Hong Dai. Design and high-level synthesis of hybrid controller. In *Networking, Sensing and Control, 2004 IEEE International Conference on*, volume 1, pages 433–438. IEEE, 2004.

[15] AFCET Commission. Normalisation de la représentation du cahier de charges d'un automatisme logique. *IEEE, TSE 31. Tourres L, Manufacturing Technology*, 34(1):pp.427–430, Aug. 1977.

[16] International Electrotechnical Commission et al. Iec 60848: Grafcet specification language for sequential function charts. Technical report, Tech. rep. International Electrotechnical Commission, 2002.

[17] Alberto Rodrigues Da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, 2015.

[18] Sajal K. Das, VK Agrawal, Dilip Sarkar, Lalit M. Patnaik, and Prem Shankar Goel. Reflexive incidence matrx (rim) representation of petri nets. *IEEE transactions on software engineering*, (6):643–653, 1987.

[19] René David. Grafcet: A powerful tool for specification of logic controllers. *IEEE Transactions on control systems technology*, 3(3):253–268, 1995.

[20] Rene David and Hassane Alla. Petri nets and grafcet: tools for modelling discrete event systems. 1992.

[21] René David and Hassane Alla. Petri nets for modeling of dynamic systems: A survey. *Automatica*, 30(2):175–202, 1994.

[22] G De Tommasi and A Pironti. An educational open-source tool for the design of iec 61131-3 compliant automation software. In *Power Electronics, Electrical Drives, Automation and Motion, 2008. SPEEDAM 2008. International Symposium on*, pages 486–491. IEEE, 2008.

[23] Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Model transformations. In *Formal Methods for Model-Driven Engineering*, pages 91–136. Springer, 2012.

[24] Samba Diaw, Rédouane Lbath, and Bernard Coulette. Etat de l'art sur le développement logiciel dirigé par les modèles. *Technique et Science Informatique TSI*, 29(505-536):71, 2008.

[25] Carla Ferreira, Sérgio Monteiro, and João Monteiro. Automatic generation of c-code or pld circuits under sfc graphical environment. In *Industrial Electronics, 1997. ISIE'97., Proceedings of the IEEE International Symposium on*, volume 1, pages SS181–SS185. IEEE, 1997.

[26] EMF: Eclipse Modeling Framework, Visited 2018.

[27] Georg Frey and Mark Minas. Editing, visualizing, and implementing signal interpreted petri nets. In *Proceedings of the AWPN 2000*, pages 57–62. Citeseer, 2000.

[28] Evelio González, Roberto Marichal, and Alberto Hamilton. Ontology-based approach to basic grafcet formalization. *Journal of the Chinese Institute of Engineers*, 39(8):946–953, 2016.

[29] OMG: Object Management Group. Mda (model driven architecture) guide version 1.0.1 [online], 2001.

[30] Charles L Hamblin. Translation to and from polish notation. *The Computer Journal*, 5(3):210–213, 1962.

[31] Jean-Marc Jézéquel, Benoit Combemale, and Didier Vojtisek. *Ingénierie Dirigée par les Modèles: des concepts à la pratique...* Ellipses, 2012.

[32] Robert Julius, Max Schürenberg, Frank Schumacher, and Alexander Fay. Transformation of grafcet to plc code including hierarchical structures. *Control Engineering Practice*, 64:173–194, 2017.

[33] Kyo C Kang, Jaejoon Lee, and Patrick Donohoe. Feature-oriented product line engineering. *IEEE software*, 19(4):58–65, 2002.

[34] Anneke G Kleppe, Jos Warmer, Wim Bast, and MDA Explained. The model driven architecture: practice and promise, 2003.

[35] Anneke G Kleppe, Jos Warmer, Jos B Warmer, and Wim Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[36] Philippe Le Parc, Dominique L'Her, J-L Scharbarg, and Lionel Marce. Grafcet revisited with a synchronous data-flow language. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 29(3):284–293, 1999.

[37] José Machado, Eurico Seabra, José C Campos, Filomena Soares, and Celina P Leão. Safe controllers design for industrial automation systems. *Computers & Industrial Engineering*, 60(4):635–653, 2011.

[38] Marga Marcos, Elisabet Estevez, Federico Perez, and Eelco Van Der Wal. Xml exchange of control programs. *IEEE Industrial Electronics Magazine*, 3(4), 2009.

[39] OMG Mof. 2.0 query/views/transformations rfp, 2002.

[40] Valery Monthe, Laurent Nana, Georges E Kouamou, and Claude Tangha. Rsaml: A domain specific modeling language for describing robotic software architectures with integration of real time properties. In *EWiLi*, 2016.

[41] B Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and P Vanderheyden. Eclipse development using the graphical editing framework and the eclipse modeling framework (2004). *IBM Redbook*.

[42] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden. Eclipse development. *Using the Graphical Editing*

Framework and the Eclipse Modeling Framework. IBM RedBooks.(IBM Corp.), 379, 2004.

[43] Gérard Nzebop Ndenoka, Emmanuel Simeu, and Rshdee Alhakim. Efficient controller synthesis of multi-energy systems for autonomous domestic water supply. *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées*, 24, 2017.

[44] K Mun Ng and Z Alam Haron. Visual microcontroller programming using extended s-system petri nets. *WSEAS Transactions on Computers*, 9(6):573–582, 2010.

[45] Thomas Messi Nguele. *DSL pour la fouille des réseaux sociaux sur des architectures Multi-coeurs*. PhD thesis, 2018.

[46] T Parr. The definitive antlr reference, building domain-specific languages, pragmatic bookshelf, dallas texas. Technical report, ISBN 0-9787392-5-6, 2007.

[47] Rajendra Patel and Arvind Rajwat. A survey of embedded software profiling methodologies. *arXiv preprint arXiv:1312.2949*, 2013.

[48] XML PLCOpen. Formats for iec 61131-3, 2005.

[49] Julien Provost, Jean-Marc Roussel, and Jean-Marc Faure. A formal semantics for grafcet specifications. In *Automation Science and Engineering (CASE), 2011 IEEE Conference on*, pages 488–494. IEEE, 2011.

[50] Julien Provost, Jean-Marc Roussel, and Jean-Marc Faure. Translating grafcet specifications into mealy machines for conformance test purposes. *Control Engineering Practice*, 19(9):947–957, 2011.

[51] Yassine Qamsane, Mahmoud El Hamlaoui, Abdelouahed Tajer, and Alexandre Philippot. A model-based transformation method to design plc-based control of discrete automated manufacturing systems. *Proceedings of Engineering and Technology–PET*, 19:4–11, 2017.

[52] Helena M Ramos, Filipe Vieira, and Dídia IC Covas. Energy efficiency in a water supply system: Energy consumption and co2 emission. *Water Science and Engineering*, 3(3):331–340, 2010.

[53] Jean-Marc Roussel and Jean-Jacques Lesage. Validation and verification of grafcets using state machine. In *IMACS-IEEE" CESA'96"*, pages pp–758, 1996.

[54] Stéphane Rubini and Dominique Lavenier. Les architectures reconfigurables. *Calculateurs Parallèles*, 9(1):9–27, 1997.

[55] Frank Schumacher and Alexander Fay. Requirements and obstacles for the transformation of grafcet specifications into iec 61131–3 plc programs. In *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–4. IEEE, 2011.

[56] Frank Schumacher and Alexander Fay. Transforming time constraints of a grafcet graph into a suitable petri net formalism. In *Industrial Technology (ICIT), 2013 IEEE International Conference on*, pages 210–218. IEEE, 2013.

[57] Frank Schumacher and Alexander Fay. Formal representation of grafcet to automatically generate control code. *Control Engineering Practice*, 33:84–93, 2014.

[58] Frank Schumacher, Sebastian Schröck, and Alexander Fay. Tool support for an automatic transformation of grafcet specifications into iec 61131-3 control code. In *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on*, pages 1–4. IEEE, 2013.

[59] Frank Schumacher, Sebastian Schröck, and Alexander Fay. Transforming hierarchical concepts of grafcet into a suitable petri net formalism. *IFAC Proceedings Volumes*, 46(9):295–300, 2013.

[60] Médésu Sogbohossou and Antoine Vianou. Formal modeling of grafcets with time petri nets. *IEEE Transactions on Control Systems Technology*, 23(5):1978–1985, 2015.

[61] Moujtahid Soukaina, Belangour Abdessamad, and Marzak Abdelaziz. Model driven engineering (mde) tools: A survey. *American Journal of Science, Engineering and Technology*, 3(2):29, 2018.

[62] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[63] Kleanthis Thramboulidis and Georg Frey. An mdd process for iec 61131-based industrial automation systems. In *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–8. IEEE, 2011.

[64] Carla Tricarico, Mark S Morley, R Gargano, Zoran Kapelan, G De Marinis, Dragan Savic, and F Granata. Integrated optimal cost and pressure management for water distribution systems. 2014.

[65] Unisim, 2015.

[66] Markus Voelter. *Generic tools, specific languages*. Citeseer, 2014.

[67] Markus Voelter. Fusing modeling and programming into language-oriented programming. In *International Symposium on Leveraging Applications of Formal Methods*, pages 309–339. Springer, 2018.

[68] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.

[69] Daniel Witsch and Birgit Vogel-Heuser. Close integration between uml and iec 61131-3: New possibilities through object-oriented extensions. In *Emerging Technologies & Factory Automation, 2009. ETFA 2009. IEEE Conference on*, pages 1–6. IEEE, 2009.

[70] Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document object model (dom) level 3 core specification, 2000.

[71] Richard Zurawski and MengChu Zhou. Petri nets and industrial applications: A tutorial. *IEEE Transactions on industrial electronics*, 41(6):567–583, 1994.

Annexe 1 : Quelques codes générés par matrices de codage

Annexe 1.1 - Code Xml du Grafcet de l'exemple

Listing 2: Une partie du code Xml du Grafcet de l'exemple (Généré par UniSim)

```
<?xml version="1.0" encoding="utf-8"?>
<project xmlns="http://www.plcopen.org/xml/tc6.xsd" xmlns:xhtml="http
    ://www.w3.org/1999/xhtml" xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance">
  <fileHeader companyName="Universita di Napoli Federico II"
      companyURL="www.unina.it" productName="UniSim" productVersion="
      0.5.0" productRelease="B" creationDateTime="2007-12-31T23:59:59Z
      " contentDescription="Application" />
  ...
  <types>
    <dataTypes />
    <pous>
      <pou name="G7_Exemple" pouType="program">
        ...
        <body>
          <SFC>
            <step name="1" height="44" width="44" localId="1"
                initialStep="true" negated="false">
              <position x="402" y="26" />
              <connectionPointIn />
            </step>
            <step name="2" height="44" width="44" localId="2"
                initialStep="false" negated="false">
              <position x="402" y="118" />
              <connectionPointIn>
                <connection refLocalId="20" />
              </connectionPointIn>
            </step>
            <step name="3" height="44" width="44" localId="3"
                initialStep="false" negated="false">
              <position x="162" y="238" />
```

```
24              <connectionPointIn>
25                <connection refLocalId="10" />
26              </connectionPointIn>
27            </step>
28            <actionBlock>
29              <position x="228" y="244" />
30              <connectionPointIn>
31                <connection refLocalId="3" />
32              </connectionPointIn>
33              <action qualifier="N" duration="00:00:00">
34                <reference name="VR1" />
35              </action>
36            </actionBlock>
37            <step name="4" height="44" width="44" localId="4"
                  initialStep="false" negated="false">
38              <position x="54" y="374" />
39              <connectionPointIn>
40                <connection refLocalId="18" />
41              </connectionPointIn>
42            </step>
43            <actionBlock>
44             ...
45            </actionBlock>
46            <step name="5" height="44" width="44" localId="5"
                  initialStep="false" negated="false">
47              <position x="274" y="374" />
48              <connectionPointIn>
49                <connection refLocalId="18" />
50              </connectionPointIn>
51            </step>
52            <actionBlock>
53              ...
54            </actionBlock>
55            <step name="6" height="44" width="44" localId="6"
                  initialStep="false" negated="false">
56              <position x="386" y="238" />
57              <connectionPointIn>
58                <connection refLocalId="11" />
59              </connectionPointIn>
60            </step>
61            <actionBlock>
62              ...
63            </actionBlock>
64            <step name="7" height="44" width="44" localId="7"
                  initialStep="false" negated="false">
65              <position x="594" y="238" />
66              <connectionPointIn>
67                <connection refLocalId="12" />
68              </connectionPointIn>
69            </step>
70            <actionBlock>
71              ...
72            </actionBlock>
```

```
73              <step name="8" height="44" width="44" localId="8"
                    initialStep="false" negated="false">
74                <position x="490" y="374" />
75                <connectionPointIn>
76                  <connection refLocalId="22" />
77                </connectionPointIn>
78              </step>
79              <actionBlock>
80                ...
81              </actionBlock>
82              <transition height="44" width="44" localId="9">
83                <position x="420" y="96" />
84                <connectionPointIn>
85                  <connection refLocalId="1" />
86                </connectionPointIn>
87                <condition>
88                  <reference name="^init" />
89                </condition>
90              </transition>
91              <transition height="44" width="44" localId="10">
92                <position x="180" y="200" />
93                <connectionPointIn>
94                  <connection refLocalId="21" />
95                </connectionPointIn>
96                <condition>
97                  <reference name="hT2+rain" />
98                </condition>
99              </transition>
100             <transition height="44" width="44" localId="11">
101               <position x="404" y="200" />
102               <connectionPointIn>
103                 <connection refLocalId="21" />
104               </connectionPointIn>
105               <condition>
106                 <reference name="!rain*bWD*ppM1" />
107               </condition>
108             </transition>
109             <transition height="44" width="44" localId="12">
110               <position x="612" y="200" />
111               <connectionPointIn>
112                 <connection refLocalId="21" />
113               </connectionPointIn>
114               <condition>
115                 <reference name="(ppM2+!bT1)*[2.X]" />
116               </condition>
117             </transition>
118             <transition height="44" width="44" localId="13">
119               <position x="180" y="316" />
120               <connectionPointIn>
121                 <connection refLocalId="3" />
122               </connectionPointIn>
123               <condition>
124                 <reference name="hT1+!mT2+!rain" />
125               </condition>
```

```
126             </transition>
127             <transition height="44" width="44" localId="14">
128               <position x="36" y="280" />
129               <connectionPointIn>
130                 <connection refLocalId="19" />
131               </connectionPointIn>
132               <condition>
133                 <reference name="[4.T&gt;5s]+bT1" />
134               </condition>
135             </transition>
136             <transition height="44" width="44" localId="15">
137               <position x="404" y="328" />
138               <connectionPointIn>
139                 <connection refLocalId="6" />
140               </connectionPointIn>
141               <condition>
142                 <reference name="hT1+!bWD+!ppM1" />
143               </condition>
144             </transition>
145             <transition height="44" width="44" localId="16">
146               <position x="612" y="328" />
147               <connectionPointIn>
148                 <connection refLocalId="7" />
149               </connectionPointIn>
150               <condition>
151                 <reference name="!bWD*bT1" />
152               </condition>
153             </transition>
154             <transition height="44" width="44" localId="17">
155               <position x="812" y="268" />
156               <connectionPointIn>
157                 <connection refLocalId="8" />
158               </connectionPointIn>
159               <condition>
160                 <reference name="[8.T&gt;10s]+bT1" />
161               </condition>
162             </transition>
163             <simultaneousConvergence localId="19">
164               <position x="0" y="0" />
165               <connectionPointIn>
166                 <connection refLocalId="4" />
167                 <connection refLocalId="5" />
168               </connectionPointIn>
169             </simultaneousConvergence>
170             <simultaneousDivergence localId="18">
171               <position x="0" y="0" />
172               <connectionPointIn>
173                 <connection refLocalId="13" />
174               </connectionPointIn>
175             </simultaneousDivergence>
176             <selectionConvergence localId="20">
177               <position x="0" y="0" />
178               <connectionPointIn>
179                 <connection refLocalId="9" />
```

```
                <connection refLocalId="14" />
                <connection refLocalId="17" />
              </connectionPointIn>
            </selectionConvergence>
            ...
            <selectionDivergence localId="21">
              <position x="0" y="0" />
              <connectionPointIn>
                <connection refLocalId="2" />
              </connectionPointIn>
            </selectionDivergence>
          </SFC>
        </body>
      </pou>
    </pous>
  </types>
...
</project>
```

Annexe 1.2 - Code interpréteur Arduino généré pour l'exemple de Grafcet

Listing 3: Extrait du code Arduino de l'interpréteur généré pour l'exemple de Grafcet

```
#include "TimerOne.h"
#include "EEPROM.h"

#define limit_RS 20
#define NMax_Trans 16
#define NMax_Steps 16
#define N_RisingEdges_Max 16

/*Declare Materials: Inputs & Outputs*/
#define NMax_Inputs 7
#define NMax_Outputs 7

const unsigned int Board_TIME_UNIT = 100;

//Effective number of Inputs :
const unsigned int N_inputs = 9;
//Declare INPUT pins
const unsigned int pins_input[] = {0, 1, 2, 3, 4, 5, 6, 7, 8};

//Effective number of Outputs :
const unsigned int N_outputs = 5;
//Declare OUTPUT pins
const unsigned int pins_output[] = {9, 10, 11, 12, 13};

```

```
25
26 unsigned int MatricesDATA[] = {
27 //N_trans, N_Steps, N_Inputs, N_actions:
28        0x0009,
29        0x0008,
30        0x0009,
31        0x0005,
32 //INIT Vector
33        0x0001,
34 //E matrix
35        0x0001, 0x0002, 0x0002, 0x0002, 0x0004, 0x0018, 0x0020, 0x0040
              , 0x0080,
36 //S matrix
37        0x0002, 0x0004, 0x0020, 0x0040, 0x0018, 0x0002, 0x0080, 0x0080
              , 0x0002,
38 //MA matrix
39        0x0060, 0x0004, 0x0070, 0x0080, 0x0008,
40 //RS matrix
41        0x0003, 0x8001, 0x3000, 0x0000,
42        0x0003, 0x8003, 0x8005, 0x0000,
43        0x0006, 0x8005, 0x2000, 0x8006, 0x1000, 0x8007, 0x1000,
44        0x0006, 0x8008, 0x8000, 0x2000, 0x0000, 0xC001, 0x1000,
45        0x0007, 0x8002, 0x8004, 0x2000, 0x0000, 0x8005, 0x2000, 0x0000
              ,
46        0x0003, 0xE332, 0x8000, 0x0000,
47        0x0007, 0x8002, 0x8006, 0x2000, 0x0000, 0x8007, 0x2000, 0x0000
              ,
48        0x0004, 0x8006, 0x2000, 0x8000, 0x1000,
49        0x0003, 0xE764, 0x8000, 0x0000
50 };
51
52 /*Declare variables to be stored in RAM*/
53 unsigned int MatricesDATA_eeprom_adr;
54 ...
55 unsigned int E[NMax_Trans];
56 ...
57
58 void initializeTimer1(){
59        unsigned int TU = 1000/Board_TIME_UNIT;
60        Timer1.initialize(1000000/TU);
61        Timer1.attachInterrupt(Step_timer_update_callback);
62 }
63
64 void Reading_Inputs()
65 {
66        for(i=0; i<N_Inputs;i++){
67               temp_IN_OUT_Reg = digitalRead(pins_input[i]) << i;
68               IN_Port = asmOR(IN_Port, temp_IN_OUT_Reg);
69        }
70 }
71
72 void Update_Outputs(void)
73 {
74        bool old_state,new_state;
```

```
75          for(i=0; i<N_Actions;i++){
76                  old_state = getBitWithIndex(OUT_Port_old,i);
77                  new_state = getBitWithIndex(OUT_Port,i);
78                  if(old_state != new_state){digitalWrite(pins_output[i],
                        new_state);}
79          }
80          OUT_Port_old = OUT_Port;
81  }
82
83  void setup(){
84          //Save Matrices into EEPROM
85          EEPROM_Write();
86          /*Copy Static Matrix from MatricesDATA to "DatainRAM"(G7 data)
                in RAM*/
87          EEPROM_Read();
88          init_IO_ports();
89          X = X_INIT;
90          initializeTimer1();
91          for(i=0; i<N_RisingEdges_Max; i++){ Old_RisingEdges[i] = false
                ;}
92  }
93
94  void loop(){
95          Reading_Inputs();
96          //Evaluate new Grafcet situation
97          VA = 0; VD = 0;
98          for (i = 0; i < N_Trans; i++)
99          {
100                 XEi = asmAND(X,E[i]);
101                 if (XEi == E[i]) {VT_i=1;} else {VT_i=0;}
102                 for (j = 1; j <= RS[i][0]; j++)
103                 {
104                         RS_Parse(RS[i][j]);
105                 }
106                 R_i = (boolean) *(--stack_position);
107                 FT_i = VT_i && R_i;
108                 if (FT_i !=0)
109                 {
110                         VA = asmOR(VA,S[i]); //S[i] is a 16 bits (2
                            bytes) data (unsigned int)
111                         VD = asmOR(VD,E[i]);
112                 }
113         }
114         AD = asmAND(VA,VD);
115         not_VD = asmNOT(VD);
116         DX = asmOR(not_VD,AD);
117         X = asmAND(X,DX);
118         X = asmOR(X,VA);
119
120         //Evaluate OUTPUTS
121         unsigned int XMA;
122         OUT_Port = 0x0000;
123         for (i = 0; i < N_Actions; i++)
124         {
```

```
125                 XMA = asmAND(X,MA[i]);
126                 if (XMA != 0)
127                 {
128                 temp_IN_OUT_Reg = 1<<i;
129                 OUT_Port = asmOR(OUT_Port,temp_IN_OUT_Reg);
130                 }
131         }
132         delay(2);
133 }
134
135 void Step_timer_update_callback()
136 {
137         for (i = 0; i < N_Steps; i++)
138         {
139                 if (getBitWithIndex(X,i) == 0){ //step i is not active
140                         Step_active_timer[i] = 0;
141                 }
142                 else
143                 {
144                         Step_active_timer[i] += 1;
145                 }
146         }
147 }
148
149 ...
150
151 void init_IO_ports()
152 {
153         //init inputs & outputs
154         for(i=0; i<N_Inputs;i++){
155                 pinMode(pins_input[i], INPUT);
156         }
157
158         for(i=0; i<N_Actions;i++){
159                 pinMode(pins_output[i], OUTPUT);
160         }
161         OUT_Port = 0x0000;
162         IN_Port = 0x0000;
163 }
164
165 /*Define usefull Assembly Functions*/
166 unsigned int asmNOT(unsigned int reg_data){
167         return ~reg_data;
168 }
169 unsigned int asmAND(unsigned int a, unsigned int b){
170         return a&b;
171 }
172 unsigned int asmOR(unsigned int a, unsigned int b){
173         return a|b;
174 }
175 ...
176 void EEPROM_Write()
177 {
178         MatricesDATA_eeprom_adr = 0;
```

```
179         unsigned int M_size = sizeof(MatricesDATA)/2;
180
181         for(i=0; i<M_size ; i++){
182                 writeNextIntInEEPROM(MatricesDATA[i]);
183         }
184 }
185 void EEPROM_Read()
186 {
187         MatricesDATA_eeprom_adr = 0;
188
189         N_Trans = readNextIntFromEEPROM();
190         N_Steps = readNextIntFromEEPROM();
191
192         N_Inputs = readNextIntFromEEPROM();
193         N_Actions = readNextIntFromEEPROM();
194
195         X_INIT = readNextIntFromEEPROM();
196         for (i = 0; i < N_Trans; i++)
197         {
198                 E[i] = readNextIntFromEEPROM();
199         }
200         for (i = 0; i < N_Trans; i++)
201         {
202                 S[i] = readNextIntFromEEPROM();
203         }
204         for (i = 0; i < N_Actions; i++)
205         {
206                 MA[i] = readNextIntFromEEPROM();
207         }
208         for (i = 0; i < N_Trans; i++)
209         {
210                 unsigned int RS_length = readNextIntFromEEPROM();
211                 RS[i][0] = RS_length;
212                 for (j = 1; j <= RS_length; j++)
213                 {
214                         RS[i][j] = readNextIntFromEEPROM();
215                 }
216                 for (j = RS_length+1; j < limit_RS; j++)
217                 {
218                         RS[i][j] = 0;
219                 }
220         }
221 }
222 void writeNextIntInEEPROM(unsigned int val)
223 {
224         unsigned char faible = val & 0x00FF;
225         unsigned char fort = (val >> 8) & 0x00FF;
226         EEPROM.write(MatricesDATA_eeprom_adr++, fort) ;
227         EEPROM.write(MatricesDATA_eeprom_adr++, faible) ;
228 }
229 unsigned int readNextIntFromEEPROM()
230 {
231         return (MatricesDATA[MatricesDATA_eeprom_adr++]);
232 }
```

Annexe 1.3 - Code ordinaire Arduino généré pour l'exemple Grafcet

Listing 4: Extrait du code généré par équation algébrique Grafcet pour l'exemple

```
/*
Time : 22/02/2019 17:59:28 Generated from the grafcet File:
       D:\DD2\THESE\_Redaction_These\grafcets\g7_express_UniSim\
           Exemple_G7_Projet_Article_BIS.xml
*/
#include "TimerOne.h"

//****  Declare INPUT pins mapped ****  Total Inputs : 9
const unsigned int pin_bT1 = 0;
const unsigned int pin_init = 1;
const unsigned int pin_hT1 = 2;
const unsigned int pin_hT2 = 3;
const unsigned int pin_mT2 = 4;
const unsigned int pin_rain = 5;
const unsigned int pin_bWD = 6;
const unsigned int pin_ppM1 = 7;
const unsigned int pin_ppM2 = 8;

//****  Declare OUTPUT pins mapped  ****  Total Outputs : 5
const unsigned int pin_A = 9;
const unsigned int pin_VR1 = 10;
const unsigned int pin_IP = 11;
const unsigned int pin_REC = 12;
const unsigned int pin_AV = 13;

//****  Declare INPUT pins states ****
boolean bT1 = false, bT1_Old = false;
boolean init = false, init_Old = false;
boolean hT1 = false, hT1_Old = false;
boolean hT2 = false, hT2_Old = false;
boolean mT2 = false, mT2_Old = false;
boolean rain = false, rain_Old = false;
boolean bWD = false, bWD_Old = false;
boolean ppM1 = false, ppM1_Old = false;
boolean ppM2 = false, ppM2_Old = false;

//****  Declare OUPUT pins states ****
boolean A = false, A_Old = false;
boolean VR1 = false, VR1_Old = false;
boolean IP = false, IP_Old = false;
boolean REC = false, REC_Old = false;
boolean AV = false, AV_Old = false;
```

```
42
43 //****  Declare STEPs variables ****
44 boolean S1_X, S1_X_Old = false;
45 boolean S2_X, S2_X_Old = false;
46 boolean S3_X, S3_X_Old = false;
47 boolean S4_X, S4_X_Old = false;
48 boolean S5_X, S5_X_Old = false;
49 boolean S6_X, S6_X_Old = false;
50 boolean S7_X, S7_X_Old = false;
51 boolean S8_X, S8_X_Old = false;
52
53 //****  Declare Validated Transitions variables ****
54 boolean VT_1 = false;
55 boolean VT_2 = false;
56 boolean VT_3 = false;
57 boolean VT_4 = false;
58 boolean VT_5 = false;
59 boolean VT_6 = false;
60 boolean VT_7 = false;
61 boolean VT_8 = false;
62 boolean VT_9 = false;
63
64 //****  Declare Receptivities of Transitions variables ****
65 boolean R_1 = false;
66 boolean R_2 = false;
67 boolean R_3 = false;
68 boolean R_4 = false;
69 boolean R_5 = false;
70 boolean R_6 = false;
71 boolean R_7 = false;
72 boolean R_8 = false;
73 boolean R_9 = false;
74
75 //****  Declare Transitions receptivities variables ****
76 boolean TR_1 = false;
77 boolean TR_2 = false;
78 boolean TR_3 = false;
79 boolean TR_4 = false;
80 boolean TR_5 = false;
81 boolean TR_6 = false;
82 boolean TR_7 = false;
83 boolean TR_8 = false;
84 boolean TR_9 = false;
85
86 //****  Declare STEPs timing variables for duration activity  ****
87 unsigned int S1_duration = 0;
88 unsigned int S2_duration = 0;
89 unsigned int S3_duration = 0;
90 unsigned int S4_duration = 0;
91 unsigned int S5_duration = 0;
92 unsigned int S6_duration = 0;
93 unsigned int S7_duration = 0;
94 unsigned int S8_duration = 0;
95
```

```
96  const unsigned int Board_TIME_UNIT = 100;
97
98  //To manage timer or measuring duration of active steps
99  void initializeTimer1(){
100   //Timer1.initialize(1000000);
101   unsigned int FT_Steps = 1000/Board_TIME_UNIT;
102   Timer1.initialize(1000000/FT_Steps);
103   Timer1.attachInterrupt(Step_timer_update_callback);
104 }
105
106 void Step_timer_update_callback(){
107       if(S1_X) {S1_duration ++ ;} else {S1_duration = 0;}
108       if(S2_X) {S2_duration ++ ;} else {S2_duration = 0;}
109       if(S3_X) {S3_duration ++ ;} else {S3_duration = 0;}
110       if(S4_X) {S4_duration ++ ;} else {S4_duration = 0;}
111       if(S5_X) {S5_duration ++ ;} else {S5_duration = 0;}
112       if(S6_X) {S6_duration ++ ;} else {S6_duration = 0;}
113       if(S7_X) {S7_duration ++ ;} else {S7_duration = 0;}
114       if(S8_X) {S8_duration ++ ;} else {S8_duration = 0;}
115 }
116
117 void setup(){
118       initializeTimer1();
119       //INIT INPUT PINs
120       pinMode(pin_bT1, INPUT);
121       pinMode(pin_init, INPUT);
122       pinMode(pin_hT1, INPUT);
123       pinMode(pin_hT2, INPUT);
124       pinMode(pin_mT2, INPUT);
125       pinMode(pin_rain, INPUT);
126       pinMode(pin_bWD, INPUT);
127       pinMode(pin_ppM1, INPUT);
128       pinMode(pin_ppM2, INPUT);
129       //INIT OUTPUT PINs
130       pinMode(pin_A, OUTPUT);
131       pinMode(pin_VR1, OUTPUT);
132       pinMode(pin_IP, OUTPUT);
133       pinMode(pin_REC, OUTPUT);
134       pinMode(pin_AV, OUTPUT);
135       //Set the initial situation
136       S1_X_Old = true;
137 }
138
139 void loop(){
140       //Reading input values
141       bT1 = digitalRead(pin_bT1);
142       init = digitalRead(pin_init);
143       hT1 = digitalRead(pin_hT1);
144       hT2 = digitalRead(pin_hT2);
145       mT2 = digitalRead(pin_mT2);
146       rain = digitalRead(pin_rain);
147       bWD = digitalRead(pin_bWD);
148       ppM1 = digitalRead(pin_ppM1);
149       ppM2 = digitalRead(pin_ppM2);
```

```
150
151         //Evaluate validated transitions variabless
152         VT_1 = (S1_X_Old);
153         VT_2 = (X2_X_Old);
154         VT_3 = (S2_X_Old);
155         VT_4 = (S2_X_Old);
156         VT_5 = (S3_X_Old);
157         VT_6 = (S4_X_Old && S5_X_Old);
158         VT_7 = (S6_X_Old);
159         VT_8 = (S7_X_Old);
160         VT_9 = (S8_X_Old);
161
162         //Evaluate Receptivities of transitions
163         R_1 = (!init_Old && init);
164         R_2 = hT2 || rain;
165         R_3 = !rain && bWD && ppM1;
166         R_4 = (ppM2 || !bT1) && S2_X;
167         R_5 = hT1 || !mT2 || !rain;
168         R_6 = (Stp4_duration > 50) || bT1;
169         R_7 = hT1 || !bWD || !ppM1;
170         R_8 = !bWD && bT1;
171         R_9 = (8_duration > 100) || bT1;
172
173         //Evaluate Clearing conditions
174         TR_1 = VT_1 && R_1;
175         TR_2 = VT_2 && R_2;
176         TR_3 = VT_3 && R_3;
177         TR_4 = VT_4 && R_4;
178         TR_5 = VT_5 && R_5;
179         TR_6 = VT_6 && R_6;
180         TR_7 = VT_7 && R_7;
181         TR_8 = VT_8 && R_8;
182         TR_9 = VT_9 && R_9;
183
184         //Save the states of Step's activity variables in Old_
                corresponding variables
185         S1_X_Old = S1_X;
186         S2_X_Old = S2_X;
187         S3_X_Old = S3_X;
188         S4_X_Old = S4_X;
189         S5_X_Old = S5_X;
190         S6_X_Old = S6_X;
191         S7_X_Old = S7_X;
192         S8_X_Old = S8_X;
193
194         //Evaluate Step variables
195         S1_X = (1_X_Old && !TR_1);
196         S2_X = TR_1 || TR_6 || TR_9 || (S2_X_Old && !TR_2 && !TR_3 &&
                !TR_4);
197         S3_X = TR_2 || (S3_X_Old && !TR_5);
198         S4_X = TR_5 || (Stp4_X_Old && !TR_6);
199         S5_X = TR_5 || (S5_X_Old && !TR_6);
200         S6_X = TR_3 || (S6_X_Old && !TR_7);
201         S7_X = TR_4 || (S7_X_Old && !TR_8);
```

```
202         S8_X = TR_7 || TR_8 || (S8_X_Old && !TR_9);
203
204         //Evaluate OUTPUTs variables
205         A = S6_X || S7_X;
206         VR1 = S3_X;
207         IP = S5_X || S6_X || S7_X;
208         REC = S8_X;
209         AV = S4_X;
210
211         //Update OUTPUTs
212         if(A_Old != A) {digitalWrite(pin_A, A);}
213         if(VR1_Old != VR1) {digitalWrite(pin_VR1, VR1);}
214         if(IP_Old != IP) {digitalWrite(pin_IP, IP);}
215         if(REC_Old != REC) {digitalWrite(pin_REC, REC);}
216         if(AV_Old != AV) {digitalWrite(pin_AV, AV);}
217
218         //Update Old OUTPUTs
219         A_Old = A;
220         VR1_Old = VR1;
221         IP_Old = IP;
222         REC_Old = REC;
223         AV_Old = AV;
224         delay(2);
225 }
```

Annexe 2: Quelques codes générés par approche IDM

Annexe 2.1 - Code du métamodèle Microcontrôleur

Listing 5: Code du métamodèle Microcontrôleur

```
1
2  package MicrocontrollerModeling : MicrocontrollerModeling = 'http://
       www.example.org/microcontrollermodeling'
3  {
4          class Microcontroller
5          {
6                  attribute name : String[?];
7                  property pins : Pin[+|1] { ordered composes };
8                  property clanguage : CLanguage[1] { composes };
9                  property processor : Processor[+|1] { ordered composes
                       };
10                 attribute family : String[?];
11                 attribute manufacturer : String[?];
12                 property rom : ROM[?] { composes };
13                 property flash : Flash[?] { composes };
14                 property ram : RAM[1] { composes };
15                 attribute wordMemory : WordSize[?];
16                 property registers : Register[*|1] { ordered composes
                       };
17                 invariant uniqueName_PerPin:
18                 self.pins->forAll(p1, p2|p1<>p1 implies p1.name<>p2.
                       name);
19                 invariant uniquePinNumber_PerPin:
20                 self.pins->forAll(p1, p2|p1<>p1 implies p1.number<>p2.
                       number);
21                 invariant asLeast_OneRomOrFlashMemory:
22                 not(self.flash = null and self.rom->isEmpty());
23         }
24         class Pin
25         {
```

```
26                  attribute name : String[?];
27                  attribute nature : PinNature[?];
28                  attribute number : ecore::EInt[1];
29          }
30          enum PinNature { serializable }
31          {
32                  literal Digital;
33                  literal Analog = 1;
34                  literal Mixed = 2;
35          }
36          class CLanguage
37          {
38                  attribute name : String[?];
39                  property libraries : Library[*|1] { ordered composes };
40                  property timerconfig : TimerConfig[?] { composes };
41                  property pinmodes : PinMode[1..2|1] { ordered composes
                        };
42                  attribute hasMain : Boolean[1];
43                  attribute filesExtension : String[?];
44                  property pinoperation : PinOperation[1..5|1] { ordered
                        composes };
45          }
46          class Function
47          {
48                  attribute type : String[?];
49                  property parameters : Parameter[+|1] { ordered composes
                         };
50                  property instructions : Instruction[+|1] { ordered
                        composes };
51          }
52          class TimerConfig extends Function
53          {
54                  attribute name : TimerOp[?];
55                  attribute period : ecore::EInt[1];
56          }
57          abstract class Memory
58          {
59                  attribute unit : MemoryUnit[?];
60                  attribute size : ecore::EInt[1];
61          }
62          class Processor
63          {
64                  attribute unit : SpeedUnit[?];
65                  attribute speed : ecore::EInt[1];
66          }
67          enum SpeedUnit { serializable }
68          {
69                  literal Hz; literal Mhz = 1;
70                  literal GHz = 2; literal MIPS = 3;
71          }
72          class ROM extends Memory;
73          ...
74  }
```

Annexe 2.2 - Code *Acceleo* du module principal

Listing 6: Code Acceleo du module principal de la transformation

```
[comment encoding = UTF-8 /]
[module generateG7MM2Code('http://www.example.org/grafcetModeling', '
    http://www.example.org/microcontrollermodeling')]
[import G7MMToCode::main::generate_G7_structures/]
[import G7MMToCode::main::genG7Services/]
[template public generateMainCode(ag7 : Grafcet, aMicro :
    Microcontroller)]
[comment @main/]
[file ((ag7.name + '/' + ag7.name + '.'+aMicro.clanguage.
    filesExtension).replaceAll(' ','_'), false, 'UTF-8')]
//Code generated from the g7 "[ag7.name/]" and the microcontroller "[
    aMicro.name/]"
//Date: [getTime()/]
[generate_header_and_global_variables(ag7, aMicro)/]
boolean transitions_fired;
void setup(){
       [generate_initializations(ag7, aMicro)/]
}
void loop(){
[generate_inputsBoardReading(ag7, aMicro)/]
       transitions_fired = 0;
[generate_next_state_calculations(ag7)/]
[generate_outputs_calculations(ag7)/]
       if(!transitions_fired){
[generate_UpdatingLevelActions_Outputs_variables(ag7, aMicro)/]
       }
[generate_UpdatingStoredActions_Outputs_variables(ag7, aMicro)/]
[generate_SaveOldModel_Variables(ag7)/]
}
[if (aMicro.clanguage.hasMain)]
int main(void){
  setup();
  for ( ; ; ) loop(); // repeat indefinitely the function loop()
  return 0;
}
[/if]
[generate_other_functions(ag7, aMicro)/]
[/file]
[/template]
```

Annexe 3: Autres contraintes OCL, Code du métamodèle et code XMI de l'exemple

Ces contraintes sont relatives aux instances du métamodèle Grafcet défini au chapitre 3.

Annexe 3.1 - Énoncé des contraintes spécifiques aux opérateurs et formalisation OCL

Étant donnée une expression, lorsque ses deux sous-expressions sont non nulles, alors elles sont forcément toutes du même type car modélisent une composition par un opérateur binaire. La formalisation OCL se fait avec :

Listing 7: ValidExpressionWithBinaryOperation

```
1        (self.operator<>null and self.subExpr1<>null and self.subExpr2
             <>null) implies (self.subExpr1.type = self.subExpr2.type);
```

Pour faciliter la tâche de débogage lors de la vérification des modèles construits, des contraintes spécifiques sont formalisées en OCL selon le type d'opérateur en présence. Ainsi, chaque fois que l'erreur `ValidExpressionWithBinaryOperation` se produit, elle est suivie de l'une des erreurs suivantes :

Cas des opérateurs de comparaison

Listing 8: ValidComparisonOperator_GT (supérieur strict)

```
1  invariant ValidComparisonOperator_GT:
2  (self.operator.oclIsTypeOf(LogicalOperator) and (self.operator.
       oclAsType(LogicalOperator)).name = LogicOpValues::GT)
3          implies (self.subExpr1.type = ExprType::Arithmetic and self.
               subExpr2.type = ExprType::Arithmetic);
```

Listing 9: ValidComparisonOperator_LT (inférieur strict)

```
invariant ValidComparisonOperator_LT:
(self.operator.oclIsTypeOf(LogicalOperator) and (self.operator.
    oclAsType(LogicalOperator)).name = LogicOpValues::LT)
        implies (self.subExpr1.type = ExprType::Arithmetic and self.
            subExpr2.type = ExprType::Arithmetic);
```

Listing 10: ValidComparisonOperator_GE (supérieur ou égal)

```
invariant ValidComparisonOperator_GE:
(self.operator.oclIsTypeOf(LogicalOperator) and (self.operator.
    oclAsType(LogicalOperator)).name = LogicOpValues::GE)
        implies (self.subExpr1.type = ExprType::Arithmetic and self.
            subExpr2.type = ExprType::Arithmetic);
```

Listing 11: ValidComparisonOperator_LE (inférieur ou égal)

```
invariant ValidComparisonOperator_LE:
(self.operator.oclIsTypeOf(LogicalOperator) and (self.operator.
    oclAsType(LogicalOperator)).name = LogicOpValues::LE)
        implies (self.subExpr1.type = ExprType::Arithmetic and self.
            subExpr2.type = ExprType::Arithmetic);
```

Cas des opérateurs logiques binaires (ET, OU)

Listing 12: invariant ValidLogicOperator_AND (ET)

```
invariant ValidLogicOperator_AND:
(self.operator.oclIsTypeOf(LogicalOperator) and (self.operator.
    oclAsType(LogicalOperator)).name = LogicOpValues::AND)
        implies (self.subExpr1.type = ExprType::Logical and self.
            subExpr2.type = ExprType::Logical);
```

Listing 13: ValidLogicOperator_OR (OU)

```
invariant ValidLogicOperator_OR:
(self.operator.oclIsTypeOf(LogicalOperator) and (self.operator.
    oclAsType(LogicalOperator)).name = LogicOpValues::OR)
        implies (self.subExpr1.type = ExprType::Logical and self.
            subExpr2.type = ExprType::Logical);
```

Cas des opérateurs arithmétiques

Listing 14: ValidArithmeticOperator_ADD(+)

```
invariant ValidArithmeticOperator_ADD:
```

```
2  (self.operator.oclIsTypeOf(AritmeticOperator) and (self.operator.
       oclAsType(AritmeticOperator)).name = ArithmOpValues::ADD)
3          implies (self.subExpr1.type = ExprType::Arithmetic and self.
               subExpr2.type = ExprType::Arithmetic);
```

Listing 15: ValidArithmeticOperator_SUB(–)

```
1  (self.operator.oclIsTypeOf(AritmeticOperator) and (self.operator.
       oclAsType(AritmeticOperator)).name = ArithmOpValues::SUB)
2          implies (self.subExpr1.type = ExprType::Arithmetic and self.
               subExpr2.type = ExprType::Arithmetic);
```

Listing 16: ValidArithmeticOperator_MULT(*)

```
1  invariant ValidArithmeticOperator_MULT:
2  (self.operator.oclIsTypeOf(AritmeticOperator) and (self.operator.
       oclAsType(AritmeticOperator)).name = ArithmOpValues::MULT)
3          implies (self.subExpr1.type = ExprType::Arithmetic and self.
               subExpr2.type = ExprType::Arithmetic);
```

Listing 17: ValidArithmeticOperator_DIV(/)

```
1  invariant ValidArithmeticOperator_DIV:
2  (self.operator.oclIsTypeOf(AritmeticOperator) and (self.operator.
       oclAsType(AritmeticOperator)).name = ArithmOpValues::DIV)
3          implies (self.subExpr1.type = ExprType::Arithmetic and self.
               subExpr2.type = ExprType::Arithmetic);
```

Annexe 3.2 - Code du métamodèle Grafcet

Listing 18: Extrait du code du métamodèle Grafcet (Issu de l'éditeur OCLinEcore)

```
1
2  import ecore : 'http://www.eclipse.org/emf/2002/Ecore' ;
3
4  package grafcetModeling : grafcetModeling = 'http://www.example.org/
       grafcetModeling'
5  {
6  abstract class G7Element{ attribute name : String[?]; }
7  class Grafcet{ ...}
8  class Step extends G7Element
9  {
10         operation createStepActivityVar();
11         attribute isInitial : Boolean[1];
```

```
12          attribute isActive : Boolean[1];
13          property grafcet#steps : Grafcet[1];
14          property inConnections : TransitionToStep[*|1] { ordered };
15          property outConnections : StepToTransition[*|1] { ordered };
16          property actions#step : Action[*|1] { ordered composes };
17          property variable : BooleanVariable[1] { derived };
18          property inTransitions : Transition[*] { derived volatile }
19          {
20          initial: (grafcet.transitions->select(trans|trans.
                outConnections->exists(outCon|self.inConnections->includes
                (outCon))))->asSet();
21          }
22          property outTransitions : Transition[*] { derived volatile }
23          {
24                  initial: (grafcet.transitions->select(trans|trans.
                        inConnections->exists(inCon|self.outConnections->
                        includes(inCon))))->asSet();
25          }
26          invariant stepVarIsInternalVar:
27          self.variable.type = VarType::Internal;
28          invariant levelActionVarIsBoolVar:
29          self.actions->forAll(act|act.oclIsTypeOf(LevelAction) implies
                act.actionVariable.oclIsTypeOf(BooleanVariable));
30          invariant uniqueNamesOfActions:
31          self.actions->forAll(a1,a2| a1<>a2 implies a1.name<>a2.name);
32  }
33  class Transition extends G7Element
34  {
35          operation parseReceptivity();
36          property transitionCondition : Expression[?] { derived
                composes };
37          attribute receptivity : String[?];
38          property grafcet#transitions : Grafcet[1];
39          property inConnections : StepToTransition[+|1] { ordered };
40          property outConnections : TransitionToStep[+|1] { ordered };
41          property inSteps : Step[*] { derived volatile }
42          {
43                  initial: (grafcet.steps->select(step|step.
                        outConnections->exists(outCon|self.inConnections->
                        includes(outCon))))->asSet();
44          }
45          property outSteps : Step[*] { derived volatile }
46          {
47                  initial: (grafcet.steps->select(step|step.inConnections
                        ->exists(inCon|self.outConnections->includes(inCon)
                        )))->asSet();
48          }
49          invariant validTransition:
50          self.inConnections->size()>=1 and self.outConnections->size()
                >=1;
51  }
52  class Expression
53  {
54          operation eval();
```

```
55          operation evalOld();
56          operation getCExpr() : String[?];
57          operation getOldCExpr() : String[?];
58          operation getName2() : String[?];
59          property subExpr1 : Expression[?] { derived composes };
60          property subExpr2 : Expression[?] { derived composes };
61          property operator : Operator[?] { derived composes };
62          attribute type : ExprType[?];
63          attribute isSimple : Boolean[1];
64          attribute name : String[?];
65          attribute boolValue : Boolean[1] = 'false';
66          attribute arithmValue : ecore::EInt[1];
67          property variable : Variable[?] { derived };
68          attribute isConstant : Boolean[1];
69          invariant VariableExistsInGrafcet:
70          (self.isSimple and not self.isConstant) implies self.variable
                <>null;
71          invariant AValidConstantExpression:
72          self.isConstant implies (self.isSimple and self.variable =
                null);
73          invariant SimpleExpressionHasNoSubExpressions:
74          self.isSimple implies (self.subExpr1 = null and self.subExpr2
                = null);
75          invariant ValidUnaryOperationExpression:
76          (self.operator<>null
77          and
78          (self.operator.oclIsTypeOf(TimingOperator)
79          or
80          (self.operator.oclIsTypeOf(LogicalOperator)
81           and ( ((self.operator.oclAsType(LogicalOperator)).name =
                    LogicOpValues::NOT)
82                 or ((self.operator.oclAsType(LogicalOperator)).name =
                        LogicOpValues::RE)
83                 or ((self.operator.oclAsType(LogicalOperator)).name =
                        LogicOpValues::FE)
84                 )
85          )
86          or
87          (self.operator.oclIsTypeOf(AritmeticOperator) and (self.
                operator.oclAsType(AritmeticOperator)).name =
                ArithmOpValues::U_MINUS)
88   )
89  )
90  implies (self.subExpr1=null and self.subExpr2<>null);
91
92          ...
93
94          invariant ValidArithmeticOperator_DIV:
95          (self.operator.oclIsTypeOf(AritmeticOperator) and (self.
                operator.oclAsType(AritmeticOperator)).name =
                ArithmOpValues::DIV)
96          implies (self.subExpr1.type = ExprType::Arithmetic and self.
                subExpr2.type = ExprType::Arithmetic);
97  }
```

```
98  abstract class Operator;
99  class TimingOperator extends Operator
100 {
101         attribute type : TimingType[?];
102         attribute duration1 : ecore::EInt[1];
103         attribute duration2 : ecore::EInt[1];
104         attribute unit1 : TimeUnit[?];
105         attribute unit2 : TimeUnit[?];
106 }
107 enum LogicOpValues { serializable }
108 {
109         literal OR;
110         literal AND = 1;
111         literal NOT = 2;
112         literal RE = 3;
113         literal FE = 4;
114         literal EQU = 5;
115         literal LE = 6;
116         literal LT = 7;
117         literal GE = 8;
118         literal GT = 9;
119         literal NEQ = 10;
120 }
121 ...
122 }
```

Annexe 3.3 - Code *XMI* du modèle Grafcet exemple

Listing 19: Extrait du code XML du modèle Grafcet exemple

```
1
2  <?xml version="1.0" encoding="ASCII"?>
3  <grafcetModeling:Grafcet xmi:version="2.0" xmlns:xmi="http://www.omg.
       org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:grafcetModeling="http://www.example.org/grafcetModeling"
       name="Grafcet Example">
4    <connections xsi:type="grafcetModeling:StepToTransition" name="con3"
         />
5    ...
6    <connections xsi:type="grafcetModeling:StepToTransition" name="con1"
         />
7    <connections xsi:type="grafcetModeling:TransitionToStep" name="con2"
         />
8    ...
9    <connections xsi:type="grafcetModeling:TransitionToStep" name="con20
         "/>
10   <transitions name="1" receptivity="re init_" inConnections="//
         @connections.9" outConnections="//@connections.10">
```

```
11      <transitionCondition name="(RE init_)">
12        <subExpr2 isSimple="true" name="init_" variable="//@variables.0"
              />
13        <operator xsi:type="grafcetModeling:LogicalOperator" name="RE"/>
14      </transitionCondition>
15    </transitions>
16    ...
17    <transitions name="9" receptivity="((N +1) > 0) AND [not 10s/X8] or
          bT1" inConnections="//@connections.8" outConnections="//
          @connections.18">
18      <transitionCondition name="((((N + 1) > 0) and [not 10 s/X8]) or
            bT1)">
19        <subExpr1 name="(((N + 1) > 0) and [not 10 s/X8])">
20          <subExpr1 name="((N + 1) > 0)">
21            <subExpr1 type="Arithmetic" name="(N + 1)">
22              <subExpr1 type="Arithmetic" isSimple="true" name="N"
                    variable="//@variables.11"/>
23              <subExpr2 type="Arithmetic" isSimple="true" name="1"
                    arithmValue="1" isConstant="true"/>
24              <operator xsi:type="grafcetModeling:AritmeticOperator"/>
25            </subExpr1>
26            <subExpr2 type="Arithmetic" isSimple="true" name="0"
                  isConstant="true"/>
27            <operator xsi:type="grafcetModeling:LogicalOperator" name="
                  GT"/>
28          </subExpr1>
29          <subExpr2 name="[not 10 s/X8]">
30            <subExpr2 isSimple="true" name="X8" variable="//@variables
                  .23"/>
31            <operator xsi:type="grafcetModeling:TimingOperator" type="
                  Limited" duration1="10" unit1="s"/>
32          </subExpr2>
33          <operator xsi:type="grafcetModeling:LogicalOperator" name="AND
                "/>
34        </subExpr1>
35        <subExpr2 isSimple="true" name="bT1" variable="//@variables.8"/>
36        <operator xsi:type="grafcetModeling:LogicalOperator"/>
37      </transitionCondition>
38    </transitions>
39    <steps name="1" isInitial="true" outConnections="//@connections.9"
          variable="//@variables.16">
40      <actions xsi:type="grafcetModeling:StoredAction" name="C"
            actionVariable="//@variables.10" storedExpression="0">
41        <expressionToEvaluate type="Arithmetic" isSimple="true" name="0"
              isConstant="true"/>
42      </actions>
43      <actions xsi:type="grafcetModeling:StoredAction" name="N"
            actionVariable="//@variables.11" moment="DeActivation"
            storedExpression="10">
44        <expressionToEvaluate type="Arithmetic" isSimple="true" name="10
              " arithmValue="10" isConstant="true"/>
45      </actions>
46    </steps>
47      ...
```

```
48    <steps name="8" inConnections="//@connections.16 //@connections.17"
          outConnections="//@connections.8" variable="//@variables.23">
49      <actions xsi:type="grafcetModeling:LevelAction" name="REC"
            actionVariable="//@variables.15" condition="true">
50        <expressionCondition isSimple="true" name="true" boolValue="true
              " isConstant="true"/>
51      </actions>
52      <actions xsi:type="grafcetModeling:StoredAction" name="N"
            actionVariable="//@variables.11" storedExpression="N - 1">
53        <expressionToEvaluate type="Arithmetic" name="(N - 1)">
54          <subExpr1 type="Arithmetic" isSimple="true" name="N" variable=
                "//@variables.11"/>
55          <subExpr2 type="Arithmetic" isSimple="true" name="1"
                arithmValue="1" isConstant="true"/>
56          <operator xsi:type="grafcetModeling:AritmeticOperator" name="
                SUB"/>
57        </expressionToEvaluate>
58      </actions>
59    </steps>
60    <variables xsi:type="grafcetModeling:BooleanVariable" name="init_"/>
61    <variables xsi:type="grafcetModeling:BooleanVariable" name="VR1"
          type="Output"/>
62    ...
63    <variables xsi:type="grafcetModeling:BooleanVariable" name="mT2"/>
64    <variables xsi:type="grafcetModeling:NumericVariable" name="C" type=
          "Output"/>
65    ...
66    <variables xsi:type="grafcetModeling:BooleanVariable" name="REC"
          type="Output"/>
67    ...
68    <variables xsi:type="grafcetModeling:BooleanVariable" name="X7" type
          ="Internal"/>
69    <variables xsi:type="grafcetModeling:BooleanVariable" name="X8" type
          ="Internal"/>
70  </grafcetModeling:Grafcet>
```

Publications scientifiques tirées des travaux

Article de conférence

G. Nzebop Ndenoka, R. Alhakim and E. Simeu, **Controller Synthesis of Multi-Energy System for Autonomous Domestic Water Supply**, *Conférence de Recherche en Informatique Ed. 2015 (CRI'15)* Yaoundé, Dec. 2015

G. Nzebop Ndenoka, M. Tchuente and E. Simeu, **Langage et sémantique des expressions pour la synthèse de modèle Grafcet dans un environnement IDM**, *Conférence de Recherche en Informatique Ed. 2019 (CRI'19)* Yaoundé, Dec. 2019

Article de journal

G. Nzebop Ndenoka, E. Simeu and R. Alhakim, **Efficient controller synthesis of multi-energy systems for autonomous domestic water supply**. *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées (ARIMA)*, 2017, vol. 24.

Efficient controller synthesis of multi-energy systems for autonomous domestic water supply

G. NZEBOP NDENOKA [a, c] — E. SIMEU [b] — R. ALHAKIM [b]

[a] UMI 209 UMMISCO, University of Yaoundé I, P.O. Box 337 Yaoundé, Cameroon. ndenokag@yahoo.fr

[b] University of Grenoble Alpes, TIMA Laboratory, 46 Avenue Félix Viallet, 38031 Grenoble Cedex France. {Rshdee.Alhakim, Emmanuel.Simeu}@imag.fr

[c] LIRIMA, IDASCO Team, Department of Computer Science, Faculty of Sciences, University of Yaoundé I , P.O. Box 812 Yaoundé Cameroon.

ABSTRACT. The continuous development of ICT facilitates the emergence and rapid proliferation of a wide variety of low-cost processors for the execution of programs in complex embedded applications. In this paper, the study explores the possibility to benefit from this wealth of computing capacity at a reasonable cost to solve concrete problems encountered in implementation of sustainable development processes, particularly in water and energy supply ... We are focusing autonomous water supply in buildings of several floors using several tanks supplied by several sources of water and pumping energy, based on a multilevel hierarchical priority access to water. The first problem is to propose pumping devices and a switching process between power sources, associated to an architectural structure guaranteeing significant reduction of pumping energy. The second problem is the system controller realization. For this, we have proposed a generic architecture justified by gains in potential energy. We also propose an automatic generation tool of control programs for different microprocessor targets taken from the functional design specification of the system given in a Grafcet form. To put them in evidence, we describe at the end a case study.

RÉSUMÉ. Le développement vertigineux des TIC favorise l'émergence et la prolifération rapide d'une grande variété de processeurs à bas coût destinés à l'exécution de programmes embarqués dans des applications complexes. Dans ce papier, l'étude explore la possibilité de tirer profit de cette profusion de capacité de calcul à coût raisonnable pour résoudre des problèmes concrets que l'on rencontre dans la mise en place de processus de développement durable, notamment ceux liés à l'approvisionnement en eau et en énergie ... Nous visons l'étude d'approvisionnement en eau autonome des bâtiments à plusieurs étages, en utilisant plusieurs citernes de stockage approvisionnées par plusieurs sources d'eau et d'énergie de pompage, basée sur plusieurs niveaux hiérarchisés de priorité d'accès à l'eau. Le premier problème est de proposer des dispositifs de pompage et un processus de commutation entre les sources d'énergie, associé à une structure architecturale garantissant une réduction significative de l'énergie de pompage. Le second problème est la réalisation du contrôleur. Pour cela, nous proposons une architecture générique justifiée par des gains d'énergie potentielle. Nous proposons aussi un outil de génération automatique de programmes de contrôle pour différentes cibles à microprocesseur à partir d'une spécification fonctionnelle sous la forme d'un Grafcet. Pour les mettre en évidence, nous décrivons à la fin un cas d'étude.

KEYWORDS : ICT, sustainable development, water supplying, photovoltaic energy, Grafcet modeling, automatic synthesis, logic controller.

MOTS-CLÉS : TIC, dveloppement durable, approvisionnement en eau, énergie photovoltaque, modélisation Grafcet, synthèse automatique, contrôleur logique.

1. Introduction

Information and Communication Technologies (ICT) have undergone enormous evolution in recent years. These changes have led to significant improvements in many application fields such as health care, education, commerce, ... Automation is considered one of the most important sectors which has remarkably been developed thanks to the ICT revolution. For example, almost all digital devices, used to control automated electromechanical systems are nowadays based on sophisticated small microcontrollers instead of hard wired logic components (such as relays, cam timers, drum sequencers), that contribute explicitly to save time, energy, materials and money. Despite all these available advantages, sustainable development applications are slow to benefit from these to solve some implementation problems, such as energy and water distribution issues. For example, in 2013 Cameroon National Institute of Statistics (INS) declared that "almost 92% of households in Yaoundé suffer from intermittent water cuts.". Hence, "To reduce the shortages, households depend on the following sources: public water supply (55%), mineral water (10%), drilled wells (6%) and other undrinkable water sources (29%)." [6].

The use of multiple water supply sources is a feasible solution allowing to face the recurring interruptions of urban water supply services. There are, however, a number of problems: some sources (wells/drillings) require electrical energy to operate. This energy can itself even be available in several possibilities including classical urban energy distribution as well as local sustainable sources. However, whether water or energy sources, their costs differ from one another. A management strategy must therefore be implemented to select available sources that are least costly at each operating time.

The importance to invest on renewable energy sources (such as solar energy, wind energy,...) is highlighted to permanently improve the availability of water ([7, 10]). An optimization method is proposed in [11] to minimize energy consumption and to reduce leakage in a water distribution system (WDS) in a hydraulic context. However, the proposed algorithm cannot be applied in a context of household. A model of multi-criteria optimization for energy efficiency has been presented in [10]. But none of the existing methods presents the operational way of handling the issue of energy and water sources management.

The objective of this paper is to study energy control strategy for water supply problem in buildings having many floors. Precisely, it is to find solution that combines sustainable solutions and the advantages of ICT tools in order to realize a complete and autonomous system for water and power supply, with a particular emphasize on saving energy. This involves the establishment of appropriate pumping mechanisms and an efficient process of switching between energy sources. The smart system should guarantee the continuous providing of water to households, and automatically switch between several water and power resources according to the source availability and the service costs.

For ICT tools applied on the automation applications, International Electrotechnical Commission (IEC) has issued five standard programming languages defined in the IEC 6331-3 standard [9]. Among them, Grafcet (or SFC) is a powerful graphical language useful for both specification and programming of industrial automation systems [5]. It is nowadays considered as one of the most used specification languages of IEC standard. Many research works have been carried out to promote and popularize this language, and many modeling and simulation tools as UniSim ([1, 8]), PLCopen XML [13], Isagraf,

... are based on Grafcet standard. On the other hand, the Grafcet language has actually been built and developed to be integrated in the design and implementation software for programmable logic controllers (PLC) devices ([1, 4, 12]).

PLC are digital devices basically used to control automated electro-mechanical systems. They are specially designed to survive in harsh industrial situations and could not be recommended for other simpler control applications. Hence, they are relatively expensive for developing countries and are not required to be operated in corresponding environments. Our investigations on Grafcet synthesis have permitted to carry out a novel software tool that permits to automatically convert any Grafcet specification code to a control program deployable on very economic microcontrollers. This tool is presented and can be exploited to generate the appropriate solution once the system is well specified in Grafcet language.

The paper is organized as follows: Section 2 presents the problem to be solved with different configurations; the proposed model is explained in Section 3. Section 4 presents the study of the logic controller modelling using Grafcet and a solution synthesis approach to obtain an automatic control system. Section 5 draws a case study with the corresponding Grafcet model and conclusions are drawn in Section 6.

2. Problem statement

Consider a water providing system, associated to a building having m floors, n water supply sources, p energy sources used to pump water, and k tanks. Each floor of the building is supplied in water by a specific tank until the tank is empty. When the corresponding tank is empty, the connected floors are stocked by a global supply pipe connected to urban water distribution or to the highest tank of the plant.

2.1. Water storage of tanks

As shown in Figure 1, each tank is supplied with water by the descent of rainwater, by pumping water from the lower tank, or by opening water supplied by the urban network water distribution. Each tank supplies k floors of the building (for example $k = 3$). Each tank T_i is equipped with three water level sensors which indicate the empty tank (bT_i), the full tank (hT_i) and the average water level (mT_i). There are two ways of conveying water from sources to tanks: energy-free filling with rainwater and the pumping based filling. The urban water is also used to supply the floors when the first two sources are not supplied.

2.2. Energy-free filling

During the rains, the tanks are filled by rainwater harvested from the roofs.

Filling is done sequentially by going from the top tank to the lowest. In the case of low rainfall, the sequential of the tanks filling makes it possible to stock the water collected in priority in the tanks which have the highest potential energy. Especially, the filling of T_i is possible if the upper tank T_{i+1} is already full ($hT_{i+1} = 1$).

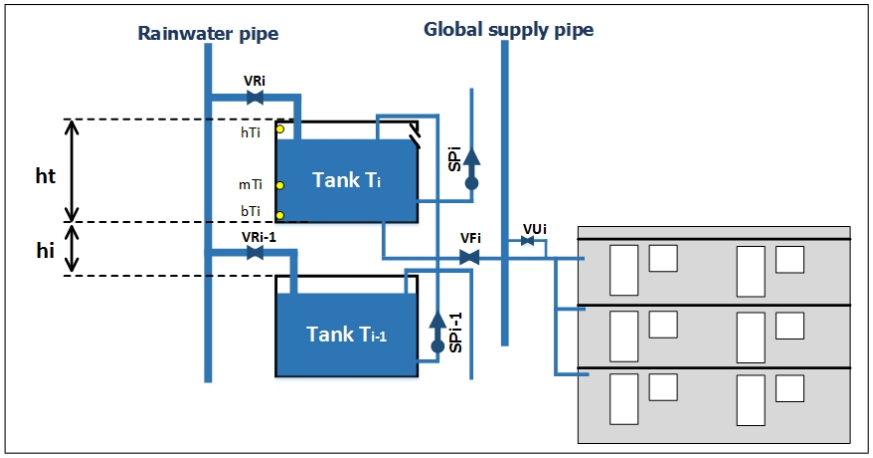


Figure 1. *Schematic of an intermediary tank*

The filling of the tank T_i consists in opening the valve VR_i when all the other valves VR_j ($j \neq i$) are closed. The filling ends when the tank T_i is filled ($hT_i = 1$), the valve VR_i is then closed and the filling of the tank T_{i-1} can begin.

2.3. Filling by pumping

In the absence of rain, the pumping allows to supply water to the tank T_i. It consists in activating the pump SP_{i-1} which transfers water from the tank T_{i-1} to the tank T_i. Depending on the form of energy used to supply the pump, we can distinguish 3 pumping modes :

– Pumping with the sun

In this pumping mode, water pumping from the tank T_{i-1} to the tank T_i is done without energy costs with solar water pumps that work with an electric motor whose power comes from photovoltaic cells located on solar panels that capture power from the sun's light.

In the presence of sufficient solar brightness ($srl = 1$), this zero energy cost pumping mode is systematically activated until the tank T_j is full ($hT_i = 1$) or the water level in tank T_{i-1} is under average level ($mT_{i-1} = 0$). This pumping mode is especially suited to rural environments in Africa. This is because sunlight is plentiful (with over six hours per day of maximum sunlight) and sufficient even for large quantity pumping, and the isolation of rural villages often makes it difficult to supply them with conventional energy supplies. Otherwise the need for water in rural village is generally low enough that it can be covered by solar pumping.

– Pumping with battery

During the night or when the solar radiation is insufficient, the pump SP_{i-1} is activated using batteries power supply when the level of water in the tank T_i is below the lower level ($bT_i = 0$). This pumping mode is activated until the tank T_i is full ($hT_i = 1$) or the water level in tank T_{i-1} is under average level ($mT_{i-1} = 0$).

Batteries accumulate excessive energy created by the photovoltaic (PV) panel system and store it to be used at night when there is no more solar energy input. Batteries can discharge rapidly and yield the current charging, ... For PV water pumping application, peep-cycle batteries are suitable to deliver few amperes required for pumping for hundreds of hours between charges. This type of battery is capable of many repeated deep cycles

and are best suited for PV power systems, compared to shallow-cycle batteries, as those used for starting a car that are designed to deliver several hundred amperes for a few seconds, then the alternator takes over and the battery is quickly recharged. These two types of batteries are designed for different applications and should not be interchanged. Ideally, a battery bank should be sized to be able to store power for 5 days of autonomy during cloudy weather. If the battery bank is smaller than 3 days pumping capacity, it is going to cycle deeply on a regular basis and the battery will have a shorter life. System size, individual needs and expectations will determine the best battery size for your system.

The energy cost of this pumping mode is almost zero because the cost is limited to the maintenance of the battery whose wear is induced by multiplicity of charge/discharge cycles.

– Pumping with urban electricity

During the nights when the batteries are faulty, power supply pumps is ensured by the urban electricity network. This method of pumping is costly since it is invoiced gradually and paid periodically to the urban electrification agency.

To reduce the energy cost, this mode of operation will be applied only if the corresponding tank is below the low level ($bT_i = 0$) and stops when the middle level is reached ($mT_i = 0$).

2.4. Purchasing to the urban distribution network

The urban network water distribution is realized by a public or private company. We assume that this network system guarantees to furnish sufficient pressure to propel water to the upper tank T_k without a need of any auxiliary pumping equipment. When a tank T_i is empty ($bT_i = 0$) and it is not possible to fill it with rainwater nor the water coming from the tank T_{i-1}, the floors of T_i are stocked by a global supply pipe connected to the urban water distribution or *to the highest tank of the plant* (as shown on Figure 3).

2.5. Input/Output configuration of an Intermediary tank

Considering the previous description, the input/output configuration of the controller of an intermediary tank T_i is described by the model presented in Figure 2.

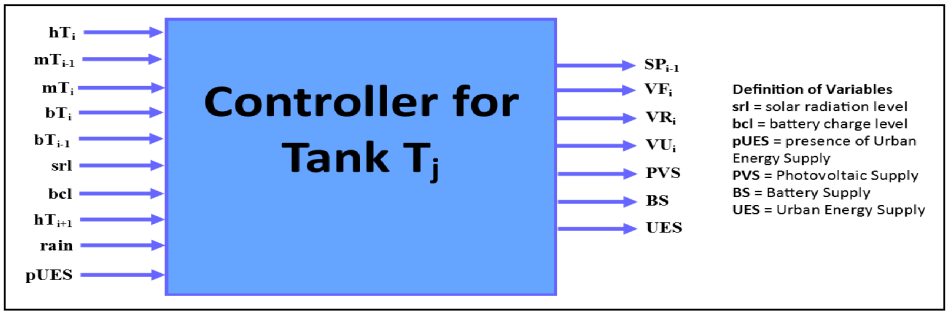


Figure 2. *Input/output configuration of the controller of tank* T_i

3. Proposed model

Once presented the key concepts of the architectural structure with pumping devices and priorities between water sources and power sources, we propose here a switching process that guarantees a significant reduction of pumping energy. In fact, each switching process permits to obtain a particular architectural structure.

There exists many possible switching processes associated to architectural structures and presenting different gain of energy. They depend on many factors: the number of tanks k, the position of tanks with respect to floors, the pumping mechanisms, ... Whatever be the case, the final aim is to choose an architectural structure permitting to guarantee the supplying of every floor of the building with significant reduction of pumping energy.

3.1. Architectural structure

Generally, we may have any number k of tanks. In the case where there is only one tank ($k = 1$), that tank should supply water in all the floors of the building, which means that it would be situated above all the floors. It is then possible that the rainwater is not collected due to the position of that unique tank, which is relatively above the roof. However, if the rainwater is not used, this leads to the loss of a zero cost water source. The rainwater will be collected in a tank near the roof to allow the supply of any level of the building with zero energy.

When there are many tanks, every floor of the building could have its own tank. But if each of the m floors has its own tank, it necessitates m tanks. This is costly since the installation cost will be very huge unnecessarily. It is advantageous to group the floors in blocks to supply them with the same tank. Let η be the number of floors supplied by each tank (for example $\eta = 3$ on Figure 3). $k = \frac{m}{\eta}$ is the number of tanks to supply the m floors of the building. Hence, every tank T_i is preferably supplying the floors $\eta \times (i-1)$, $\eta \times (i-1) + 1$, $\eta \times (i-1) + 2 \ldots$, $\eta \times (i-1) + \eta - 1 = \eta \times i - 1$. It avoids waste of energy, because the water present in that tank may be probably pumped since the tanks at the bottom.

It would be possible (when there is a need) to drive water from a tank to another. If the tank from which water is conveyed to the other one is above, none energy is necessary.

To summarize it, we present on Figure 3 a generic architecture having m floors supplied by k tanks with $k-1$ surface pumps associated and one immersed pump. On that figure, there are $\forall i \in \{1, 2, 3, \ldots, k\}$:

– VR_i is the valve opening rainwater from the rainwater pipe to the tank T_i.

– VF_i is the valve opening water from the tank T_i to the floors associated to it.

– VU_i is the valve opening water from the urban pipe to the floors associated to the tank T_i.

– SP_i is the surface pump pumping water from the tank T_i to the tank T_{i+1} ($i \leq k-1$).

There is also the following valves:

– VR is the valve allowing to evacuate the surplus of rainwater when all the tanks are full.

– VU is the valve driving water from urban distribution network to the floors. It closes whenever the water level in all the k tanks is above the low level ($bT_i = 1$).

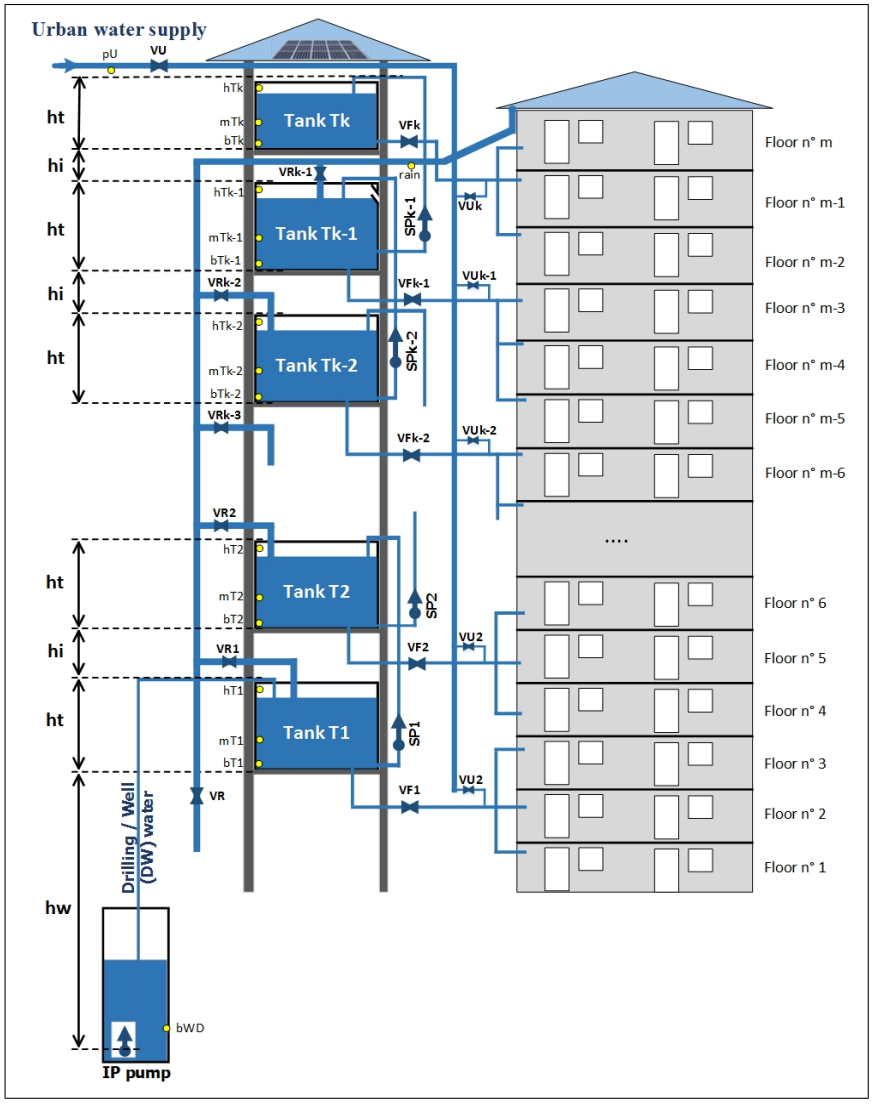


Figure 3. *General physical architecture*

Some sensors are also used to read and send some information to the controller :

– The sensor $rain$ indicates that it is raining. When it is raining, $r = 1$ and all pumping mechanisms stop, except for T_k.

– The sensor pU indicates that the urban water distribution network is supplied ($pU = 1$).

– The sensor bWD located inside the well/drilling tells the controller that there is water inside to be pumped by the immersed pump IP.

3.2. Tanks supplying and energy issues

Whether the pumping energy is zero cost or not, it is necessary and imperative to avoid any waste of energy. The tanks must then be placed and the pumps positioned in

such a way that they can be operated by avoiding the losses of potential energy. Here, we evaluate this energy to justify the chosen pumping mechanisms.

Let $E_{i,j}$ be the amount of energy necessary to supply the tank T_j with the tank T_i, $i, j \in \{1, 2, \ldots, k\}$ and $i \neq j$ with a water volume of $V(m^3)$. Considering that hi is the height between two successive tanks and ht is the height of each tank,

If $i > j$, water of volume V can be conveyed directly from the tank T_i to the tank T_j without been pumping. Then $E_{i,j} = 0$.

If $i < j$, to move the volume V of water from the tank T_i to the tank T_j directly, the height is $(j-i)(ht+hi)+ht$ and $E_{i,j}(1) = \rho gV[(j-i)(ht+hi)+ht]$. We have:

$$E_{i,j}(1) = \rho \times g \times (j-i)(ht+hi)+ht \quad [1]$$

Where $\rho(Kg/m^3)$ is the density of water, and $g(N/Kg)$ is the gravity of Earth.

When $i < j$, if the same volume V of water is moved from T_i to T_j passing through the tanks $T_{i+1}, T_{i+2}, \ldots T_{j-1}$, then $E_{i,j}(2) = E_{i,i+1} + E_{i+1,i+2} + \ldots + E_{j-1,j} = \sum_{l=i}^{j-1} E_{l,l+1}$. $E_{l,l+1} = \rho gV(2ht+hi)$, which is constant. Then $E_{i,j}(2) = \sum_{l=i}^{j-1} \rho gV(2ht+hi) = \rho gV(j-i)(2ht+hi)$ and

$$E_{i,j}(2) = \rho \times g \times (j-i)(2ht+hi) \quad [2]$$

The $\rho gV(j-i)ht$ energy surplus is due to the fact that the water first enters the intermediate tanks being sent higher. The difference between those two amounts of energy is given by $\rho \times g \times V[(j-i)(2ht+hi) - (j-i)(ht+hi) - ht]$. Then

$$E_{i,j}(2) - E_{i,j}(1) = \rho \times g \times V(j-i-1)ht \quad [3]$$

In summary, when $i < j$, since $\rho gV(j-i-1)ht \geq 0$, $E_{i,j}(2) - E_{i,j}(1) \geq 0$. It is then better to move water directly from the tank T_i to the tank T_j without passing through the intermediate tanks. But this advantage is apparent because the realization of such an architecture is very difficult due to the following reasons:

– **Costly and cumbersome**: this requires an amount of $(j-i)$ different water lines equipped with pumps, from the tank T_i to the tanks T_{i+1}, …, T_{j-1}, T_j. For all the building, it requires a total of $1+2+\ldots+(k-1) = \frac{(k-1)k}{2}$ lines. All this is cumbersome and too expensive.

– **Unknowing of the floor where the water will be used**: the volume V of water may be used elsewhere in building floors, except those supplied by the tank T_j.

Water should then be pumped steps by steps from tanks T_i to tanks T_{i+1}, $i = 1, 2, \ldots$

3.3. Number of tanks and energy issues

If there is only one tank, it implies a too wasting of pumping energy, which may not be available. The main reason is that the rainwater may not be used. In fact, it is difficult to place the highest tank in order to collect rainwater and supply the highest floors. Then, **there shall be at least two (02) tanks**.

Let's show that it is better to pump water from tanks to tanks, from the drilling/wells to a specific floor, instead of pumping it up to the highest tank before redistributing it to floors.

3.3.1. Basic case with two tanks

Having two ($k = 2$) tanks T_1 and T_2, we consider two cases: water is pumped directly from the wells/drilling up to T_2 (Case 1), and water is pumped from wells/drilling up to T_1 before beeing moved to T_2 (Case 2).

The following is the calculation of the amount of energy required to pump a volume $V(m^3)$ of water from wells/drilling to the floors. We consider that $V = V_1 + V_2$, where V_1 is the volume of water used to supply the lower floors and V_2 the volume used to supply the upper floors.

Case 1: water is pumped directly up to T_2

Water is pumped directly from the well/drilling to the upper tank T_2, from which all the building is supplied. E_1 is the necessary amount of energy for that and is given by:

$E_1 = \rho \times g(hw + 2ht + hi)(V_1 + V_2)$, then:

$$E_1 = \rho \times g[(hw + ht)(V_1 + V_2) + (ht + hi)(V_1 + V_2)] \quad [4]$$

Case 2: water is pumped to T_1 before beeing moved to T_2

In presence of two tanks, water is pumped from the well/drilling to the lower tank T_1 (supplying the lower floors with V_1) and from T_1 to the upper tank T_2 (supplying the upper floors with V_2). E_2 is the necessary amount of energy for that and is given by $E_2 = \rho \times g[(hw + ht)(V_1 + V_2) + (2ht + hi)V_2]$, then :

$$E_2 = \rho \times g[(hw + ht)(V_1 + V_2) + (2ht + hi)V_2] \quad [5]$$

Where hw is the height between the well/drilling and the tank T_1.

Gain of energy

The difference of energy $E_1 - E_2$ is given by

$$E_1 - E_2 = \rho \times g[(hw + hi)V_1 - ht \times V_2] = \rho \times g[hi \times V_1 + ht(V_1 - V_2)] \quad [6]$$

In practice, ht is very less than hi. It may be possible to have $hi = 6 \times ht$.

If $V_1 = V_2 = \frac{V}{2}$ then $E_1 - E_2 = \rho \times g \times hi \times \frac{V}{2} > 0$

It shows that water needs to be pumped from the well/drilling to T_1 and secondly from T_1 to T_2.

3.3.2. Generalization with k tanks

The previous calculations takes into consideration only two tanks. Let's generalize it with k tanks. For this, we consider the general architecture of Figure 3, and $V = V1 + \ldots + V_k$, $V_i \geq 0, \forall i \in \{1, 2, \ldots, k\}$. V_i is used by the tank T_i to supply the floors numbered $\eta \times i - j, j \in \{0, 1, \ldots, \eta - 1\}$, where η is the number of floors supplied by every tank T_i.

Case 1: water is pumped directly up to T_k

When the volume V of water is pumped from the well/drilling to the tank T_k before providing the other tanks $(T_{k-1}, \ldots, T_1)$, the corresponding amount of energy E_1 is :

$$E_1 = \rho g \sum_{i=1}^{k} V_i[hw + ht + (k-1)(hi + ht)] = \rho g(hw + ht) \sum_{i=1}^{k} V_i + \rho g[(k-1)(hi + ht)] \sum_{i=1}^{k} V_i$$

$$\frac{E_1}{\rho g} = (hw + ht) \sum_{i=1}^{k} V_i + (k-1)(hi + ht) \sum_{i=1}^{k} V_i$$

Case 2: water is pumped steps by steps from T_i to T_{i+1}

E_2 is the amount of energy necessary to pump the volume V of water from tanks to tanks and up to T_k, pumping only the necessary water volume $(V_{i+1} + \ldots + V_k)$ from T_i to T_{i+1}, steps by steps. Hence we have :

$$E_2 = \rho \times g \times (\sum_{i=1}^{k} V_i)(hw + ht) + \rho \times g \times (\sum_{i=2}^{k} V_i)(hi + 2ht) + \rho \times g \times (\sum_{i=3}^{k} V_i)(hi + 2ht) + \ldots + \rho \times g \times (V_{k-1} + V_k)(hi + 2ht) + \rho \times g \times V_k(hi + 2ht)$$

$$\frac{E_2}{\rho g} = (hw+ht) \sum_{i=1}^{k} V_i + (hi+2ht) \sum_{i=2}^{k} V_i + (hi+2ht) \sum_{i=3}^{k} V_i + \ldots + (hi+2ht)(V_{k-1} + V_k) + (hi+2ht)V_k \Rightarrow \frac{E_2}{\rho g} = (hw+ht) \sum_{i=1}^{k} V_i + (hi+2ht)(0V_1 + 1V_2 + \ldots + (k-1)V_k)$$

Then,

$$\frac{E_2}{\rho g} = (hw + ht) \sum_{i=1}^{k} V_i + (hi + 2ht) \sum_{i=1}^{k} (i-1)V_i \quad [7]$$

Gain of energy

The difference of energy $E_1 - E_2$ is then given by:

$$\frac{E_1 - E_2}{\rho g} = (k-1)(hi + ht) \sum_{i=1}^{k} V_i - (hi + 2ht) \sum_{i=1}^{k} (i-1)V_i$$

By reorganizing it into factors of hi and ht, we have

$$\frac{E_1 - E_2}{\rho g} = hi[(k-1) \sum_{i=1}^{k} V_i - \sum_{i=1}^{k} (i-1)V_i] + ht[(k-1) \sum_{i=1}^{k} V_i - 2 \sum_{i=1}^{k} (i-1)V_i] \quad [8]$$

Hence, the difference is given by

$$\frac{E_1 - E_2}{\rho g} = hi \sum_{i=1}^{k} (k-i)V_i + ht \sum_{i=1}^{k} (k - 2i + 1)V_i \quad [9]$$

Sign of $E_1 - E_2$: considering that $k > 1$

$\sum_{i=1}^{k} (k-i)V_i > 0$ and $\sum_{i=1}^{k} (k - 2i + 1)V_i$ may be negative. But, considering the fact that in reality $ht < hi$ (possibly $ht = \frac{hi}{6}$), it can be shown that $|\sum_{i=1}^{k} (k-i)V_i| > |\sum_{i=1}^{k} (k - 2i + 1)V_i|$ and then $E_2 - E_1 > 0$.

If $V_1 = V_2 = \ldots = V_k = \frac{V}{k} = V_0$, we have

$$\frac{E_1 - E_2}{\rho g V_0} = hi \sum_{i=1}^{k}(k-i) + ht \sum_{i=1}^{k}(k-2i+1) \quad [10]$$

But, $\sum_{i=1}^{k}(k-i) = \frac{k(k-1)}{2}$ and $\sum_{i=1}^{k}(k-2i+1) = 0$, then $\frac{E_1-E_2}{\rho g} = \frac{k(k-1)}{2} hi \times \frac{V}{k}$.

In other words,

$$E_1 - E_2 = \frac{(k-1)}{2} \rho \times g \times hi \times V > 0 \quad [11]$$

In summary, the use of a single tank leads to neglect rainwater and a waste of pumping energy. It is necessary to use at least $k = 2$ tanks. To gain energy, water must be pumped from the well/drilling to the lowest tank T_1 using an immersed pump, from any tank T_i to T_{i+1} $(1 \leq i \leq k-1)$ using surface pumps. Also, The rainwater must be collected first in T_k then in T_{k-1}, T_{k-2} ... to gain in potential energy. This water is not lowered before it is brought up again, by loosing energy.

4. Logic Control using Grafcet

From the physical layout architectural water supply system (as shown in Figure 3), we can describe several functional requirements of systems considered in particular contexts. In addition to the case of the whole system, some interesting systems may be obtained considering the fact that there is the absence of one of the following functions: urban electricity supply, drilling/well, urban water supply or battery. In this section, our attention is focused on the whole system from which the solution to other systems cases can be obtained. Initially, let's look at an overview of the Grafcet modeling language. Next, we will propose pieces of grafcet to model the basic functionalities of this system and finish it with the presentation of the synthesis solution strategy.

4.1. The Grafcet modeling language

The Grafcet (Graphe Fonctionnel de Commandes et d'Etapes/Transition), a graphic language for modeling automated systems, is the most international formalism used for high-level description of complex sequential systems [5]. It is widely used in several domains such as home automation, automotive, power generation, manufacturing, etc. A grafcet specifies the behavior of a logic control system. This language has been standardized by the International Electrotechnical Commission IEC 60848 [3], and is understandable and readable compared to other specification languages as Petri net, Relay ladder, Instruction list, etc.

A grafcet (program written in Grafcet language) is a graph that consists of two types of nodes: steps and transitions. A step is represented by a square, while a transition is represented by a horizontal line. The initial steps have a double square. Directed edges are necessarily used to connect steps to transitions or transitions to steps. Each step may be associated with several actions which represent the outputs of a Grafcet graph. An action is symbolized by a rectangle which is connected to a step. The same action can be connected to many steps and it becomes true if at least one of those steps is active. Some

elements are used to separate structurally steps and transitions: when there are many steps in the upstream of a transition, a "junction AND" represented with a double-line is put between those steps and this transition. A "distribution AND", represented with a double-line, is used to separate a transition with many steps when this transition has many steps in the downstream. When there are many transitions in the upstream of a step, a "junction OR" represented with a simple horizontal line is used. A "distribution OR" also represented with a simple horizontal line is used to separate a step to many transitions that are in the downstream of this step.

The Grafcet evolution from the initial state is done by firing transitions according to the following five evolution rules [3]:

– **Rule 1**: At the initial time, all the initial steps are active; all the other steps are inactive.

– **Rule 2**: A transition is enabled when all the steps that immediately precede this transition are active. A transition is fireable when it is enabled and when the associated transition condition is true. A fireable transition must be immediately fired.

– **Rule 3**: Firing a transition provokes simultaneously the activation of all the immediately succeeding steps and the deactivation of all the immediately preceding steps.

– **Rule 4**: When several transitions are simultaneously fireable, they are simultaneously fired.

– **Rule 5**: When a step shall be both activated and deactivated, by applying the previous evolution rules, it is activated if it was inactive, or remains active if it was previously active.

These rules enable the calculation of the subsequent state and the corresponding output signals caused by an input event.

4.2. Grafcet specification model for the system

Here, we propose some general Grafcet models for the specification of processes under control of the logic controller whose the general architecture is presented in Figure 3. In Figure 2, there is the I/O specification of intermediary tanks controller. This is a part of entire controller which should control the modeling of water providing of tanks & floors according to multilevel hierarchical priority access to water, and the switching process between water & energy sources.

4.2.1. Grafcet modeling water providing of tanks

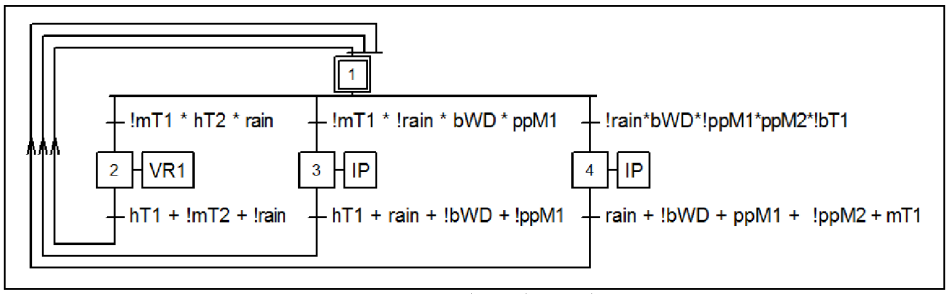
According to the description of the tanks supplying given in subsections 2.2 & 2.3, there are three cases of tanks:

– The lower tank T_1: water is pumped with the immersed pump IP.

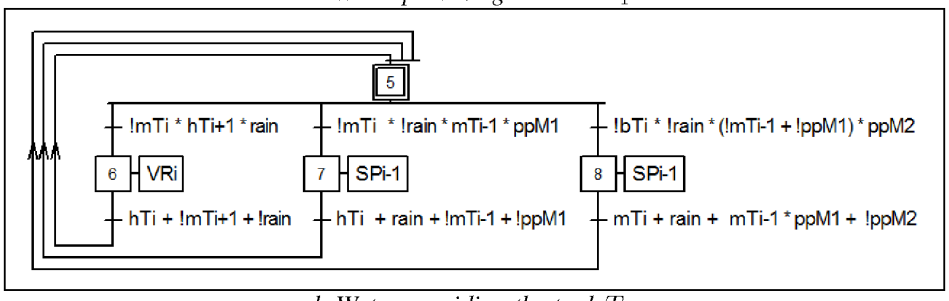
– The highest tant T_k : water is pumped with a surface pump SP_{k-1}.

– The other T_i ($2 \leq i \leq k$) tanks: water is pumped with the same condition as T_k, and the filling of T_i with rainwater is conditioned by the fact that T_{i+1} is already full ($hT_{i+1} = 1$).

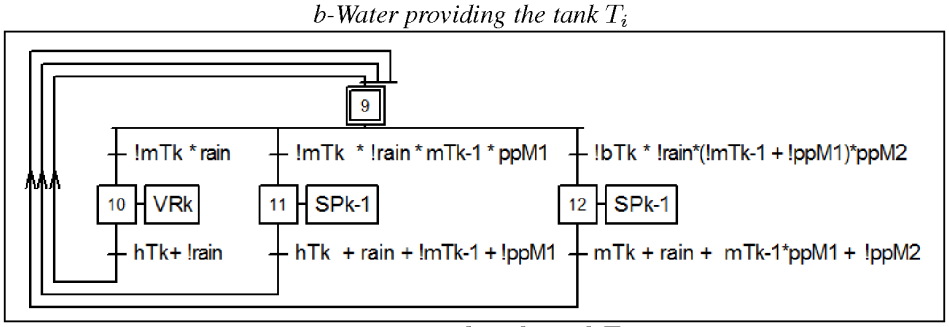
On transitions receptivities of Grafcet , we sometimes use the notation "!" to indicates the negation logical operator.



a-Water providing the tank T_1



b-Water providing the tank T_i



c-Water providing the tank T_k

Figure 4. *Grafcet specification of water providing tanks*

The water providing of tanks is well specified during the modeling of the switching process presented in 4.2.3. This process can be specified with the Grafcet model of Figure 4. Variables $ppM1$ and $ppM1$ indicate the presence of pumping energy. $ppM1 = srl + \overline{srl} * bcl$ concerns the first mode inherent to pumping with zero energy cost (with sun or with storage batteries) and the variable $ppM2 = \overline{srl + bcl} * pUES$ is related to pumping with energy cost, where $pUES$ informs on the presence of urban electricity.

Figure 4.b presents the providing of any tank T_i: When T_i lacks water ($mT_i = 0$) and if there is rainwater ($rain = 1$), the valve VR_{i+1} is opened and T_i is provided until it is full ($hT_i = 1$). If there is not rainwater ($rain = 0$), the second alternative is used when T_{i-1} is provided ($mT_{i-1} = 1$) and there is power to supply the surface pump SP_i. The action SP_{i-1} is put on to pump water from T_{i-1} to T_i.

It is the same with T_1 (Figure 4.a) except that water is pumped from IP pump. Also, bWD indicating the level of water in the drilling/wells. For T_k, when it is raining, the opening of VR_k is not subject to the conditions hT_{k+1} (the tank T_{k+1} does not exists), but only to the condition $rain*!mT_k$ (Figure 4.c).

To manage the fact that when it is raining, tanks are filled from the highest to the lowest, we consider that T_k is filled with rainwater with the condition $!mTk * rain$ (receptivity of the transition going from the step 9 to the step 10); and $\forall T_i, i \in \{1, \ldots, k-1\}$ the condition to fill T_i with rainwater is $!mT_i * mT_{i+1} * rain$ (receptivity of the transition from the step 5 to the step 6), so T_{i+1} must be fulfilled first ($hT_{i+1} = 1$).

In addition, the action VR allows the evacuation of the surplus of rainwater when all the tanks are full($hT_1 * hT_2 * \ldots * hT_k = 1$). It stops when there exists a tank T_j whose water level is below the medium level($mT_j = 0$). In this case, the filling process of T_j starts automatically. To avoid damages between the time all tanks are full and when at least one tank has a water level below the medium, exhaust outlets may be put on each tank. The process of opening and closing VR is specified with the Grafcet model of Figure 5.

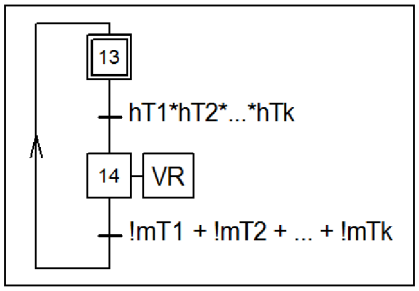


Figure 5. *Grafcet specification of opening/closing the exhaust valve VR*

4.2.2. Grafcet modeling water providing of floors

Every floor is attached to a tank T_i from which it is provided.

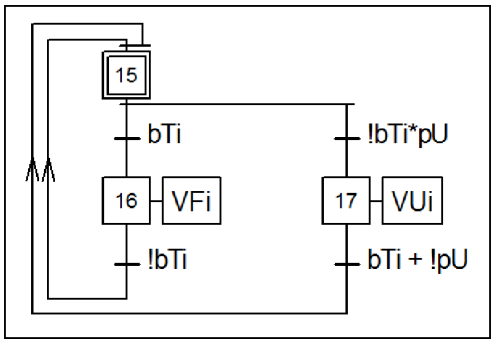


Figure 6. *Gafcet specification of water providing floors*

The valve VF_i should remain opened while there is water inside T_i ($bT_i = 1$). VF_i is closed as soon as the tank is empty ($bT_i = 0$), in which case the urban distribution network takes over with the valve VU_i when it is supplied ($pU = 1$). VU_i is also closed as soon as T_i becomes supplied again ($bT_i = 1$) or the urban network becomes unavailable ($pU = 0$).

The conditions bT_i, $!bT_i$, $!bT_i * pU$ and $bT_i + !pU$ present in the Grafcet specification model of Figure 6 are then justified.

Table 1. *Truth table of MX1 and MX2 multiplexers*

Addresses: $j = j_\lambda j_{\lambda-1} \dots j_2 j_1$					$E_{out} = E_j$
MX_λ	$MX_{\lambda-1}$	...	MX_2	MX_1	Output
0	0	...	0	0	$None$
0	0	...	0	1	E_1
...	...	...	...	...	...
1	1	...	1	1	E_p

4.2.3. Grafcet modeling switching process between energy sources

When modeling water providing of tanks (4.2.1) and water providing floors (4.2.2), the switching process between water sources is handled by activating/deactivating the gates control. Because every pump can function with many energy sources, the switching process between energy sources issue should also be solved. For that, it is necessary to use another electronic component electrically controllable, as multiplexers for example.

With p sources of energy, we rank them from the cheapest to the most expensive: $E_1, E_2, \dots, E_p$. We associate a variable pE_j (presence of energy to the source E_j) to every power source E_j. $pE_j = 1$ when the source E_j has energy ($j = 1, 2, \dots, p$).

A mechanism implemented by a multiplexer is then putted in place. It permits to select between the available sources of the p sources a particular source E_j having the lowest cost, to be connected on the multiplexer output E_{out}.

Let $\lambda = \lceil \log_2 p \rceil$, such that $p < 2^\lambda$ (there should be one emty energy in input). We define the variables $MX_1, \dots, MX_{\lambda-1}$ used in input of a multiplexer MX for the selection between the sources $E_1, E_2, \dots, E_p$, as presented on Figure 7.

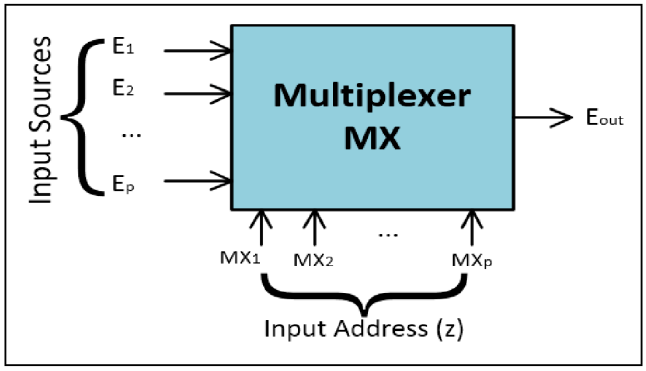


Figure 7. *Multiplexer MX for power sources switching*

To select a particular source E_j (i.e. to connect E_j to E_{out}), $\forall j \in \{1, 2, \dots, p\}$ we can have $j = (j_\lambda, j_{\lambda-1}, \dots, j_1) = (MX_\lambda, \dots, MX_1)$, where $(j_\lambda, j_{\lambda-1}, \dots, j_1)$ is the binary representation of j. We then define the actions MX_{j_z} ($z \in \{1, 2, \dots, \lambda\}$), which are activated when $j_z = 1$ in that representation of j. Table 1 presents the truth table of that mechanism.

As long as there is at least one available source, E_{out} remains supplied until none source has power. Also, when E_{out} is not supplied, it becomes supplied as soon as one source of energy becomes available. The energy is provided independently of the fact that E_{out} is used by a pump or not. It can be specified by the grafcet of Figure 8.

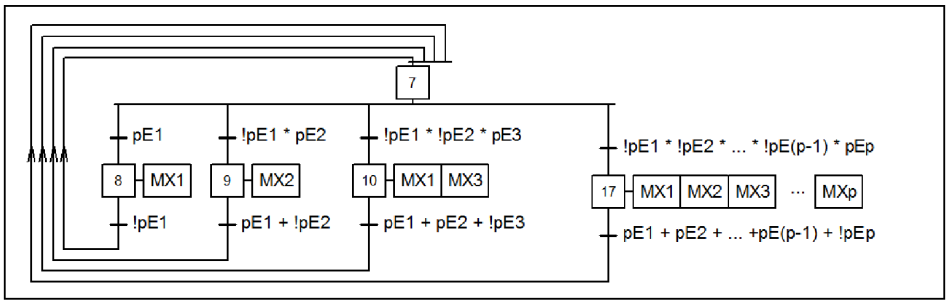


Figure 8. *Grafcet specification of the switching process between power sources providing a pump*

In the specific context of water providing of households, the sensors pE_j correspond to the signals srl, bcl, $pUES$, ... whereas the sources E_j are PVS, BS, UES, ... with $j \in \{1, 2, \ldots, p\}$. Here, we consider that all the pumps (surface & immersed) are supplied with the same sources of energy. In contrary, it is possible to have many types of pumping sources (24vols, 220vols, ...). Sources can then be reorganized in groups and this switching mechanism can easily be applied to each group.

4.3. Synthesis solution strategy

Given the technical design specifications of the water supply system, the grafcet specification model is derived from. We present in this subsection the synthesis process for logical control systems specified in Grafcet. A grafcet model can be realized on programmable controllers (such as microcontrollers or PLCs). However, they do not allow a direct implementation on control targets like microcontrollers, which cost comparatively less than PLCs. For this, we have conducted research on the automatic and robust synthesis of logic controllers specified in Grafcet language and realized on microcontroller targets.

A program synthesis tool is then designed for microcontrollers, according to the Grafcet algebraic equations approach broadly presented in sub-section 4.4.

Referring to Figure 9 , the synthesis process can be split into multiple steps which are broadly described as follows:

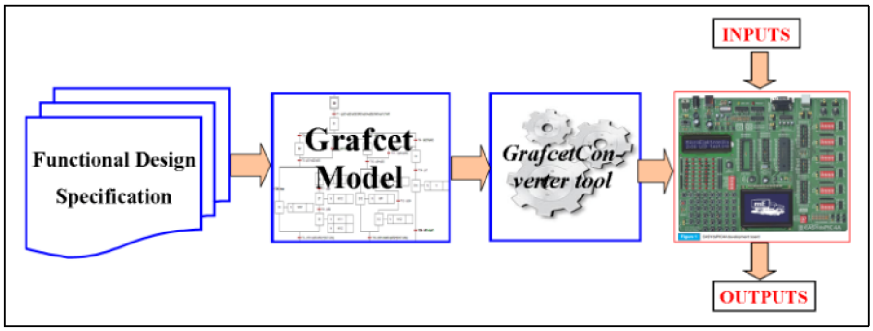


Figure 9. *Synthetic approach solution*

– **Step 1. Functional Design Specification (FDS) of the system** :
The design and development of any system starts with the description of requirements. An FDS is the most important step in the design of any control system. It provides details of the proposed solution to be implemented, to meet user requirements. Thus, FDS is the documentation that defines what the system should do and what functions and facilities should be provided by this system in order to satisfy the requirements. After the FDS specifying all the functions (e.g. pumping, switching, measuring ...), the objects (sensors, pumps, switches ...) and sequential interactions which are associated with the system, the controller is broadly specified with an $Inputs/Outputs$ system presenting all the signals in input and all actions in output.

– **Step 2. Grafcet model** :
The next step is to represent the technical specification of the system by Grafcet model. We propose UniSim software tool in order to edit Grafcet chart [8]. It is a graphical user interface (GUI) that offers gallery of specific graphical objects (shapes) for all Grafcet elements (such as steps, transitions, transition-conditions and actions). The user can easily select these graphical objects, place them onto a drawing page and attach them together in order to create specific grafcet charts. Furthermore, UniSim tool permits to reformat or serialize Grafcet data into XML format [13], this option ensures that this information can be easily processed by another software application (interoperability property), especially the one that we have realized.

– **Step 3.** $GrafcetConverter$ **tool** :
We have built this software tool in order to directly read and extract the Grafcet data from XML format file and generate the appropriate program according to the Grafcet algebraic equations approach presented in subsection 4.4. This program in C language is ready to be compile by the appropriate compiler to build the corresponding binary program. Furthermore, $GrafcetConverter$ tool provides a simulation engine with which we can simulate the Grafcet program and verify whether it meets the requirements or not before generating the C-program for the selected microcontroller. This tool can generate code for microcontrollers, especially the EASYdsPIC4 of the Microship dsPIC family [14].

– **Step 4. Integrated Circuit** :
We are using the EASYdsPIC4A microcontroller, a full-feature and very powerful development system for PIC microcontrollers from Microchip [14] that holds a dsPIC30F4013 microprocessor. In C language, we have implemented the Grafcet interpretation algorithm [5] using the MPLAB IDE. The board is programed by generating and transferring a binary program to the EEPROM board through USB cable or RS-232 COM port. When the code is uploaded in the board and the board reinitialized, the board runs and serves as the sit control of the system.

4.4. Grafcet implementation with algebraic equations

In the program outputted by the synthesis tool, to make sure that the evolution of the Grafcet situation is done according to the Grafcet evolution rules, we automatically generate Grafcet algebraic equations as stated by J. Machado et al in [2]. They use algebraic equations as a common model for simulation and formal verification to obtain a safe controller specification that will be implemented on PLC.

This control program runs according to a scan-based perception which is an effect of the PLC cyclic mode of operation. Figure 10 presents the scan-based cycle processed in every step of the microcontroller (μC) during its sequential execution.

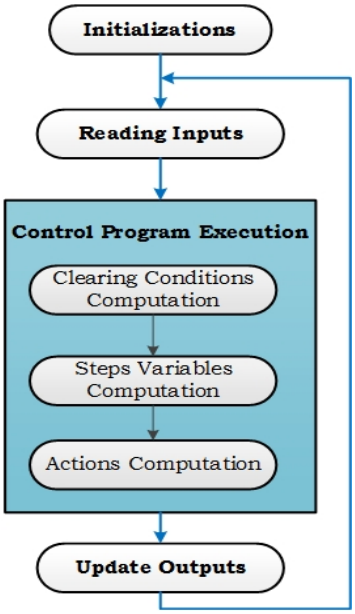


Figure 10. *μC/PLC Scan cycle*

The Grafcet algebraic equations generated when producing a control program are defined as follows:

Let $CC(tr)$ (Clearing Condition) be a Boolean variable associated to every transition tr of a Grafcet. A transition tr can be cleared if it is enabled and if its associated transition condition $TC(tr)$ is true. $CC(tr)$ is calculated as shown on equation E15 as follows:

$$CC(tr) = (\prod_{j=1}^{m} X_j^{tr}) \times TC(tr) \qquad [12]$$

where :

– X_j^{tr} is the step activity Boolean variable associated to step j and directly preceding transition tr,

– $TC(tr)$ is the transition condition associated to transition tr and

– m is the number of steps immediately preceding the transition tr.

Any Boolean step activity associated to each Grafcet step is computed as follows:

$$X_i(0) = \begin{cases} 1 & \text{if } X_i \text{ is an initial step} \\ 0 & \text{else} \end{cases} \qquad [13]$$

$$X_i(t+1) = \sum_{j=1}^{p} CC(tr_j^{i-}) + X_i(t) \times \prod_{j=1}^{q} \overline{CC(tr_j^{i+})} \quad [14]$$

where :

– $X_i(t)$ is the step activity variable of step i in the t^{th} scan cycle,

– $X_i(t+1)$ is the step activity variable of step i in the $(t+1)^{th}$ scan cycle,

– p is the number of transitions directly preceding the step i,

– q is the number of transitions directly succeeding the step i,

– $CC(tr_j^{i-})$ is the clearing condition of transition j, directly preceding the step i and

– $CC(tr_j^{i+})$ is the clearing condition of transition j, directly succeeding the step i.

For actions computation, a Boolean variable A is associated to each action $\mathcal{A}$. An action $\mathcal{A}$ may be associated to several steps and its value $A(t)$ is obtained by computing the $OR(+)$ operation of the step activity variables $X_i^{\mathcal{A}}, i = 1, 2, \ldots, h$ of h steps to which this action is associated. Hence,

$$A(t) = \sum_{i=1}^{h} X_i^{\mathcal{A}}(t) \quad [15]$$

Where $X_i^{\mathcal{A}}(t)$ is the activity variable of a step to which the action $\mathcal{A}$ is associated.

5. Case study

In this section, we present a case study, describing the physical layout architecture of a water & energy supply system and its synthetic scheme.

5.1. System Features

Figure 11 illustrates the water distribution system for a six-floor building, consisting of: several water sources ($n = 3$), several sources of pumping energy ($p = 5$), several water access valves with different priority levels, control system, in addition to two ($k = 2$) water tanks (T_1 and T_2).

As described in the general case (sub-section 3.1), we consider:

Three water sources:

– Urban network water distribution,

– Well/Drilling Water and

– Reserve of rainwater.

Five sources of pumping energy:

– Source E_1 or PVS: is 24 volts solar panel. Its presence is indicated by the electrical power sensor srl.

– Source E_2 or BS: is 24 volts battery, can be charged with solar power. The electrical power sensor bcl indicates its presence.

– Source E_3: is 220 volts AC to 24 volts DC transformer, from Source **E5**. The electrical power sensor $pUES$ indicates its presence.

– Source E_4: is inverter generator, provides 220 volts. Its presence is known thanks to the electrical power sensor bcl.

– Source E_5 or UES: is public electricity network, provides 220 volts. Its presence is known thanks to the electrical power sensor $pUES$.

Pumping mechanisms

– **The surface pump** SP is setup for 24 volts DC; the power supply is provided either by direct solar panels **E1**, or by solar charged battery **E2** or by transformer **E3**. SP permits to pump the water from T_1 up to T_2.

– **Immersed pump** IP is setup for 220 volts; the power supply is provided either by the inverter **E4** or by the urban network **E5**. **IP** permits to pump the well water up to T_2.

Water access valves: VU, VR, VR_1, VF_1, VF_2, VU_1 and VT_2.

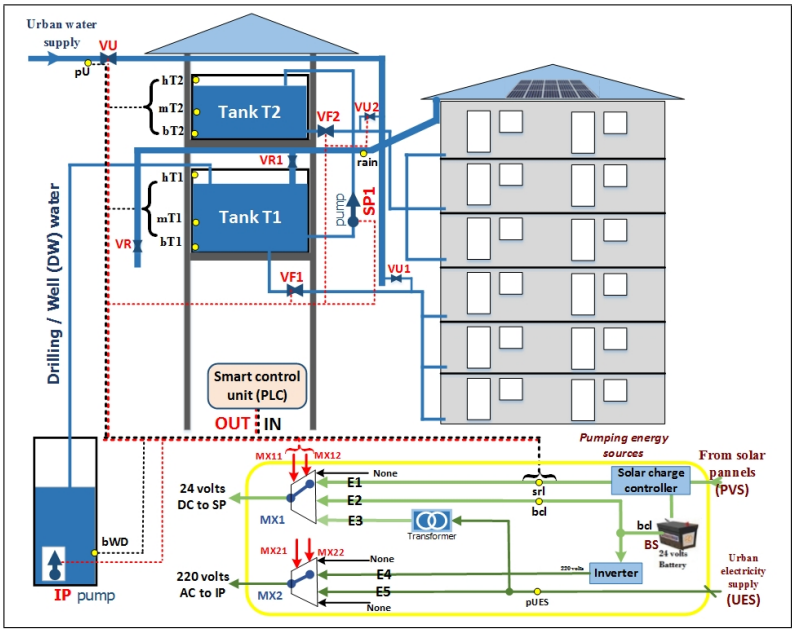


Figure 11. *Synthetic diagram of the system*

This architecture of this case study works according to the detailed description presented in Section 2 where there are water storage of tanks (2.1), the ways of conveying water from sources to tanks: energy-free filling with rainwater (2.2), the pumping based filling (2.3) and pumping purchased to the urban distribution network (2.4). The switching process between power sources and valves is then presented.

Multiplexers are used as presented in 4.2.3 for the switching process between sources of pumping energy. The **surface pump** SP is supplied either by $E1$, $E2$ or $E3$. The selection of the desired source among them can then occur by means of a multiplexer switch MX_1. The **immersed pump** IP is supplied either by $E4$ or $E5$. Hence, the selection of the desired source can occur by means of a multiplexer switch MX_2.

5.2. The logic control system

The water distribution system is completely controlled by a smart control system (controller). It has its proper power supply that makes it autonomous.

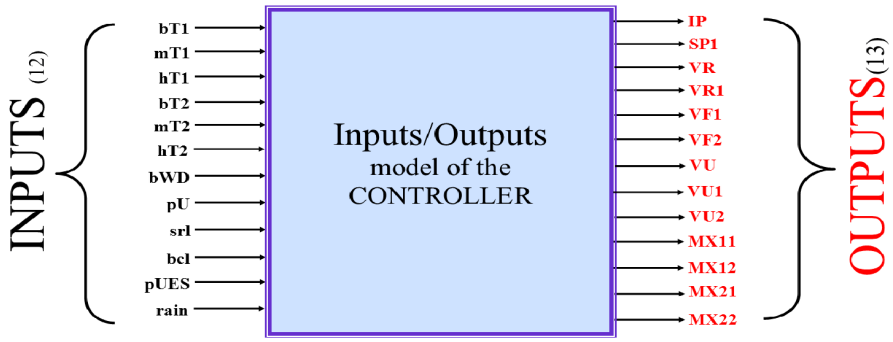


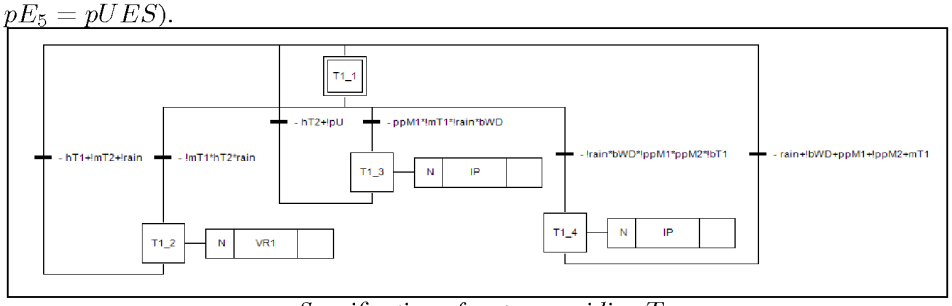
Figure 12. *Input/Output model of the case study logic control system*

The input points of the controller are connected to sensors, which permit to report events and transmit this information as signals to the controller. The 13 digital input sensors used in our system are summarized as follows:

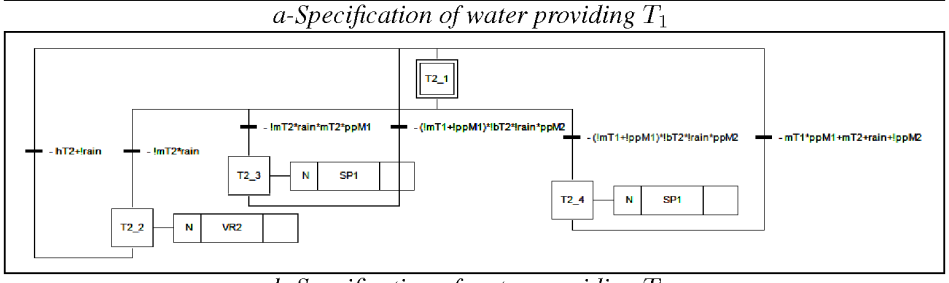
– **Water level sensors** : $bT_1, mT_1, hT_1, bT_2, mT_2, hT_2, bWD$.

– **Pressure sensors** : $pU, rain$.

– **Electrical power sensors**: srl, bcl and $pUES$ ($pE_1 = srl, pE_2 = bcl, pE_4 = bcl$, $pE_5 = pUES$).

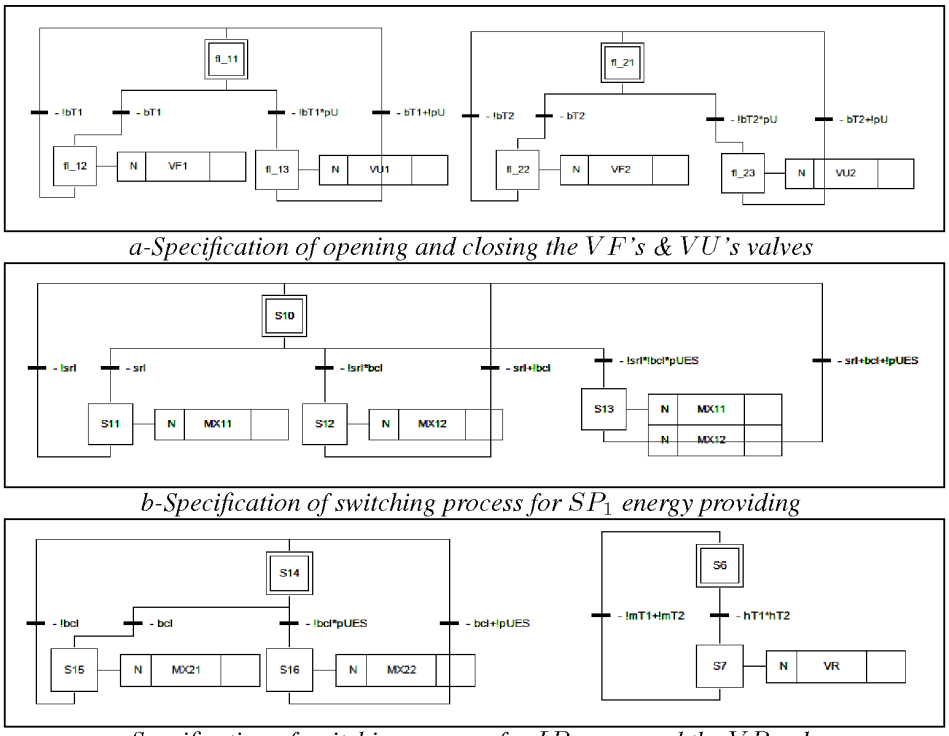


a-Specification of water providing T_1



b-Specification of water providing T_2

Figure 13. *Case study: Grafcet specification of tanks providing*



a-Specification of opening and closing the VF's & VU's valves

b-Specification of switching process for SP_1 energy providing

c-Specification of switching process for IP pump and the VR valve

Figure 14. *Case study: Grafcet specification of valves and swithing process*

The controller will then read, analyze the input signals and activate actions to be activated, according to the μC/PLC scan-base cycle of Figure 10. These generated actions are interpreted as output signals transmitted to output devices (such as actuators and relays) to control the water circulation system. The 13 digital output devices used in our system are :

– **Inductors** : IP, SP.

– **Relays** : $VR, VR_1, VF_1, VF_2, VU, VU_1, VU_2$.

– **Inputs of multiplexer switches** : MX_{11} and MX_{12} (for MX_1 multiplexer to select either $E1$, $E2$ or $E3$ pumping energy sources), MX_{21} and MX_{22} (for MX_2 multiplexer to select either $E4$ or $E5$ pumping energy sources).

The Input/Output model of that logic control system is presented on Figure 12. In that model, power sources are replaced by MX_{11}, MX_{12}, MX_{21} and MX_{22} which permit to command the selection of one of the power sources.

5.3. Grafcet specification model for the system

From functional specifications of the water supply system described in the previous subsection, we derive the corresponding Grafcet specification model, according to the general method described in subsection 4.2. The Grafcet specification of the functioning of this logic control system is shown on the following figures : Figure 13 and Figure 14.

As described in the synthesis process in 4.3, when the control program generated from the Grafcet specification model of the water supply system is compiled an sent to

the EASYdsPIC4A board, this board serves as the core of the smart control unit. Thus, it receives input signals from the sensors (presence of water, pressure, and electric power), analyzes them and generates output signals in form of commands or actions to perform specific acts (open/close valves, turn on/off the pumps or choose a power source).

6. Conclusion and Perspectives

In this paper, we have presented the ability to exploit ICT advantages in order to resolve critical issues ongoing in developing countries, particularly households water supplying with electrical energy issue. We have then designed a generic and sophisticated control system that permits the analysis and the study on the availability of several energy & water sources having different costs; and automatically take a decision on the best source to be selected for ensuring continuous water service in the whole building without any interruption and with the lowest cost. The pumping mechanisms and the switching process between power sources is based on energy calculations which guarantee a considerable reduction in pumping energy.

Furthermore, the designed architecture of the water distribution system presents a generic concept of water supply that may be integrated in a building project as well as in an already existing building, having several floors and supplied through several tanks. The controller of such events-driven systems can be realized thanks to the synthesis tool, *GrafcetConverter*, that reads the Grafcet model of the system under study and produces a C-program which, after compilation and loading in the board, runs at each step by reading inputs and performing the necessary actions

7. References

[1] G. DE TOMMASI, A. PIRONTI, "An educational open-source tool for the design of IEC 61131-3 compliant automation software", *IEEE International Symposium on Power Electronics, Electrical Drives, Automation and Motion*, 2008, pp. 486-491.

[2] MACHADO J, SEABRA E, CAMPOS JC, SOARES F, LE C., "Safe controllers design for industrial automation systems", *Computers & Industrial Engineering*, 2011, Vol. 60(4), pp.635-653.

[3] IEC 60848, "GRAFCET specification language for sequential function charts (3rd ed.)", *IEC*, 2012.

[4] FRANK SCHUMACHER, SEBASTIAN SCHRÖCK, ALEXANDER FAY, "Tool support for an automatic transformation of GRAFCET specifications into IEC 61131-3 control code", *IEEE-ETFA*, 2008, pp. 486-491.

[5] R. DAVID, "Grafcet: A Powerful Tool for Specification of Logic Controllers", *IEEE Transactions on Control Systems Technology*, vol. 3, num. 2, 1995, pp. 253-268.

[6] INSTITUT NATIONAL DE LA STATISTIQUE, "Qualité des eaux de surface et souterraines dans la ville de Yaoundé et son impact sanitaire", *Journal*, July 2013, pp. 2-3.

[7] MINEPAT, "LES ÉNERGIES RENOUVELABLES : une piste à explorer", *La lettre économique du Cameroun*, June 2015, pp. 2-3.

[8] "UNISIM", "UNISIM WEBSITE", *http://wpage.unina.it/detommas/unisim/*, 2006, Accessed: Aug. 2016.

[9] IEC 61131-3, "Programmable Controllers - Part 3: Programming languages (3rd Ed.)", *International Electrotechnical Commission*, Feb. 2013.

[10] HELENA M. RAMOS, FILIPE VIEIRA, DÍDIA I. C. COVAS, " Energy efficiency in a water supply system: Energy consumption and CO2 emission ", *Water Science and Engineering*, 2010, 3(3):pp. 1659-1668.

[11] TRICARICO CARLA, MORLEY MARK S., GARGANO R., KAPELAN ZORAN, DE MARINIS G., SAVIC, DRAGAN, GRANATA F., "Integrated optimal cost and pressure management for water distribution systems", *12th CCWI2013, Procedia Engineering 70*, 2014, 3(3):pp. 331-340.

[12] AFCET, "Normalisation de la représentation du cahier de charges d'un automatisme logique", *Tech. Rep.*, Nov-Dec 1977, pp. 61-62.

[13] PLCOPEN TECHNICAL COMITTEE 6, "XML Formats for IEC 61131-3", *Technical report*, 2005.

[14] MIKROELECTRONIKA, "EasydsPIC4A Development System", *http://www.mikroe.com/products/view/313/easydspic4a-development-system/*, Accessed: Aug. 2015.