Model-driven co-simulation of Cyber-Physical Systems

Sahar Guermazi

▶ **To cite this version:**

Sahar Guermazi. Model-driven co-simulation of Cyber-Physical Systems. Modeling and Simulation. Université Paris Saclay (COmUE), 2017. English. NNT : 2017SACLS333 . tel-01865810

HAL Id: tel-01865810
https://theses.hal.science/tel-01865810

Submitted on 2 Sep 2018

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

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

NNT : 2017SACLS333

Thèse de doctorat

Model-driven co-simulation of Cyber-Physical Systems

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université Paris-Sud

École doctorale n°580 Sciences et Technologies
de l'Information et de la Communication (STIC)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue au CEA Nano-Innov, le 28 Septembre 2017, par

Mme Sahar Guermazi

Composition du Jury :

M. Hans Vangheluwe Professeur, Université d'Anvers – MSDL	Rapporteur
M. Jean-Philippe Babau Professeur, Université de Brest – Lab-STICC	Rapporteur
M. Frédéric Boulanger Professeur, CentraleSupélec – LRI	Examinateur
Mme Laurence Pierre Professeur, Université Grenoble Alpes – TIMA	Examinatrice
Mme Lena Buffoni Maître de Conférences, Université de Linköping – PELAB	Examinatrice
M. Sébastien Gérard Directeur de recherche, Université Paris-Saclay – CEA/LIST/LISE	Directeur de thèse
M. Arnaud Cuccuru Ingénieur de recherche, CEA/LIST – LISE	Co-encadrant
Mme Saadia Dhouib Ingénieure de recherche, CEA/LIST – LISE	Co-encadrante

Je dédie cette thèse

A mes très chers parents Hela et Abderraouf

Je vous dédie ce travail en témoignage de mon profond amour et toute ma gratitude pour les sacrifices que vous avez consenti pour mon instruction, et pour l'amour et l'attention que vous m'avez donnés. C'est grâce à vous que je suis arrivée là.

Malgré la distance qui nous sépare, mes pensées sont avec vous tous les jours. Je serai toujours là pour vous comme vous l'êtes pour moi.

Une pensée particulière à toi, papa. Que dieu te donne la force et le courage.

Que Dieu vous garde pour moi maman, papa, et vous donne santé, bonheur et longue vie afin que je puisse vous combler à mon tour.

A mon très cher époux Bilel

Je te remercie d'être toujours à mes côtés avec ton soutien, ton attention et ton amour. Je te remercie aussi pour ta patience tout au long de mon parcours.

En témoignage de mon amour, de mon admiration et de ma grande affection, je te dédie ce travail en expression de mon estime et mon sincère attachement.

Je prie Dieu pour qu'il te donne bonheur, santé et prospérité et te garde pour moi.

Je t'aime…

A ma très chère sœur Fatma, son mari Mourad et ma nièce Baya,

A mon très cher frère Hsan et son épouse Sirine

Fatma et Hsan, vous êtes mes anges gardiens. Malgré la distance, vous êtes toujours présents pour me soutenir, m'aider et m'encourager. Je ne vous remercierai jamais assez pour tout ce que vous faites pour moi. Je vous aime fort.

Sirine et Mourad : Vous étiez toujours présents pour me soutenir et m'aider. Merci encore.

Baya : Tu es mon rayon de soleil et ma joie de vivre. Je t'aime et t'aimerai toujours…

En témoignage de mon affection et de ma profonde reconnaissance, je vous souhaite tous une vie pleine de bonheur et de succès.

Que Dieu vous protège et vous garde.

A mes grands-parents Hamida et Mohsen

Je vous remercie pour tout ce que vous avez fait pour moi depuis mon enfance. Mon amour pour vous est très profond.

Que Dieu vous préserve santé et longue vie.

A ma belle-mère Sonia et mon beau père Ridha

Je vous remercie pour votre soutien, votre amour. Je vous remercie aussi de m'avoir considérée comme votre fille. Je suis heureuse que vous fassiez partie de ma famille.

Vos prières et vos conseils m'ont toujours accompagné.

Puisse Dieu vous accorder meilleure santé et longue vie.

A mon beau-frère Rami et ma belle-sœur Mariem

Puisse Dieu vous garder, éclairer votre route et vous aider à réaliser à votre tour vos vœux les plus chers.

A ma belle grand-mère Chefia

Je prie Dieu qu'il te prodigue santé et longue vie.

A toute ma famille et tous mes chers amis…

Je vous souhaite tout le succès et le bonheur du monde.

A la mémoire de mon grand-père paternel Hsan, ma grand-mère paternelle Aicha, de ma tante Najoua, du mari de ma tante Fakher, de mon oncle Mohamed et de ma belle grand-mère Mahsouna

Que Dieu, le miséricordieux, vous accueille dans son éternel paradis.

REMERCIEMENT

Je souhaite remercier mon directeur de thèse, Sébastien Gérard, ainsi que mes deux encadrants, Arnaud Cuccuru et Saadia Dhouib, pour leur encadrement, leurs conseils et le temps qu'ils ont consacré pour mener à terme ces travaux de thèse.

Je tiens à remercier Mme Laurence Pierre, Mme Lena Buffoni et M. Frédéric Boulanger d'avoir accepté de faire partie du Jury de ma thèse. Ils m'ont fait l'honneur d'évaluer mes travaux.

Mes remerciement von également à M. Hans Vangheluwe et à M. Jean Philippe Babau pour le temps qu'ils ont consacré pour rapporter mes travaux. Leurs remarques m'ont permis d'envisager des améliorations pour mes travaux. Aussi, ils m'ont fait honneur de faire partie du Jury de ma thèse.

Je remercie tous les membre du LISE qui ont contribué à mener à bien ces travaux.

J'exprime ma gratitude à tous mes amis pour leur écoute, leurs encouragements et leur disponibilité. Je les remercie aussi pour tous les beaux moments qu'on a partagé ensemble.

Je remercie enfin tous ceux qui ont contribué de près ou de loin à l'élaboration de ce travail.

Table of contents

Synthèse en français **9**

Introduction **29**

PART I: RELATED WORK

1. Chapter 1: Foundations and Techniques of Cyber-Physical Systems Modeling and Simulation

1.1. Foundations of modeling and simulation 35
1.1.1. Modeling languages and modeling formalisms 35
1.1.2. Model of Computation (MoC) 35
1.1.3. Simulation tools 38
1.2. Technique of CPS modeling and simulation 38
1.2.1. Translation of models 39
1.2.2. Composition of modeling languages 39
1.2.3. Unification of semantics 40
1.2.4. Composition of models 40
1.2.5. Co-simulation 41
1.3. Discussion and conclusion 42

Chapter 2: Toward FMI-based Co-Simulation of CPS

2.1. About FMI for co-simulation 44
2.1.1. Functional Mock-up Unit (FMU) 45
2.1.2. Master algorithm 48
2.2. Limitations of FMI regarding CPS domain 50
2.2.1. Untimed semantics are not supported (I1) 50
2.2.2. Time events are not handled (I2) 51
2.2.3. Conclusion on FMI issues related to CPS domain 51
2.3. How to address those FMI limitations? 51
2.3.1. Adaptation of semantics at the FMU level 52
2.3.2. Extension of the FMI API 52
2.3.3. Adaptation of semantics at master level 53
2.4. Discussion and positioning 54

Chapter 3: UML models execution - Overview and Key aspects

3.1. Tools for UML models simulation 57
3.1.1. Tools evaluation 57
3.1.2. Discussion and positioning 59
3.2. Computational components modeling and simulation with fUML* 59
3.2.1. Systems of interest 59
3.2.2. Executable models within fUML* 63
3.2.3. Non-executable models within fUML* 65
3.2.4. Addressing fUML* limitations 68
3.3. Outline of the proposed approach 70

3.4. Conclusion 71
PART II: ABOUT THE CONTRIBUTION
Chapter 4: UML-based Master Simulation Tool for modeling and simulation of CPSs
4.1. Architecture of the framework 74
4.1.1. Graphical User Interface features 75
4.1.2. MST-engine features 79
4.2. Validation of the framework implementation 84
4.2.1. Use case: The TankPISystem 84
4.2.2. Import of the FMUs and the definition of the co-simulation scenario 84
4.2.3. Simulation results 85
4.3. Conclusion 87
5. Chapter 5: Integration of untimed UML models in FMI-based co-simulation
5.1. Untimed Models of Transformational Systems 89
5.1.1. Modeling rules for integration in FMI co-simulation 89
5.1.2. Adapting fUML* execution semantics to FMI API 92
5.1.3. Pseudocode of the master algorithm 94
5.1.4. Experience on a representative example 95
5.2. Untimed models of reactive systems 97
5.2.1. Modeling rules for integration in FMI co-simulation 98
5.2.2. Extension of fUML semantics 100
5.2.3. Adapting fUML execution semantics to FMI API 101
5.2.4. Pseudocode of the master algorithm 104
5.2.5. Experience on a representative example 104
5.3. Conclusion 107
6. Chapter 6: Integration of timed UML models in FMI-based co-simulation
6.1. Timed models of transformational systems 109
6.1.1. Modeling rules for interation in FMI co-simulation 109
6.1.2. Adapting fUML semantics to FMI API 112
6.1.3. Pseudocode of the master algorithm 114
6.1.4. Experience on a representative example 115
6.2. Timed models of reactive systems 116
6.2.1. Modeling rules for integration in FMI-based co-simulation 116
6.2.2. Extension of fUML semantics 120
124
6.2.3. Adapting fUML execution semantics to FMI API 124
6.2.4. Pseudocode of the master algorithm 128
6.2.5. Experience on a representative example 128
6.3. Conclusion 130
PART III: EXPERIMENTS
7. Chapter 7: The Case Study: Energy auto-consumption management in smart energy building
7.1. Context 132
7.2. Specification of the case study 133
7.2.1. The ‘ElectricLoad’ 134
7.2.2. The ‘ESS’ 134

7.2.3. The ‘ElectricityGrid’ 134
7.2.4. The ‘PV’ 135
7.2.5. The ‘ControlSelfConsumption’ 135
7.3. Modeling of the case study in Papyrus 139
7.3.1. Modeling of FMUs in Papyrus 139
7.3.2. Modeling of ‘SelfConsumptionControl’ component in Papyrus 140
7.4. Simulation of the case study in Papyrus/Moka 144
7.4.1. The basic control scenario 144
7.4.2. The advanced control scenario 145
7.4.3. Interpretation of the simulation results 147
7.5. Summary of the proposition validation 147
PART IV: CONCLUSION AND PERSPECTIVES
8. Conclusion and perspectives 150
PART IV: ANNEXES
A. ANNEX A: foundational UML (fUML) and PSCS for UML models execution: Syntax and Semantics
A.1. The syntax 154
A.2. The semantics 155
A.2.1. Behavioral semantics 156
A.2.2. Instantiation semantics 159
B. ANNEX B: FMI for co-simulation Standard
B.1. FMU content 161
B.1.1. Structure (XML file) 161
B.1.2. Dynamics (DLL/C-functions) 167
B.2. The master Algorithm 169
B.2.1. Procedures calls order 169
B.2.2. Pseudocode of a basic master algorithm 169
C. ANNEX C: Papyrus/Moka support for FMI for co-simulation standard
C.1. Moka Overview 172
C.1.1. Execution of models based on standards 172
C.1.2. Interactive execution 172
C.1.3. Extension for new execution semantics 173
C.2. Moka extended for the FMI standard 174
C.2.1. Moka as a master for co-simulation 174
C.2.2. Moka as a slave for co-simulation 181
D. ANNEX D: Topological sort on directed graphs
D.1. Introduction to topological sort on directed graphs 182
D.2. Application to a co-simulation graph 183
References 184

List of Figures

Figure I-5-1. CPS modeling and simulation 29
Figure 1-1. The Data Flow MoC 36
Figure 1-2. The Reactive Synchronous MoC 36
Figure 1-3. The Discrete Event MoC 37
Figure 1-4. The Continuous Time MoC 37
Figure 2-1-a. FMI for model exchange 45
Figure 2-1-b. FMI for Co-simulation 45
Figure 2-2. Principle of an FMU simulation 48
Figure 2-3. Basic master algorithm for FMI co-simulation 49
Figure 3-1. A transformational system 60
Figure 3-2. Interaction of a reactive system with its environment 61
Figure 3-3. Transformation is a passive class 63
Figure 3-4. The specification of the operation ‘transform’ with an activity 63
Figure 3-5. The specification of the operation 'multiply' with an activity 64
Figure 3-6. The Game system represented with a UML composite structure 64
Figure 3-7. The players are active classes 64
Figure 3-8. The behaviors of the players represented with activities 65
Figure 3-9. A timed behavior 65
Figure 3-10. A behavior reactive to a change on a value 67
Figure 3-11. Extract semantic elements of PSCS as extension to the fUML semantic model. 69
Figure 3-12. Composition of FMI based co-simulation cases 70
Figure 3-13. The proposed approach for the integration of fUML* and FMI 71
Figure 4-1. The co-simulation Framework 75
Figure 4-2.The Co-simulation Profile 76
Figure 4-3. FMU to UML model transformation 78
Figure 4-4. An example of a CPS composed of four FMUs in a UML composite structure diagram 79
Figure 4-5. FMI-based Co-simulation from UML model to native code 79
Figure 4-6. The FMI standard implementation 80
Figure 4-7. The Variable-order algorithm 80
Figure 4-8. The master step algorithm 81
Figure 4-9. The basic master algorithm for FMUs orchestration enriched with rollback feature and co-simulation graph analysis 82
Figure 4-10. The Master algorithm expressed with UML 82
Figure 4-11. Extensions of the PSCS semantic model for FMI based co-simulation 83
Figure 4-12. The tankPI system and its decomposition 84

Figure 4-13. The import of the TankPI_TankPI_FMU in the framework 85
Figure 4-14. The definition of the co-simulation scenario of the TankPI system.................. 85
Figure 4-15. Execution results in the proposed Dymola 86
Figure 4-16. Execution results in the proposed framework 86
Figure 5-1. An untimed model of a transformational system: structure and behavior........... 91
Figure 5-2. Pseudocode of a master algorithm for a Co-simulation graph connecting FMUs with untimed UML model of a transformational system. 95
Figure 5-3. Co-simulation graph connecting an imported FMU to an untimed model of a transformational system 96
Figure 5-4. Co-simulation results of an untimed model of a transformational system-Basic master 97
Figure 5-5. Co-simulation results of an untimed model of a transformational system-Advanced master 97
Figure 5-6. An untimed UML model of a reactive system: structure and behavior.............. 100
Figure 5-7. Semantic elements which capture semantics of change events 101
Figure 5-8. Co-simulation graph connecting an imported FMU to an untimed model of a reactive system 104
Figure 5-9. Co-simulation results of an untimed model.. 106
Figure 5-10. Co-simulation results of an untimed model.. 106
Figure 6-1. Two-path behavior... 110
Figure 6-2. A timed model of a transformation system.. 111
Figure 6-3. Pseudocode of a master algorithm for a Co-simulation graph connecting FMUs with timed models of a transformational system. 114
Figure 6-4.Co-simulation graph connecting an imported FMU to an timed model of a transformational system 115
Figure 6-5. Co-simulation results of a timed model.. 116
Figure 6-6. Co-simulation results of a timed model.. 116
Figure 6-7. A simple example of a timed model of a reactive system 119
Figure 6-8. Algorithm of discrete event simulation 121
Figure 6-9. Events order in the FEL... 121
Figure 6-10. The DE scheduler model 122
Figure 6-11. Interaction description of the DE scheduler with a semantic element capturing the execution semantics of a timed activity action.. 122
Figure 6-12. Extension of behavioral semantics for timed execution: Object and ObjectActivation 123
Figure 6-13. Extension of behavioral semantics: the activity nodes activations.................. 124
Figure 6-14. Extension of instantiation semantics: Locus and ExecutionFactory 124
Figure 6-15. Pseudocode of a master algorithm for a Co-simulation graph connecting FMUs with a timed model of a reactive system. 127
Figure 6-16. Co-simulation graph connecting an imported FMU to a timed model of a reactive system 128
Figure 6-17. Co-simulation results of a timed model.. 129
Figure 6-18.Co-simulation results of a timed model... 129
Figure 7-1. The ‘Energy Auto-consumption’ system... 133

Figure 7-2. The 'ElectricLoad' component structure ... 134
Figure 7-3. The 'ESS' component structure ... 134
Figure 7-4. The 'ElectricityGrid' component structure ... 134
Figure 7-5. The ‘PV’ component structure ... 135
Figure 7-6. The basic ‘ControlSelfConsumption’ component structure ... 135
Figure 7-7. The basic ‘ControlSelfConsumption’ component behavior ... 136
Figure 7-8. The ‘ControlSelfConsumption’ component structure in Simulink ... 136
Figure 7-9. PeakHourIndicator component Structure ... 137
Figure 7-10. ControlSelfConsumption component structure in Papyrus ... 137
Figure 7-11. The ‘PeakHoursIndicator’ component behavior ... 138
Figure 7-12. The advanced ‘ControlSelfConsumption’ component behavior ... 139
Figure 7-13. The ‘Grid_3inputsPac’ FMU imported in Papyrus ... 140
Figure 7-14. The ‘Load’ FMU imported in Papyrus ... 140
Figure 7-15. The ‘ESS’ FMU imported in Papyrus ... 140
Figure 7-16. The 'PV' FMU imported in Papyrus ... 140
Figure 7-17.UML Model of the basic ‘SelfConsumptionControl’ - Structure and Behavior 141
Figure 7-18. Modeling the Peak-up indicator component with UML ... 142
Figure 7-19.UML model of the advanced self-consumption control – Structure and Behavior ... 143
Figure 7-20. Co-simulation scenario connecting FMUs to the basic 'SelfConsumptionControl' ... 144
Figure 7-21. Simulation results of the basic control scenario in Simulink ... 145
Figure 7-22. Simulation results of the basic control scenario in Papyrus ... 145
Figure 7-23. Co-simulation scenario connecting FMUs to the advanced 'SelfConsumptionControl' ... 146
Figure 7-24. Simulation results of the advanced control scenario in Simulink ... 146
Figure 7-25. Simulation results of the advanced control scenario in Papyrus ... 147
Figure A-1. Syntax and semantics of fUML ... 155
Figure A-2. Extract of semantic visitors considered by fUML ... 156
Figure A-3. Behavioral semantic elements related to the Object visitor ... 157
Figure A-4. The ActivityExecution$^{(sem)}$ and ActivityNodeActivation$^{(sem)}$ visitor ... 158
Figure A-5. Semantic visitor of activity nodes ... 159
Figure A-6. The Locus, the executor, and the Execution Factory in fUML ... 160
Figure B-1. XML schema of the FMI standard (version 2.0) ... 162
Figure B-2. The attributes associated with the 'CoSimulation' element in the XML schema 163
Figure B-3. Attributes of the ‘DefaultExperiment’ element in the XML schema ... 164
Figure B-4. Attributes of 'ScalarVariable' element in the XML schema ... 165
Figure B-5. Example of a model description xml file of an FMU for co-simulation ... 166
Figure B-6.’Unknown’ element attributes in the XML schema ... 166
Figure B-7. 'fmi2Iinstantiate’ function ... 167
Figure B-8. 'fmi2SetupExperiments' function ... 167
Figure B-9. 'fmi2EnterInitializationMode' and ‘fmi2ExitInitializationMode’ ... 168
Figure B-10. 'fmi2GetXXX' function ... 168
Figure B-11.'fmi2SetXXX' function ... 168

Figure B-12. 'fmi2DoStep' function 168
Figure B-13. 'fmi2Terminate' function 169
Figure B-14. Calling sequence of Co-Simulation C functions.. 169
Figure B-15. Co-simulation scenarion composed of two FMUs.. 170
Figure B-16. Pseudocode of the master algorithm 171
Figure C-1. Interactive execution in Papyrus/Moka 173
Figure C-2. The support of several execution engines in Moka 173
Figure C-3. Papyrus/Moka support for FMI for co-simulation.. 174
Figure C-4. The QVTo transformation 178
Figure C-5. The import of FMUs in Papyrus/Moka 179
Figure C-6. The definition of co-simulation scenario in papyrus 180
Figure C-7. The definition of a run configuration in papyrus/Moka 181
Figure D-1. Kahn's algorithm for topological sort on directed graph 182
Figure D-3-a. Directed Acyclic graph.. 183
Figure D-3-b. Directed cyclic graph 183

List of Tables

Table 1-1. Evaluation of verification techniques according to the chosen criteria 43
Table 2-1. Mapping between the FMI API functions and the formalization functions........... 48
Table 2-2. Semantic gap between FMI and non-CT MoCs: model of time and control 50
Table 2-3. Summary of the FMI-based techniques evaluation... 54
Table 3-1. UML tools evaluation 57
Table 4-1. Stereotypes of UML profile for FMI. 77
Table 4-2. Mapping between formalization functions and wrapper functions......................... 83
Table 5-1. Mapping of transformational systems properties to UML modeling concepts...... 89
Table 5-2. Stereotypes to apply for an untimed model of transformational systems 91
Table 5-3. fUML routine for instantiation and initialization of an untimed model of a transformational system 92
Table 5-4. fUML routines for stepwise simulation and data propagation of an untimed model of a transformational system 94
Table 5-5. fUML routines for termination of an untimed model of a transformational system 94
Table 5-6.Mapping of reactive systems properties to UML modeling concepts..................... 98
Table 5-7. Stereotypes to apply for an untimed model of reactive systems 99
Table 5-8. fUML routines for instantiation and initialization of an untimed model of a reactive system 102
Table 5-9. fUML routines for stepwise simulation and data propagation of an untimed model of a reactive system 103
Table 5-10. fUML routines for termination of an untimed model of a reactive system........ 104
Table 6-1. Time modeling in UML models of transformational systems 109
Table 6-2. Stereotypes to apply for a timed model of transformational systems 111
Table 6-3. fUML routines for instantiation and initialization of a timed model of a transformational system 112
Table 6-4.fUML routines for stepwise simulation and data propagation of a timed model of a transformational system 113
Table 6-5. fUML routines for termination of an untimed model of a transformational system 114
Table 6-6. Time modeling with UML for transformational systems 117
Table 6-7. Stereotypes to apply for a timed model of reactive systems 119
Table 6-8.fUML routines for instantiation and initialization of a timed model of a reactive system 125
Table 6-9. fUML routines for stepwise simulation and data propagation of a timed model of a reactive system 126
Table 6-10. fUML routine for termination of a timed model of a reactive system 127

Synthèse en français

I. Introduction

I.1. Contexte et motivations

De nos jours, on utilise de plus en plus des systèmes cyber-physiques (CPS). Nous les trouvons dans plusieurs domaines tel que le domaine de l'automotive, le domaine avionique, le domaine des bâtiments intelligents et de la manufacture. Les CPS intègrent de manière fortement couplée des composants physiques et des composants logiciels [25]. Par exemple, nous trouvons du logiciel embarqué sur des unités de contrôle dans les véhicule, dans un système de gestion de vol sur un avion, ou aussi sur des unités de contrôle pour la gestion d'énergie des réseaux intelligents.

L'utilisation étendue de la modélisation et de la simulation tout au long du cycle de vie du développement des systèmes est l'une des façons les plus utilisées pour concevoir efficacement des systèmes sûrs, sécurisés, performants et fiables. Les CPS sont des systèmes particulièrement difficiles à modéliser et à simuler. En effet, de par la nature hétérogène de leurs composants, leur conception nécessite l'utilisation de différents formalismes de modélisation. En pratique, les composants physiques sont représentés par des modèles physiques qui s'appuient sur le modèle de calcul « Continuos Time » (CT) tandis que les parties cybers sont représenté par des modèles qui s'appuient sur des modèles de calcul tels que le « Discrete Event » (DE) et le « Dataflow » (DF). Ces modèles sont hétérogènes, modélisés avec des langages différents et simulés avec des outils différents. L'une des techniques pour la validation du comportement global du système est la co-simulation. En particulier, la norme « Functional Mock-up Interface » (FMI) offre une interface normative pour coupler plusieurs simulateurs dans un environnement de co-simulation, nommé « Master ». Celui-ci est chargé de fournir un algorithme pour une orchestration et une synchronisation efficace des différents composants du système, nommés « Functional Mock-up Unit » (FMU). Cette norme s'impose de plus en plus dans l'industrie, et est supportée par de nombreux environnements de modélisation et de simulation. Cependant, FMI est initialement conçu pour la co-simulation des processus physiques, avec un support limité des formalismes à événements discrets qui est modèle de calcul et de communication largement utilisé dans les environnements de modélisation spécifiques au logiciel. En particulier, bien qu'UML soit un des langages de référence pour la modélisation de logiciels et soit très couramment utilisé dans l'industrie, aucune des solutions actuelles de co-simulation basées sur FMI ne permet de le prendre en considération.

Notre thèse est que les concepteurs logiciels bénéficieront de l'intégration de leur modèles UML dans une approche de co-simulation basée sur FMI. Cela leur permettra en effet d'évaluer le comportement de leurs composants logiciels dans un environnement simulé, et donc de les aider à faire les meilleurs choix de conception le plus tôt possible dans leur processus de développement. Cette intégration pourrait également ouvrir de nouvelles perspectives intéressantes pour les concepteurs des CPS en leur permettant d'envisager l'utilisation d'un langage largement utilisé pour la modélisation des composants logiciels de leurs systèmes. Et donc de renforcer les interactions entre communautés. Néanmoins, nous n'avons pas trouvé dans la littérature de travaux qui intègrent les modèles UML dans une démarche de co-simulation basée sur FMI.

I.2. Problématiques

Une première analyse de l'état de l'art sur l'intégration des modèle UML dans une approche de co-simulation basée sur FMI montre qu'il y'a des problématiques à trois niveaux :

❖ *Une problématique liée à l'utilisabilité du standard FMI pour les composants logiciels.*

En effet, FMI était historiquement destiné pour la co-simulation de processus physiques (ce qui explique que son API s'appuie sur le modèle de temps CT). L'utilisation de FMI pour la co-simulation de composants logiciels n'est donc pas immédiate.

❖ *Une problématique liée à l'exécutabilité des modèles UML.*

En effet, l'intégration des modèles UML dans une approche de co-simulation nécessite qu'ils soient exécutables. Pour cela, nous avons choisi de s'appuyer sur les standards de l'OMG autour de l'exécution des modèles UML, fUML (« Semantics of a foundational subset for executable UML models ») et PSCS (« Precise Semantics of UML Composite Structures ») que nous noterons fUML* dans la suite. fUML* définit une sémantique précise pour l'exécution d'un sous ensemble de UML. Ces deux normes constituent notre socle de définition qui donne une base intéressante et formelle pour l'intégration des modèles UML dans les approches de co-simulation de systèmes cyber-physiques. Néanmoins le sous-ensemble couvert est limité pour représenter tous les comportements qu'on peut avoir pour des composants logiciels.

❖ *Une problématique liée à la synchronisation entre les des modèles hétérogènes.*

En effet, les modèles diffèrent dans la façon avec laquelle ils interagissent avec l'environnement, dans la façon avec laquelle ils traitent leurs comportements et gèrent le temps. Cette problématique est une problématique classique de la co-simulation de modèles hétérogènes qui a été le centre d'intérêt de plusieurs travaux de recherche. Nous devons traiter, en particulier, l'hétérogénéité entre la sémantique des modèles UML et celle de l'API FMI.

Ces problématiques seront détaillées par la suite pour différents types de systèmes représentatifs des composants logiciels dans les CPS. Pour une bonne compréhension de ces problématiques, nous allons commencer par introduire brièvement les standard FMI pour co-simulation et fUML* pour l'exécution des modèles UML.

I.3. Introduction à FMI

FMI est un standard qui fournit une interface standard pour la co-simultation et pour l'échange de modèles dynamiques originairement conçus avec des outils de simulations différents. Il a été initialement lancé dans le cadre du projet MODELISAR et se poursuit maintenant grâce à la participation de 16 entreprises et instituts de recherche sous le toit de l'Association Modelica. Aujourd'hui, FMI est soutenu par plus de 89 outils [17].

Une entité qui implémente le standard FMI est appelée FMU. Elle est obtenue par l'export d'un modèle à partir d'un outil de simulation conformément au standard FMI : ce qui veut dire que la description du modèle et le solveur contenu dans la FMU sont conformes respectivement à un métamodèle et une API fournis par le standard. L'API définit en particulier des procédures pour l'initialisation et l'instanciation de la FMU, la simulation pas à pas et la terminaison de la simulation. La FMU peut être utilisée :

- ❖ *Pour un échange de modèles*, où la FMU contient un modèle qui sera exécutée par le solveur fournit par l'environnement de simulation qui l'importe (Figure 2) ;
- ❖ *Pour une co-simulation*, où l'objectif est de fournir une interface standard pour coupler deux FMUs ou plus dans un environnement de co-simulation. L'échange de données entre ces

FMU est limité à un ensemble discret de points de communication où chaque FMU est exécutée indépendamment avec son propre solveur (Figure 1).

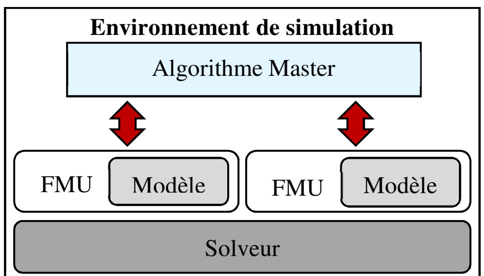

Figure 1. FMI pour échange de modèles

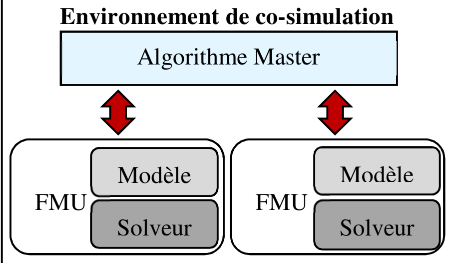

Figure 2. FMI pour co-simulation

L'échange de modèles n'est pas adapté à ce travail vu qu'il n'est pas courant pour les outils UML de fournir des solveurs pour l'exécution de modèles physiques. Nous nous intéressons à la co-simulation où un système est vu comme une interconnexion de FMUs (les 'slaves'), importées dans un environnement de co-simulation (le 'Master'). Le Master est chargé de fournir un algorithme pour orchestrer et synchroniser les FMUs.

La spécification FMI ne standardise pas l'algorithme du 'Master'. L'environnement de co-simulation est donc responsable de fournir son propre algorithme. L'algorithme le plus simple (Figure3) va instancier et initialiser un ensemble de FMUs puis les simuler du début ($tc_0=t_{start}$) à la fin ($tc_n=t_{stop}$) de la simulation par pas de simulation fixe h_{FMU}. Les FMUs sont d'abord instanciées et initialisées (à $t_{c0}=t_{start}$) puis exécutées indépendamment entre deux points de communication discrets "tc_i" et "tc_{i+1}". Le temps avance localement sur les FMU de $h_{FMU} = tc_{i+1}-tc_i > 0$. A ces points de communication, le Master récupère les sorties et met à jour les entrées de toutes les FMUs, puis avance le temps de h_{FMU}.

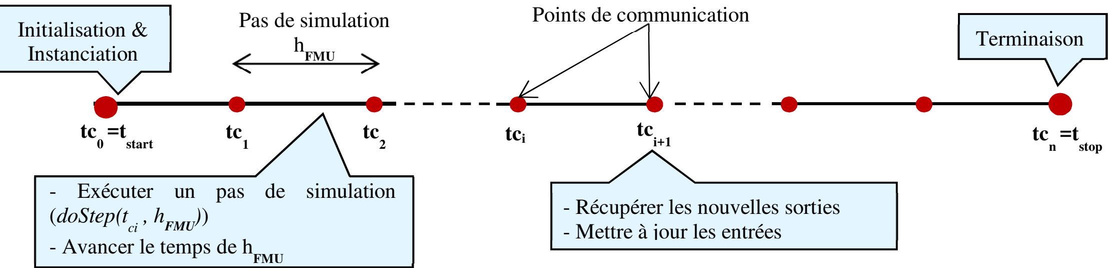

Figure 3. Master de co-simulation basique

I.4. Introduction à fUML*

fUML définit une sémantique d'exécution précise pour un sous-ensemble de UML (à savoir respectivement, un sous ensemble de la modélisation structurelle à base de classes et un sous ensemble de la modélisation comportementale à base d'activité et d'actions, et la partie modélisation à base de classes composites structurées de UML). Le standard PSCS est une extension de fUML notamment pour les structures composites (Figure 4).
A chaque élément syntaxique de ce sous-ensemble est associé un visiteur sémantique qui capture sa sémantique d'exécution. L'ensemble de ces visiteurs forme le modèle sémantique de fUML*.

L'instanciation des visiteurs sémantiques est gérée par deux éléments spécifiques :

- Le « Locus » : représente la mémoire dans laquelle sont stockés tous les visiteurs sémantiques des éléments du modèles.

- La « ExecutionFactory » : responsable pour l'instanciation des visiteurs sémantiques qui capturent une sémantique comportementale.

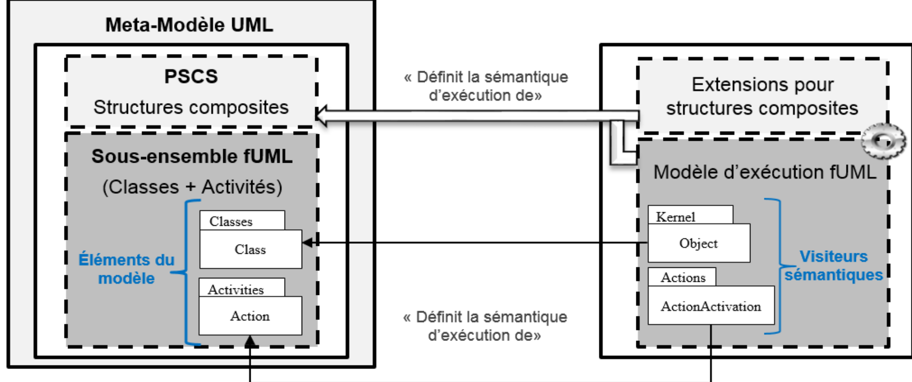

Figure 4. Syntaxe et sémantique de fUML*

Les types de composants logiciels nous voulons modéliser en UML et intégrer dans une approche de co-simulation sont identifiés à partir d'un cas d'étude du domaine des réseaux intelligents. Pour chaque type de composant identifié, nous présentons une projection des problématiques identifiées auparavant.

II. Projection des problématiques sur un cas d'étude

Le cas d'étude en question est un système de gestion de l'autoconsommation d'énergie électrique sur un réseau intelligent. Le système intègre des parties physiques et autres cybers. Les modèles de la consommation d'énergie électrique (« Load »), de la production de l'énergie électrique par des panneaux photovoltaïques (« PV »), du réseau électrique (« Electricity grid ») ainsi que celui de la batterie (« ESS ») dans laquelle l'énergie produite est stockée sont spécifiés par des modèles physiques et donc sont représentés avec des FMUs. L'ensemble de ces composants physiques est contrôlé par un composant logiciel qui définit une stratégie de contrôle (« SelfConsumptionController »), et est représenté par un modèle UML. Celui-ci calcule une consigne de charge ou de décharge pour la batterie dans le but de favoriser l'autoconsommation en énergie du système.

Deux variantes de stratégie de contrôle sont à modéliser et à simuler :

❖ Une première stratégie implémente un contrôle basique qui consiste simplement à donner une consigne de charge ou de décharge à la batterie qui alimente de réseau en comparant la

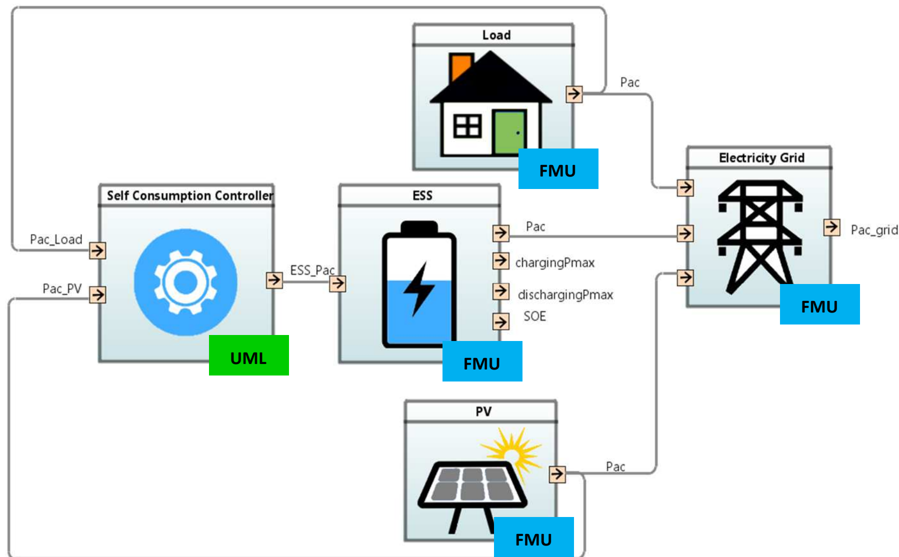

Figure 5. Système de gestion de l'autoconsommation avec stratégie de contrôle basique

consommation d'énergie sur le réseau à la production photovoltaïque (Figure 5). En littérature, ce type de composant logiciel est dit transformationnel [55] : système passif dont ses sorties ne dépendent que de ses entrées, et s'appuie sur le temps logique (modèle de calcul DF).

❖ Une deuxième stratégie plus intelligente qui va calculer une nouvelle consigne en prenant en compte l'état du système (ici l'état de charge de la batterie) ainsi que la période de la journée, i.e. heures creuses ou heures pleine (Figure 6). C'est un composant réactif aux changements de valeurs à l'entrée [21].
Un autre composant (« Peak Hour Indicator ») est nécessaire. Il indique le passage d'une période d'heures creuses à une période d'heures pleines (et inversement). Ce dernier est réactif aux évènements temporels [21].

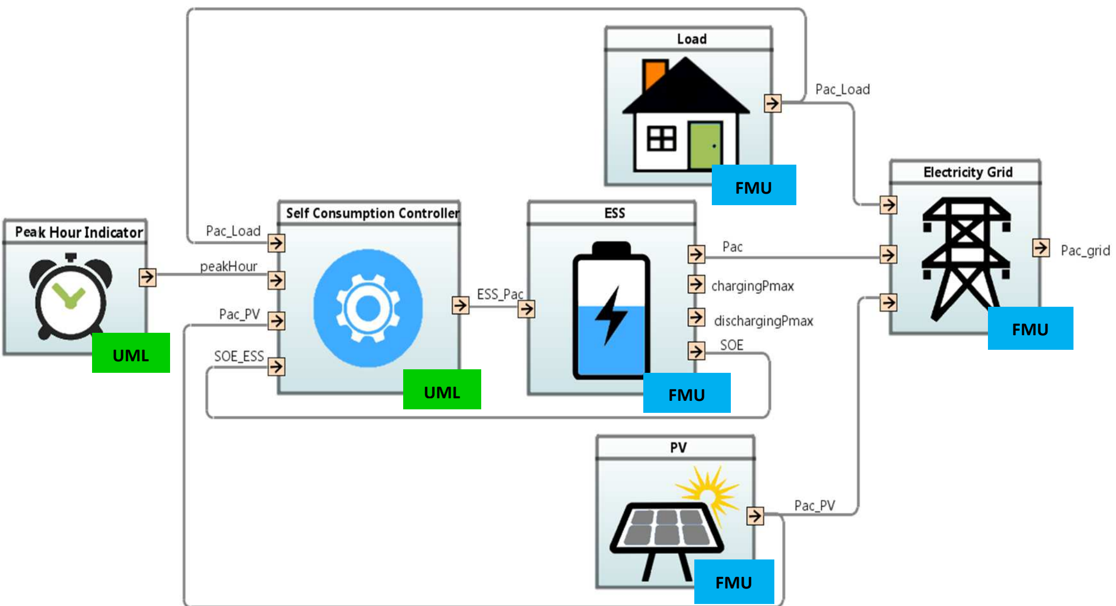

Figure 6. Système de gestion de l'autoconsommation avec stratégie de contrôle avancée

Dans la suite nous détaillerons les problématiques énoncées dans la section I.2. Pour chaque type de système représentatif de composants logiciels, i.e, système transformationnel, système réactif aux changements de valeurs, et systèmes réactifs aux évènements temporels.

II.1. Problématiques dans le cas des systèmes transformationnels

❖ *Utilisabilité du standard FMI pour les composants logiciels.*

Les systèmes transformationnels s'appuient sur le modèle de temps logique. Ce type de comportement non temporisé n'est pas pris en compte par le standard FMI. En effet, d'après la spécification FMI, on doit pourvoir affecter une valeur à une variable à tout instant t. De plus, le pas de simulation nul n'est pas autorisé.

❖ *Exécutabilité des modèles UML.*

Le sous-ensemble fUML* couvre l'ensemble des éléments nécessaires à la modélisation de la structure et du comportement de ce type de système et à son exécution.

❖ *Synchronisation entre les des modèles hétérogènes.*

Le comportement souhaité pour les systèmes transformationnels est de produire la sortie une fois calculée. Etant donné que le composant s'appuie sur du temps logique, le temps de calcul est supposé nul. Par conséquent, la sortie d'une entrée reçue à $t=t_{in}$ doit être propagée au même moment que la réception de l'entrée ($t_{out}=t_{in}$).

Si on exporte ce composant en une FMU conforme au standard FMI, et on le simule avec l'algorithme de master décrit dans la section I.3, la sortie d'une entrée reçue à $t=t_{in}$ sera propagée à $t_{out}=t_{in}+h$ (h>0, étant le pas de simulation choisi par le master).
Cela engendre donc un retard de propagation des données comme le montre la Figure 7.

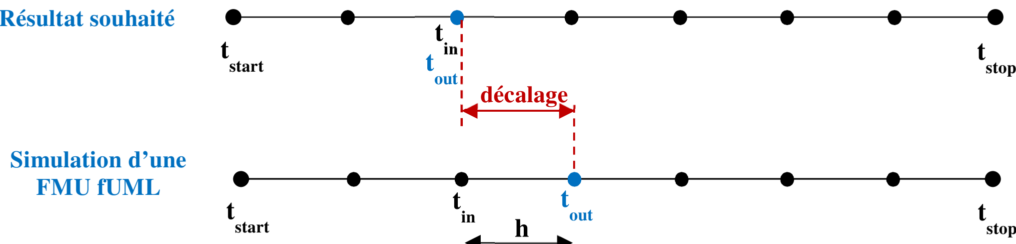

Figure 7. Problème de synchronisation entre FMI et modèle de temps 'Data-Flow'

II.2. Problématiques dans le cas des systèmes réactifs aux changements de valeurs

- *Utilisabilité du standard FMI pour les composants logiciels.*

FMI ne supporte pas les réactions instantanées aux évènements, en particulier aux changements de valeurs à l'entrée.

- *Exécutabilité des modèles UML.*

Le sous-ensemble fUML* ne couvre pas l'ensemble des éléments nécessaires à la modélisation de la structure et du comportement de ce type de système et à son exécution. En effet, ce sous-ensemble ne fournit pas d'éléments pour la modélisation de comportements réactifs aux changements de valeurs à l'entrée.

- *Synchronisation entre les des modèles hétérogènes.*

Le comportement souhaité pour les systèmes transformationnels est de réagir instantanément à un changement de valeur à l'entrée. Par conséquent, la réaction d'une entrée reçue à $t=t_{in}$ doit être propagée au même moment to que la réception de l'entrée ($t_{out}=t_{in}$).

Si on exporte ce composant en une FMU conforme au standard FMI, et on le simule avec l'algorithme de master décrit dans la section I.3, la réaction à une entrée reçue à $t=t_{in}$ sera propagée à $t_{out}=t_{in}+h$ (h>0, étant le pas de simulation choisi par le master). Cela engendre donc un retard de propagation des données comme le montre la Figure8.

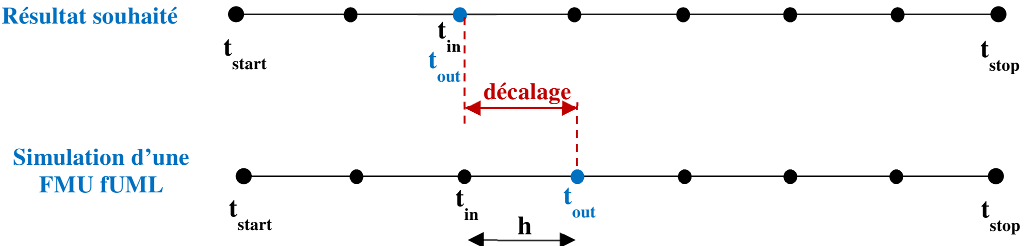

Figure 8. Problème de synchronisation entre FMI et modèle de temps 'Synchronous Reactive'

II.3. Problématiques dans le cas des systèmes réactifs aux évènements temporels

- *Utilisabilité du standard FMI pour les composants logiciels.*

FMI ne supporte pas les réactions instantanées aux évènements, en particulier aux évènements temporels.

- *Exécutabilité des modèles UML.*

Le sous-ensemblede fUML* ne couvre pas l'ensemble des éléments nécessaires à la modélisation de la structure et du comportement de ce type de système et à son exécution. En effet, ce sous-ensemble ne fournit pas d'éléments pour la modélisation de comportements réactifs aux évènements temporels.

❖ *Synchronisation entre les des modèles hétérogènes.*

Le comportement souhaité pour les systèmes réactifs aux évènements temporels est de produire une sortie à des moments discrets définis par des évènements temporels. Par conséquent, si deux évènements e_1 et e_2 sont prévus aux instants te_1 et te_2 respectivement, une première sortie doit être produite à $t_{out}=te_1$ puis une deuxième sortie à $t_{out}=te_2$.

Si on exporte ce composant en une FMU conforme au standard FMI, et on le simule avec l'algorithme de master décrit dans la section I.3, la réaction à e_2 sera produite à $tout=t_{in}+h> te_2$ alors que l'évènement e_1 ne sera pas pris en compte vu qu'une nouvelle sortie a été calculée à un instant postérieur. Cela engendre donc un retard de propagation des données ainsi qu'une perte d'évènements comme le montre la Figure9.

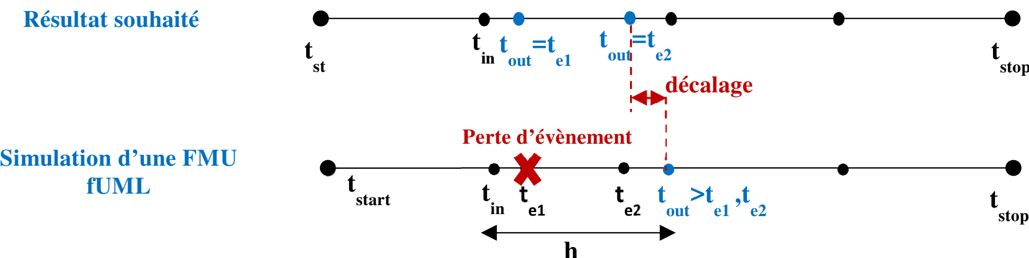

Figure 9. Problème de synchronisation entre FMI et modèle de temps 'Discrete Event'

II.4. Récapitulatif sur les problématiques

Pour récapituler (Tableau1), nous avons remonté quatre problèmes spécifiques à l'intégration de fUML* et FMI.

Tableau 1. Récapitulatif des problématiques d'intégration des modèles UML dans une approche de co-simulation basée sur FMI

	Système transformationnel	Système réactif aux évènements temporels	Système réactif aux changements de valeurs
FMI	P1	P2	P3
fUML*	-	P2	P3
Synchonisation FMI et fUML*	P4	P4	P4

❖ *P1 : Pas de support pour comportement non temporisé.*

Une première problématique qui ressort de la spécification du standard FMI. Elle concerne le fait que les composants conformes à FMI ne supportent pas les comportements non temporisés.

❖ *P2 : Pas de support pour les comportements réactifs aux évènements temporels.*

Une deuxième problématique qui ressort des standard FMI et fUML*, elle concerne le fait que les évènements temporels ne sont pas supportés par les ces standards syntaxiquement et sémantiquement.

❖ *P3 : Pas de support pour les comportements réactifs aux changements.*

Une troisième problématique ressort des deux standards aussi, elle concerne le fait que les réactions instantanées aux changements de valeurs ne sont pas supportées par les standards FMI et fUML*.

- *P4 : Décalage de propagation des valeurs et/ou perte d'évènements.*

Une dernière problématique qui ressort du gap sémantique entre l'API FMI et la sémantique d'exécution des modèles UML qui entraine un décalage de propagation des données et la perte des évènements temporels.

Après avoir déterminé les différents problèmes spécifiques à l'intégration des modèles UML dans démarche de co-simulation basée sur FMI, nous avons exploré l'état de l'art pour analyser des solutions qui ont été déjà proposées pour des problématiques semblables si elles existent. Cet état de l'art nous a permis de se positionner et de construire une solution globale à ces problèmes.

III. Positionnement et solution proposée

III.1. Etat de l'at autour de fUML*

Pour résoudre les problématiques P2 et P3 liées fUML*, il est nécessaire de permettre l'exécution des comportements réactifs aux changements de valeurs et aux évènements dans fUML*. Trois approches peuvent être utilisées :

- *La première approche* consiste à utiliser le langage CCSL introduit par le standard MARTE [29] pour appliquer des contraintes de temps aux modèle UML. Ce langage fournit plusieurs modèles de temps. Une possibilité est d'utiliser des outils qui implémentent cette approche. Le problème rencontré était que les approches proposées faisaient des hypothèses fortes sur la manière dont la syntaxe et la sémantique du langage sont définie. Leur intégration avec fUML* n'était pas directe puisque cela nécessitait l'adaptation du modèle sémantique fUML* à l'architecture proposée.
- *Une deuxième approche* consiste à mettre en place une entité qui contrôle les exécutions temporisées. Cette approche est utilisée dans des outils de simulation tel que SystemC[1] et Ptolemy[2]. Cette approche fournit un seul modèle de temps, le Discrete Event. L'avantage est que son implémentation est indépendante de l'architecture de fUML*. Cette entité sauvegarde une liste d'évènements temporel et les exécute dans un ordre chronologique. Le temps est avancé par pas discret correspondant à la date relative d'un évènement.
- *Une troisième approche* consiste à étendre le modèle sémantique de fUML* avec la syntaxe et la sémantique nécessaire. Cette approche a été déjà utilisée pour étendre le modèle d'exécution de fUML pour le support des structures composites. Une première étape consiste à déterminer un ensemble minimal d'éléments UML nécessaires à la modélisation de comportements réactifs aux changements et aux évènements temporels, et/ou utiliser les profils UML si nécessaire pour expliciter des informations importantes propriétés du système, puis la définition de nouveaux sémantiques visiteurs qui capturent la sémantique d'exécution de cet ensemble.

Nous avons opté pour une combinaison des deux dernières approches comme une solution partielle de P2 et P3.

III.2. Etat de l'art autour de FMI

En se basant sur la spécification FMI, nous avons constaté l'existence de trois méthodes pour la

[1] http://hdl.telecom-paristech.fr/sc_intro.html
[2] http://ptolemy.eecs.berkeley.edu/ptolemyii/

prise en compte des nouveaux modèles de calcul dans une approche de co-simulation basée sur FMI :

➢ L'adaptation de sémantique au niveau de la FMU.

Cette méthode consiste à exporter un modèle en une FMU. La FMU obtenue est conforme au standard FMI. Cette méthode est donc bien acceptée en industrie. Néanmoins, elle ne résout pas les problèmes P1, P2, P3 et P4 liés à l'intégration des modèles UML dans une approche de co-simulation basée sur FMI.

➢ L'extension de l'API de FMI.

Cette approche consiste à étendre FMI pour permettre d'exposer plus d'informations sur les modèles, et d'offrir plus de capacités quant à leur exécution. Cette approche permet de proposer des solutions à P1, P2, P3, et P4 complètement conformes au standard et donc acceptées en industrie. Par conte cela nécessite la validation du consortium.

➢ L'adaptation de sémantiques au niveau du master.

La meilleure solution serait de combiner les deux premières méthodes : une extension du standard FMI avec de nouvelles capacités, et l'export de modèles en FMUs. Néanmoins, cette approche est à présent non applicable vu que les extensions nécessaires aux standard FMI ne sont pas prises en compte.

La troisième méthode est une approche intermédiaire. Elle permet la réutilisation de FMUs pour les parties physiques, et la résolution de P1, P2, P3 et P4 tout en gardant les modèles logiciels en boites blanches. Le seul souci de cette approche est qu'elle impose une restriction quant à l'environnement de co-simulation utilisé, i.e, il doit être un outil UML.

III.3. Positionnement et solution proposée

En conclusion, trois solutions partielles ont été retenues pour la résolution de P1, P2, P3 et P4 :

❖ **S1 :** étendre la syntaxe et sémantique fUML*. Il faut :
- Identifier les éléments UML nécessaires.
- Implémenter leur sémantique d'exécution (définition de visiteur sémantiques et leur instanciation).

❖ **S2 :** déléguer le contrôle des exécution temporelles à une entité externe. Il faut :
- Implémenter un ordonnanceur pour l'exécution des comportements temporisés

❖ **S3 :** implémenter un master de co-simulation avancé responsable pour :
- L'orchestration des composants impliqués où il faut identifier les routines équivalentes de fUML* (et ses extensions).
- La synchronisation où il faut adapter le pas de simulation au type du composant. Pour cela le master a besoin d'avoir les informations nécessaires sur modèle de temps du modèle UML ; cela peut être fait en ajoutant des annotations (stéréotypes) sur les modèles UML.

Comme le résume le Tableau2, la résolution de P1 et P4 nécessite l'utilisation de S3. La résolution de P2 nécessite l'utilisation de S1, S2 puis S3. La résolution de P3 nécessite l'utilisation de S1 puis S3.

Tableau 2. Récapitulatif des solutions aux problématiques identifiées

	S1	**S2**	**S3**
P1: Pas de support pour comportement non temporisé (FMI)			**X**

P2: Pas de support pour les comportements réactifs aux évènements temporels (FMI et fUML*)	**X**	**X**	**X**
P3: Pas de support pour les comportements réactifs aux changements (FMI et fUML*)	**X**		**X**
P4: Décalage de propagation des valeurs et perte d'évènements (Synchronisation FMI et fUML*)			**X**

La solution proposée pour l'intégration des modèles UML dans une approche de co-simulation basée su FMI est illustrée dans la Figure10.

Figure 10. Approche proposée pour l'intégration des modèles UML dans une approche de co-simulation basée sur FMI

IV. Contributions

IV.1. Environnement de co-simulation dans Papyrus

Cette approche est outillée dans Papyrus, modeleur UML/SysML intégré à Eclipse. Il fournit une implémentation des standard OMG relatifs à l'exécution des modèles UML via son moteur d'exécution Moka. Nous avons implémenté un environnement de co-simulation basé sur le standard FMI. Dans un premier temps, cet environnement permet de définir des scenarios de co-simulation qui assemble des FMUs importées, simuler ces scenarios par un algorithme de co-simulation intégré dans Moka et enfin stocker et visualiser les résultats de simulation. Les détails de l'implémentation sont présentés dans le chapitre 4 du manuscrit.
Le but est d'étendre cet environnement pour permettre en plus la définition de scénarios de co-simulation qui assemblent des FMUs à des modèles UML et leur simulation (Figure11).

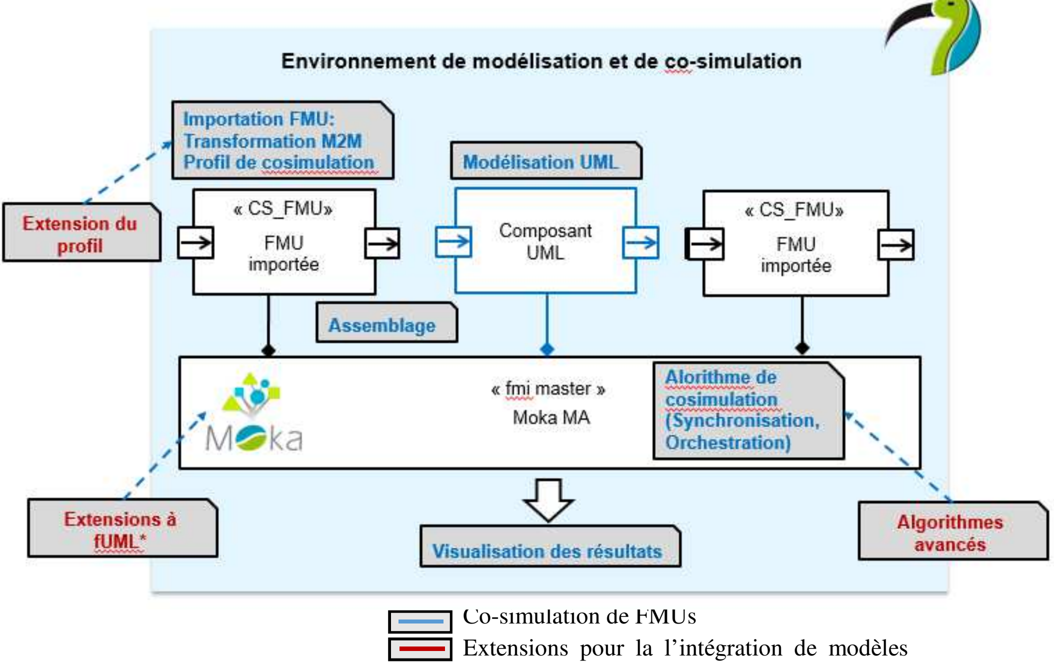

Figure 11. Environnement de co-simulation dans Papyrus

IV.2. Application de l'approche aux systèmes transformationnels

IV.2.1. Règles de modélisation UML

La Figure12 illustre les différents éléments UML nécessaires à la modélisation de composant logiciels transformationnel. Un système transformationnel est représenté par une classe passive auquel est appliqué le stéréotype « CS_Untimed ». Ces deux éléments reflètent le modèle de calcul Data Flow (DF) du composant. Cette classe doit avoir des ports d'entrée, des ports de sortie, et une opération qui implémente le calcul à faire quand le composant est invoqué. Pour la distinguer, le stéréotype « CS_Operation » lui est appliqué. Cette opération est définie par une activité qui va lire les valeurs sur les ports d'entrée, faire le calcul nécessaire, et mettre à jour les valeurs des ports de sortie.

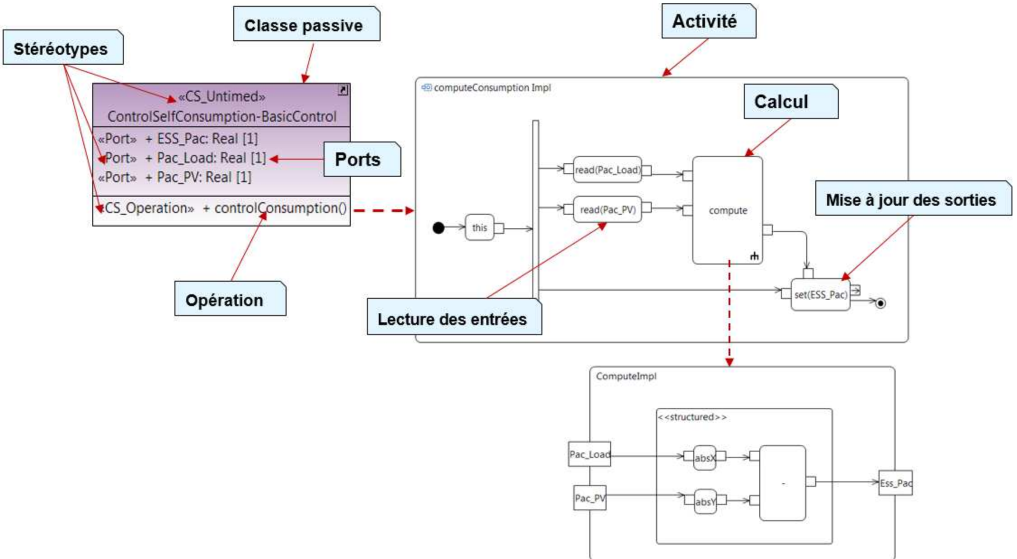

Figure 12. Règles de modélisation UML de système transformationnel

*IV.2.2. Identification de routines fUML**

Le Tableau3 représente le mapping entre les routines définies dans le modèle sémantique de fUML* et les procédures indispensables de l'API du standard FMI.

Tableau 3. Routines fUML* pour l'exécution de modèle UML de système transformationnel

API FMI	**Modèle sémantique de fUML***
Instanciation	c.locus.instantiate();
Initialisation	Les valeurs des propriétés (ports) sont automatiquement initialisées à l'instanciation par les valeurs par défaut dans le modèle.
Simulation pas à pas *doStep$_c$(h)*	c.dispatch(operationToExecute).execute();
Mise à jour des entrées *set$_c$(inPort,value)*	c.setFeatureValue(inPort,value)
Récupération des sorties *get$_c$(outPort)*	c.getFeatureValue(inPort);
Terminaison	Ne rien faire

IV.2.3. Matser de co-simulation

L'algorithme de master proposé pour la co-simulation de composants logiciels transformationnels avec des FMUs (Figure13) prend en compte la nature de chaque composant. Il adapte son pas de simulation, i.e, il utilise un pas de simulation h_{UML}=0 pour les composants UML représentatif de systèmes transformationnels, et un pas de simulation proposé par les FMUs h_{FMU}>0. Pour assurer une propagation instantanée des données, il exécute les composants UML en même temps que la propagation.

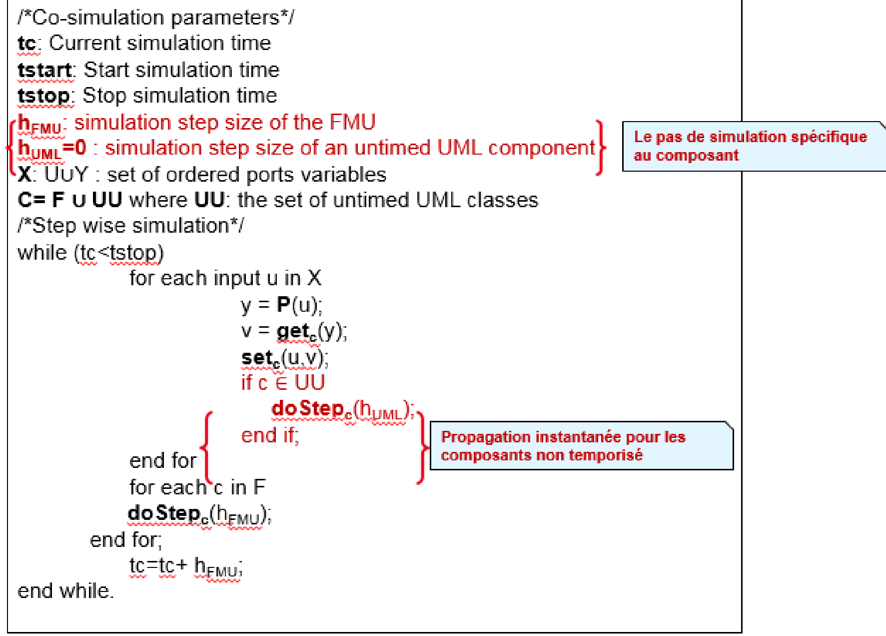

Figure 13. Master de co-simulation de FMUs et de modèles UML représentant des systèmes transformationnels

V.3. Application de l'approche aux systèmes réactifs aux changements de valeurs

IV.3.1. Règles de modélisation UML

Un système réactif aux changements de valeurs est représenté par une classe active auquel est appliqué le stéréotype « CS_Untimed ». Ces deux éléments reflètent le modèle de calcul Synchronous Reactive (SR) du composant. Cette classe doit avoir des ports d'entrée, des ports de sortie, et un comportement qui lui est associé. Ce comportement est défini par une activité qui va lire détecter les changements de valeurs sur les ports d'entrée. Si un changement est détecté, elle va lire les valeurs sur les ports d'entrée, réagir à ce changement, et mettre à jour les valeurs des ports de sortie.

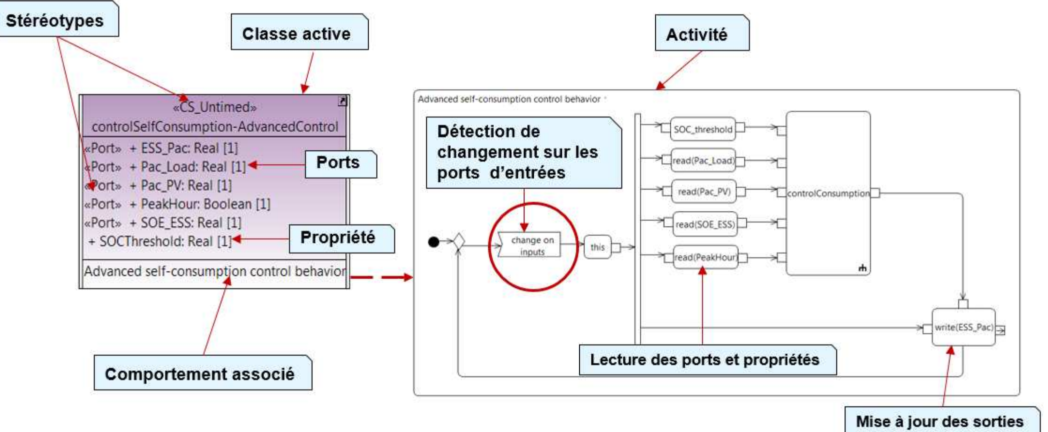

Figure 14. Règles de modélisation UML de système réactif aux changements

*IV.3.2. Extension de fUML**

Les réactions aux changements de valeurs n'est pas pris en compte dans fUML*. Comme mentionné dans la section I.4, une extension du modèle sémantique de fUML* par l'ajout de nouveaux visiteurs sémantiques est nécessaire (solution S1). La Figure15 illustre l'extension apportée à fUML*.

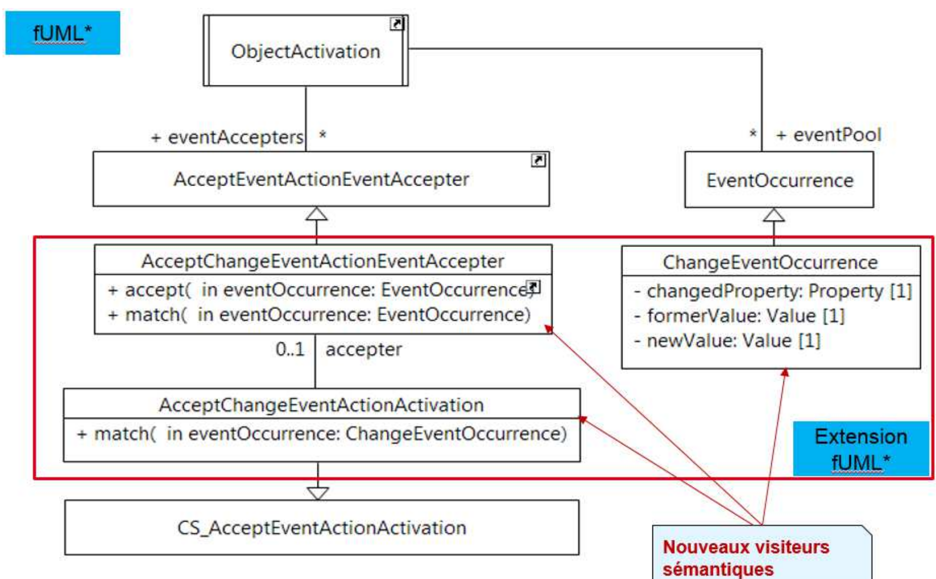

Figure 15. Extension de fUML* pour les comportements réactifs aux changements de valeurs

*IV.3.3. Identification de routines fUML**

Le Tableau4 représente le mapping entre les routines définies dans le modèle sémantique de fUML* et les procédures indispensables de l'API du standard FMI.

Tableau 4. Routines fUML* pour l'exécution de modèle UML de système réactif eux changements

API FMI	**Modèle sémantique de fUML***
Instanciation	c.locus.instantiate();
Initialisation	Les valeurs des propriétés (ports) sont automatiquement initialisées à l'instanciation par les valeurs par défaut dans le modèle.

Simulation pas à pas
doStep$_c$(h)
	if (firstSimulationStep) then
	c.actionActivation.startBehavior();
	if (c.objectActivation.eventPool.size() > 0) then
	c.actionActivation.dispatchNextEvent();
	Else do nothing;

Mise à jour des entrées
set$_c$(inPort,value)
	if (c.inPort.oldValue != value) then
	c.setFeatureValue(inPort,value);
	if (inPort.observed) then
	evt=new changeEventOccurrence(inPort,c.inPort.oldValue, value);
	c.objectActivation.eventPool.add(evt);
	endif;
	else doNothing;

Récupération des sorties *get$_c$(outPort)*	c.getFeatureValue(inPort);
Terminaison	c.objectActivation.stop();

IV.3.4. Master de co-simulation

L'algorithme de master proposé adapte son pas de simulation, i.e, il utilise un pas de simulation h_{UML}=0 pour les composants UML représentatif de systèmes transformationnels, et un pas de simulation proposé par les FMUs h_{FMU}>0. Pour assurer une propagation instantanée des données, il exécute les composants UML en même temps que la propagation.

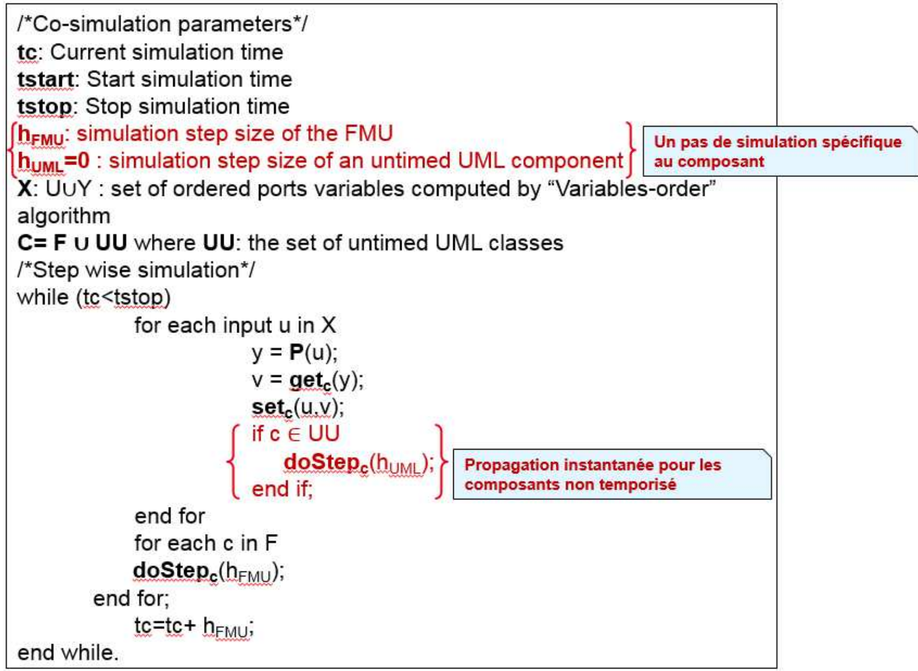
/*Co-simulation parameters*/
tc: Current simulation time
tstart: Start simulation time
tstop: Stop simulation time
hFMU: simulation step size of the FMU
hUML=0 : simulation step size of an untimed UML component
X: U∪Y : set of ordered ports variables computed by "Variables-order" algorithm
C= F ∪ UU where UU: the set of untimed UML classes
/*Step wise simulation*/
while (tc<tstop)
for each input u in X
y = P(u);
v = getc(y);
setc(u,v);
if c ∈ UU
doStepc(hUML);
end if;
end for
for each c in F
doStepc(hFMU);
end for;
tc=tc+ hFMU;
end while.

Figure 16. Master de co-simulation de FMUs et de modèles UML représentant des systèmes réactifs aux changements de valeurs

V.4. Application de l'approche aux systèmes réactifs aux évènements temporels

*IV.4.1. Règles de modélisation fUML**

Un système réactif aux évènements temporels est représenté par une classe active auquel est appliqué le stéréotype « CS_Timed ». Ces deux éléments reflètent le modèle de calcul Discrete Event (DE) du composant. Cette classe doit avoir des ports de sortie et un comportement qui lui est associé. Ce comportement est défini par une activité qui va produire une réaction à des instants discret définis par les évènements temporels (évènements temporels relatifs 'After(2)' et 'after(22)', ou évènements temporels absolus 'at(19)') en mettant à jour les ports de sortie.

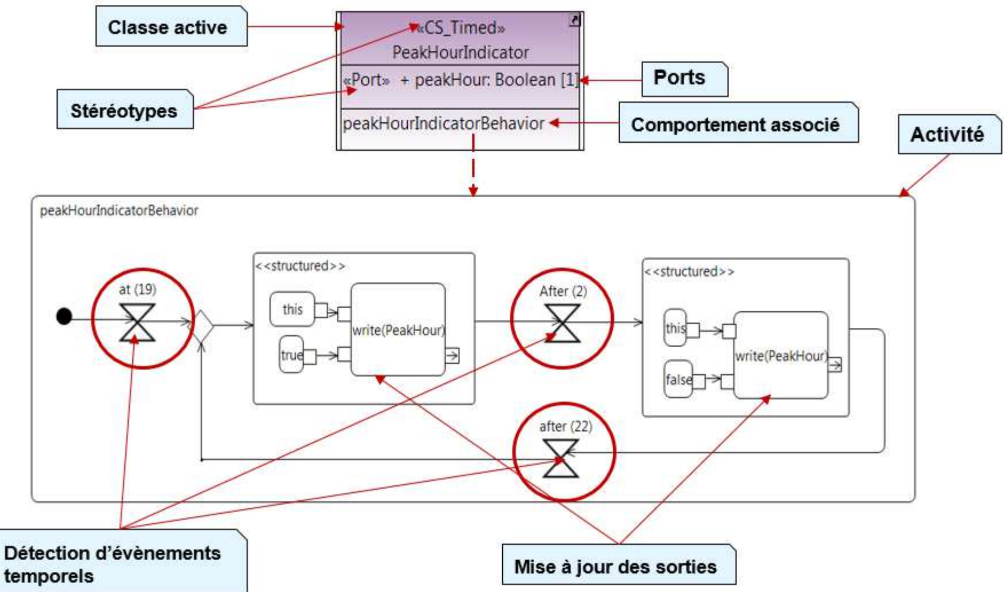

Figure 17. Extension de fUML* pour les comportements réactifs aux évènements temporels

*IV.4.2. Extension de fUML**

Les évènements temporels n'est pas pris en compte dans fUML*. Comme mentionné dans la section I.4, une extension du modèle sémantique de fUML* par l'ajout de nouveaux visiteurs sémantiques

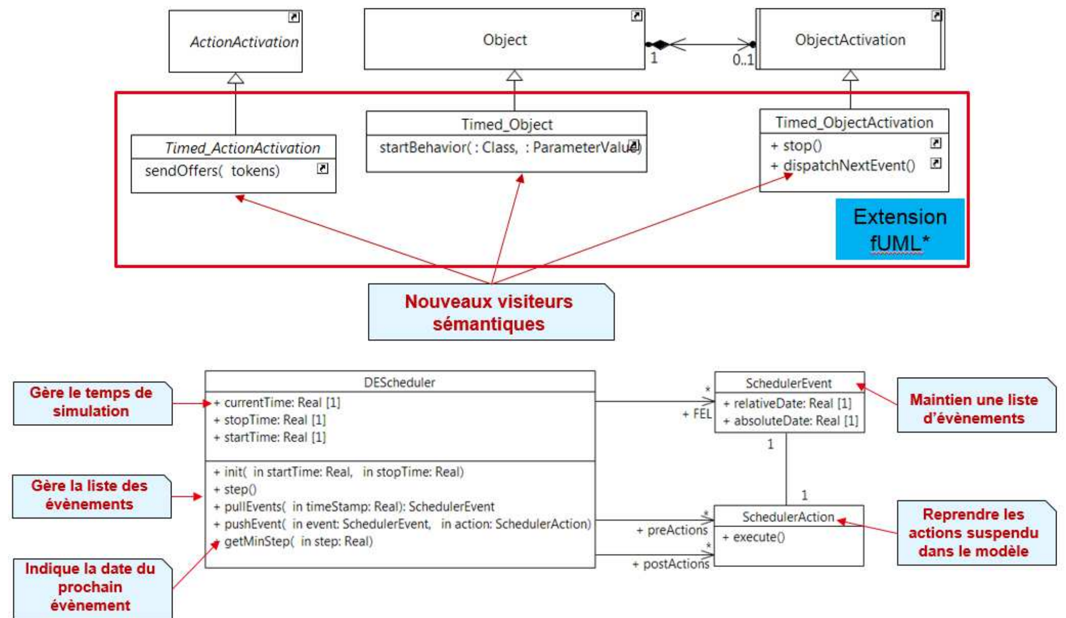

Figure 18. Extensions de fUML* pour les comportements réactifs aux évènements temporels

ainsi qu'une entité responsable de l'exécution des évènements temporels sont nécessaires. La Figure18 illustre l'extension apportée à fUML* ainsi qu'un diagramme de classe spécifiant le 'Scheduler' implémenté pour l'exécution des comportements associés aux systèmes réactifs aux évènements temporels.

*IV.4.3. Identification de routines fUML**

Le Tableau5 représente le mapping entre les routines définies dans le modèle sémantique de fUML* et les procédures indispensables de l'API du standard FMI.

Tableau 5. Routines fUML* pour l'exécution de modèle UML de système réactif aux évènements temporels

API FMI	**Modèle sémantique de fUML***
Instanciation	c.locus.instantiate();
Initialisation	Les valeurs des propriétés (ports) sont automatiquement initialisées à l'instanciation par les valeurs par défaut dans le modèle. DEScheduler.init(startTime, stopTime);

Simulation pas à pas
$doStep_c(h)$
	if (firstSimulationStep) then c.actionActivation.startBehavior();
	end if;
	if (DEScheduler.FEL.size() > 0) then DEScheduler.step(h);
	else do nothing;
	end if;

Mise à jour des entrées $set_c(inPort,value)$	c.setFeatureValue(inPort,value);
Récupération des sorties $get_c(outPort)$	c.getFeatureValue(inPort);
Terminaison	c.objectActivation.stop();

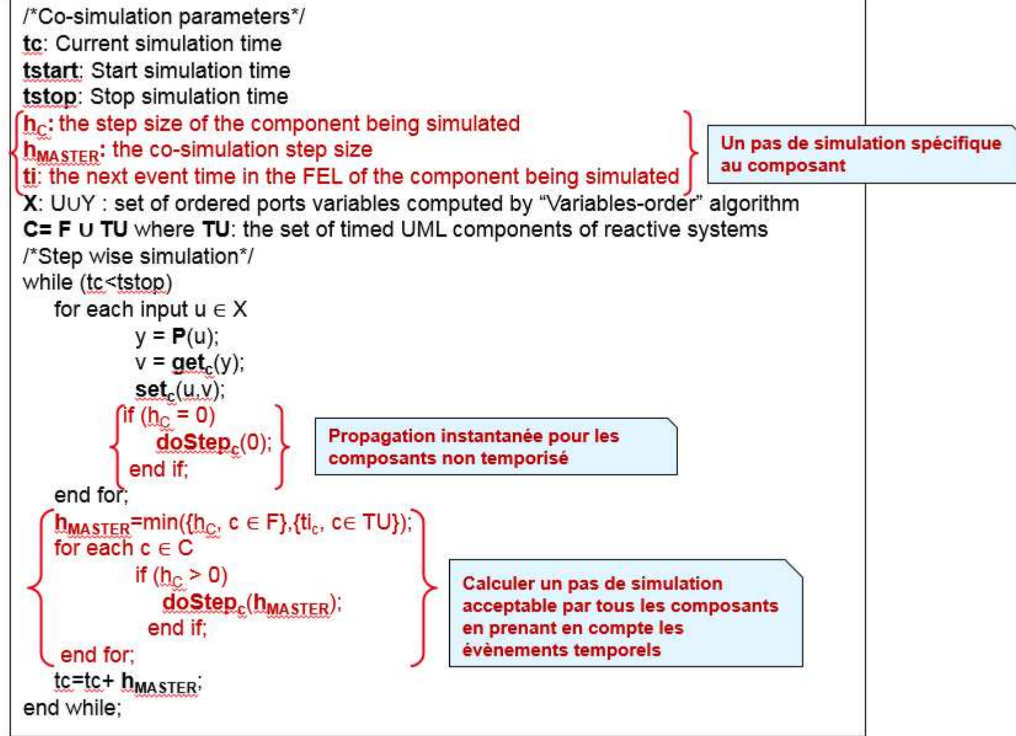

Figure 19. Master de co-simulation de FMUs et de modèles UML représentant des systèmes réactifs aux évènements temporels

IV.4.4. Master de co-simulation

Le master de co-simulaion proposé (Figure19) adapte le pas de simulation au type de composant. Il calcule le minimum entre le pas de simulation proposé par les FMUs et le prochain évènement temporels dans la file d'attente. Il prend aussi en compte les réactions instantanées (pas de simulation nul).

V. Validation de l'approche

Pour la validation de notre approche, nous avons simulé le modèle du cas d'étude dans Simulink en se plaçant dans les mêmes conditions que dans notre environnement de co-simulation dans Papyrus. Cela veut dire que nous avons utilisé les mêmes FMUs et gardé le composant de contrôle en boite blanche.

Nous avons effectué la simulation sur une période de 24 heures avec un pas de simulation de 1heure. Puis nous avons en particulier observé les instants auxquels une nouvelle consigne est produite par le composant de contrôle et les instants auxquels cette consigne est prise en compte

V.1. Simulation du cas d'étude avec stratégie de contrôle basique

Les Figure20 et Figure21 illustrent les résultats de simulation respectivement dans Simulink et dans Papyrus.

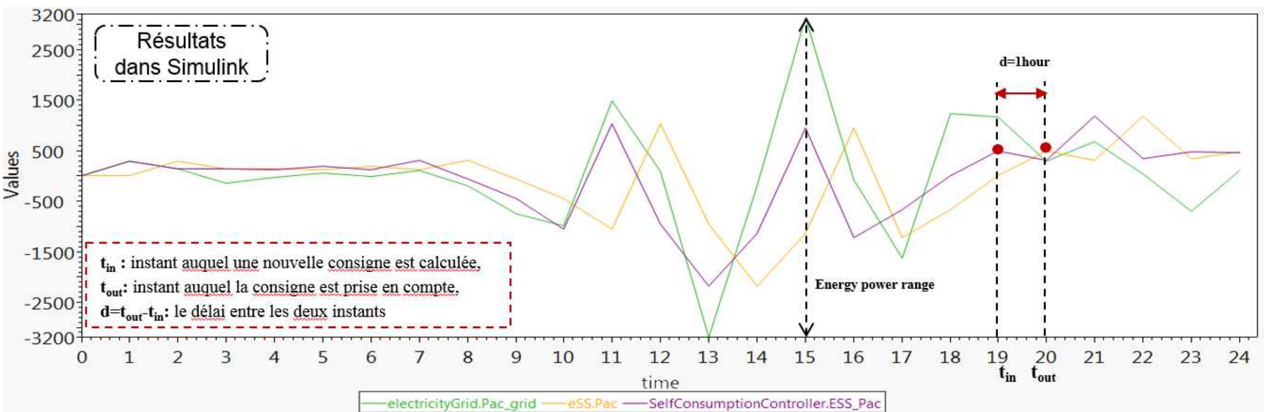

Figure 21. Résultats de simulation du cas d'étude avec contrôle basique dans Simulink

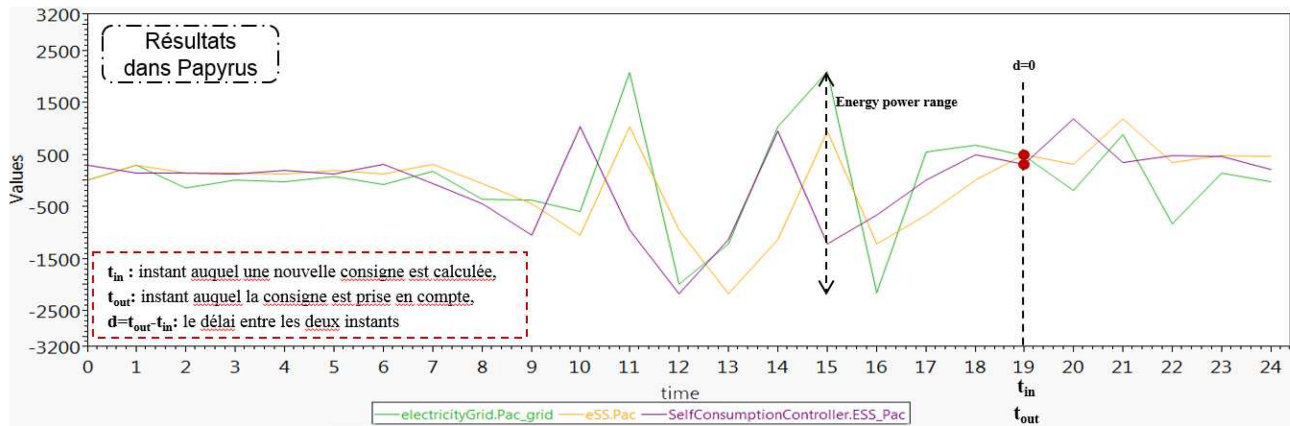

Figure 20. Résultats de simulation du cas d'étude avec contrôle basique dans Papyrus

V.2. Simulation du cas d'étude avec stratégie de contrôle avancée

Les Figure23 et Figure22 illustrent les résultats de simulation respectivement dans Simulink et dans Papyrus.

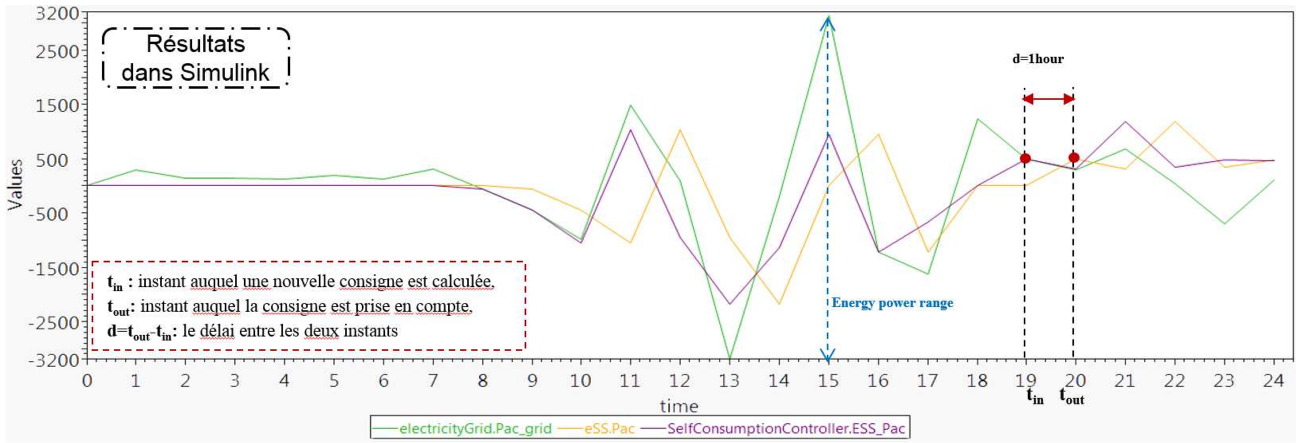

Figure 22. Résultats de simulation du cas d'étude avec contrôle avancée dans Simulink

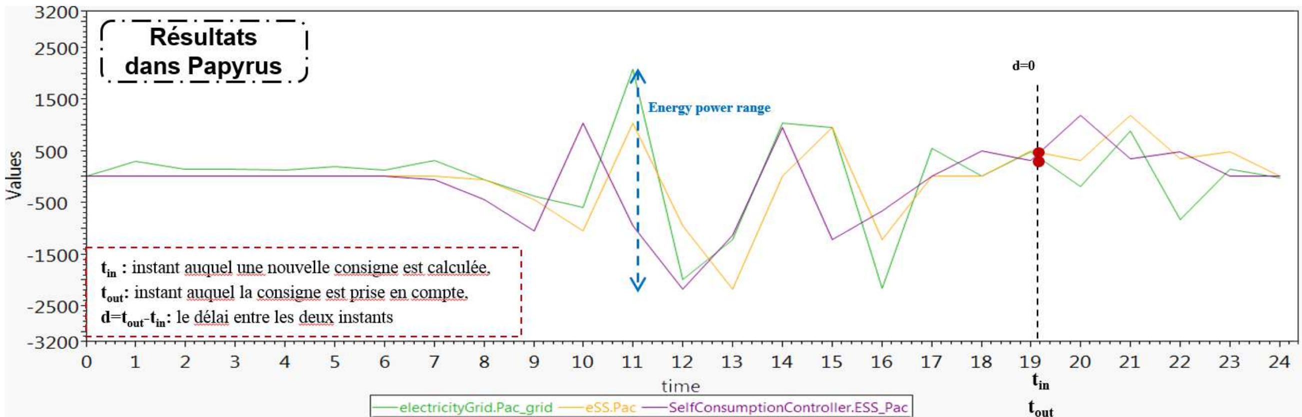

Figure 23. Résultats de simulation du cas d'étude avec contrôle basique dans Papyrus

V.3. Interprétation des résultats de simulation

Les résultats de la simulation dans Papyrus démontrent que, en appliquant la démarche que nous avons proposée, nous sommes capables de :

- Obtenir des informations sur la puissance énergétique qui doit être fournie par l'unité de stockage au réseau électrique ainsi que sur l'état de charge de l'unité de stockage plus tôt. En effet, comme illustré sur la Figure20 et la Figure21 (respectivement Figure22 et Figure23) la nouvelle consigne (la sortie 'Ess_Pac') ainsi que l'état de charge et la quantité d'énergie à injecter dans le réseau (la sortie 'Pac' du composant 'ESS') sont calculées et propagées avant une heure (qui correspond au pas de simulation) par rapport aux résultats de Simulink. Cela assure un meilleur fonctionnement du réseau intelligent. En effet, nous pouvons éviter les pannes en anticipant les demandes d'énergie (délivrer la puissance énergétique nécessaire dans le temps) et en maintenant un état de charge acceptable en chargeant l'unité de stockage le plus rapidement possible lorsque le niveau est inférieur au seuil requis.

- Minimiser et maintenir un taux de distribution d'énergie plus homogène dans le réseau électrique. Comme illustré sur la Figure20 la Figure21 (respectivement Figure22 et Figure23), en utilisant les algorithmes de master que nous proposons dans Papyrus, le taux d'énergie sur

le réseau est plus petit que lors de la simulation dans Simulink. Cela permet de mieux dimensionner et calibrer le système, et donc de minimiser le coût de conception du système, facteur très important pour les fournisseurs d'énergie.

VI. Conclusion et perspectives

VI.1. Conclusion

Dans ces travaux de thèses, nous nous sommes intéressés à la co-simulation des systèmes cyber physiques où les composants cyber sont spécifiés avec des modèles UML. Nous avons traité différents types de systèmes représentatifs des composants logiciels : transformationnels, réactifs aux changements de valeurs et réactifs aux évènements temporels.
Le contexte technologique que nous avons choisi s'appuie sur les standards : le standard FMI pour la co-simulation, et fUML* pour l'exécution des modèles UML. Nous avons rencontré des problématiques pour l'intégration des modèles UML dans une approche de co-simulation basée sur FMI à trois niveaux :

- Problématique liée au standard FMI : le standard ne supporte pas les comportements non temporisés et les réactions instantanées aux évènements.

- Problématique liée au standard fUML* : Celui-ci définit une sémantique précise pour l'exécution des modèles UML pour un sous-ensemble de modèles UML. Néanmoins, ce sous-ensemble ne couvre pas tous les éléments nécessaires à la modélisation des systèmes que nous souhaitons intégrer.

- Problématique liée à la synchronisation entre FMI (Modèle de calcul CT) et fUML* (Modèles de calcul : DataFlow, DE et SR) : le gap sémantique entre les modèles à intégrer peut engendrer des retards de propagation des données et la perte d'évènements.

Notre contribution était de faire cohabiter les modèles UML avec des FMUs tout en assurant la bonne synchronisation entre les différents composants impliqués. Cette contribution intervient à deux niveaux : localement sur les modèles UML où on a défini des règles de modélisation pour représenter chaque type de système en UML, et proposé des extensions au standard fUML* pour la prise en compte des comportement réactifs aux changements de valeurs et aux événements temporels. Puis globalement, où nous avons proposé des algorithmes de Masters pour différents scénarios de co-simulation qui intègrent des modèles UML avec des FMUs.
L'approche est spécifique aux modèles UML au niveau de la modélisation mais générale au niveau des algorithmes de co-simulation proposés. Les algorithmes de master proposés peuvent être réutilisées pour d'autres formalismes qui représentent les modèles de calcul DE, DataFlow et SR.

Enfin l'approche a été validée par comparaison des résultats de simulation obtenus dans l'environnement de co-simulation dans Papyrus par rapport aux résultats de simulation obtenus dans Simulink.

Note : les systèmes transformationnels temporisés ont été aussi traités dans le cadre de la thèse

VI.2. Perspectives

- *Perspective liée aux capacités des modèles UML à intégrer*

Les capacités en simulation d'un algorithme de co-simulation est lié entre autres aux capacités des composants. En particulier, pour pouvoir aller jusqu'au bout d'une simulation, un master peut avoir besoin de demander à une FMU de refaire un pas de simulation si elle ne parvient pas à aller

jusqu'au du pas de simulation proposé auparavant. Cette capacité de 'refaire un pas de simulation' appelée 'Rollback'. Son implémentation nécessite la sauvegarde de l'état de l'objet (notamment ensemble des valeurs dans le locus, son pool d'évènement et l'endroit où l'activité a été suspendue). Pour l'instant, cet aspect n'est pas traité dans le modèle d'exécution fUML*. Même si cette capacité reste pour le moment optionnel dans le standard FMI, son implémentation pour les modèles UML peut avoir un impact très important pour l'amélioration des algorithmes proposés.

A noter que pour les modèles transformationnels un rollback est possible vu que le système ne possède pas de variable d'état et donc ne garde pas l'historique des valeurs précédentes (une sortie ne dépend que d'une nouvelle entrée).

- *Perspective liée aux modèles UML supportés*

La deuxième perspective consiste à prendre en compte un ensemble plus large de modèles UML et aussi de formalismes. En effet, Le standard Precise Semantics of UML State Machines (PSSM) définit la sémantique d'exécution des machines à états UML et est une extension de fUML*. La même approche peut être appliquée pour des modèles UML qui implémente des machines à états. Ces travaux sont en cours dans le projet ITEA OpenCPS.

- *Perspective liée au standard FMI*

Une troisième perspective concerne l'extension du standard FMI. Ces travaux de thèse nous ont permis d'avoir le retour et la réflexion suffisante pour savoir les capacités qui manque à la spécification FMI pour permettre une intégration facile et directe de modèles de calcul autres que le CT (e.g. Dataflow, Synchronous Reactive et Discrete Event) sur lesquels s'appuient les composants logiciels dans une approche conforme à FMI.
Les informations et capacités à ajouter concerne spécifiquement le modèle de temps du composant : en particulier le besoin de permettre un pas de simulation nul nécessaire à la prise en compte des comportements non temporisés et des réactions instantanées aux évènements, et le besoin d'exposer la date du prochain évènement. Cette dernière proposition s 'aligne avec une proposition en cours[3].

[3] JP.Tavella & al.,Toward an Hybrid Co-simulation with the FMI-CS Standard, 2016.

Introduction

A. Context

As shown in Figure I-1-24, Cyber Physical Systems (CPS) are integrations of physical and computational components interacting in a tight coordination [25]. Examples include applications that enable the monitoring and controlling of the physical environments such as smart cities, automotives controllers, medical devices and robotics. CPS are systems that are particularly difficult to model and verify because the heterogeneous nature of their components requires many different modeling formalisms, and therefore rely on different Models of Computation (MoCs). Models of physical components usually rely on Continuous-Time (CT) MoC, whereas those of computational components rely on MoCs such as Discrete-Event (DE) or Data-Flow (DF). The verification of the overall system requires the composition of these components, which is not trivial. The models involved differ, in the way they interact with their environment, execute their behaviors, and manage time/events. Therefore, the most challenging issue is the coordination and synchronization between the involved models. Such global verification can be achieved by co-simulation of the different components composing the system.

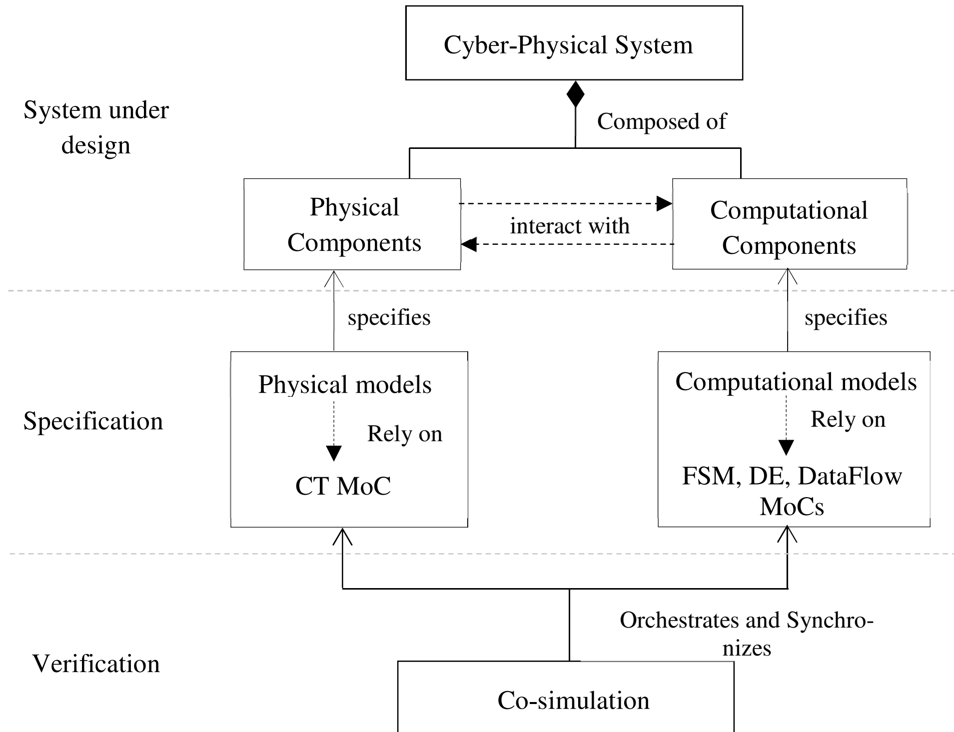

Figure I-1-24. CPS modeling and simulation

The Functional Mock-up Interface standard (FMI) in particular, offers a standard interface to couple two or more simulators in a co-simulation environment, known as master. This latter is responsible for providing an algorithm with efficient orchestration and synchronization of the

involved components, known as Function Mock-up Units (FMUs). FMI standard is gaining popularity in the industry, and it is being supported by many modeling and simulation environments. FMI was originally intended for co-simulation of physical processes, with limited support for other MoCs such as DE and Data Flow, even if they are commonly used to model the logic of software parts of a system. In particular, while UML is the reference standard for software modeling and is very commonly used in the industry, none of the present-day FMI-based co-simulation solutions consider UML models. This lack is a real issue while the importance of software concern of CPS is ever growing.

B. Contributions

Our thesis is that system engineering in general would greatly benefit from the consideration of UML in the FMI-based co-simulation approach. It would enable a significant number of software designers to evaluate the behavior of their software components in their simulated environment, as soon as possible in their development processes, therefore allowing them to make early and better design decisions. It would also open new interesting perspectives for CPS system engineers, by allowing them to consider a widely-used modeling language for the software parts of their systems. In this context, the objective of this work is to define and formalize an FMI-based co-simulation environment for CPS with integration of UML models for the software part of the system. It tackles the issue of bridging the execution semantics of UML models, and FMI and their synchronization. It also proposes an environment to specify and execute co-simulation scenarios composed of UML model elements, connected to FMUs imported from continuous time simulation tools.

We set up an incremental approach where we address different kinds of systems characterizing the computational components, reactive systems and transformational systems, and different kinds of models characterizing their behaviors, timed models and untimed models. Transformational systems are systems that simply transform a set of inputs into a set of outputs [55] whereas reactive systems are systems that maintain interaction with their environment [21].
Given a co-simulation model composed of FMUs and an UML component, four scenarios are therefore possible:

- Co-simulation of an untimed UML model of transformational systems with FMUs
- Co-simulation of an untimed UML model of reactive systems with FMUs
- Co-simulation of a timed UML model of transformational systems with FMUs
- Co-simulation of a timed UML model of reactive systems with FMUs

We base our proposals on OMG standards related to the execution semantics of UML models, fUML (known as "Semantics of a Foundational subset for executable UML models") and PSCS (the fUML extension for UML composite structure, also known as "Precise Semantics of composite Structure). They define precise execution semantics for a subset of UML (namely classes for structural modeling, and activities and actions for behavioral modeling in fUML extended with composite structures in PSCS). Both OMG standards provide a formal basis for the integration of UML models in CPSs co-simulation approaches, and in particular, FMI-based co-simulation approaches.

Our contribution is twofold: locally at the level of UML models, and globally at the master level:

- At the local level, for each aforementioned kind of system (i.e, transformational and reactive), we first identify a set of rules to model it with UML (i.e, the UML syntactic elements and their properties) and potential extensions to fUML and PSCS in cases where execution semantics of the required UML elements are not defined by fUML and PSCS,
- Then at the global level, we propose a master algorithm for each scenario. The proposed masters take into account not only external and internal dependencies between components and their capabilities, but also and especially their MoCs. They rely on, in particular, the adaptation of the untimed semantics of fUML and PSCS to timed semantics of FMI, and the adaptation of both Data Flow and DE MoCs (on which rely the execution semantics of untimed and timed UML models) to CT MoC (on which rely the FMUs). Based on these adaptations, the master algorithms are able both to propagate data between components and to trigger them at the correct points of time.

No extensions to the FMI standard are required for the realization of our approach, which means that imported FMUs are totally FMI-compliant. The approach is experienced and validated with use cases from the energy domain where the purpose is to verify energy management strategies defined as software components at different levels of the control module of an energy system.

C. Outline of the manuscript

The first part of this manuscript is a review of the related literature and studies. It is organized into three chapters.

- ***Chapter 1***: This chapter reviews the foundations and techniques of CPS modeling and simulation. It gives an overview of the most used MoCs for modeling and simulation of computational and physical components. It also introduces the challenges of CPS, enumerates the proposed techniques in the literature for their modeling and simulation, as well as provides an evaluation of each of them. We opt for co-simulation technique using the FMI standard.
- ***Chapter 2***: This chapter focuses on the FMI for co-simulation standard. Firstly, it gives an overview of the standard principles, the standard API and the standard misses regarding the co-simulation of CPS. Secondly, evaluates works in the literature which propose FMI-based co-simulation approaches for CPS and deal with the identified issues. Three techniques are identified: the extension of the FMI standard, the adaptation of semantics at FMU level and the adaptation of semantics at the master level. We opt for the third alternative, where the idea is to benefit from the use of the FMI standard and to enable the use of UML in co-simulation contexts.
- ***Chapter 3:*** This chapter is dedicated to the execution of UML models. Fistly, it evaluates UML tools for their support for the execution of UML models and the integration of FMI standard. Based on this evaluation, we decided to base our approach on the OMG standard fUM. Secondly, it identifies a set of UML models (representing computational

components) we need to consider in our approach, and evaluates fUML for its ability to model and simulate this set of models. At the end of this chapter, we present our approach for the integration of UML models in FMI-based co-simulation.

The second part of this manuscript concerns our contributions and is organized into four chapters.

- ***Chapter 4:*** This chapter introduces an UML-compliant master simulation tool, which is the environment we propose for modeling and simulation of CPSs based on the FMI standard. It describes the architecture of the implementation and the features it proposes.
- ***Chapter 5:*** This chapter deals with the integration of untimed UML models for both kinds of systems. It follows the two steps identified previously, which is, identifying the modeling rules for untimed models in the context of FMI and proposing a master algorithm for each kind of system.
- ***Chapter 6:*** This chapter deals with the integration of timed UML models for both kinds of systems. It follows the two steps identified previously, by first identifying modeling rules for timed models in the context of FMI, then proposing a master algorithm for each kind of system. This chapter also proposes an implementation of a control entity responsible for the simulation of timed UML models.

The third of this manuscript concerns the validation of the contributions. It compares the results of the co-simulation when applying our approach against original co-simulation results.

The fourth and last part of the manuscript gives the conclusion and the perspectives on this work and draws some perspectives for the extension of the proposed approach.

PART I: RELATED WORK

This first part of the manuscript is a review of the literature and studies related to the modeling and simulation of CPS. It aims at highlighting the challenges of CPS with regard to the solutions proposed in the literature. It is organized into three chapters.

- ***Chapter 1**: **Foundations and techniques of CPSs***
  This chapter reviews the foundations and techniques of CPS modeling and simulation. It gives an overview of languages and formalisms as well as the most used Models of Computation for modeling and simulation of computational and physical components. This chapter also introduces the challenges of CPS. It enumerates the proposed techniques in the literature for their modeling and simulation, while providing an evaluation for each of them. We chose the co-simulation technique using the FMI standard.
- ***Chapter 2**: **Towards FMI-based co-simulation of CPSs***
  This chapter concentrates on the FMI for co-simulation standard. It gives an overview of the standard principles, the standard API and the standard shortcomings regarding the co-simulation of CPS. Then, it evaluates works in the literature which propose FMI-based co-simulation of CPS. Finally, it presents our positioning.
- ***Chapter 3: Overview and key aspects of UML models' execution***
  This chapter aims at the identification of entry points for the integration of UML models in the frame of a FMI based co-simulation. We identify a set of systems we would like to model with UML and propose to rely on OMG standards fUML and PSCS for their execution. At the end of the chapter we represent our approach.

Chapter 1: Foundations and Techniques of Cyber-Physical Systems Modeling and Simulation

Outline

1.1. Foundations of modeling and simulation
- 1.1.1. Modeling languages and modeling formalisms
- 1.1.2. Model of Computation (MoC)
  - 1.1.1.2. Data Flow (DF)
  - 1.1.2.2. Synchronous Reactive (SR)
  - 1.1.2.3. Discrete-Event (DE)
  - 1.1.2.4. Continuous-Time (CT)
- 1.1.3. Simulation tools

1.2. Technique of CPS modeling and simulation
- 1.2.1. The translation of models
- 1.2.2. The composition of modeling languages
- 1.2.3. The unification of semantics
- 1.2.4. Composition of models
  - 1.2.4.1. Ptolemy
  - 1.2.4.2. ModHel'X
- 1.2.5. Co-simulation
  - 1.2.5.1. HLA Standard
  - 1.2.5.2. FMI for co-simulation standard

1.3. Discussion and conclusion

Cyber-physical systems (CPS) are the synergy between the physical world and the cyber world. They are the integration of computation and physical processes interacting in a tight coordination, involve several domains, and are inherently heterogeneous [25]. Examples include applications monitoring and controlling physical environments such as smart cities, automotive controllers, medical devices and robotics systems.

The heterogeneous nature of CPS implies the heterogeneity of its related model. In fact, a model of a CPS comprises models of physical components as well as models of computational components. In practice, these models are provided by different teams, each of them possibly using specific modeling language - and therefore model of computation (MoC) and simulation tool - well suited to the underlying domain. The heterogeneity we consider is that of the modeling languages, the MoCs and the simulation tools. The latter are the foundations of modeling and simulation and will be explained in the section 1.1 of this chapter. The heterogeneity is the most challenging issue of CPSs models. The main point in this context is to determine the global behavior of the model. How integrated and simulated are these heterogeneous models? This research question is the focus of section 1.2 of this chapter.

1.1. Foundations of modeling and simulation

A model is a abstracted representation of a real system under study. It keeps only the features which are important for a given goal. This model should consider aspects of the system that affect the problem under investigation [26]. In particular, models intended for simulation should specify the structure and the behavior of the real system. Various formalisms may be used to describe the behavior of the system. The model may be a representation of the activity flow of the system, the possible states of the system or a mathematical representation of the system. Each of these representations requires specific modeling formalism. This work focuses on the modeling and simulation of Cyber Physical Systems (CPS), which are typically made up of components of different natures, such as physical components and computational components. This section gives an overview about foundations and techniques for the modeling and simulation of each part of CPS. Section 1.2 then concentrates on techniques and challenges for modeling and simulating CPSs.

1.1.1. Modeling languages and modeling formalisms

A model is described using a modeling language, which is a particular implementation of a modeling formalism. A modeling language consists of: (a) An abstract syntax which specifies the concepts supported by the language, potential relationships between them and the way they can be combined. In the context of model driven engineering, it is often described by a meta-model, (b) A concrete syntax which defines the notation –textual, graphical, tabular, etc.– of each element in the abstract syntax, and (c) A semantics which defines how abstract concepts should be interpreted. The language semantics is a key feature for the definition of the model of computation on which the model relies.

1.1.2. Model of Computation (MoC)

A model of computation (MoC) is a formal set of semantics that provides the rules for interpreting the structure and behavior of a model. It corresponds to a class of modeling languages for which the computation and the communication characteristics are similar. A MoC is distinguished from another based on three semantic components [10]:

- The data: it specifies the data structures exchanged between the components of the system (e.g, signals, messages and events)
- The control: it characterizes the system kind (e.g, instantaneous, reactive and concurrent) and specifies the order in which the components execute their behavior and the instant at which the system should be observed
- The time/event management: it specifies the model of time that defines how time progresses if the notion of time exists in the model (e.g, continuous time, discrete events, or also iterations)

In the following material, we represent a variety of MoCs including Data Flow (DF), Synchronous Reactive (SR), Discrete Event (DE), and Continuous Time (CT) as defined in [34]. These MoCs are the most representative of the modeling languages currently used in the industry. Later in sections 5.1.2, 5.2.3, 6.1.2, 6.2.3, we will position our models in relation to these MoCs.

1.1.2.1.Data Flow (DF)

In the Data Flow MoC (Figure 1-1), the behavior is considered as a graph where nodes represent computations to execute and edges represent dependencies between these nodes. The execution (firing) of a node starts when all required data are available at the input pins. Data tokens are consumed when the node fires. The DF MoC has no notion of time. The execution is purely causal. The control consists in determining the activation order of the nodes and the propagation of the execution flow as much as possible in the model**.** Communication between components relying on the DF MoC is via sequences of data tokens. Each token is an arbitrary data structure that is treated monolithically by the MoC.

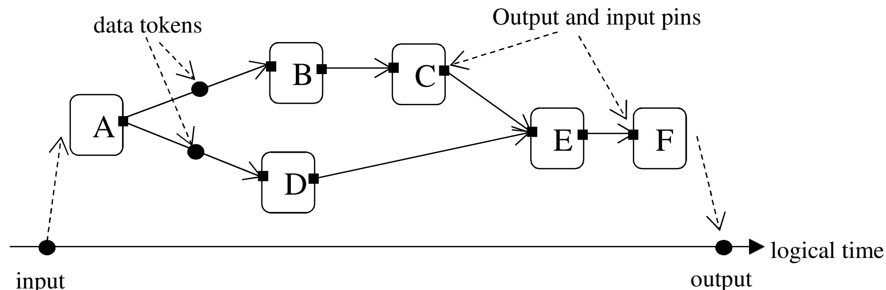

Figure 1-1. The Data Flow MoC

This MoC can be used to model software components and digital signal processing systems. fUML [32] provides precise semantics for UML activities execution which relies on the DF MoC.

1.1.2.2. Synchronous Reactive (SR)

In the Synchronous Reactive (SR) MoC (Figure 1-2), a component reacts to the events it receives from the environment in which it is placed. The components which rely on SR MoC are supposed to provide outputs and communicate instantaneously. The execution of the behavior is triggered when a new input is received. If the component has more than one input, then partial input arrival is sufficient to trigger its behavior.

The SR MoC can be described as logically timed systems. Time is divided into discrete instants called reactions or ticks in which the system is observed. Although steps are ordered, there is no notion of time delay between steps. Thus, we refer to time in this domain as logical time rather than discrete time.

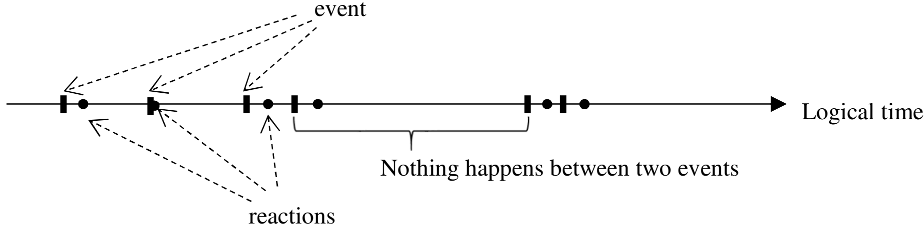

Figure 1-2. The Reactive Synchronous MoC

This MoC is usually used for describing reactive systems and the real-time controller's behaviors in which pieces of the program react simultaneously and instantaneously at each tick of a

global clock. Primary among languages which rely on this MoC are Esterel [9], Lustre [19] and Statecharts [22].

1.1.2.3. Discrete-Event (DE)

Components which are governed by Discrete-Event MoC (Figure 1-3) react to events that occur at a given time instant and produce other events (either at the same or at a future time instant). Time is an integral part of the model. Each event has a specific time instant that is global on the level of the model, called a time stamp. A DE simulator maintains a global events' queue sorted by their time stamp and has an internal notion of simulated time. The Discrete-Event MoC therefore relies on timed execution semantics. During a simulation step, the control consists in resuming all processes that have sent events with the same time stamp. The execution is chronological and serves as a basis for expressing concurrency in the model, that is, the first event to execute is the first one in the queue of events. This MoC is usually used to model systems in network and communication systems, manufacture or also management systems.

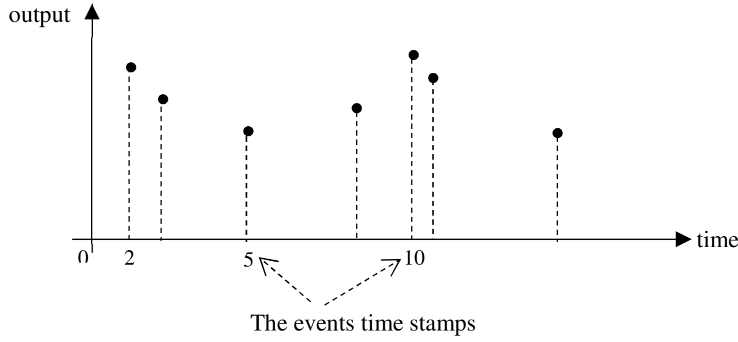

Figure 1-3. The Discrete Event MoC

1.1.2.4. Continuous-Time (CT)

Continuous-Time (CT) models consist of components that have continuous time signals as their inputs and outputs, and may have a state that changes over time advancement (Figure 1-4). These systems are usually represented with differential equations. During a simulation step, the simulation tool consists in solving the equations by fixed or variable integration steps by means of numerical solvers (continuous-time solving algorithm). Time is advanced in increments of exactly the integration step size noted "h". After each update of the clock, the state variables are updated for the time interval [t, t+h].

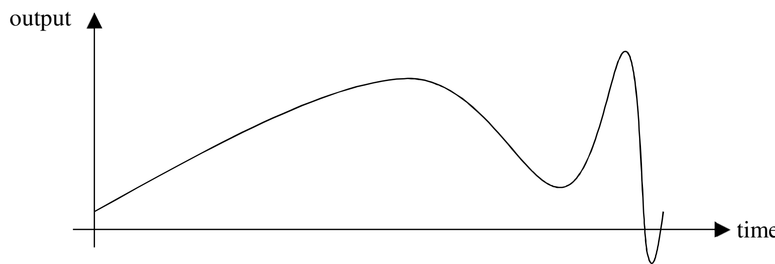

Figure 1-4. The Continuous Time MoC

The CT MoC is commonly used for modeling and simulation of physical systems such as mechanical and hydraulic systems.

1.1.3. Simulation tools

The simulation aims at the verification of system behavior early in the design process. It is performed with the help of a simulator, also called a simulation software/tool or an execution engine. The simulation tool is usually a modeling tool for a specific language and provides an execution engine responsible for controlling a model according to computation rules defined by the MoC of the modeling language.
Here are examples of language/simulation tool. Papyrus/Moka[4] is an execution engine for UML models where the execution comply to the DF MoC. SCADE[5] (Safety-Critical Application Development Environment) is an environment provided by Esterel Technologies. It builds on Lustre [19] and Esterel [9] to provide execution of models relying on the SR MoC. SystemC[6] is a framework that provides facilities to describe software and hardware components and a simulator that coordinates the execution of components relying on DE MoC. Open Modelica[7] and Dymola[8] are simulation tools for models expressed with the Modelica language where the execution relies on the CT MoC.

In the next section, we will see that a model of a CPS involve several components made using various modeling languages, and therefore several simulations tools and MoCs. The verification of a CPS is therefore not possible using a unique simulation tool. A new technique allowing to bring together all these components is then required.

1.2. Technique of CPS modeling and simulation

CPSs are complex. They involve several components operating in different domains (electric, mechanic, hydraulic, network, control, etc.) which must interact with each other. Their related models cannot be built in a monolithic manner. They comprise models of physical components as well as models of computational components which are of different natures. A modular approach should be used where models of individual (physical or computational) components are first built, then integrated in the same environment to obtain the model of the overall system.

In practice, the modeling and simulation of individual components is performed by specialized suppliers (domain specialists) using modeling languages and simulation tools they are familiar with. The resulting models are made of different languages and most likely rely on different MoCs. In addition to that, due to the complexity of CPSs, the reuse of IP components is becoming essential for coping with tight time-to-market. Suppliers from different domains provide their models to industrials while protecting their IPs by restricting the access to the component model [28].

In summary, CPSs are not easy to verify due to the following reasons:

- they embed components operating in different domains,
- their modeling brings together different languages/simulation tools and MoCs,
- their modeling may need the use of legacy code and IP-protected models

[4] Refer to eclipse.org/papyrus
[5] Refer to www.esterel-technologies.com
[6] Refer to http://hdl.telecom-paristech.fr/sc_intro.html
[7] Refer to www.modelica.org
[8] Refer to www.3ds.com/dymola

The simulation of the whole system requires the integration of heterogeneous models where the main issue is to determine the global behavior of the model. Several works, such as [38, 44, 35], state that the integration of heterogeneous models is not trivial since their semantics are pretty different in terms of data, control and model of time. These models, in fact, differ in the way the components interact with their environment, execute their behavior and manage time/events (refer to section Model of Computation (MoC) 1.1.2). The coordination and synchronization between the coupled models is a challenging issue for CPS modeling and simulation. The IP protection issue aggravates this task since the models' suppliers do not expose sufficient information about their models.

New design methodologies and frameworks for CPSs modeling and simulating of CPSs are required to close the gap between heterogeneous models and to integrate legacy code and IP models in order to be accepted in the industry [28].
Authors in [20] have identified five academic and industrial initiatives which address this kind of heterogeneity issue. In the next section, each technique will be evaluated independently based on the three following criteria:

- (criteria 1) the capabilities to cope with the heterogeneity of CPS,
- (criteria 2) the possibility to apply the approach to CPSs,
- (criteria 3) the acceptance level of the approach in the industry.

1.2.1. Translation of models

This technique aims at supporting the transformation between modeling formalisms. ATOM3[9] for example, is a tool for multi-formalism modeling which provides support for several modeling formalisms (such as Data Flow Diagrams, State Machines and Petri Nets) and utilities for their simulation. Models are described as graphs. Their translation from one formalism to another consists in graph rewriting.

<u>*Evaluation in the context of CPSs modeling and simulation*</u>:
This technique allows the support of heterogeneous modeling formalisms and is scalable *(criteria 1 probably satisfied)*. Providing support for a new modeling formalism, in fact, consists in integrating of its meta-model in the framework and the implementation of the transformation rules independent of the previously made transformations. The feasibility of the approach in the context of CPS *(criteria 2 possibly not satisfied)* depends on the number and size of the involved languages meta-models, and on their accessibility. One need to put a lot of effort to translate a large meta-model (e.g, for Modelica into another meta-model. In addition, switching between two different approaches for modeling components is not possible when details about the models are not sufficiently accessible (i.e, black boxes based on legacy library) *(criteria 3 probably not satisfied)*.

1.2.2. Composition of modeling languages

The composition of modeling languages is a traditional technique and consists in defining a common language for the specification of systems by composition of existing languages. The resulting language should be rich enough to support the heterogeneity of the system and may

[9] Refer to: http://atom3.cs.mcgill.ca/indesystem

be obtained by the extension of an existing one. In [14], authors enumerate several methods for the composition of modeling languages.

Evaluation in the context of CPSs modeling and simulation: This approach is not feasible in the context of CPSs (*criteria 2 not satisfied*) because their modeling and simulation are handled by different domain experts. Using this approach, all domain experts are asked to model their components using a new language. This is not practical (*criteria 3 not satisfied*) since they use languages and tools they are familiar with. In addition, this approach is costly due to the number of involved languages and its lack of scalability (*criteria 1 not satisfied*). That is, each time a new modeling language is considered, the target hybrid language must be rebuilt to account for new concepts.

1.2.3. Unification of semantics

This technique proposes a unique semantic support for describing heterogeneous models. The difference of this technique compared to the composition of the modeling languages is that it aims at fitting the models in an already defined semantics instead of enlarging the modeling language with new concepts. For example, in Metropolis [6] the process networks were chosen as the semantics basis for the definition of unified heterogeneous semantics.

Evaluation in the context of CPSs modeling and simulation: This approach is not applicable in the context of CPSs (*criteria 2 not satisfied*) due to the diversity of the involved models. As stated previously, CPSs integrate models from different domains (electric, mechanic, control, etc.) which are provided by different teams, and which rely on different MoCs. The unification of the semantics requires the domain experts to work together in order to share their knowledge (*criteria 3 not satisfied*), to think about the unification of the domains concepts (*criteria 1 not satisfied*), and to bridge the semantic gap between the MoCs.

1.2.4. Composition of models

In this technique, heterogeneous models are assembled together at model description level. That is, instead of defining a common modeling language for the specification of the system (as proposed in section 1.2.2), the system components are specified using different modeling languages then assembled and adapted. The most popular approach is Ptolemy[10] and we can also notice Modhel'x[11]. This kind of approach is based on the concept of MoC and consist in adaptating the involved MoCs while defining relations between the semantics of these MoCs.

1.2.4.1. Ptolemy

Ptolemy proposes an environment for the modeling and the simulation of heterogeneous systems. It uses an actor-oriented design approach to model structure of components. The rules of interaction and communication between actors are defined by the MoC, and implemented by a director. Heterogeneous models combine directors realizing distinct MoCs. Ptolemy addresses the problem of mixing heterogeneous models by providing adaptation of semantics (at data, control and model of time levels) between the involved MoCs.

[10] Refer to: http://ptolemy.eecs.berkeley.edu/ptolemyii/
[11] Refer to: http://wwwdi.supelec.fr/software/modhelx/

1.2.4.2.ModHel'X

ModHel'X is a framework for simulating multi-formalism models. Models are made of blocks (a generalization of actor in Ptolemy). ModHel'X in particular relies on the notion of interface blocks. An interface block includes an adaptation layer which allows the modeler to specify explicitly how the semantics of the MoCs are adapted at the boundary between two heterogeneous models. Modhel'x was proposed as an extension to Ptolemy. It allows flexible heterogeneous modeling without modifying the original assembled models.

Evaluation of the approach in the context of CPSs: This approach provides a way to address the heterogeneity issue while preserving the modularity of models *(criteria 1 probably satisfied)*. The advantage of Ptolemy and ModHel'x is that they support the most widespread MoCs (e.g, CT, DE and several types of dataflow) and proposes their adaptation. They are good candidates for experimenting the adaptation and the synchronization of heterogeneous MoCs *(criteria 2 possibly satisfied)*. However, the use of this solution for the CPS is not well accepted industrially especially when the CPS model is built using legacy models *(criteria 3 possibly not satisfied).*

1.2.5. Co-simulation

Co-simulation is the joint simulation of models developed with different languages and tools. It enables tools interoperability to facilitate the simulation of the intrinsically heterogeneous CPSs.

Co-simulation has been extensively investigated in literature. Some approaches such as [24] and [23] propose the coupling of a fixed and restricted set of simulators. An adaptation is required to connect a tool to the others for their synchronization during simulation. The main drawback of these point to point solutions is that the synchronization is specific to each simulation tool. Therefore, synchronization modules built for one tool may not be easily reused for other tools. This approach also lacks scalability, that is, the integration of a new tool in the co-simulation approach requires synchronization with all integrated tools or the designation of a central component responsible for their synchronization and orchestration. Other co-simulation approaches aim at supporting the connection of any type of simulator. The most popular initiatives are the High Level Architecture (HLA) [1] and the Functional Mock-up Interface for co-simulation standard (FMI) [17]. Subsections 1.2.5.1 and 1.2.5.2 give further details about both initiatives.

1.2.5.1.HLA Standard

HLA is a standard initially developed by the US Department of Defense (DoD) in an effort to facilitate the interconnection of distributed simulators. A distributed simulation, so-called "federation", interconnects several simulators known as "federates". These federates are processes which exchange data relying on publish/subscribe patterns. Synchronization of federates, data exchange and event passing between all federates is managed by the Real-Time Infrastructure (RTI). HLA provides a standard architecture which eases the interconnection of simulators but is not an implementation by itself. CERTI[12] , for example, provides open source implementation of an HLA RTI.

[12] Refer to: www.openrobots.org/certiHLA

1.2.5.2. FMI for co-simulation standard

FMI for co-simulation specification aims at co-simulation of separately developed components. FMI is a result of the MODELISAR research project. It emerges from industrial needs, where the goal is to facilitate the cooperation between different companies while preserving the intellectual properties (IPs). A model of a given system is a set of interconnected black-box slaves, the so-called FMUs (Functional Mockup Units). The FMUs are passive entities whose simulation is triggered and orchestrated by a master algorithm (MA). FMI restricts the communication and exchange between FMUs to discrete communication points where, in between, each component is solved independently with its specific tool.

Evaluation in the context of CPS modeling and simulation: HLA and FMI have close goals but have different architectures and provide different features. In HLA, the synchronization and coordination of the heterogeneous simulators during the simulation is handled through the RTI services (data exchange, time advance and events handling mechanisms) whereas, in FMI, these services are not provided in the standard and should be handled by a MA. In [5], authors attempt the combination of these two standards by proposing the RTI as a MA orchestrating a set of FMUs encapsulated in federates.

Both HLA and FMI have limitations, but they present attractive characteristics in terms of modularity of the solution and the interoperability between the model's simulators (*criteria 1 and criteria 2 probably satisfied*). They are standards which are already supported in several tools. Most of HLA-compliant tools are network simulators such as NS3[13] and OMNeT++[14] since HLA is mainly used for distributed simulations. At the same time, FMI standard is becoming well-accepted industrially *(criteria 3 satisfied)*. It is supported by more than 90 simulation tools and is considered an important driver in enabling tool interoperability in the area of cyber-physical systems.

1.3. Discussion and conclusion

In this first chapter, we introduced foundations and techniques for CPS modeling and simulation. We first defined the notions of modeling languages, MoCs, and simulators in order to become familiar with terms later used in our discussion. The second part of this chapter focused on techniques for simulating heterogeneous systems. We stated that CPS are not easily verified due to the heterogeneity of the involved components regarding the domains, the modeling languages and tools, and MoCs. Methodologies for CPS modeling and simulation not only have to cope with this heterogeneity, but they also have to become accepted within the industry.

Five techniques for verification of heterogeneous systems were presented and evaluated based on these criteria. A synthesis of this evaluation is given in **Table *1-1***. We concluded that the co-simulation is the most suitable technique for the modeling and simulation of CPS.

[13] Refer to: www.networkSimulator-ns3.org
[14] Refer to: omnetpp.org

Table 1-1. Evaluation of verification techniques according to the chosen criteria

Techniques	Criteria 1: The capability to cope with the heterogeneity of CPS	Criteria 2: The possibility to apply the approach to CPSs	Criteria 3: The acceptance in the industry
Technique1: Translation of models	Probably satisfied	Possibly not satisfied	Probably not satisfied
Technique2: Composition of modeling languages	Not satisfied	Not satisfied	Not satisfied
Techniques3: Unification of semantics	Not satisfied	Not satisfied	Not satisfied
Technique4: Composition of models	Probably satisfied	Possibly satisfied	Possibly not satisfied
Technique5: Co-simulation	Probably satisfied	Probably satisfied	Satisfied

Two co-simulation standards, HLA and FMI, were presented in section 1.2.5. We stated that both represent attractive characteristics with some limitations. We choose FMI standard as a basis for our contribution. FMI is considered as an important driver in enabling tool interoperability in the area of cyber-physical systems. The state of the art, as well as the number of simulation tools which support FMI, give an indication on the popularity of the FMI standard for the co-simulation of CPS. However as stated previously, FMI has some limitations. These later are addressed in the next chapter.

Chapter 2: Toward FMI-based Co-Simulation of CPS

Outline

2.1. About FMI for co-simulation
- 2.1.1. The Functional Mock-up Unit (FMU)
  - 2.1.1.1. The co-simulation description schema
  - 2.1.1.2. The co-simulation interface (FMI API)
- 2.1.2. The master algorithm

2.2. Limitations of FMI regarding CPS
- 2.2.1. Untimed semantics are not supported (I1)
- 2.2.2. Time events are not handled (I2)

2.3. Adressing FMI limitations
- 2.3.1. Adaptation of semantics at the FMU level
- 2.3.2. Extension of the FMI API
- 2.3.3. Adaptation of semantics at master level

2.4. Discussion and positioning

In the previous chapter, we stated that co-simulation is the most suitable technique for dealing with related heterogeneity of CPS simulation. We also explained why we chose FMI for co-simulation as a basis for our work. Although FMI provides a standard interface for co-simulation of models from different languages/tools, it also has lacks to cope completely with the heterogeneity of the involved MoCs. To better understand the limitations of the FMI standard, we will represent the architecture of the standard section 2.1, with the notions of Master Algorithm and FMU respectively, in subsections 2.1.1 and 2.1.2. Further, we will represent the life cycle of the FMU in order to identify the problems related to the integration of different MoCs in section 2.2. Then we will outline the solution we propose to solve these problems in section 2.3. The chapter ends with a discussion about the proposed solution and our positioning in regards to the integration of UML models in the FMI-based co-simulation.

2.1. About FMI for co-simulation

FMI [17] is a standard that supports both model exchange and co-simulation of dynamic models originally designed with different simulation tools. Its development was initially launched as part of the MODELISAR project and continues now through the participation of 16 companies and research institutes under the roof of the Modelica Association. Today, FMI is supported by over 89 tools [17]. A component that implements FMI standard is called FMU. This can later be used for either:

- A model exchange, where the goal is to allow a generated FMU to be imported and executed in a different simulation tool. The FMU is executed using the solver provided by the host simulator (Figure 2-1-a);

- A co-simulation, where the intention is to provide a standard interface to couple two or more FMUs in a co-simulation environment. The data exchange between these FMUs is restricted to a discrete set of communication points where each FMU is executed independently with its own solver (2-1-b).

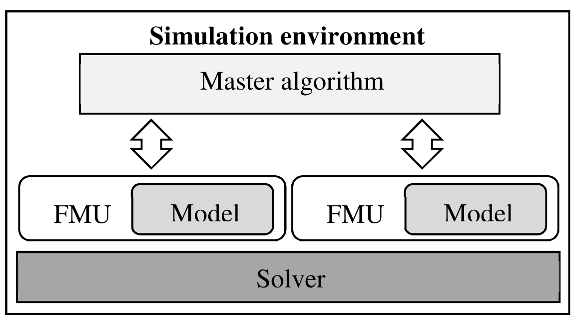

Figure 2-1-a. FMI for model exchange

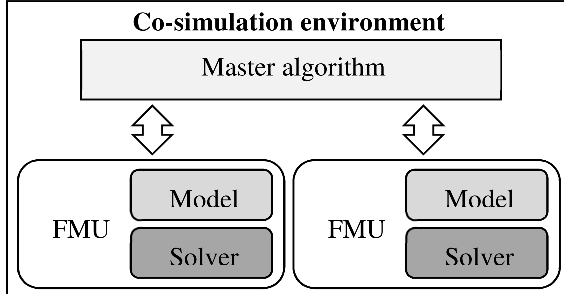

Figure 2-1-b. FMI for Co-simulation

It is not common for UML tools to support execution of physical models. The model exchange is therefore not possible to adopt in this work. We are interested in FMI for co-simulation, which is based on master/slave architecture. A system is seen as an interconnection of slaves, the so-called FMUs, imported in a co-simulation environment, the so-called master. This latter is in charge of providing an algorithm to orchestrate and synchronize the slaves. The FMI specification provides a standard interface for the representation of the models as FMUs, but it does not standardize the master algorithm.

The following material provides further detail about the FMUs and the master algorithm while pointing out important information that concerns our work. It specifically illustrates the FMUs' composition and the master algorithm as defined in the FMI specification. Then, it identifies the shortfalls of the FMI standard regarding simulation of CPS.

2.1.1. Functional Mock-up Unit (FMU)

The FMI specification creates a distinction between the description of data and functionality. It consists of two parts: the co-simulation description schema defined as an xml schema, and the co-simulation interface defined as an API written in C. An FMU is an implementation of the FMI interface. It contains a zipped-file, which therein contains a "modelDescription.xml" file providing static information about the model, and code files or binaries implementing the dynamics of the model. The FMU may contain additional data and functionalities.

Subsections 2.1.1.1 and 0 outline details about the important elements of the co-simulation description schema and the co-simulation interface.

2.1.1.1. Co-simulation description schema

The co-simulation description schema defines the structure and the content of the XML file contained in the FMU. Figure B-1**.** XML schema of the FMI standard (version 2.0) in Annex B depicts an extract of the schema taken from the FMI specification. Each FMU has a specific XML file (called "modelDesciption.xml") that complies with that schema. It contains static information relevant for the communication in the co-simulation environment specified in tags (expressed between '< >'), in particular:

- The model variables and parameters (expressed in the element <ScalarVariable>),
- The model structure which represents the inputs and outputs and identifies the initial inputs and derivatives of the model (expressed in the element <ModelStructure>),
- The solver/simulator capabilities which characterize the ability of the slave to support advanced master algorithms e.g, the usage of variable communication step sizes, higher order signal extrapolation, or others (expressed in the element <CoSimulation>),
- The information about the default simulation configuration of the simulator e.g, the simulation start time, the simulation stop time and the preferred step size (expressed in the element <DefaultExperiment>).

Figure B-5. Example of a model description xml file of an FMU for co-simulationof Annex B depicts an example of a model description file of an FMU for co-simulation together with an explanation of the different elements of the XML file.

FMI 2.0 introduces new optional features compared to the early version 1.0 as follows:

- The ability to save and then restore the complete state of an FMU during simulation represented by the flag "canGetAndSetFMUState" in the XML file. This feature enables an FMU to go back in the simulation time in order to perform a simulation step again if this later fails. This feature is interesting especially for continuous time simulation where the solver may fail to solve differential equations because the step size is too large.
- The input/output (I/O) dependency feature provides information about potential dependency relation between the outputs and the inputs of a given FMU. It is specified as an output dependency in the "modelStructure" element of the XML file. Together with the external dependency between FMUs (as a result of their connections), this information can be used by the master algorithm in order to detect cycles when connecting FMUs, and therefore avoid algebraic loops when connecting FMUs together. In addition, it can be used to establish the order in which the data should be propagated.

These features, when supported by the involved FMUs, can be used to empower the master algorithm.

2.1.1.2. Co-simulation interface (FMI API)

FMI for Co-Simulation defines interface routines for the communication between the master and all FMUs in a co-simulation environment. The co-simulation interface is a set of C functions (an API) which an FMU should implement. FMUs by themselves are passive objects, in the sense that they do not execute. For that reason, they are called slaves. We need a so-called master algorithm for controlling the FMUs and for data exchange of input and output values as well as status information.

An FMU is therefore seen as a black-box which implements the methods defined in the FMI co-simulation interface and, the simulation of one FMU consists of a sequence of API functions called by a master. The most important functions concern the instantiation and initialization of an FMU, the propagation of variables from one FMU to another, the stepwise simulation of the FMU and the termination of the simulation.

a. Instantiation and initialization

The instantiation is handled by a call to the *fmi2Instantiate* function that returns a new instance of an FMU. The instantiation is mandatory to run the simulation of an FMU.
Once instantiated, the FMU can be initialized. The initialization is handled by a call to the functions *fmi2EnterInitializationMode* and *fmi2ExitIntializationMode* where in between some variables could be set or get before the beginning of the simulation.
The variables that can be initialized are input variables and variables that must have an exact value at time zero. The variables that can be retrieved at initialization phase are output variables. At the initialization phase, the simulation parameters should be set by a call to *fmi2SetupExperiments*. The simulation parameters indicate the simulation start time and the simulation stop time.

b. Stepwise simulation and data propagation

At this stage, the FMU is instantiated and initialized. The co-simulation computation can therefore start. The *fmi2DoStep* function advances the co-simulation by the simulation step size $h>0$ from time t_{ci} to time t_{ci+1} ($t_{ci} <> t_{ci+1}$) and returns a status which indicates whether the FMU succeeded the simulation step or not. It returns *fmi2OK* to indicate that the slave has performed the simulation up to the requested point in time. In turn, fmi2Discard is returned to indicate that only a part of the time interval could be computed successfully, while fmi2Error is to indicate that the computation could not be performed at all.
At the end of a simulation step, the *fmi2SetXXX* and *fmi2GetXXX* commands are used to set the input variables values and retrieve the output variables values of an FMU. The XXX is replaced with the data type, for example, *fmi2GetReal* for real variables. In a network of connected FMUs, these functions allow the master algorithm to propagate data from one FMU to another.

c. Termination

The termination of an FMU is allowed only if the FMU successfully performed the last simulation step. The termination of an FMU is handled with function *fmi2Terminate*.

A formalization of the FMI API - more specifically, the procedures we introduced - and the connections of FMUs in a model was proposed by Broman and al. in [12] as follows[15]:

Given a co-simulation model **M**:
C = F: is the set of FMU instances
$\mathbf{c \in C}$: an FMU
$\mathbf{S_c}$: the set of states of c
$\mathbf{U_c}$: the set of input port variables for c
$\mathbf{Y_c}$: the set of output port variables for c
V: the set of values that a variable may take
$\mathbf{I \subseteq V}$: the set of default variables values.
$\mathbf{D_c \subseteq U_c \times Y_c}$: I/O dependency for instance c,
$D_{c1} = (u_{c1}, y_{c1})$ means that the output y_{c1} of the FMU instance c1 depends on the input u_{c1} of the FMU instance c1 (internal dependency)

[15] This formalization will be used later to specify the master algorithms proposed in the literature as well as those proposed in this work

$U = S_{c \in C}$ U_c: the set of all input variables in M
$Y = S_{c \in C}$ Y_c: the set of all output variables in M
P: U → Y: Port mapping constructed from connectors of M
$P(u_{c2}) = y_{c1}$ means that the output y_{c1} of the FMU instance c_1 is connected to the input u_{c2} of the FMU instance c2 in M (external dependency)
$inst_c$: C → S_c instantiates the FMU c and returns its state.
$init_c$: R ≥ 0, R ≥ 0 → S_c **initializes** c with a given start time *tstart* and stop time *tstop*. Input ports variables can be set during initialization phase with values from I. It returns the state of c.
set_c: U_c × V → Sc sets a given input u ∈U_c with a value v∈ V and returns the new state of c.
get_c: Y_c → V returns the value v∈V of a given output y∈Y_c.
$doStep_c$: R > 0 → S_c × R ≥ 0 takes as input a step size h ∈ R>0. It performs a simulation step and returns the new state and the last successful simulation time h'∈ R>0 of c.
$terminate_c$: → Sc: terminate the simulation of c.

Table 2-1 gives the mapping between the FMI API and the functions in the formalization.

Table 2-1. Mapping between the FMI API functions and the formalization functions

Formalization	FMI API
$inst_c$()	fmi2Instantiate();

$init_c$(tstart, tstop)	fmi2EnterInitializationMode ();
	fmi2SetupExperiments(tStart, tStop);
	fmi2Setxxx(v);
	fmi2ExitInitializationMode();

get_c(y)	fmi2Getxxx(y); where xxx is one of Real, Integer, Boolean and String
setc(u,v)	fmi2Setxxx(u,v); where xxx is one of Real, Integer, Boolean and String
$doStep_c$(h)	fmi2DoStep(h);
$terminate_c$()	fmi2Terminate();

2.1.2. Master algorithm

A master algorithm triggers and orchestrates a collection of FMUs to co-simulate the different parts of a system. Figure 2-2 depicts the principle of the master algorithm for an FMU simulation.

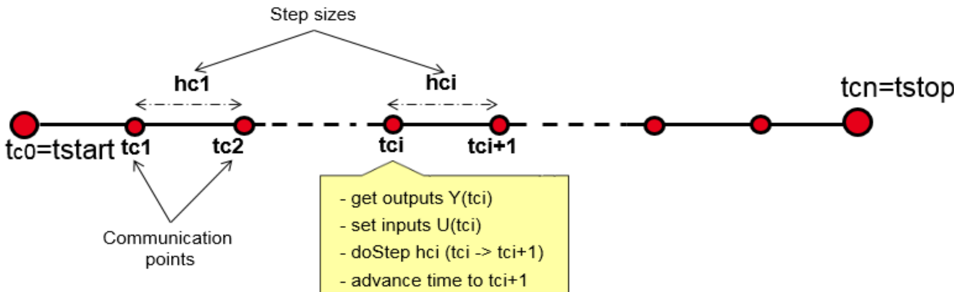

Figure 2-2. Principle of an FMU simulation

The master algorithm performs stepwise co-simulation from time “tstart” to time “tstop”. Time advances locally on FMUs by the step chosen by the master algorithm. When the co-simulation is started, the FMUs are simulated independently between two discrete communication points “tci” and “tci+1” with a step size “$h=t_{ci+1}-t_{ci}>0$”. At these communication points, the master algorithm collects the outputs “y(tci)” and sets the inputs “u(tci)” of all FMUs.
In this way, the master algorithm synchronizes the FMUs in the manner that it waits for all FMUs to simulate up to the communication point at each simulation step before it advances the time.
A simple co-simulation master algorithm uses the sequence of FMI commands depicted in Figure 2-3.

```
/*Co-simulation parameters*/
tc: Current simulation time
tstart: Start simulation time
tstop: Stop simulation time
h: simulation step size>0
/*Instantiate and initialize components c ∈ C */
For each component c ∈ C:
        Instc();
        Initc(tstart, tstop);
/*Step wise simulation*/
While (tc<tstop)
        For each connection between an input u and an output y:
                v = getc(y);
                setc(u,v);
                doStepc(h);
        End for
        tc=tc+ h;
end while.
/* Termination of the simulation*/
For each component c ∈ C:
        terminatec() ;
end simulation
```

Figure 2-3. Basic master algorithm for FMI co-simulation

The FMI specification does not standardize a master algorithm for co-simulation. It only provides a pseudocode of a basic master algorithm (depicted in Figure B-16 of Annex B) and outlines the supported calling sequences from master to slave. Tool vendors are responsible for providing their own master algorithms according to their specific needs.
Simple master algorithms can be found in FMI for co-simulation standard specification [17] and in [7] where they propose a fixed-step platform-independent MA. Authors in [37] investigate adaptive communication size control in the FMI to improve the accuracy of simulations. Acker and al. in [43] propose a method to automatically generate an optimal master algorithm compliant to FMI standard starting from an explicitly modeled co-simulation model. Authors in [12] propose a class of advanced master algorithms where they demonstrate the importance of the rollback and the I/O dependency information regarding the efficiency of the simulation.

2.2. Limitations of FMI regarding CPS domain

FMI was originally intended for continuous time models (which rely on CT MoC) with limited support of models with non-CT MoCs. The procedures defined in the FMI API are, in fact, very representative of continuous time systems simulations. A model relying on the CT MoCs can be directly wrapped into an FMU. However, MoCs such as Data-Flow (DF), Synchronous-Reactive (SR) and Discrete-Event (DE) are not easily wrapped to the FMI API due to the semantics gap between their execution semantics and that of the FMI API [12].
For CPSs, we need to bridge the semantics gap between the aforementioned MoCs and FMI, particularly at time and control levels. Table 2-2 is a reminder of the model of time as well as the instants at which a system should be observed with respect to the MoC on which it relies.

Table 2-2. Semantic gap between FMI and non-CT MoCs: model of time and control

MoC	Model of time	Control
FMI	Continuous	The system is observed throughout the simulation
DF	Logical (no notion of time)	The system is observed at the end of the execution (Causal)
SR	Logical (instantaneous reactions)	The system should be observed at system reactions
DE	Discrete time instants (differed reactions)	The system is observed at time events occurrences

Some issues regarding the integration of non-CT MoC are observed in the call sequence of the master algorithm given in Figure 2-3:

- CPS & FMI Issue 1 (I1): The first issue is related to the inability of the master algorithm to account for untimed behaviors (behaviors relying on the DF MoC) and instantaneous reactions (behaviors relying on the SR MoC). The master, in fact, does not consider simulation step of size zero.
- CPS & FMI Issue 2 (I2): The second issue is related to the choice of the simulation step size. This latter is chosen by the master without any assumption regarding the instants at which the component should be observed.

These issues flow directly from limitations in the FMI API which are identified and explained in sections 2.2.1 and 2.2.2.

2.2.1. Untimed semantics are not supported (I1)

<u>*Context:*</u> Untimed (and instantaneous reactive) behaviors do not take into account time to execute. There is indeed no notion of time. Outputs are supposed to be produced and propagated from one component to another at the same instant inputs. If the *fmi2DoStep* is called on an untimed component at time ‘t_c’, then data should be produced and propagated from this component to the other components at ‘t_c’. The master should use a zero-step size for such components.

FMI specification: The use of zero step size is not allowed [17]. The standard does not account for untimed behaviors and does not provide a mechanism allowing the component to output an instantaneous reaction to a changed value during simulation. As stated in section 2.1.2, the master algorithm performs stepwise simulation on the connected components where each component is supposed to execute and advance for an amount of time ‘h>0’ chosen by the master. If the *fmi2DoStep* is called on an FMU at time ‘t_c’ with a simulation step size ‘h>0’, then data will be propagated from one component to another at the next communication point ‘$t_{c+1}=t_c+h$’.

Issue: **When wrapped to FMI API, untimed and instantaneous reactive behaviors will not behave correctly. In fact, the data propagation will be delayed with an amount of time Δt=h>0. This delay may considerably affect the simulation results since the component will not produce outputs to its environment at the desired time.**

2.2.2. Time events are not handled (I2)

Context: Semantics of DE components is timed but not continuous. The component produces new outputs at a discrete set of time instants t_e (time event occurrences). For an efficient simulation, the component should be observed at each t_e. A *doStep* should therefore account for these specific time instants.

FMI specification: The FMI specification does not provide a way for the component to express the instants at which it will produce new values. The FMI specification provides a way for an FMU to express a desired static step size (optionally expressed in the xml file). However, the API does not provide routines that allow the FMU to express a change according to the desired simulation step size during the simulation.

Issue: **When wrapped to FMI API, DE components may not behave correctly. For example, suppose that the master performs a *doStep* with a step size h>0 at t_c, and the component has a time event at time $t_e<t_c+h$. The event will be missed and the data propagation will be delayed with an amount of time $\Delta t=t_c+h-t_e>0$. This delay may considerably affect the simulation results since the component will not produce outputs to its environment at the desired time.**

2.2.3. Conclusion on FMI issues related to CPS domain

The FMI standard provides an interesting basis for the modeling and simulation of CPSs with some limitation. We identified in this section 2.2.1 and section 2.2.2 two important the limitations of FMI for CPS domain: untimed semantics is missing and time events are not supported. It should be noted that we are not interested in limitations identified in the literature regarding the CT components and we will not provide solutions to them because they do not belong to our area of expertise. The next section focuses on solutions proposed in the literature for the simulation of CPS using FMI for co-simulation standard. The proposed solutions will be evaluated for their ability to handle untimed execution semantics and time events during simulation as well as for their applicability in the context of our work.

2.3. How to address those FMI limitations?

Existing works related to FMI aim at the consolidation of simulation of CPS that particularly combine CT and non-CT dynamics. Based on the investigation of FMI standard specification

and on works in the literature, we conclude that solving that aforementioned limitations of FMI for CPS domain could be performed by adopting one of the following techniques.

2.3.1. Adaptation of semantics at the FMU level

Principle: FMI aims at enabling the reuse of models in different co-simulation environments. One natural way to integrate a new modeling formalism in FMI-based co-simulation approaches is to export models as FMUs for co-simulation. An exported FMU must comply with the FMI standard, and it should contain the structure of the model and implement at least the mandatory functions of the FMI API mentioned in Section 2.1.1.2 of this chapter. The challenge would be to wrap semantics of various modeling formalisms (untimed semantics, discrete events semantics, etc.) into the FMI API. The export should be done by wrapping semantics of the original modeling formalism to that of FMI API, and model to model transformation for the structure of the model.

Related works: In [42], Tripakis and al. address principles of encoding different MoCs, including finite state machines (FSM), discrete-event (DE), and synchronous dataflow (SDF) as FMUs. Feldman and al. [16] developed an approach to generate FMI code from Rhapsody SysML models that wraps state charts as FMUs. The authors acknowledge problems with FMI co-simulation of state charts due to the standard's lack of support for instantaneous reaction to events. Pohlmann et al [33] generate FMUs from a UML model described as real-time state charts.

Evaluation: This technique certainly enables simulation of heterogeneous systems by integration of new formalisms. But it doesn't provide a solution for events handling since the models are wrapped into FMUs that comply with the FMI API which does not provide a way to get information about time events and instantaneous reactions. It is particularly interesting in industrial contexts where the goal is to facilitate the cooperation between different companies while preserving the IPs without any insurance about the efficiency of the simulation.

2.3.2. Extension of the FMI API

Principle: Extension of the FMI API essentially concerns the capabilities of FMUs to expose, or not, some information. This technique was already used in the second version of FMI, where the first version of the standard was extended to introduce new features (refer to section 2.1.1.1 for further details related to the added features).

Related works: Using these features, Broman and al. in [12] have formally proven the importance of these new features to ensure efficiency of the master algorithm in terms of determinacy (i.e, given a set of inputs, different runs of the simulation produce the same output) and successful termination of an integration step. They propose a preprocessing algorithm called "order-variables" that statically analyzes the dependencies in a co-simulation model based on the connectors that link the FMUs and the I/O dependency information exposed by the FMUs. The latter is used to detect potential loops in the co-simulation model and, if no cycle exists, the order in which the input and output variables should be accessed. Then, they propose the master algorithm called "Master-step" that requires all FMUs to support rollback.

The drawback of this algorithm is that the implementation of the rollback mechanism is still optional and may be difficult to achieve in practice. For this reason, they propose in addition,

an extension enabling a master algorithm to query an FMU for the time of events that are expected in the future. They propose to add a procedure to FMI API called "fmiGetMaxStepSize" that returns an upper bound on the step size that the FMU can accept. The last MA they propose uses the proposed extension to the FMI standard. They argue that the latter relaxes constraint imposed by the previous MA and correctly handles models containing a mix of FMUs that support rollback, FMUs that do not support rollback but implement the proposed extension to FMI for predictable step sizes, and at most, one FMU that supports neither.
Authors in [41] propose an equivalent extension which aims at improving the "fmi2DoStep" primitive by adding a new status "fmi2Event" returned by the FMU when an event occurs (either a time event or a state event) before the completion of the current simulation step. If the status "fmi2Event" is returned, that means that the FMU succeeds to perform simulation up to the time instant of the event. The MA should consider that the simulation was successfully performed until the time instant of the event and continue the simulation from this point of time without the need of the rollback mechanism.
Evaluation: These extensions certainly enable events handling, but the solution requires the agreement of FMI standard consortium to accept or reject such propositions. In fact, providers of models try to expose the most minimum amount of information in order to protect their IPs as stated in [28]. In addition, the efficiency of simulation not only depends on the capabilities of the FMU (i.e, the exposed information and the performance of the solver), but also on the ability of the master algorithm to account for all important information, which is why works that propose extensions have also to propose an advanced master algorithm as a part of their contribution.

2.3.3. Adaptation of semantics at master level

Principle: This technique consists in coupling FMU and non-FMU models in a specific co-simulation environment. The idea emerges from the need to integrate new modeling formalisms without the need to export the original models as FMUs for co-simulation either because there is no intention to reuse the models or because the export of models as FMUs is not trivial. The co-simulation environment is responsible for providing master algorithm for the orchestration of the connected models (since there are FMUs involved in the global model), which should cope with their heterogeneity. It should make a difference between an FMU and a non-FMU in order to simulate each component relying on their specific execution semantics.
Related works: Savicks et al. [36] propose an approach which enables co-simulation of FMUs with Event-B models without wrapping them as FMUs for co-simulation. The approach is tooled within the Event-B platform Rodin [2]. It is based on the master/slave architecture of the FMI standard. It proposes a simple MA for their orchestration but does not provide a way to handle time events.
Denil and al. in [13] define an adaptation of semantics between pure event models and continuous models. This adaptation is required in imported FMUs and in the master algorithm in order to set up an efficient co-simulation between heterogeneous formalisms. They define two extra FMUs that manage adaptation of semantics between the involved heterogeneous formalisms.
Evaluation: The main drawback of this approach is that it imposes the co-simulation environment to be compliant with a specific modeling formalism. The main strength of this approach is that models which are not exported as FMUs are accessible, that is, all information about the

model are exposed. Therefore, it ensures the reuse of models and their IP protection (the FMUs), while providing flexibility to cope with the heterogeneity of the involved models and easily handling events produced by the non-FMUs models.

2.4. Discussion and positioning

We presented in this second chapter the FMI for co-simulation standard. We focused particularly on its shortcomings for the modeling and simulation of CPS and identified two challenges:

- Issue 1: Untimed semantics are missing.
- Issue 2: Time events are not supported.

Three techniques proposed in the literature were evaluated on their ability to meet or not meet these challenges as shown in Table 2-3.

Table 2-3. Summary of the FMI-based techniques evaluation

Technique	Evaluation
Adaptation of semantics at FMU level	*Advantage:* The approach is completely compliant with FMI co-simulation and therefore well accepted in industry. *Drawback:* Issue 1 and Issue 2 cannot be solved with the current version of FMI.
Extension of the FMI API	*Advantage:* The approach is completely compliant with FMI co-simulation and therefore well accepted in industry. *Drawback:* Issue 1 and Issue 2 can be solved but necessitates the agreement of FMI standard consortium.
Adaptation of semantics at master level	Advantage: Issue 1 and Issue 2 can be solved. Drawback: The co-simulation environment must be compliant with a specific modeling formalism.

The best solution will be the combination of the two first techniques: the extension of the standard for the support of new formalisms, along with the export of models as FMUs for co-simulation and a MA which accounts for the new extensions. This technique definitely ensures the integration of new modeling formalisms (necessary for CPS), as well as handling events.

However, it is not trivial to do that in practice because, for instance, the proposed extension of the standard (in particular related to the time of the next event) is not approved by FMI consortium; by extending the standard, the providers of models are constrained to expose information which is not necessarily compliant with their policy [28].

The third technique is an intermediate solution. It enables the reuse of models while protecting their IPs in a standardized way thanks to FMI standard, as well as the integration of new modeling formalisms in FMI-based co-simulation. It provides a solution to cope with the heterogeneity of systems by integrating FMUs and non-FMUs models. It also enables events handling and untimed semantics in non-FMUs components, since all important information about these models are exposed and known by the environment. We will demonstrate how, using adaptation of semantics at master level, we are able to integrate the UML models in FMI-based co-simulation while accounting for their event-driven and instantaneous behaviors.

UML models are required to be executable for their integration in FMI-based co-simulation. The next chapter, focus on UML models execution as well as on challenges of their integration using the third technique.

Chapter 3: UML models execution - Overview and Key aspects

Outline

3.1. Tools for UML models execution
  3.1.1. Tools evaluation
  3.1.2. Discussion and positioning
3.2. Computational components modeling and simulation with fUML*
  3.2.1. Systems of interest
    3.2.1.1 Transformational vs Reactive systems
    3.2.1.2. Untimed vs Timed systems
  3.2.2. Executable models within fUML*
    3.2.2.1. Passive behaviors
    3.2.2.2. Active behaviors
  3.2.3. Non-executable models within fUML*
    3.2.3.1. Timed Behaviors
    3.2.3.2. Behaviors reacting to change events
  3.2.4. Addressing fUML* limitations
    3.2.4.1. Extending fUML* (F1)
    3.2.4.2. Introducing the control of timed execution (F2)
3.3. Methodology for the integration of fUML* and FMI
3.4. Conclusion

UML is the reference standard for software modeling and is very commonly used in the industry [4]. Our thesis is that system engineering in general would greatly benefit from the consideration of UML in the FMI-based co-simulation approach. It would indeed enable a significant number of software designers to evaluate the behavior of their software components in their simulated environment as soon as possible in their development processes, and therefore enabling them to make better design decisions earlier. It would also open new, interesting perspectives for CPS system engineers, as it allows them to consider a widely-used modeling language for the software parts of their systems.

In the previous chapter, we explained why we chose the adaptation of semantics at master level as a technique for the integration of new modeling formalisms in FMI-based co-simulation, and stated that we are particularly interested in UML as a language for computational components modeling. The main purpose of this chapter is to identify the key entry points for the application of this technique on UML models.

Simulation capacity of UML models is available in several tools. These tools will be enumerated and evaluated in section 3.1 of this chapter. The goal of this evaluation is to find a tool or a framework on which we could rely on for the co-simulation of UML models

in the FMI context. We are particularly interested in tools which support FMI for co-simulation or at least can provide capabilities for the support of FMI features and the integration of UML models.

In section 3.2, we will take a closer look at the modeling and simulation of computational components with UML. We will establish a list of UML models we want to model and simulate, based on a classification of discrete systems, as well as on a systems design methodology (subsection 3.2.1). After that, we will evaluate the ability of fUML* to simulate these models (subsections 3.2.2 and 3.2.3) and point out the key elements regarding the integration of UML models in FMI-based co-simulation (subsection 3.2.4).

3.1. Tools for UML models simulation

The integration of UML models in co-simulation approaches requires the UML models to be executable. Simulation capacity of UML models is available in several tools. They will be enumerated and evaluated in subsection 3.1.1. They will be evaluated on their support of FMI features and their integration of UML models in FMI-based co-simulation. At the end of this section, we will give our positioning and choose our start point for the integration of UML models in FMI-based co-simulation.

3.1.1. Tools evaluation

Table 3-1 checks the support for FMI standard in UML tools (the first column of the table) as well as the integration of UML models in FMI-based co-simulation, either by exporting UML models to FMUs, or by providing a master which adapts semantics of UML models' execution to that of the FMI standard (second column of the table).

Table 3-1. UML tools evaluation

UML Tools	FMI for co-simulation features	UML to FMI for co-simulation adaptation and integration
Magic Draw - Cameo Simulation Toolkit	Yes Master, version 1.0	No
Rodin	Yes Master, version 1.0	No
Cosimate	Yes Master, version 2.0	Yes
IBM Rational Rhapsody	Planned	No
Gemoc Studio	Planned	-
Mentor Graphics Bridgepoint	No	-
Entreprise Architect - AMUSE	No	-
Moliz	No	-

- **Magic Draw - Cameo Simulation Toolkit** provides an execution engine for the UML state charts and fUML*. This toolkit also provides an implementation of FMI compliant with the version 1.0 of the standard. FMUs for co-simulation can be imported, represented, connected and co-simulated in SysML models.
  Although the tool provides both execution of UML models and an implementation of FMI, the integration of both features does not exist. In the context of co-simulation, SysML is only used for the representation of the imported FMUs and the definition of co-simulation scenarios.
- **Rodin** is a tool developed within the European project FP7. It allows the verification and simulation of formal models specified using Event-B (a variant of B language). Rodin embeds a module for a formal specification of systems using UML classes and state machines. This module then transforms the UML specifications into Event-B models for the animate of the UML model using the model checker Pro-B.
  The Rodin framework also provides a master algorithm for the co-simulation of Event-B models with FMUs exported from Ptolemy. The approach is based on the version 1.0 of FMI standard. Although, the Rodin framework provides a way for UML models simulation as well as support of the FMI standard, no work links these capabilities as far as we know.
- **Cosimate** is an open architecture enabling engineers to connect various simulation environments together. This framework supports various language interfaces, in particular FMI for co-simulation, and simulators, more specifically, IBM rational rhapsody for UML models' simulation. Cosimate provides an implementation of the FMI standard version 2.0 and allows for the co-simulation between FMI and non-FMI models. It supports heterogeneous co-simulation between solvers (e.g., Simulink) and event-driven (HDL, UML) or sequential (C) simulators. Using the Cosimate framework, [...] proposed a co-simulation between UML models designed in rhapsody with models designed in numerical simulation tools, such as Simulink and AMESim.
  However, no details concerning the co-simulation approach are available, particularly in terms of whether the co-simulation of UML models with numerical simulators is ad'hoc or based on the FMI interface.
- **IBM Rational Rhapsody** is an UML modeling and simulation tool. It allows for the execution of UML models build using classes, activities and state machines.
  This tool provides the export of UML models as FMU for model exchange, but not for co-simulation. The support of FMI for co-simulation is planned[16].
- **Gemoc Studio** is an eclipse package which contains components that offer a framework for building and composing MOF-based executable Domain Specific Modeling Languages (xDSML). It addresses the execution of fUML activities using the xMOF Execution engine [11]. The support of a master algorithm for FMI co-simulation in the gemoc studio is under investigation. Gemoc studio deals with both UML models execution and the support of FMI for co-simulation. However, no explicit integration between the two features is done.

[16] Refer to: http://fmi-standard.org/tools/

3.1.2. Discussion and positioning

In the previous section, we evaluated UML tools (which provide the simulation of UML models) for their support to the FMI standard. This evaluation underlines the lack of the integration of UML models in FMI-based co-simulation, and emphasizes that there is no concrete, useable solution on which we can base our work. As a result, we have to propose a UML compliant environment with support for FMI (Chapter 4) together with an approach which properly integrates UML models in FMI-based co-simulation (Chapter 5 and Chapter 6). By properly, we mean the synchronization and the coordination of the models' executions while preventing events missing and respecting the execution semantics of each component (i.e, providing solutions to Issue 1 and Issue 2 of FMI for CPS domain introduced in Section 502.2).

As stated in section 1.2.4, the adaptation of semantics between heterogeneous models requires thorough knowledge of each of them. In particular, the integration of UML models with FMI requires the knowledge of the FMI standard API (refer to Chapter 2) and the UML models' execution semantics. In this context, Works around execution of UML models are carried out. PragmaDev[17] tool, for example, proposes a combinaison between UML and SDL[18] (Specification and Description language) for the description of real time systems. SDL is interesting language for the modeling of systems [3] and provides precise semantics for the execution of interaction diagrams and state machines. For this work, we propose the use of the OMG standards related to the execution of UML models: foundational UML (fUML) and Precise Semantics for Composite Structures (PSCS). In the rest of the text we will refer to fUML and PSCS as fUML*. This choice is motivated by two reasons: (a) fUML* propose a standard basis for UML models' execution in which it is essential to capitalize, and (b) fUML* is already supported in several tools (MagicDraw - Cameo Simulation Toolkit, Enterprise Architect - AMUSE, Papyrus – Moka, and Moliz), which allow the proposed contributions to be adapted to other fUML* compliant tools.

fUML* define precise semantics for the execution of a subset of UML models (refer to Annex A for an overview of fUML* syntax and semantics). We will see in section 3.2 that this subset does not cover all the UML models we would like to model and simulate, but the fUM* semantic model does provide features (mechanisms) to tackle this limitation.

Section 3.2 focuses on the modeling and simulation of UML components with fUML*. It outlines the systems we can simulate as well as those we are not yet able to simulate with fUML*. Section 683.2.4 proposes a set of fUML* features as key points for enabling the simulation of a larger scope of UML models. This chapter concludes with our positioning and an introduction to the contributions part of the manuscript.

3.2. Computational components modeling and simulation with fUML*

3.2.1. Systems of interest

UML is sufficiently expressive to model software specifications. The set of systems we would like to model with UML and integrate in co-simulation approaches can be classified according to two dimensions found in literature. The first one concerns the fact that the systems can be

[17] http://www.pragmadev.com/
[18] https://www.irit.fr/Chap5SDL.pdf

transformational or reactive (section 3.2.1.1). The second one concerns the fact that the systems can be untimed or timed (section 3.2.1.2).

3.2.1.1.Transformational and Reactive systems

By referring to computational components classifications in the literature [31, 55] we identified two kinds of systems we want to model with UML: systems which simply perform a computation, the so-called transformational systems, and systems which are reactive to events occurrences, the so-called reactive systems. For each class of systems, three semantic properties are outlined. They concern:

- (P1) Activation: It indicates the way the system is activated. For the corresponding UML model elements, it indicates the instants of its instantiation and initialization.
- (P2) Behavior: It indicates the way the system behaves. For the corresponding UML model element, it indicates the way it should be executed during simulation and the instant it should be terminated.
- (P3) Output/input relationship: It indicates whether the output of the system depends on its input.

P1 and P2 allows us to later identify the equivalent routines of fUML* for the functions defined in the FMI API (inst(), init(), doStep() and terminate()). Refer to sections 5.1.2, 5.2.3, 6.1.2 and 6.2.3 for further details on the mapping we propose between the fUML* routines and the FMI API. P3 is the equivalent of the I/O dependency expressed on FMUs and is used by the master algorithm to compute the order of data propagation between ports as explained previously in this chapter.
The following subsections describe the transformational and reactive systems according to these properties.

a. Transformational systems

Transformational systems are systems that simply transform a set of inputs into a set of outputs [55]. When switched on, a transformational system accepts inputs, performs some computations and produces outputs, then terminates (Figure 3-1). Examples of transformational systems, also called passive components, include process applications which are used in embedded systems to encapsulate a piece of behavior that execute synchronously in a short cycle time.
A transformational system has the following properties [45]:

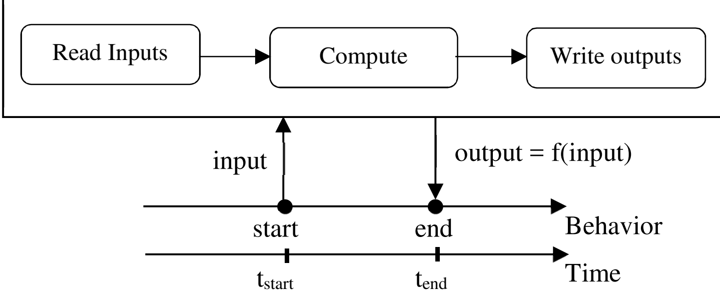

Figure 3-1. A transformational system

- (P1) Activation: A transformational system runs computations only when asked to do so by another component. That is, it should be handled by or contained in another component.

- (P2) Behavior: A transformational system transforms a set of inputs into a set of outputs and terminates.
- (P3) Output/input relationship: The output values of a transformational system depend on its input values.

b. Reactive systems

Reactive systems are systems that maintain interaction with their environment [21], which means that, when switched on, they are able to create the desired effect on their environment as a response to a received event (Figure 3-2). The response to an event generally depends on both the type of event and on the internal state of the system.

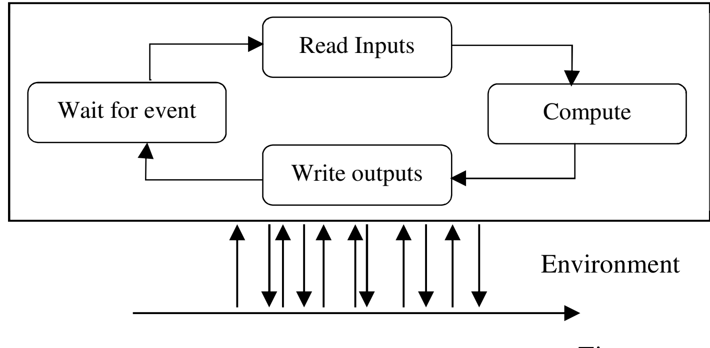

Figure 3-2. Interaction of a reactive system with its environment

Reactive systems continuously wait for the occurrence of some external or internal events.

- External events refer to some change on the environment conditions such as an On/Off button **press or a liquid** which exceeds the allowed level in a tank. Usually, the instant at which the change event occurs is not known at the system specification phase.
- Internal events refer to events which occur among the component. Time events (such as a timer tick or a calendar event) are an example of internal events and represent significant moments in time. Unlike change events, the instants at which time events occur are known and may be specified in the system behavior (which may be absolute or relative to a reference).

After recognizing the event, such systems react by performing the appropriate computations. Once the event handling is complete, the system goes back to waiting for the next event.
The properties of a reactive system are as follows [45]:

- (P1) Activation: A reactive system constantly interacts with its environment.
- (P2) Behavior: The system shall react to a received event and produce the correct reaction to the environment.
- (P3) Input/output relationship: The outputs of a reactive system depend on the stimuli it receives as well as on the current state of the system.

Both kinds of aforementioned systems, transformational and reactive, can be seen through a system design methodology where details (i.e, temporal information and reaction to changes) are progressively introduced at different design levels.

3.2.1.2. Untimed and timed systems

Models intended for simulation should be organized in a way that enables reasoning about the structure and the behavior of the real system. The structural viewpoint concentrates on the static information about the structure of the system (the inputs, the outputs and the components composing the system). UML class and UML composite structure diagrams are the most popular diagram kinds for structural modeling. UML also allows for the expression of relations between two or more UML elements (dependency relation, for example).
The behavioral viewpoint captures the dynamics of the system and describes its evolution over time. In UML, this perspective is usually handled by activity or state machine diagrams. Models intended for simulation purposes should, in particular, account for the time. As a result, we distinguish two kinds of UML behaviors: untimed behaviors and timed behaviors.

a. Untimed behaviors

An untimed model describes the system behavior in a way where only the logic of how the system accepts inputs, executes computations and produces outputs is important. This kind of representation is used for a functional simulation where the goal is to verify the partial correctness of the system behavior (i.e, check whether the system produces the correct result/reaction to a given input/stimuli or not), or for the representation of the instantaneous computations/reactions of the system. The integration of untimed behaviors in FMI-based co-simulation is the focus of the Chapter 5.

b. Timed behaviors

A timed model describes the system behavior in a way where both the logical and the timing information are considered. The model provides additional details about the behavior of the system, that is, it represents the instants at which the system receives inputs and produces outputs. It also expresses how long it takes to run computations or to react to events. The system is evaluated at a discrete set of time instants. This kind of representation is used in timed simulations where the correctness of the system behavior is not only defined based on producing the correct output, but also on producing it at the right time, such as in real-time and control systems design. This representation is in particular very relevant for co-simulation purpose since the model will be placed in its environment where it evolves and depends on time.
The UML standard [30] proposes a model of time which enables the representation of time in the applicative models. This latter comprises meta-classes to represent time and durations, as well as actions to observe the passing of time. The UML profile MARTE [29] also allows to annotate UML models with temporal parameters such as durations, periods, and deadlines. It introduces a model of Time and Timing Constraints, dealing with both physical and logical time. The integration of untimed behaviors in FMI-based co-simulation is the focus of Chapter 6.

3.2.1.3. Summary about systems of interest

In short, the models we would like to describe and integrate in a co-simulation approach are:

- An untimed UML model for a transformational system,
- A timed UML model for a transformational system,
- An untimed UML model for a reactive system; and
- A timed UML model for a reactive system.

For instance, we specified the need of modeling a set of systems with UML. In the rest of the chapter, we will focus on how to simulate these models using fUML*. The purpose of fUML* is not to give execution semantics to the whole UML set, but to a minimum essential subset which assumes the most general type of systems. This subset includes elements to model both the structure and the behavior of a given specification and is restricted to classes, activities and composite structures (refer to Annex A for more details about the syntax and semantics considered by fUML*). Next section enumerates the systems we can and cannot execute with fUML*.

Two mentions will be used in the rest of the chapter:

- *(syn) mention indicates that the element is a syntactic element.*
- *(sem) mention indicates that the element is a semantic element.*

3.2.2. Executable models within fUML*

3.2.2.1. Passive behaviors

A simple executable model we can handle with fUML* is a model of a system performing some computations. An example of such systems are transformational systems.
Figure 3-3 illustrates a simple example of a transformational system: a passive UML Class$^{(syn)}$ called 'Transformation'. This class owns two Property$^{(syn)}$ ('in' and 'out'). The behavior of the class consists of applying a transformation on the property 'in' and storing the result in the property 'out'. This behavior can be defined using an Operation$^{(syn)}$. We defined two operations: 'multiply(in x: integer):integer)' and 'transform()'. The latter represents the main behavior of the class.

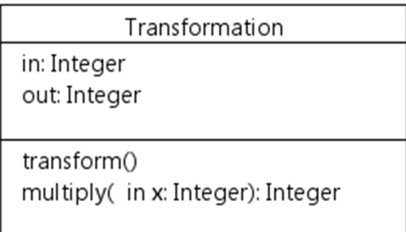

Figure 3-3. Transformation is a passive class

An Operation$^{(syn)}$ behavior can be specified with an Activity$^{(syn)}$. In this example, the activity associated with the operation ('multiply(in x: integer):integer)' simply returns the result of the multiplication of the parameter 'x' by a constant (Figure 3-4).

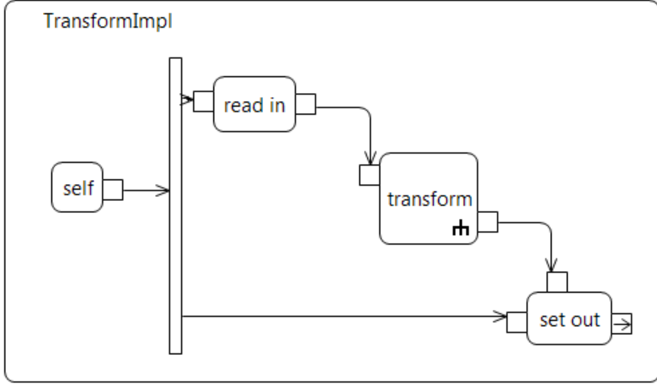

Figure 3-4. The specification of the operation 'transform' with an activity

We can then define the transformation on the properties (‘in’ and ‘out’) of the class ‘Transformation’. Figure 3-5 illustrates the specification of the ‘transform()’ operation. It models the logic of the transformation: it reads the value of the property ‘in’ using a ReadStructuralFeatureValue(syn), calls the ‘multiply’ operation using a CallOperationAction(syn)and sets the value of the property ‘out’ with the returned result using an AddStructuralFeatureValueAction(syn). When the simulation starts, no behavior is executed automatically. The ‘Transformation’ class waits for some other object to invoke it.

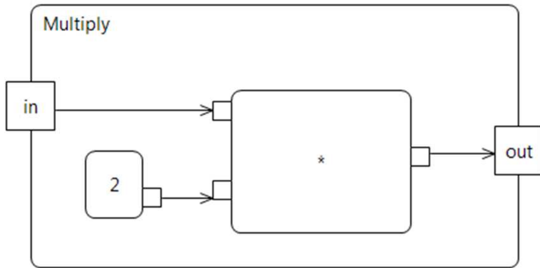

Figure 3-5. The specification of the operation 'multiply' with an activity

3.2.2.2. Active behaviors

fUML* provide syntax and semantics to model UML components whose behavior is triggered by a signal event. A common example is the Ping-Pong game. The game consists of two players that synchronize with each other by exchanging signals.

The game is represented with the structured Class(syn) ‘Game’ as shown in Figure 3-6. The players are represented with the parts ‘player1’ and ‘player2’ of type ‘Player’. Each of them owns a Port(syn) (respectively ‘portA’ and ‘portB’) through which the Signal(syn) is propagated from one player to another.

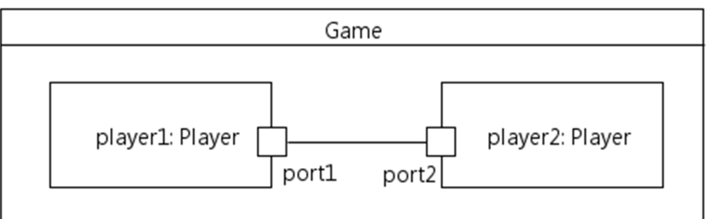

Figure 3-6. The Game system represented with a UML composite structure

The class ‘Player’ should be active since its behavior is triggered by signals receptions (‘Ping’ and ‘Pong’ in Figure 3-7). An active Class(syn) must own a classifier behavior that defines its main behavior (‘Player Classifier Behavior’ in Figure 3-7).

Figure 3-7. The players are active classes

The classifier behavior is expressed with an Activity(syn), which describes the synchronization logic from each individual player’s standpoint in terms of signal emissions and receptions. The classifier behavior of the class ‘Player’ is defined with the activity depicted in Figure 3-8. The

shape represents a SendSignalAction(syn) and the shape represents an AcceptEventAction(syn) triggered by a SignalEvent(syn).
When the simulation starts, the classifier behavior is automatically started. Player "player1" begins playing by sending a "Ping" signal then waits for a "Pong" signal from player "player2".

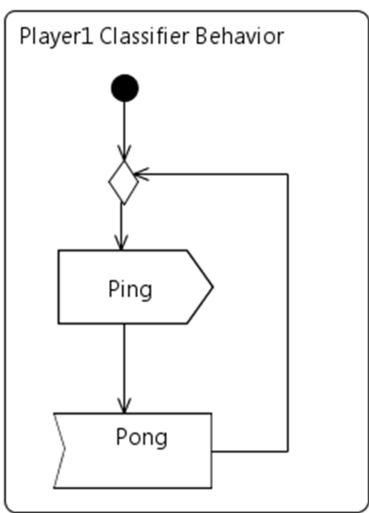

Figure 3-8. The behaviors of the players represented with activities

As soon as Player "player2" receives a "Ping" signal from player "player1", it sends a "Pong" signal, then waits again for a "Ping" signal from Player "player1". As soon as the player 'player1' receives a "Pong" signal from player "player2", it sends again a "Ping" signal and so on.
In the rest of the manuscript, we will not be interested in such reactive behaviors. Signal events, in fact, assume that components communicate and synchronize directly with each other. This is in contradiction with the requirements of the FMI standard where no direct communication between components is allowed and the master is responsible for their synchronization.
Instead, we are interested in the modeling of behaviors reactive to the change and time events. They are not in the scope of fUML* as we will also explain in the next section. Nevertheless, we will rely on the same syntactic and semantic mechanisms for the execution of behaviors reactive to change events and time events.

3.2.3. Non-executable models within fUML*

3.2.3.1.Timed behaviors

UML provides elements for time modeling. Figure 3-9 gives an example of timed behavior where the object is supposed to wait for five units of time ('after(5)') before sending a signal 'Ok'. The shape is an AcceptEventAction(syn) triggered by a TimeEvent(syn).

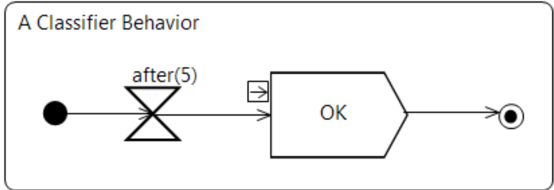

Figure 3-9. A timed behavior

As stated in section 3.2.1, we are interested in timed behaviors execution. We identified some shortcomings in fUML* for the modeling of such behaviors. The TimeEvent^(syn)^ does not belong to the fUML*. fUML*, in fact, does not account for time modeling and simulation. The behaviors are supposed to happen instantaneously.

Nevertheless, as explained in the fUML specification [32], "The execution model is agnostic about *the semantics of time. This allows for a wide variety of time models to be supported, including discrete time (such as synchronous time models) and continuous (dense) time."*

Initiatives including tools and languages related to the execution of timed UML models are given below. We are particularly interested in the possibility of their application in the context of fUML*.

- **Moliz**: It proposes a framework (i.e., a library) [39] for performance analysis that enables the integration of time representation in execution traces. The approach supports a discrete event model of time but relies on implementation of opaque behaviors to deal with simulation time advance.
- **MARTE/CCSL:** The UML profile MARTE [29] proposes a Time Model which extends the simplistic Simple Time models defined in UML specification. It offers a broad range of time models including discrete/dense time and chronometric/logical time. MARTE also introduces a time structure and proposes a Clock Constraint Specification Language (CCSL) to specify time constraints within the context of UML. A way to capture time semantics in fUML* is to rely on the formal semantics of CCSL clock constraints.

  Gemoc enables time support in simulation thanks to clocks defined in CCSL [TTC'15]. The approach is interesting in the sense that it can combine various time models. However, using it in the context of fUML* is not straightforward, since the approach makes strong assumptions about the way both syntax and semantics of a language are defined.

  Coupling fUML with existing approaches such as TimeSquare[19] could also be considered as a solution for enabling the execution of timed UML models using CCSL. TimeSquare provides an environment for modeling and analyzing timed systems. It supports an implementation of the time model introduced in the UML MARTE profile and the CCSL language (Clock Constraint Specification Language). TimeSquare takes an UML model as input, to which a CCSL model is applied. The CCSL model is used to specify time constraints and apply a specific behavioral semantics on a model. The result produced by TimeSquare is a sequence of steps (a Scheduling Map) that can be used by external tools for analysis/simulation purposes. Concretely, coupling the fUML semantic model would mean that a CCSL model must be generated for a given application model, and that the generated model reflects the time semantics of the application domain for which a profile is defined. Scheduling maps generated by TimeSquare could then be "played" by the execution model. Modifications in the architecture of the semantic model would be required, and would mainly consist in adding an explicit entity responsible for triggering executions of active objects and actions, with respect to the scheduling map generated by TimeSquare.

[19] Refer to: http://timesquare.inria.fr/

None of these proposals provide a framework or a way to directly execute timed behavior with fUML*. We will demonstrate in chapter 6 that the Simple Time model of UML proposes sufficient syntactic elements to describe timed behaviors for in both kinds of systems of interest, (i.e, transformational and reactive). The execution of timed behaviors simply requires the extension of fUML* by giving execution semantics of this Simple Time model (or a subset of it) and the control of the timed execution.

In widespread simulation tools and frameworks such as Ptolemy II[20] and SystemC[21], time is usually managed by an explicit control entity (scheduler-like), which appropriately schedules the execution of the various model elements in order to reflect the timing aspects. The fUML execution model does not include this type of control entity. Implementations are thereby responsible for providing timing mechanisms if needed. In regards to the execution model, the fUML* stated: "*Furthermore, it does not make any assumptions about the sources of time information and the related mechanisms, allowing both centralized and distributed time models*".

The extension of fUML* with the execution of timed UML models will be handled in Chapter 6 using features F1 and F2, which we will explain in section 3.2.4.

3.2.3.2. Behaviors reacting to change events

In CPS, computational components usually interact with their environment. Change events are interesting in this context in the sense that the cyber part of the CPS (typically a control component) could instantaneously detect and react to a change occurring in the physical part (typically the environment with/on which the control components interacts).

The modeling of behaviors reactive to changes of some values is possible using an AcceptEventAction$^{(syn)}$ triggered by a ChangeEvent$^{(syn)}$. The example represented in Figure 3-10 consists in comparing an input value with a constant and to produce a verdict. The comparison should be executed only if a change on the value of 'in' is detected.

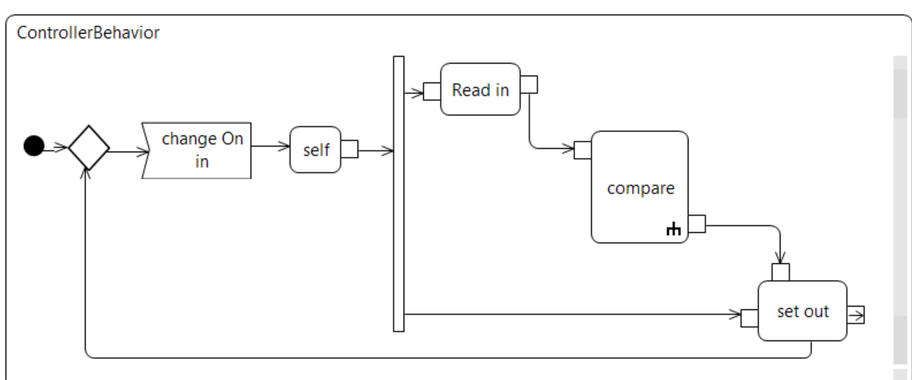

Figure 3-10. A behavior reactive to a change on a value

fUML* enables the modeling of behavior reactive to signal events (as stated in subsection 3.2.2.2) but not to ChangeEvent$^{(syn)}$. The ChangeEvent$^{(syn)}$, in fact, does not belong to the

[20] Refer to: http://ptolemy.eecs.berkeley.edu/ptolemyii/

[21] Refer to: http://hdl.telecom-paristech.fr/sc_intro.html

fUML* subset. An extension to the fUML* semantic model with new semantics is then required. Refer to section 5.2 for details about the extensions of fUML* with the ChangeEvent$^{(syn)}$.

The fUML* subset restrict the scope of the systems we can model and simulate, but at the same time, they are endowed with features allowing us to expand this scope. The next section explains how we can address limitations of fUML* regarding the execution of timed and reactive behaviors.

3.2.4. Addressing fUML* limitations

3.2.4.1. Extending fUML* *(F1)*

Principle: As stated in the Annex A, the fUML semantic model is built around the visitor pattern. According to this architecture, a natural way of building an extension to the semantic model is to introduce additional semantic visitors. Two scenarios are possible depending on the syntactic element *SynElt* to which the visitor is defined:

- The *SynElt* is considered by the fUML syntactic subset
  In this case, the semantic visitor associated to SynElt in fUML semantic model can be extended using object oriented mechanisms such as inheritance and polymorphism.
- The *SynElt* is not considered in the fUML syntactic subset.
  In this case, the first common meta-class *C* between *SysElt* and a syntactic element already considered in the fUML subset should be identified. A new visitor is then defined as a specialization to the visitor defined for the meta-class *C* in the fUML semantic model.

The introduction of new semantic visitors has, as consequence, the definition of an extension for the definition of new instantiation rules. This implies the extension of the classes:

- *Locus*$^{(sem)}$: the extension of this element, and therefore its operation *instantiate(),* is used when the new visitor is defined as a specialization to *Object*$^{(sem)}$. A common example is the definition of an extension to the meta-class *Classifier*$^{(syn)}$ or one of its specializations.
- *ExecutionFactory*$^{(sem)}$
  The extension of this element, and therefore its operation *instantiateVisitor(),* should be performed when the new semantic visitor cannot be created by the *Locus*$^{(sem)}$ class. A common example is the extension of the semantic model with new syntactic elements for the specification of a *Behavior*$^{(syn)}$.

Use case example: This extension strategy has been proposed in [40] and was successfully applied to propose a systematic approach for specifying the execution semantics of UML profiles as an increment to preliminary proposals developed in [8]. The most interesting case using this strategy is the OMG standard PSCS (Precise Semantics of UML Composite Structures), a normative extension of fUML dealing with the semantics of UML composite structures including informative annexes on the semantics of a subset of the MARTE and SysML profiles.

PSCS introduces new syntactic elements and their respective semantics such as *Connector*$^{(syn)}$ and *Port*$^{(syn)}$ in order to allow the communication of composite classes through ports. The semantic model of fUML is, therefore, extended with new semantic elements. PSCS first defines

CS_Object$^{(sem)}$ as a specialization of *Object*$^{(sem)}$ in the fUML semantic model. This extension is required in order to allow the representation of composite classes in the locus. Semantics of the operation calls and the signal reception for communication through ports are defined in the methods of the *CS_Object*$^{(sem)}$ class. The definition of new semantic visitors also requires the definition of the two factories *CS_Locus*$^{(sem)}$ and *CS_ExecutionFactory*$^{(sem)}$ as extensions to Locus and ExecutionFactory classes in the semantic model of fUML respectively. Other than the semantic visitors and the factories, PSCS defines the class *CS_DefaultConstructStrategy*$^{(sem)}$ as a specialization of the *SemanticStrategy*$^{(sem)}$ of the fUML semantic model. It defines the instantiation strategy of models using composite structures. Figure 3-11 depicts an extract of the semantic model of PSCS.

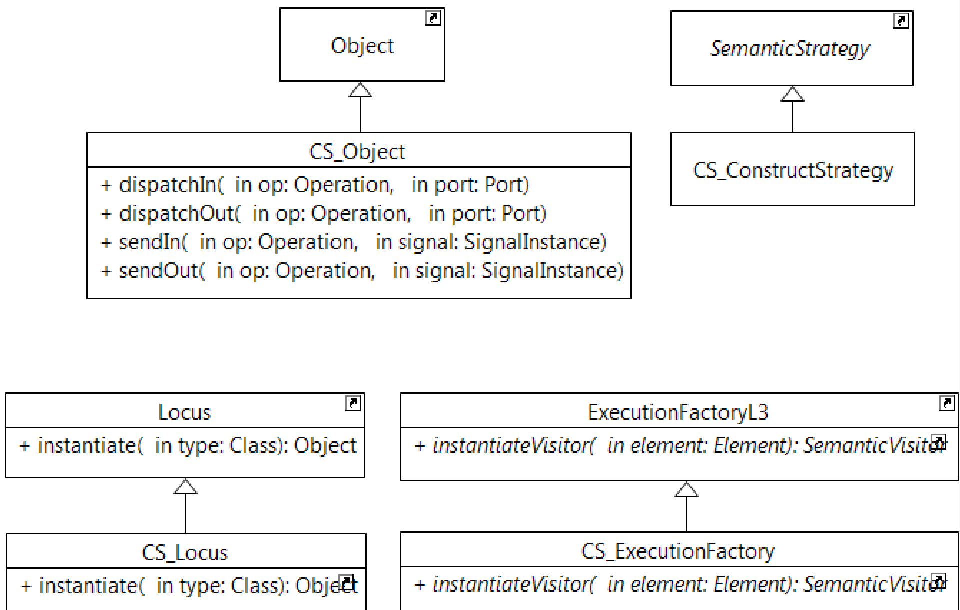

Figure 3-11. Extract semantic elements of PSCS as extension to the fUML semantic model.

3.2.4.2. Controlling executions ***(F2)***

Principle: The causality resulting from the execution semantics of activities must be respected by any fUML-compliant execution engine (i.e, a particular implementation of fUML). Therefore, a fUML-compliant execution engine that intends to provide facilities which are out of the scope of fUML needs to reroute the usual token propagation flow through external control entities while preserving the original causality. The flexibility of fUML resides in the fact that the specification does not provide any strict recommendations on how this should be implemented.

Use case example: This generic principle has been used to establish a connection between the fUML execution engine of Moka and the Eclipse Debug framework. For example, the semantics of activity nodes (captured by the visitor *ActivityNodeActivation*$^{(sem)}$) has been overloaded so that, when they receive offered tokens (operation *receiveOffer()*), the control is rerouted towards an external control entity. This control entity is responsible for the management of debugging events, as well as the animation of nodes on diagrams. In another experiments, the same delegation mechanism has been used to produce execution traces, by rerouting through a tracing entity [18].

Based on these elements, we believe that the limitations of fUML* regarding the execution of the timed and the reactive behaviors (refer to section 3.2.3) can be handled. The integration of UML models in FMI-based co-simulation (using the adaptation at master level technique) consists therefore in adapting the fUML* execution semantics to that of the FMI API. Next section identifies the steps to follow for the integration of fUML* and FMI.

3.3. Outline of the proposed approach

Figure 3-12 depicts a class diagram summarizing the different co-simulation cases we want **to define** and simulate. As stated in section 2.4, the adaptation at master level is the most suitable technique for the integration of fUML* and FMI, where the FMUs are black boxes connected to the white box UML models. The definition of these cases requires the co-simulation environment to be UML tool which provides an implementation of fUML* as well as an implementation of FMI standard. As stated in the section 3.1.2, no useable solution was found for the purpose of this work, but UML tools providing support to fUML* do exist. We will then use one of them and extend it with an implementation of the FMI for co-simulation standard as a first step of our contribution.

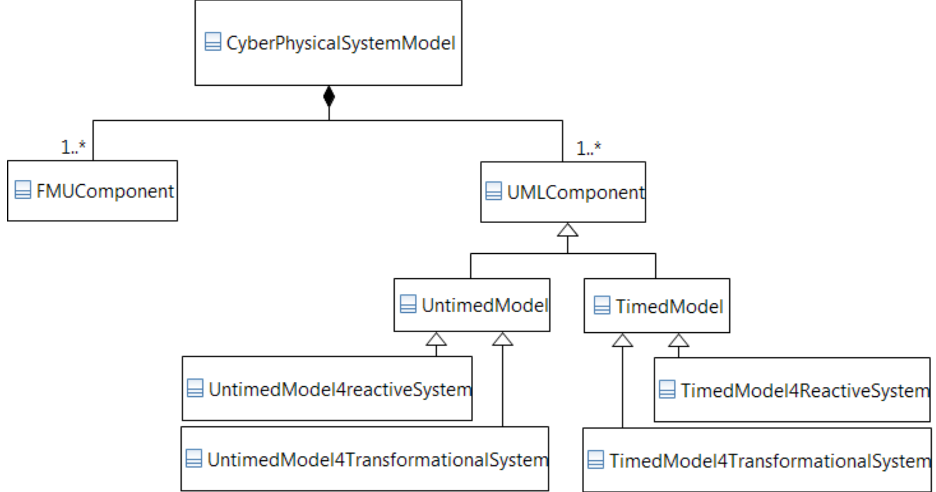

Figure 3-12. Composition of FMI based co-simulation cases

The components execute independently of each other. FMUs are executed with respect to the FMI API, and UML components execute with respect to the fUML* semantics. As stated in Chapter 0, the master is then responsible for the orchestration of the components and the synchronization of their simulations as follows:

- The orchestration of the involved components: For this task, the master propagates data from the **outputs** to the **inputs** of the connected components while accounting for their **I/O dependencies**, and performs stepwise simulation where it requires equivalent fUML* routines for the functions defined in the formalization given in section 462.1.1.2 (inst(), init(),doStep(), terminate()),
- The synchronization of the involved components' simulations: For this task, the master bridges the semantic gap between UML models execution semantics and the FMI API. It provides adaptation of semantics of untimed UML models to timed FMUs semantics, and discrete UML models to continuous time FMUs semantics. For this purpose, the master computes the suitable simulation step size in order to propagate data

and trigger the components at the correct instants. The UML components reflect the **MoC** on which it relies, in particular, information about the **model of time.**

We note that the efficiency of the adaptation between fUML* and FMI depends on the information available on the components. That is, for an efficient simulation, it is also necessary to:

- Explicitly express in the UML models information written in bold above in this section,
- Identify for each kind of computational component a minimal set of UML structural and behavioral syntactic elements to model it. It is also necessary to check whether fUML* covers the whole set. If not, we need to extend the fUML* with the new syntax and semantics using the feature F1 and F2 as explained in section 3.2.4.

Figure 3-13 illustrates the steps of the methodology we propose for the integration of the UML models in FMI-based co-simulation.

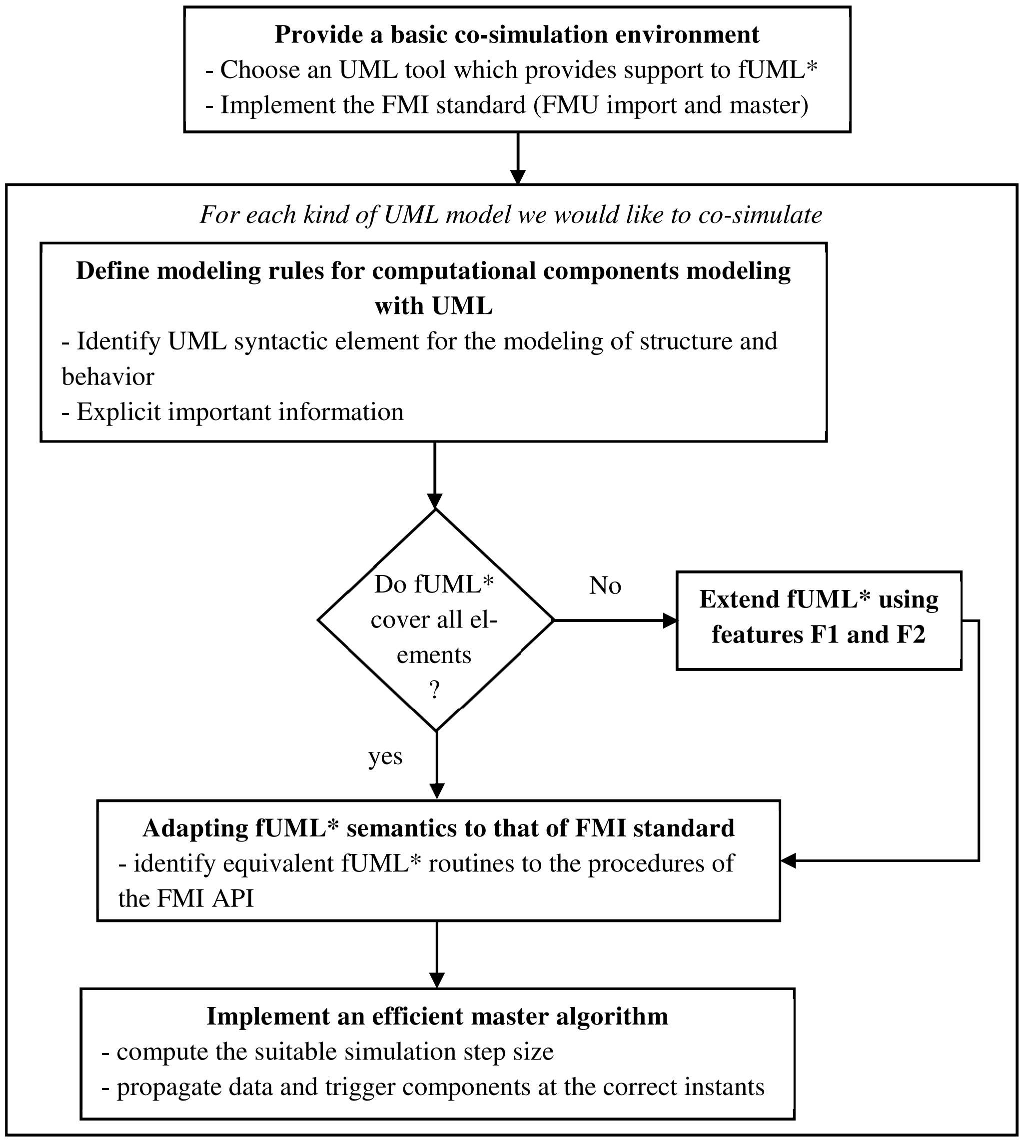

Figure 3-13. The proposed approach for the integration of fUML* and FMI

3.4. Conclusion

The main goal of this chapter was to identify the key elements for the application of the adaptation of semantics at master level technique for the integration of UML models in the FMI-based co-simulation approach.
We proposed to base our approach on fUML* standards which define precise semantics for UML models' execution. fUML* is an interesting basis for the purpose of this work but have some limitations. We identified the kind of systems we would like to describe, transformational and reactive systems, in order to determine our specific needs regarding their modeling with UML and their simulation with fUML*.

Our contribution, which is to properly integrate UML models in FMI-based co-simulation using the adaptation at master level technique, shall operate at two levels: locally at the level of UML models (modeling), and globally at the master level (simulation). Locally, for each kind of computational components models, we shall first identify a set of rules to model it with UML, and potential extensions to fUML* in cases where execution semantics of the required UML elements are not defined by fUML*. Globally, we shall propose a master algorithm for the orchestration and synchronization of each kind of computational components models with a set of FMI based on the adaptation of the execution semantics of the fUML* to that of FMI API.

In the contribution part, we will begin with a technical contribution which consists of the development of a co-simulation environment in a UML-compliant tool in chapter 4. Then, we will continue with the description of scientific contribution concerning the integration of untimed and timed UML models in FMI based co-simulation in chapter 5 and chapter 6 respectively.

PART II: ABOUT THE CONTRIBUTION

We propose an incremental approach where we address various co-simulation scenarios. The contribution part is organized into three chapters, each of them dealing with a particular co-simulation scenario as follows:

- ***Chapter 4*: *UML-based Master Simulation Tool for modeling and simulation of CPSs***
  This chapter introduces the framework we set up for the modeling and simulation of CPS. The framework is based on an implementation of the FMI for co-simulation standard in a UML-based tool. This framework allows the definition of co-simulation scenarios composed of a set of FMUs. It provides capabilities for the import and the connection of FMUs as well as basic and sophisticated master algorithms for their orchestration.
- ***Chapter 5*: *Integration of untimed UML models in FMI-based co-simulation***
  This chapter concentrates on the integration of untimed UML models in FMI-based co-simulation. It aims at enabling the definition of co-simulation scenarios composed of FMUs and untimed UML components. For each kind of systems, transformational and reactive, we in turn identify a set of modeling rules for their modeling with UML in the context of FMI and potential extensions of the fUML* semantic model. The contribution consists then in adapting the semantics of fUML* with that of the FMI API, as well as, in proposing an efficient master algorithm for the orchestration and synchronization of the co-simulation scenarios. The efficiency of the master algorithm relies in particular on its ability to trigger the execution of each component at the correct instants while avoiding events missing.
- ***Chapter 6: Integration of timed UML models in FMI-based co-simulation***
  This chapter is dedicated to the integration of timed UML models in FMI-based co-simulation. It aims at enabling the definition of co-simulation scenarios composed of FMUs and timed UML components. This chapter is organized in the same way as Chapter 5.

Chapter 4: UML-based Master Simulation Tool for modeling and simulation of CPSs

Outline

4.1. Architecture of the framework
- 4.1.1. The Graphical User Interface features
  - 4.1.1.1. UML Profile for FMI co-simulation scenarios
  - 4.1.1.2. The import of FMUs for co-simulation
  - 4.1.1.3. The definition of co-simulation scenarios
- 4.1.2. The MST-engine features
  - 4.1.2.1. Overview
  - 4.1.2.2. A wrapper for FMI
  - 4.1.2.3. The master algorithm

4.2. Validation of the framework implementation
- 4.2.1. The use case: The TankPISystem
- 4.2.2. The import of the FMUs and the definition of the co-simulation scenario
- 4.2.3. The simulation results

4.4. Conclusion

The integration of UML models in FMI-based co-simulation requires a framework compliant with UML and FMI. This framework shall also provide either the possibility to export UML models as FMUs for co-simulation, or a master algorithm that is able to orchestrate heterogeneous co-simulation scenarios composed of FMUs and UML models. Unfortunately, as stated in Chapter 0, there is no usable FMI based co-simulation solution that considers UML models. For this reason, we decided to set up our own co-simulation framework and propose an UML compliant framework for FMI-based co-simulation. We present the architecture of the framework as well as the main features it offers in section 4.1. Then we validate the framework with a common CPS example in section 4.2. As a first step, this framework only provides the co-simulation of a set of imported FMUs. This first step is not part of the scientific contribution but it is necessary to provide the basis for the experimentation and the validation of the scientific contributions.

The second step consists in extending this framework for the integration of UML models. As stated in Chapter 0, we decided not to export our UML models as FMUs for co-simulation. Instead, we chose to provide a master algorithm responsible for the orchestration of heterogeneous co-simulation scenarios composed of FMUs and UML components. This step is performed in Chapters 5 and 6.

4.1. Architecture of the framework

An FMI compliant co-simulation environment shall provide facilities for the import of FMUs and their connection, and a master algorithm that controls the data exchange between the FMU

and the synchronization of their simulations based on the FMI API. We propose a master simulation tool (MST) composed of a graphical user interface (GUI) and a master simulation tool engine (MST-engine) as depicted in the class diagram of Figure 4-1. The GUI is essentially responsible for the modeling of CPS composed of imported FMUs. It shall at least provide facilities for import, configuration and connection of the FMUs.
The configuration of the co-simulation graph (important information about FMUs and the simulation parameters) are stored in the "MST-configuration" and used later by the MA. The MST-engine proposes one or many algorithms that are mainly responsible for the orchestration of the connected FMUs (i.e. data propagation between FMUs and the synchronization of their simulations).

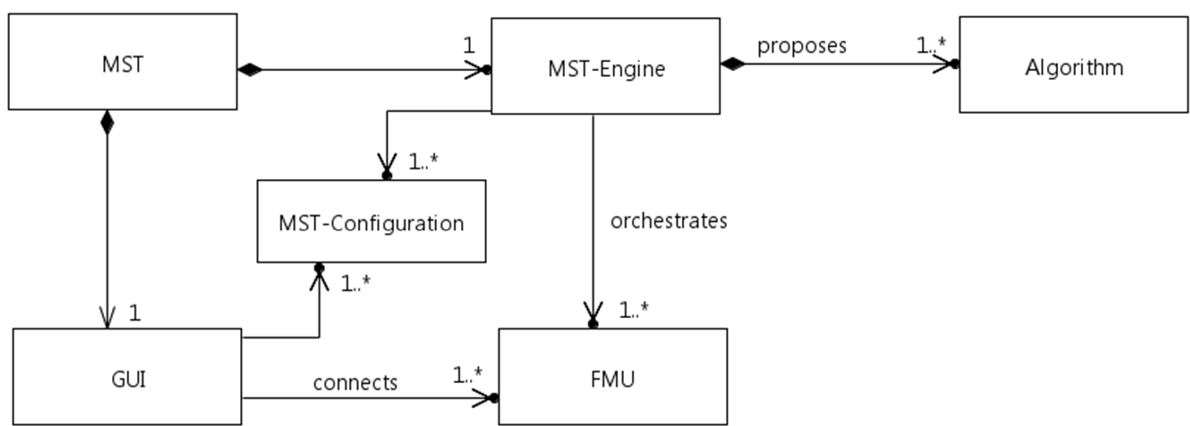

Figure 4-1. The co-simulation Framework

Section 4.1.1 and section 4.1.2 outline essential information, respectively, about the GUI and the MST-engine features.

4.1.1. Graphical User Interface features

Some of the facilities offered by the GUI are mandatory. They are those required by any FMI-based co-simulation environment such as the import and the connection of FMUs. Some others, such as displaying results in graphics at the end of the simulation, are optional.

In this section, we outline how we implement the mandatory features in a UML-based framework. This section is organized into three subsections that concern the co-simulation profile we propose for annotation of both the imported FMUs and the co-simulation graph, the import of FMUs for co-simulation, and the definition of co-simulation scenarios.

4.1.1.1. UML Profile for FMI co-simulation scenarios

UML is a generic modeling language whose expressivity gives modelers the ability and freedom to use it in many fields of engineering. UML can also be customized for a given domain thanks to the profile mechanism. Profiles define extensions to enrich the syntax and semantics of the UML language. Extensions are usually expressed with stereotypes. Each stereotype defines an extension to an element of the UML syntax and owns a set of attributes through which semantic information can be added.

A profile is particularly interesting in our work. It allows for the customization of our models to the co-simulation domain. The MST-engine requires information about the components it simulates and information about the simulation parameters. The "modelDescription.xml" file on the FMU contains all these data (refer to Chapter 2 of the state of the art for further details about the information contained in the xml file of the FMUs). We only need to preserve them when importing the FMUs in the UML-based framework. We propose an UML profile, the so-called *CoSimML* profile, that resumes all these data by mapping them to a set of stereotypes. These stereotypes are applied both to the co-simulation model and to the imported FMUs and are illustrated in Table 4-1.

A snapshot of the *CoSimML* profile is given in Figure 4-2. 'CS_Graph' stereotype is used for the class representing the co-simulation graph, 'CS_FMU' stereotype is used for an instance that represents an imported FMU, while 'CS_Port' stereotype designates ports through which data are propagated from one FMU to another. We also will apply the 'CS_Dependency' stereotype to the I/O dependency information. This latter is important to expose to the MA in order to analyze cycles in the co-simulation graph and determine the order in which data should be *get* from /set to ports.

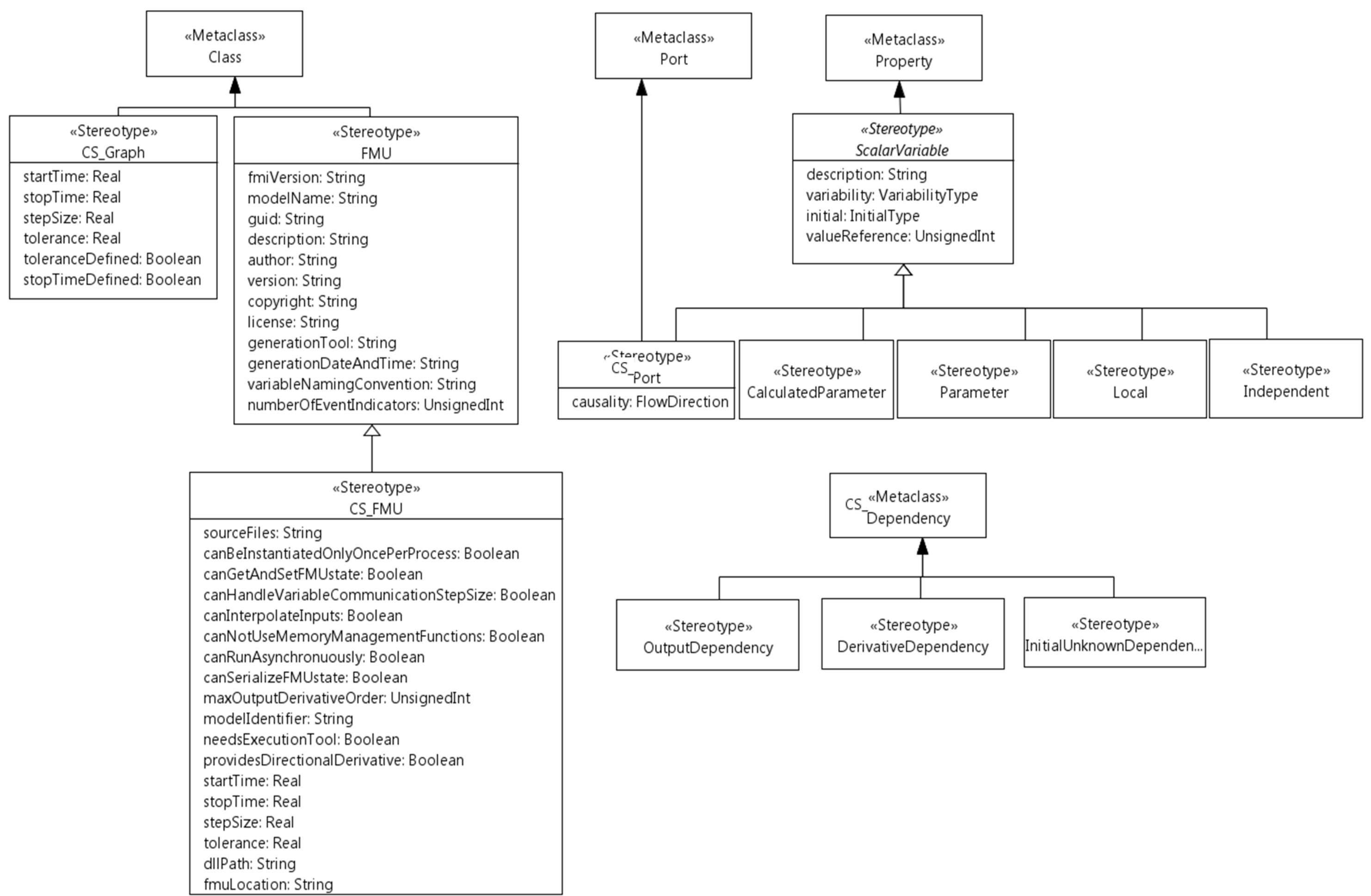

Figure 4-2.The Co-simulation Profile

Table 4-1. Stereotypes of UML profile for FMI.

Stereotype	UML Metaclass	Important Attributes	Specification
« CS_Graph »	*Class*$^{(syn)}$	- startTime - stopTime - stepSize	Used both to identify the class that represents the CPS and, to define a predefined configuration for the simulation.
« CS_FMU»	*Class*$^{(syn)}$	- canHandleVariableCommunicationStepSize - canGetAndSetFmuState	Applied to identify an UML class that represents an imported FMU. When set to true, *canHandleVariable-CommunicationStepSize* and *canGetAndSetFmuState* respectively mean that the FMU can accept different step sizes and, that the FMU supports rollback.
«CS_Port »	*Port*$^{(syn)}$	- causality	Used first to identify the ports through which the data will be propagated during simulation and then, to narrow down/precise the direction of data propagation by setting the causality attributes to "in" or "out" value. The modeler and the MST-engine should be aware of the flow direction of each port. This minimizes errors when connecting the components and ensures a correct analysis of the dependencies between components. Other attributes can be added to avoid modeling errors such that the unit and a brief description of the data passing through the port.
«Parameter», «Local», «calculatedParameter», and «independent»	*Property*$^{(syn)}$		Are applied to scalar variables respectively of kind parameter, calculated parameter and independent as defined in the "modelDescription.xsd" of the FMI specification.
«Output_Dependency»	*Dependency*$^{(syn)}$		Used to identify UML elements that specify I/O dependencies between outputs and inputs of a component.

4.1.1.2. Import of FMUs for co-simulation

A co-simulation environment which imports an FMU should preserve information contained in the model description file and ensure access to the procedures of the dll. The first task requires

transforming the "modelDescription.xml" file to a model compliant with the modeling formalism supported in the co-simulation environment. The second task requires a wrapper, which enables access to native code using the programming language supported in the co-simulation environment. This subsection focuses on the first task. The second task will be explained in section 4.1.2.2 of this chapter.

Since the framework we propose is UML-compliant, then all features it proposes rely on UML concepts. When imported, the FMUs shall be transformed into a set of UML elements. We propose a model to model transformation using QVTo [31] language as depicted in Figure 4-3.

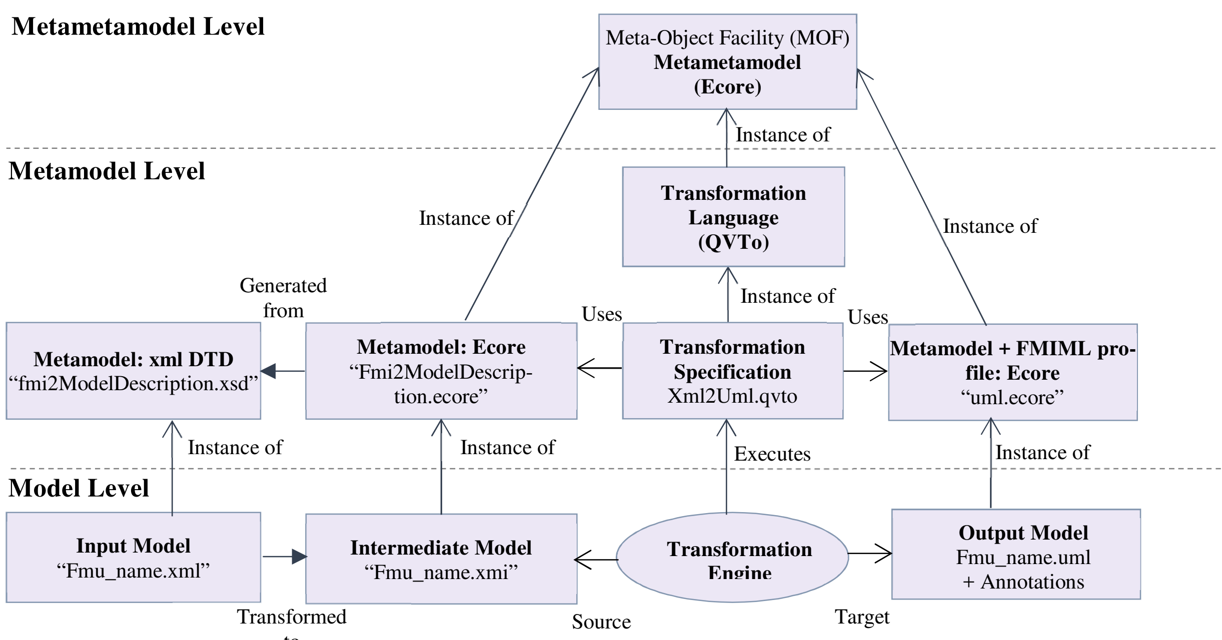

Figure 4-3. FMU to UML model transformation

The QVT transformation maps the model structure of an FMU, described in the "model description.xml" file, to a UML class and a set of UML dependencies. The class exposes input and output variables via UML ports and contains a set of UML properties that represent scalar variables of the imported FMU. The UML dependencies represent the I/O dependency information between the output and the input of the model. The co-simulation profile we proposed previously is applied to the resulting UML model.

Details about the specification of the QVTo transformation are given in annex B.

4.1.1.3. Definition of co-simulation scenarios

A co-simulation scenario defines the components making up the CPS and their connections. The model that specifies a co-simulation scenario is called "Co-simulation Graph". This model can be handled using UML composite structure diagram. As stated in the UML specification, the composite structure diagram could be used to show internal structure of a classifier namely ports, parts and their relationships [30].

In this chapter, we focus on co-simulation scenarios that are composed of imported FMUs as depicted in Figure 4-4. The '*CosimulationGraph*' is the composite class that represents the CPS and is composed of four parts (the 'fmuA', 'fmuB', 'fmuC' and, 'fmuD'), which represent imported FMUs (respectively 'FmuA', 'FmuB', 'FmuC' and, 'FmuD'). Each part has ports that

represent the interfaces for data exchange. Connectors represent a relationship between two connected FMUs.

The start time and the stop time, as well as the default step size of the co-simulation, should be set in the "CS_Graph" stereotype.

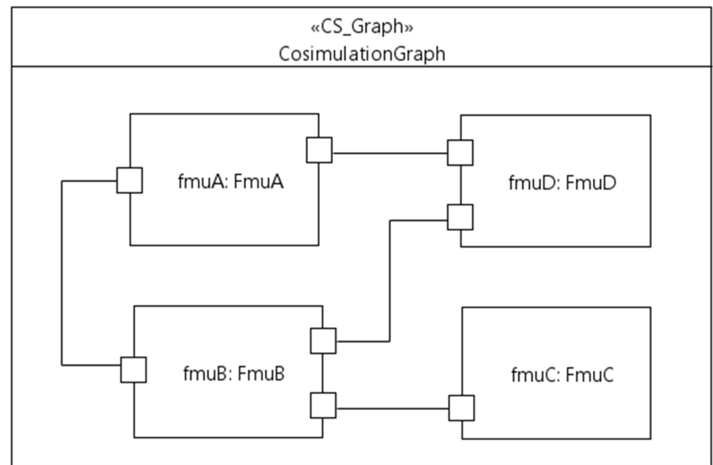

Figure 4-4. An example of a CPS composed of four FMUs in a UML composite structure diagram

4.1.2. MST-engine features

4.1.2.1. Overview

The MST engine is responsible for the synchronization of the imported FMUs. The MST-engine communicates with the GUI for getting configuration parameters, the properties of the FMUs composing the CPS and the connected ports at the beginning of the simulation. It then launches the master algorithm, which takes all this information as input and performs step wise simulation. At the end of the simulation, the MST-engine is responsible for communicating results to the GUI to display results.

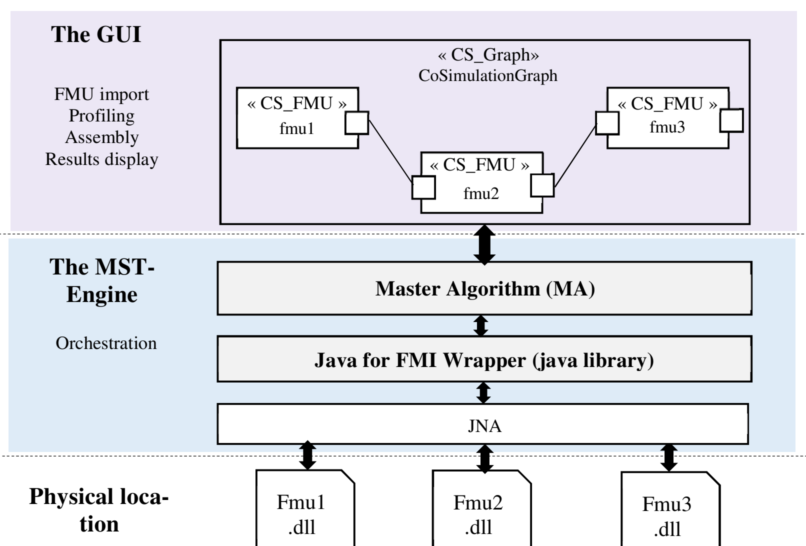

Figure 4-5. FMI-based Co-simulation from UML model to native code

The MA performs calls to dlls contained in the FMUs. A java wrapper for FMI API was implemented to enable calls to native code (i.e. the dll contained in the FMU) from the co-simulation

framework. The developed library is part of the proposed framework and uses the Java Native Access (JNA)[22], which is a community-developed library that provides Java programs easy access to native shared libraries. Figure 4-5 illustrates the co-simulation framework from the UML model connecting the imported FMUs to native calls to the dlls performed by the MA.

4.1.2.2. A wrapper for FMI

The framework implements the communication interface defined by the FMI standard, so called *Fmi2Library*. This interface provides a representation of all types and functions contained in the FMI standard API. The *Fmu2Proxy* implements the *Fmi2Library* interface as depicted in Figure 4-6. It performs calls to native code using the JNA library and represents the bridge between the framework and the dlls. Each class representing an imported FMU should be a specialization of the *FmiProxy*.

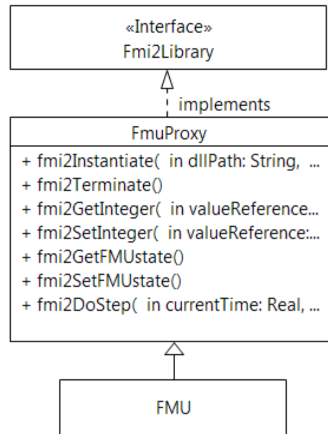

Figure 4-6. The FMI standard implementation

4.1.2.3. Master algorithm

The framework supports both basic and sophisticated MAs. We implemented the two preprocessing algorithms proposed in [12]. The first one is called "variable-order algorithm" depicted in Figure 4-7.

```
/* The algorithm parameters */
Inputs:
    P: the port mapping of the co-simulation graph
    D: the global dependency relation of the co-simulation graph
    U, Y: the set of ports in the co-simulation graph
Output:
    X (an ordered list of ports variables) or error
/*Algorithm*/
    ➢ Construct a directed graph G where the vertices are represented by the ports
      U,Y and the edges are constructed using P and D
    ➢ Perform a topological sort on G
          If a cycle is found return error
          Else return X
```

Figure 4-7. The Variable-order algorithm

[22] Refer to: https://github.com/java-native-access/jna

This algorithm is executed before the start of the simulation in order to detect cycles in the co-simulation model, in which case it doesn't perform simulation and alert the user. It builds a global inputs/outputs dependency graph based both on I/O dependency information contained in the XML files of the FMUs and on connections between FMUs in the co-simulation graph. This algorithm applies topological sort on this graph (the algorithm is given in Annex D). If no cycle is detected in the graph, then an ordered list of variables is generated. This list determines the order in which the information should be propagated from one FMU to another, that is, the order in which the MA should call *fmi2GetXXX* and *fmi2SetXXX* on ports at communication points.

The second is called the "master step" algorithm depicted in Figure 4-8. It determines the progress of the simulation in terms of time and state at each simulation step. If the FMUs support rollback, the MST-engine is able to go back in the time to remake a simulation step in case an FMU does not succeed to perform a simulation step.

```
/*Master step algorithm parameters*/
F: the set of FMUs in the co-simulation graph
P: the ports mapping
X: the ordered list of ports variables
hmaster: simulation step size
hmax: a default simulation step size
/*algorithm*/
  ➢ Set the simulation step size to the default one
        h_master=h_max
  ➢ Propagate data
        For each input u in X
            y = P(u);
            v = get_c(y);
            set_c(u,v);
        End for
  ➢ Save the states of all FMUs to enable rollback
  ➢ Find a simulation step size h acceptable by all FMUs
        For each c ∈ F:
            doStepc(h_master);
            If c is not able to perform the stepthen
                h= the last successful time of c;
                h_master=min(h_master,h);
            End if;
        End for;
  ➢ If (h_master<h_max) then restore the last state of the FMUs
  ➢ Remake the simulation step
        For each c ∈ F:
            doStepc(h_master);
        End for;
```

Figure 4-8. The master step algorithm

The resulting algorithm is depicted in Figure 4-10. It is well suited for co-simulation scenarios composed of FMUs which rely on CT MoC and support the rollback feature and expose their I/O dependencies. However, this algorithm is not suited for heterogeneous scenarios composed

of FMUs and UML components, since it does not account for the MoCs on which UML components rely. In particular, it does not allow zero-time step size and the detection of time events. Further advanced master algorithms will therefore be proposed in Chapters 5 and 6.

```
/*Co-simulation parameters*/
F: the set of FMUs in the co-simulation graph
P: the ports mapping
X: the ordered list of ports variables
tc: Current simulation time
tstart: Start simulation time
tstop: Stop simulation time
h_master: simulation step size
/*Instantiate and initialize components c ∈ C */
For each component c ∈ C:
        Instc();
        Initc(tstart, tstop);
/*Step wise simulation*/
Call the ‘variable-order’ algorithm;
While (tc<tstop)
        Call the ‘master-step’ algorithm;
        tc=tc+ h_master;
end while.
/* Termination of the simulation*/
For each component c ∈ C:
        terminatec() ;
end simulation
```

Figure 4-9. The basic master algorithm for FMUs orchestration enriched with rollback feature and co-simulation graph analysis

The master itself is expressed in UML: an active *Class*$^{(syn)}$, so called Master, whose classifier behavior is represented with a simplified *Activity*$^{(syn)}$ specifying the algorithm of the co-simulation, the so-called *MasterBehavior* depicted in Figure 4-9. An instance of this *Master* class is integrated in the *CoSimulationGraph* and is executed by relying on fUML and PSCS semantic model.

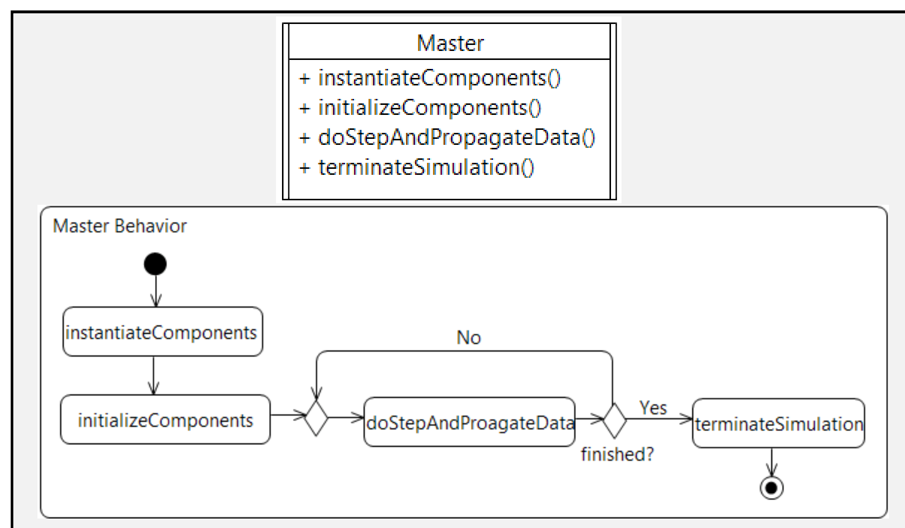

Figure 4-10. The Master algorithm expressed with UML

The *MasterBehavior* orchestrates the set of UML instances contained in the *CosimulationGraph* which represent instances of the imported FMUs. All parts contained in the C*osimulationGraph* are composite *Class*$^{(syn)}$. Therefore, in the locus, they are represented with a *CS_Object*$^{(sem)}$ or a specialization of it. As explained previously, the behavioral semantics of the imported FMUs are captured by the dlls they contain. The *instantiateVisitor()* shall be overridden in order to represent an FMU with an *FmuProxy* instance instead of an fUML *Object(sem)* at runtime.
The master is then able to perform the calls to external dlls using the feature F3. Thus, the activity nodes *instantiateComponents*, *initializeComponents*, *doStepAndPropagateData*, and *terminateSimulation* are *OpaqueBehavior*$^{(syn)}$ whose execution is dispatched to the execution of the corresponding method in the FMU instance. The extension of the fUML semantic model with this new visitor implies the definition of a new locus, so called *Cosim_Locus*$^{(sem)}$, as specialization of the *CS_Locus*$^{(sem)}$ as well as a new execution factory, so called *Cosim_Factory*$^{(sem)}$, as a specialization of the *CS_Factory*$^{(sem)}$ (Figure 4-11).

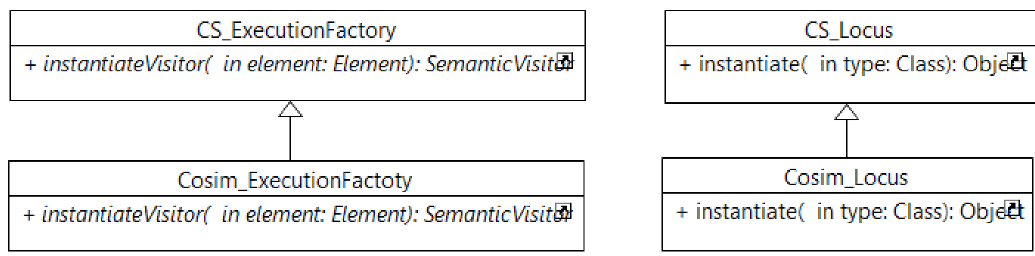

Figure 4-11. Extensions of the PSCS semantic model for FMI based co-simulation

The master is in charge of data propagation by calling fmi2GetXXX and fmi2SetXXX on FMUs. Calls to these routines update the values of the ports in the virtual memory of the FMUs but not in the UML *CosimulationGraph* itself. The update of the values in the CoSimulationGraph during the simulation procures the co-simulation environment with further capabilities such as simulation tracing. The setting and getting of features values in the fUML semantic model is performed by calling *setFeatureValue()* and *getFeatureValue()* respectively on the *Object*$^{(sem)}$ class. The *setFeatureValue()* operation takes as parameters the instance of the *Port*$^{(syn)}$, a *Value*$^{(syn)}$, and the position in which the value will be inserted. The getFeatureValue() operation takes as parameter the instance of the *Port*$^{(syn)}$ and the position from which the value will be get.

Table 4-2. Mapping between formalization functions and wrapper functions

Formalization functions	Wrapper functions
getc(y)	fmi2Getxxx(y) c.getFeatureValue(y,0) Where xxx is one of Real, Integer, Boolean and String
setc(u,v)	fmi2Setxxx(u,v) c.setFeatureValue(u,v,0) Where xxx is one of Real, Integer,e Boolean and String

4.2. Validation of the framework implementation

4.2.1. Use case: The TankPISystem

Control systems are an example of CPS. A traditional control problem is the control of the liquid level in a tank. The 'TankPI' is a common example conceived for that purpose. This example is a cyber physical system composed of a physical part (called Plant) and a cyber part (called controller) as illustrated in Figure 4-12.

The tank receives a liquid flow produced by a liquid source. These two model elements represent the physical part of the system. The controller (piContinuous) receives the indication of the level of the liquid in the tank from the level sensor and returns a control signal to an actuator. The control signal is computed based on that indication, the amount of liquid received at this moment, and the capacity of the tank. Based on that signal, the actuator will either activate a valve to let the water flow out of the tank or deactivate it.

Two FMUs for co-simulation are exported using the simulation Dymola (the 'TankPI_TanPIController.fmu' and 'TankPI_TankPIPlant.fmu'). Both FMUs rely on CT MoCs. This example is used for the validation of the FMI standard implementation in the framework we propose.

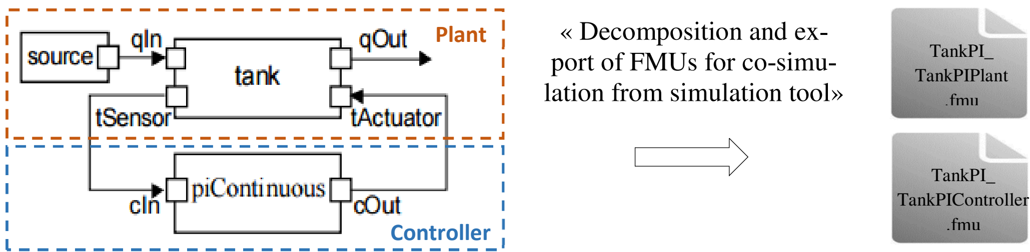

Figure 4-12. The tankPI system and its decomposition

4.2.2. Import of the FMUs and the definition of the co-simulation scenario

When imported, each FMU is transformed into an UML package. The package contains an UML *Class*$^{(syn)}$ representing the structure of the FMU (i.e, parameters, inputs and outputs) and a set of UML *Dependency*$^{(syn)}$ representing the potential dependencies of the outputs to the inputs of the FMU.

Each class is annotated with « CS_FMU» stereotype. The ports used for data propagation from one FMU to another are annotated with « CS_Port» stereotype to indicate to the MA the ports it should use for data propagation (i.e, ports which are not annotated with that stereotype should not be considered by the MA).

The 'TankPIController' FMU has an input port 'cIn' which indicates the level of the liquid in the tank and an output port 'cOut,' which represents the instruction of the controller to activate the actuator or not. The 'TankPIPlant' FMU receives the instruction of the controller on its input port 'tActuator' and outputs the level of the liquid in the tank on its output port 'tSensor'. It has another output port 'qOut' which indicates the amount of liquid flowing out of the tank.

Figure 4-13 illustrates the import of the FMU 'TankPI_TankController' in the proposed framework. It depicts the generated package as well as an extract of the properties of the stereotypes « CS_Port» and « CS_FMU».

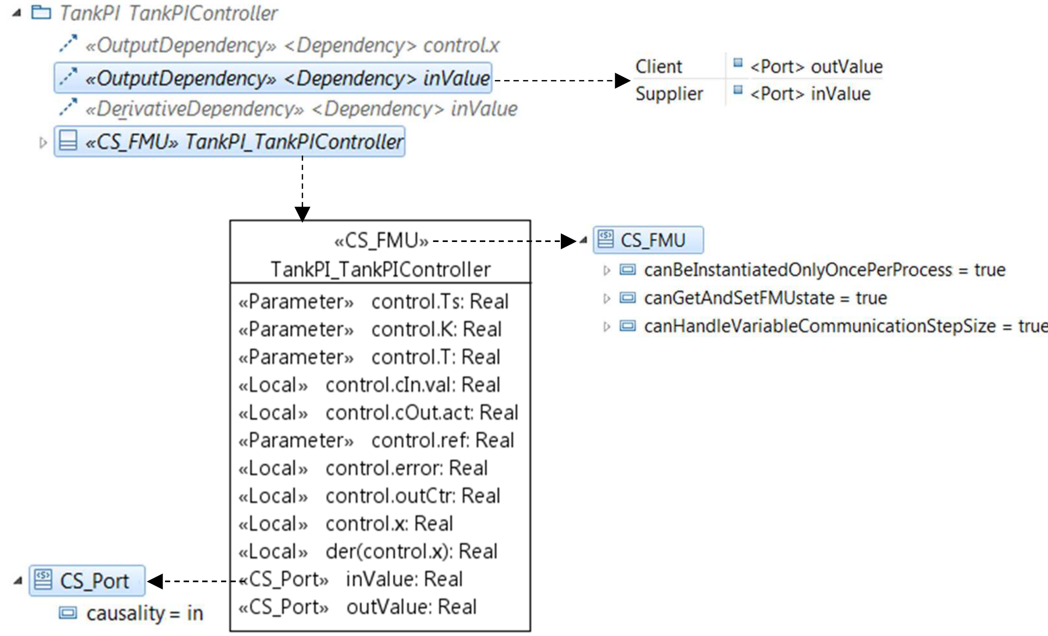

Figure 4-13. The import of the TankPI_TankPI_FMU in the framework

The imported FMUs are connected in the UML composite structure 'CosimulationGraph' (Figure 4-14). The simulation starts at t=0 and terminates at t=200 with a step size h=0.01. These parameters are indicated in the « CS_Graph» stereotype.

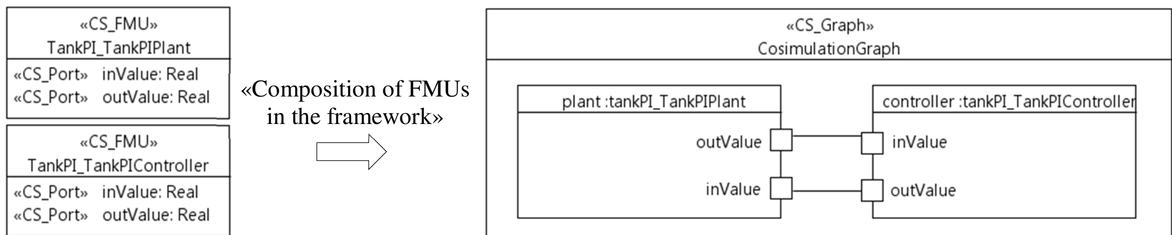

Figure 4-14. The definition of the co-simulation scenario of the TankPI system

4.2.3. Simulation results

The co-simulation scenario is simulated in simulation tool Dymola[23] (a simulation tool which provide an implementation of the FMI for co-simulation standard) as well as in the proposed framework. The simulation results in the proposed framework are the same that obtained in Dymola. This comparison confirms the validity of the implementation presented in this chapter.

Results of the co-simulation in Dymola as well as in the proposed framework are depicted respectively in Figure 4-15 and Figure 4-16. A demo is also available online.[24]

[23] Refer to: https://www.3ds.com/dymola/
[24] Refer to: https://www.youtube.com/demoFMICoSimulation

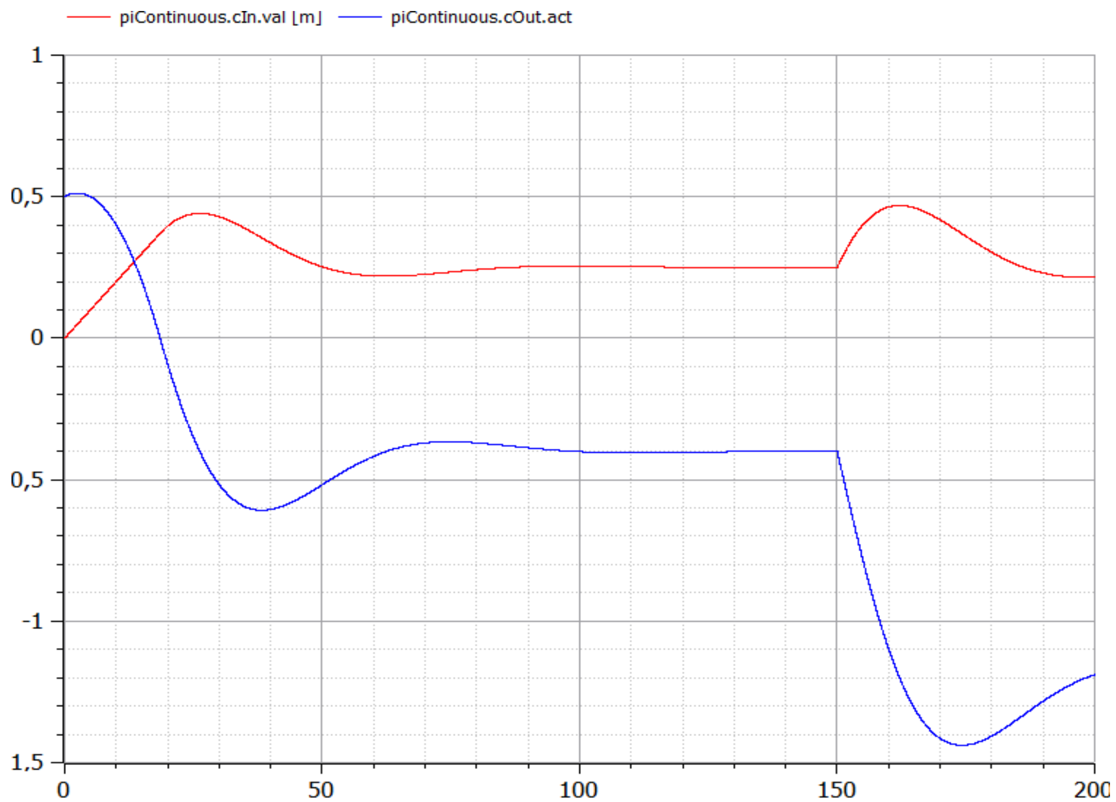

Figure 4-15. Execution results in the proposed Dymola

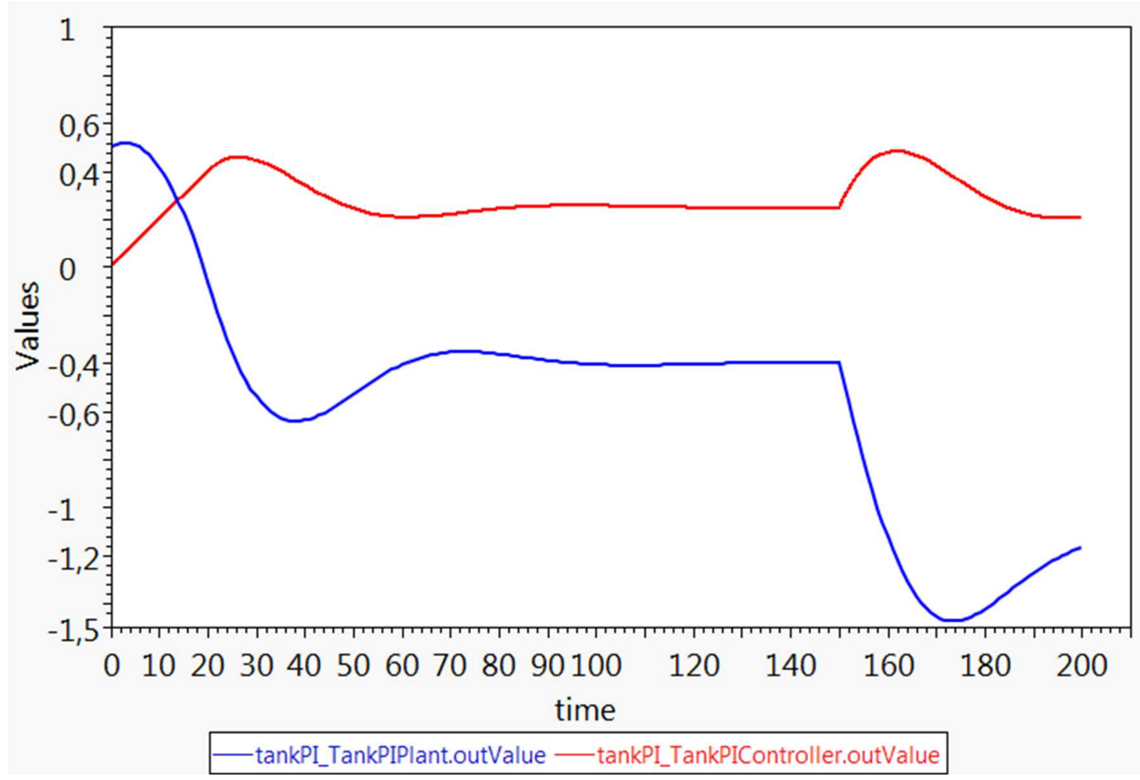

Figure 4-16. Execution results in the proposed framework

For instance, the framework offers FMI-based co-simulation of a set of imported FMUs (physical components). This is the basis for the integration of UML models (computational components) in FMI-based co-simulation. The extension of this framework for the support of UML models is the discussed in Chapters 5 and 6.

4.3. Conclusion

In this chapter, we introduced the framework we propose for FMI-based co-simulation of CPS. We listed features of each part of the framework, in particular the GUI and the MST-engine. The GUI is responsible for the definition of co-simulation scenarios using imported FMUs. The MST-engine is responsible for providing the MAs for FMUs' orchestration and the required libraries which enable it to perform calls to FMI API procedures.

For instance, we only provide an implementation of the FMI specification in an UML compliant tool. A CPS model connects a set of FMUs and their simulation is totally based on FMI.

In next chapters, this framework will be extended for integration of UML models in which case a CPS model connects FMUs representing the physical components with UML models, which represent the cyber part of the system.

Chapter 5: Integration of untimed UML models in FMI-based co-simulation

Outline

5.1. Untimed Models of Transformational Systems
5.1.1. Modeling rules for integration in FMI co-simulation
5.1.1.1. Model Structure and behavior
5.1.1.2. Applied stereotypes
5.1.2. Adapting fUML execution semantics to FMI API
5.1.2.1. Instantiation and initialization
5.1.2.2. Stepwise simulation and data propagation
5.1.2.3. Termination
5.1.3. Pseudocode of the master algorithm
5.1.4. Experience on a representative example
5.1.4.1. Definition of the simulation scenario
5.1.4.2. The simulation of the co-simulation scenario
5.2. Untimed models of reactive systems
5.2.1. Modeling rules for integration in FMI co-simulation
5.2.1.1. Model Structure and behavior
5.2.1.2. Applied stereotypes
5.2.2. Extension of fUML semantics
5.2.3. Adapting fUML execution semantics to FMI API
5.2.3.1. Instantiation and initialization
5.2.3.2. Stepwise simulation and data propagation
5.2.3.3. Termination
5.2.4. Pseudocode of the master algorithm
5.2.5. Experience on a representative example
5.2.5.1. Definition of the simulation scenario
5.2.5.2. The simulation of the co-simulation scenario
5.3. Conclusion

This chapter deals with the integration of untimed UML models in FMI-based co-simulation approaches. It is organized into two main parts: the first one focuses on untimed UML models of transformational systems, and the second focuses on untimed UML models of systems reactive to environmental changes. For each kind of systems, we define a set of modeling rules and potential extensions to fUML, in the case where fUML does not cover all UML syntactic elements. Then we map untimed semantics of fUML to timed semantics of FMI. This mapping enables the MST-engine to coordinate the imported FMUs and the untimed UML models by adapting its behavior to the kind of the model it simulates. Section 5.1 and section 5.2 deal with untimed UML models' integration, respectively of transformational systems and reactive systems.

5.1. Untimed Models of Transformational Systems

5.1.1. Modeling rules for integration in FMI co-simulation

5.1.1.1. Model structure and behavior

A transformational system model must specify the logical structure of the input and output data, and the algorithm that computes the transformation. We refer to transformational systems properties to define a set of requirements that the UML model should satisfy. Then, we refer to UML specification to identify the appropriate set of UML syntactic elements for transformational systems modeling. We try to remain as close as possible to fUML* (i.e, the subset of UML elements for which an execution semantics is defined) in order to minimize the efforts related to the extension of fUML* with new semantics.
Table 5-1 depicts a mapping between the model requirements and syntactic UML elements based on the system properties.

Table 5-1. Mapping of transformational systems properties to UML modeling concepts

System property	Model requirement	UML concept
P1: Activation A transformational system runs computations only when asked by another component.	R1: The behavior of the class representing the system shall be called to execute some computations.	C1: Passive *Class*$^{(syn)}$ with at least one *Operation*$^{(syn)}$ and no *Property*$^{(syn)}$
P2: Behavior A transformational system's computations are not interrupt-driven, and are usually sequential and terminating.	R2: The behavior of the class representing the system shall specify a sequence of actions (i.e, corresponding to computations), and shall not wait for any event to occur and should terminate.	C2: *Activity*$^{(syn)}$ as implementation of an operation
P3: Output/input relationship A transformational system's output is defined in terms of its input.	R3: The structure of the class representing the system shall provide interfaces through which it accepts and outputs data,	C3: *Port*$^{(syn)}$ C4: *ReadStructuredFeatureAction*$^{(syn)}$ and *AddStructuredFeatureAction*$^{(syn)}$
	R4: The output depends on the input.	C5: UML directed *Dependency*$^{(syn)}$

The rest of this section gives a review about the aforementioned UML concepts as defined in UML specification [30] and argues the choice of this set of syntactic elements for transformational systems modeling.

An instance of a passive class executes within the context of some other object, that is, it waits for another object to call it. A passive class has a behavior defined by its operations *(C1)*. Therefore, the behavior of a passive class only starts when one of its operations is invoked and terminates when this operation returns. fUML* restrict the behaviors modeling to activities, which may describe procedural computation *(C2)*.
The structure and the behavior of the environment are not particularly important for transformational systems. However, an interface is required to acquire sufficient information to produce the output. *Port*$^{(syn)}$ *(C3)* provides a way to model interfaces of an object through which it receives input data and produces output data.

Note that a transformational system is stateless. As a result, a UML model of a transformational system must not have properties (*Property*$^{(syn)}$) other than ports.
ReadStructuralFeatureAction$^{(syn)}$ and *AddStructuralFeatureAction*$^{(syn)}$ *(C4)* are actions which enable it to respectively retrieve and add values from/to a structural feature. Therefore, an activity that specifies the behavior of a transformational system calls *ReadStructuralFeatureAction*$^{(syn)}$ on the input port to retrieve the input value which will be passed as a parameter to a sequential flow of actions (i.e, the computations) and finally calls *addStructuralFeatureAction*$^{(syn)}$ to write the result of the transformation on the output port. Information related to the dependency between the output and the input is particularly important to model. It is used by the MST-engine to analyze the co-simulation model, and then to determine the order and the instants at which it should propagate data from one component to another. That means that the output port requires the input port for its specification. This kind of relationship is expressed in UML with *Dependency*$^{(syn)}$ *(C5)*, which proposes a way to express a supplier - client relationship between model elements. The supplier provides something to the client, and thus the client (i.e, the output port) is in some sense incomplete while being dependent on the supplier (i.e, the input port).

fUML defines precise execution semantics to concepts *C1, C2,* and *C4*. *Port*$^{(syn)}$ (C3) does not belong to the fUML* subset but a little extension to the fUML* semantic model was implemented. fUML* are especially appropriate for the execution of untimed sequential behavior which fits the data flow MoC. The concept *C5* is used only to provide extra static information related to the structure of the model, that is, no semantics are required to be defined for this element. Therefore, the fUML semantic model can be used as it is for simulation of untimed models of transformational systems.

UML models are annotated with new stereotypes in order to enable the MST-engine to recognize, in particular, the MoC on which the component it relies and then to adapt its behavior. The following section identifies important information that an untimed model of a transformational system need to expose and, extends the co-simulation profile (presented in section 4.1.1.1) by adding new stereotypes to the co-simulation profile.

5.1.1.2. Applied stereotypes

This section enumerates the stereotypes we will apply to an untimed UML model of a transformational system in order to expose important information to the MST-engine. Table 5-2 illustrates these stereotypes and their semantics. Important information of an untimed model of

transformational system concerns, the MoC on which it relies, the computations to execute and the direction of ports and their dependencies. The new stereotypes extend the co-simulation profile we presented in Chapter 4 where 'CS' stands for co-simulation.

Table 5-2. Stereotypes to apply for an untimed model of transformational systems

Stereotype	UML meta-class	Semantics
« CS_Untimed »	*Class$^{(syn)}$*	Used to indicate that the class has no semantics of time.
« CS_Port »	*Port$^{(syn)}$*	Identifies ports which should be considered by the MST-engine for data propagation.
« CS_Operation »	*Operation$^{(syn)}$*	Applied to one, and only one, operation of the model. It enables the MST-engine to identify the operation to invoke during a simulation step (the operation specifying the behavior of the system).
« CS_Dependency »	*Dependency$^{(syn)}$*	Used to identify UML elements that specify I/O dependencies between an outputs and inputs of a component.

Figure 5-1 depicts an example of an untimed model of a transformational system.

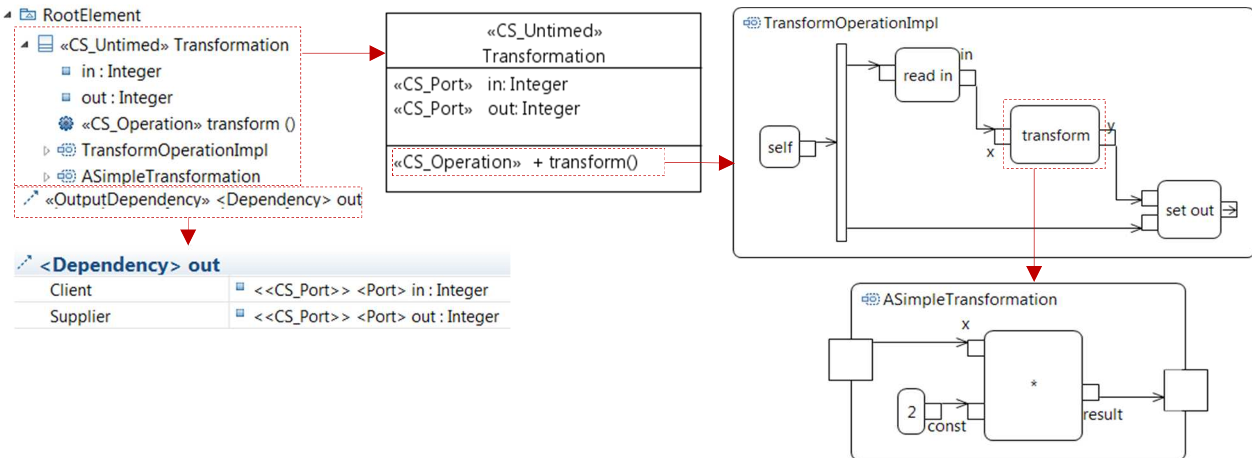

Figure 5-1. An untimed model of a transformational system: structure and behavior

The '*Transformation'* class is a passive *Class$^{(syn)}$* that represents the transformational system structure. The class owns an input port '*in*', an output port '*out'* and an operation '*transform()*'. This operation is the one that represents the computations performed by the system and that should be called by the MST-engine during a simulation step. The activity *'TransformOperationImpl'* is the specification of the operation '*transform()'*. It is composed of three action

nodes: *'read in'*, *'transform'*, and *'set out'*. The *'read in'* and *'set out'* nodes are, respectively, a *ReadStructuredFeatureAction^(syn)^* on the input port and an *AddStructuredFeatureAction^(syn)^* on the output port of the '*Transformation'* class which enable to get/set the values from/to ports. The *'transform'* node is a *CallBehaviorAction(syn),* which is a specific action that invokes another behavior. The invoked behavior is specified by the activity *'ASimpleTransformation'*. It takes an integer as input, computes a multiplication by two and returns the result. The choice to encapsulate the computations of the system in a *CallBehaviorAction^(syn)^* ensures that there is only one computation node in the activity that implements the operation and thus, simplifies the activity and makes it usable as a sample for future examples. This model will be used later in this chapter in a representative example (in section 5.1.4) for the co-simulation of an untimed UML model of a transformational system with an imported FMU.

5.1.2. Adapting fUML* execution semantics to FMI API

The execution of untimed UML models for transformational systems relies on the Data Flow MoC. Their execution is purely causal and do not consider any notion of simulated time as state in section 1.1.2.1. These semantics are covered by fUML*. The execution of the FMUs, on the contrary, continuously depends on time. This difference raises an untimed vs timed semantics issue.

This section focuses on that issue and proposes adaptation of semantics between execution semantics of UML models defined by fUML* and semantics of FMI. It is organized into three subsections following the formalization of co-simulation given in section 2.1.1.2. It refers to the semantic model of UML models introduced in Annex A and gives equivalent routines in the fUML execution semantics for each function defined in the formalization.

5.1.2.1. Instantiation and initialization

According to fUML specification, when a simulation is launched, the instances of the elements in the UML model are automatically created, initialized and then placed in the locus. The instantiation result is the instance of the class '*Transformation'* and the instances of all its features (i.e, the input port, the output ports, and the operation '*transform ()'*) in the locus.

No time semantics[2] are supported in the UML model. Thus, the initialization only sets the features values with default values in the model.

Below in Table 5-3 are given the equivalent routines in the fUML semantic model which correspond to the instantiation and initialization functions in the FMI API:

Table 5-3. fUML routine for instantiation and initialization of an untimed model of a transformational system

Formalization functions	fUML or PSCS semantic model
instc()	c.locus.instantiate()
initc()	Features values are automatically initialized with values in the model during their instantiation.

5.1.2.2. Stepwise simulation and data propagation

The behavior of a passive class is not automatically started after its instantiation. At each simulation step, the MST-engine launches the activity implementing the operation which defines

the computations of the transformational system. It first calls the *dispatch()* operation of the *Object*$^{(sem)}$ class, which dispatches the given operation to a method execution. Then by calling the *execute()* operation of the *ActivityExecution()* class, the dispatch operation takes as parameter the operation annotated with the « CS_Operation » stereotype. The activity of a passive object does not wait for any event to occur to propagate control from one activity node to another. Therefore, once launched, the whole activity is executed. The *doStep*$_c$*()* function corresponds to the execution of the whole activity associated with the operation.

At the end of a simulation step, the MST-engine propagates data from one port to another in the co-simulation model and synchronize the different components. The MST-engine considers the I/O dependency that exists between the output port and the input port of a transformational system. This dependency means that the input must be set before the computation of the output. The preprocessing algorithm *"variable-order algorithm"*, presented in section *4.1.2.3,* is executed before the beginning of the simulation. It takes into consideration the I/O dependencies of the UML component and generates the order in which the outputs/inputs of the co-simulation model should be get/set. The question is: at which instant must the data be propagated to the environment of such untimed models? The response is directly related to the step size used for the execution of these models.

Consider a simulation scenario connecting a set of FMUs with one or more untimed UML models of transformational systems. Let h_{FMU} be the step size of the FMUs, h_{UML} be the step size of the UML components, and h_{MATSER} be the step size chosen by the master to perform the co-simulation.

The absence of time makes the situation ambiguous. Two possibilities are identified:

- *The computations are assumed to take time to execute*. Since the model does not provide exact information, one can assume that the computations run for a time equal to the step size used for the FMUs. In this case, the simulation step size for the UML component is the same as for the FMUs $h_{UML}=h_{FMU}$. For a simulation step starting at tc, all components are simulated with the same step size $h_{Master}=h_{FMU}$ and data are propagated at $t_C+ h_{Master}= t_C+h_{FMU}$. It corresponds to the traditional behavior of a master algorithm. This alternative is unsatisfactory when the computations run for a longer or a shorter duration than h_{FMU}.

  Let d>=0 be the duration taken by the computations in reality. For a simulation step of size h_{FMU} starting at tc, three situations are possible:
  - o $d=h_{FMU}$: this is the ideal situation, the outputs are computed and propagated at $t_C+d = t_C+h_{FMU}$.
  - o $d>h_{FMU}$: the outputs are actually produced at t_C+d, and therefore at $t_c+h_{FMU}<t_c+d$ the outputs are supposed to be absent, which is not respected when using a step size $h_{FMU}<d$ and may considerably affect the reliability of the simulation results.
  - o $d<h_{FMU}$: the outputs are produced at t_c+d and propagated at $t_c+h_{FMU}>t_c+d$, therefore data are propagated with a delay of $h_{FMU}-d$, which also affect the accuracy of the simulation results.
- *The computations are assumed to take zero time to execute*. The output of the UML component should be, therefore, available and propagated in the co-simulation model at t_C. As a result, each time an UML component is met, the master sets the value of its

input port, executes a $doStep_c()$ of size $h_{UML}=0$ on the component, and gets the value of its output before executing and advancing time in the FMUs.
The master adapts the simulation step size to the kind of the component being simulated, that is, $h_{MASTER}=h_{FMU}$ for the FMUs and $h_{MASTER}=h_{UML}=0$ for UML components. In addition, UML models are simulated at the beginning of a simulation step before the FMUs. This alternative is not satisfactory because the duration of the computations is not considered. However, this choice covers the case where the duration of the computations is almost equal to zero.

In both cases, the impact of the choice on the simulation results and on the decisions to make depends on the properties of the system under design. For the rest of this section, we will consider the second alternative. First because it covers a particular use case (i.e. the instantaneous transformational systems), and second because of the direct dependency which exists between the outputs and the inputs of the component.

Below in Table 5-4 are given the equivalent routines in the fUML* semantic model which correspond to the stepwise simulation and data propagation functions in the FMI API.

Table 5-4. fUML routines for stepwise simulation and data propagation of an untimed model of a transformational system

Formalization functions	fUML or PSCS semantic model routines
$doStep_c(0)$	c.dispatch(operationToExecute).execute();
$set_c(inPort,value)$	c.setFeatureValue(inPort,value)
$get_c(outPort)$	c.getFeatureValue(inPort)

5.1.2.3.Termination

The termination of an *Object*$^{(sem)}$ corresponds to the termination of its behavior. The execution of an activity representing an *Operation*$^{(sem)}$ is automatically terminated when the result is returned.

Below in Table 5-5 is given the equivalent routine in the fUML semantic model which corresponds to the termibation function in the FMI API:

Table 5-5. fUML routines for termination of an untimed model of a transformational system

Formalization functions	fUML or PSCS semantic model
$terminate_c()$	do nothing

5.1.3. Pseudocode of the master algorithm

In order to account for these adaptations, we need to enrich the MST-engine with a new master algorithm that orchestrates a set of imported FMUs with one or more untimed UML models for transformational systems. The pseudocode of this latter is given in Figure 5-2. This algorithm can be enriched with the rollback functionality as done in the master algorithm orchestrating a set of FMUs. In fact, the transformational systems are memoryless. The produced outputs only depend on the current inputs. Therefore, in case where the master needs to remake a simulation

step, this only requires that all FMUs support rollback capabilities. The master can therefore save and then restore the previous states of the FMUs when needed. For the UML components, there is no particular computations to perform since no notion of system state exists.

```
/*Co-simulation parameters*/
tc: Current simulation time
tstart: Start simulation time
tstop: Stop simulation time
hFMU: simulation step size of the FMU
hUML=0 : simulation step size of an untimed UML component
X: U∪Y : set of ordered ports variables computed by "Variables-order" algorithm
C= F ∪ UU where UU: the set of untimed UML classes
/*Assumptions*/
On the co-simulation graph: no cycles exist in the co-simulation model, the co-simulation graph
is composed of a set of FMUs and one or more untimed UML models
On UML components: zero step size is allowed
On the FMUs: no particular assumptions
/*Instantiate and initialize components c ∈ C */
for each component c ∈ C:
        instc();
        if c ∈ F
                initc(tstart, tstop);
        end if;
/*Step wise simulation*/
while (tc<tstop)
        for each input u in X
                y = P(u);     //returns the output 'y' linked to the input 'u' in the co-simulation graph
                v = getc(y);
                setc(u,v);
                if c ∈ UU     //c is the one which has u as input
                        doStepc(hUML);
                end if;
        end for
        for each c in F
                doStepc(hFMU);
        end for;
        tc=tc+ hFMU;
end while.
/* Termination of the simulation*/
for each component c ∈ C:
        terminatec() ;
end simulation
```

Figure 5-2. Pseudocode of a master algorithm for a Co-simulation graph connecting FMUs with untimed UML model of a transformational system.

5.1.4. Experience on a representative example

5.1.4.1.Definition of the simulation scenario

The representative example consists of the definition and the simulation of a co-simulation scenario composed of an FMU, so-called 'Inc' provided by the FMU SDK[25], and an untimed UML

[25] www.qtronic.de/fmusdk

model of a transformational system, the so-called '*Transformation*' represented in section 5.1.1.2 (Figure 5-3).
The part '*inc*' in the figure below represents the FMU 'Inc.' It has no inputs, and only one integer output 'counter.' It increments the counter at each instant 't+*n*' where n∈N and uses a step of size '*h=0.5*'. The counter is therefore incremented after two calls to '*doStep()*' on the '*Inc*' FMU. The value of the 'counter' is propagated to the input 'in' of the 'untimedTransformation' component.

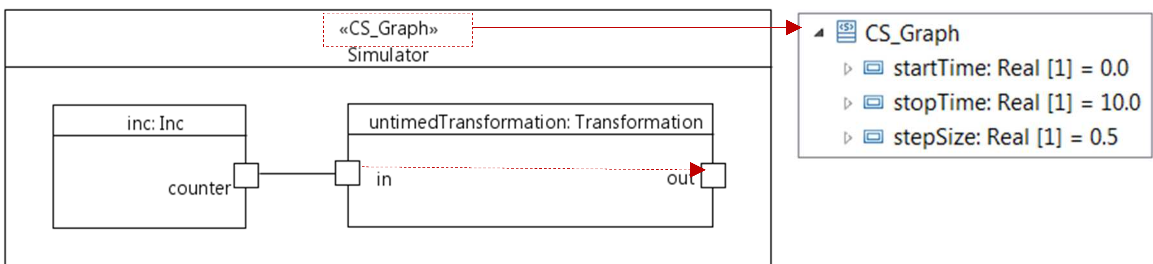

Figure 5-3. Co-simulation graph connecting an imported FMU to an untimed model of a transformational system

The 'untimedTransformation' component in Figure 5-3. Co-simulation graph connecting an imported FMU to an untimed model of a transformational system represents the class 'Transformation' defined in section 5.1.1.2. The red arrow indicates the direct dependency of the output port *'out' to* the input port *'in'* of '*Transformation*' class. The expected behavior of the '*untimedTransformation*' component is that it will output a result at the same time it receives values on its input port, that is, the result must be computed and output before the time advances in the '*inc*' component.

The co-simulation scenario will be simulated from 't_{start}=0' to 't_{stop}=10' with a default simulation step size 'h_{master}=0,5' as indicated in the 'CS_Graph' stereotype. The simulation results are given in the next subsection.

5.1.4.2.Simulation of the co-simulation scenario

Figure 5-4 and Figure 5-5 depict simulation results of the co-simulation scenario defined in the previous subsection using, respectively, the basic master (defined in the FMI standard in section 2.1.2) and the advanced master algorithm proposed in section 5.1.3.

Using the basic master algorithm given in section 2.1.2 (where the simulation step size h_{master}=h_{FMU}>0 for all components), we notice a delay between the instant at which the inputs arrive to the 'untimedTransformation' component (t_{in}=2) and the instant at which this latter produces the corresponding output (t_{out}=2,5) (Figure 5-4). This delay is introduced at all simulation steps from the beginning to the end of the simulation.

Figure 5-5 demonstrates that this delay does not exist when using the master algorithm we proposed in section 5.1.3. The outputs are in fact produced at the same time as the inputs' arrival (t_{in}=t_{out}) from the beginning to the end of the simulation which corresponds to the expected behavior.

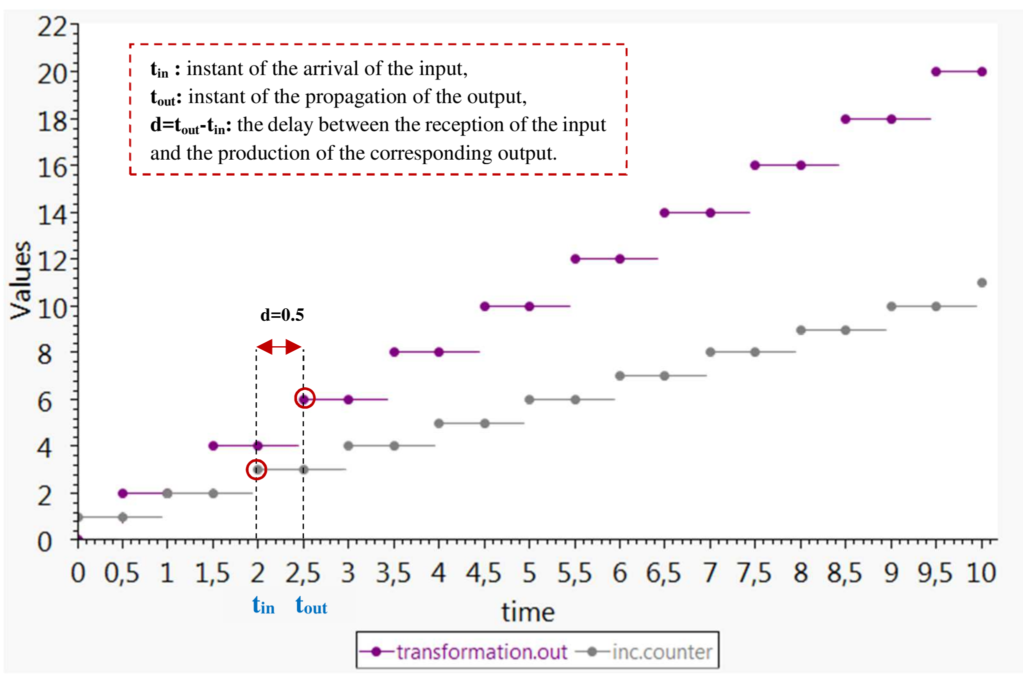

Figure 5-4. Co-simulation results of an untimed model of a transformational system-Basic master

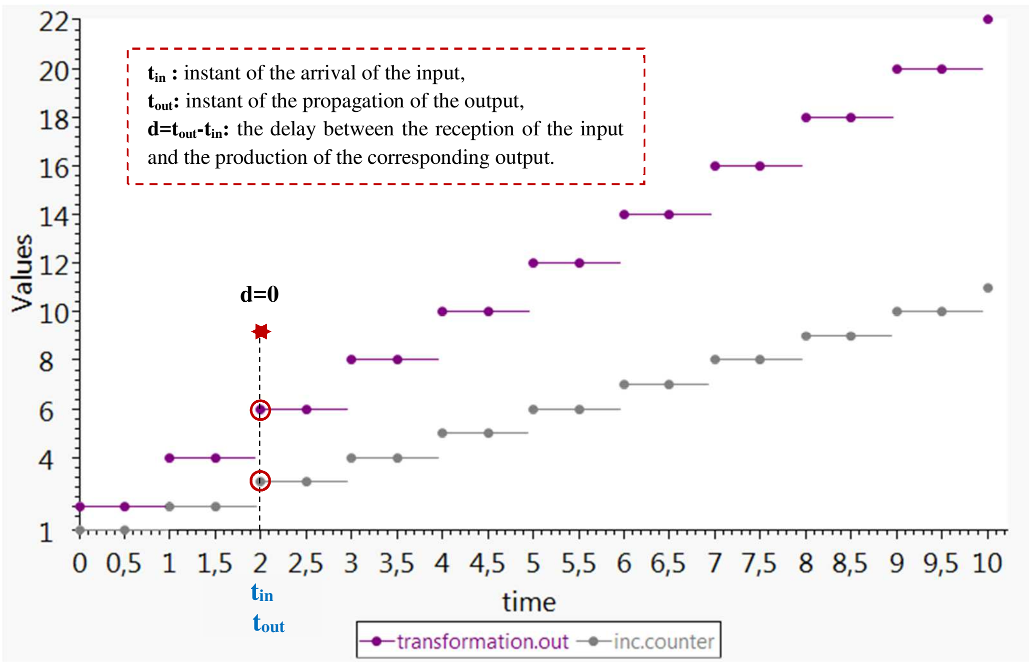

Figure 5-5. Co-simulation results of an untimed model of a transformational system-Advanced master

5.2. Untimed models of reactive systems

5.2.1. Modeling rules for integration in FMI co-simulation

5.2.1.1.Model structure and behaviors

A model of a reactive system must contain, in addition to the logical behavior of the system, the elements that ensure interaction with the environment. By contrast to transformational systems, reactive systems are in continuous interaction with the environment. Therefore, they always wait for some inputs to which they output a reaction.

Similar to what we do for transformational systems modeling with UML, we refer to reactive systems properties to define a set of requirements that the UML model should satisfy. Then, we refer to UML specification to identify the appropriate set of UML syntactic elements for modeling of reactive systems.

Table 5-6 depicts a mapping between the model requirements and syntactic UML elements based on reactive systems properties.

Table 5-6.Mapping of reactive systems properties to UML modeling concepts

System property	Model requirement	UML concept
P1: Activation A reactive system constantly interacts with its environment.	R1: The behavior of class representing the system shall be active from the beginning of the simulation and should wait for some stimuli on its input port to continue processing.	C6: Active *Class*$^{(syn)}$ with a classifier behavior.
P2: Behavior Reactive system computations are interrupt driven and are usually non-terminating. The system should react to external stimuli and produce the correct actions to the environment.	R2: The behavior of the class shall specify a stimuli/response behavior.	C2: *Activity*$^{(syn)}$ representing the classifier behavior C7: *AcceptEventAction*$^{(syn)}$ triggered by a *ChangeEvent*$^{(syn)}$.
P3: Input/output relationship Reactive system outputs depend on the external stimuli it receives as well as on the current state of the system.	R3: The structure of the object shall provide interfaces through which it accepts and outputs data.	C3: *Port*$^{(syn)}$ C4: *ReadStructuredFeatureAction*$^{(syn)}$ and *AddStructuredFeatureAction*$^{(syn)}$.

The rest of this section serves as a reminder about the UML concepts C6 and C7 as defined in UML specification [30] and argues the choice of these syntactic UML elements for reactive systems modeling.

An instance of an active *Class*$^{(syn)}$ is an object that begins to execute its behavior as soon as it is created and does not cease until either the complete behavior is executed or the object is terminated by some external object. The points at which an active object responds to communications received on its ports is determined by its behavior and not by the invoking object. An active object has a behavior defined as its classifier behavior.
The designer needs techniques for specifying stimulus-reaction behavior. Reactive behaviors modeling is handled by a specific UML action, the *AcceptEventAction*$^{(syn)}$. This action is introduced in UML to handle the processing of events during the execution of a behavior. It waits for the occurrence of an event meeting a specified condition to continue execution. In particular, the occurrence of a *ChangeEvent*$^{(syn)}$ is based on some condition becoming true, which meets the semantics of stimuli. An *AcceptEventAction*$^{(syn)}$ triggered by a *ChangeEvent*$^{(syn)}$ is then well suited for the modeling of stimuli-response behavior of a reactive system.
fUML defines the execution semantics of behaviors designed with activities and associated to active classes as a classifier behavior *(C6),* but not for the change events. An extension to fUML is then required and will be detailed in section 5.2.2.

Similar to imported FMUs and untimed models of transformational systems, an untimed model of a reactive system must expose information about its MoC. Section 5.2.1.2 outlines the set of stereotypes we will apply to an untimed model of a reactive system.

5.2.1.2.Applied stereotypes

Table 5-7 illustrates the stereotypes used for an untimed UML model of a reactive system and gives the semantics of each of them. These stereotypes have been already introduced in section 4.1.1.1.

Table 5-7. Stereotypes to apply for an untimed model of reactive systems

Stereotype	UML meta-class	Semantics
« CS_Untimed »	*Class*$^{(syn)}$	Indicates that the class has no semantics of time.
« CS_Port »	*Port*$^{(syn)}$	Identifies ports which must be considered by the MST-engine for data propagation.

A simple example of an untimed reactive system is depicted in Figure 5-6.*'Controller'* is an active *Class*$^{(syn)}$ that represents the reactive system structure. The class owns an input port *'in'*, an output port *'out'* and a classifier behavior *'ControllerClassifierBehavior'*. The classifier behavior describes the dynamics of a system. It controls the value of the input to produce some effect on its environment. It is specified with the activity '*ControllerClassifierBehaviorImpl'*. This activity is composed of four action nodes: 'change on in', 'read in', 'control' and 'set out'. The activity first waits for a change in the input port value represented by the '*change on in'* node which is an *AcceptEventAction*$^{(syn)}$ triggered by a *ChangeEvent*$^{(syn)}$ related to the port 'in'. Once a change is detected, the control is propagated in the activity. The *'read in'* and *'set out'*

nodes respectively, are a *ReadStructuredFeatureAction*$^{(syn)}$ on the input port and an *AddStruc-*

Figure 5-6. An untimed UML model of a reactive system: structure and behavior

turedFeatureAction$^{(syn)}$ on the output port of the '*Transformation*' class which enable to get/set the values from/to ports. The *'control'* action node is a *CallBehaviorAction*$^{(syn)}$. This specific action is used to simplify the activity and make it usable as a basis for future examples. The behavior called by this action is specified by the activity *'ASimpleControl'*. This takes an integer as input, compares it to a threshold value and returns a boolean result. This model will be used later in this chapter in a representative example in section 5.2.5 for the co-simulation of an untimed UML model of a reactive system with an imported FMU.

5.2.2. Extension of fUML semantics

fUML considers the *AcceptEventAction*$^{(syn)}$ triggered by a *SignalEvent*$^{(syn)}$ but not one triggered by a *ChangeEvent*$^{(syn)}$. The fUML should be extended in both syntax and semantics. The syntax subset should be enriched by the syntactic element *ChangeEvent*$^{(syn)}$. The semantic model should then provide semantic elements to capture the semantics brought by this new syntactic element. Following the visitor pattern explained in Chapter 3, this implies:

- The introduction of a new visitor *ChangeEventOccurence*$^{(sem)}$ as a specialization of the *EventOccurence*$^{(sem)}$ class in the fUML semantic model. It represents a single occurrence of a *ChangeEvent*$^{(syn)}$. The change event occurrences are placed in the event pool of the *ObjectActivation*$^{(sem)}$.
  The occurrence of a change event is based on some expression becoming true. The expression is checked continuously or at specific instants so that each time the value of the expression changes from false to true, a change event is generated. As stated in the UML specification: **"***It is a semantic variation when the change expression is evaluated. For example, the change expression may be continuously evaluated until it becomes true. It is further a semantic variation whether a change event remains until it is consumed, even if the change expression changes to false after a change event."* [30]

In the context of FMI, we are interested in detecting a change on the value of a specific input port. We, therefore, propose to evaluate the condition when values of input ports are set by the MSt-engine. The *ChangeEventOccurence*$^{(sem)}$ concerns a specific feature represented by *changedProperty* attribute. A change event is generated when the new value (*newValue)* of the feature changes compared to the former one (*formerValue*).

- The definition of a new visitor *AcceptChangeEventActionEventAccepter*$^{(sem)}$ as a specialization of *AcceptEventActionEventAccepter*$^{(sem)}$. This event Accepter handles events reception on behalf of a specific accept event action activation waiting for a change event occurrence.
- The definition of a new visitor *AcceptChangeEventActionActivation*$^{(sem)}$ as a specialization of *CS_AcceptEventActionActivation*$^{(sem)}$ together with a new execution factory *U4CosimExecutionFactory*$^{(sem)}$ as a specialization of *CosimExecutionFactory*$^{(sem)}$. The *instantiateVisitor()* operation is overridden to take into account the *AcceptChangeEventActionActivation*$^{(sem)}$ visitor. The latter is associated with the *AcceptChangeEventActionEventAccepter*$^{(sem)}$. The operations *match()* and *accept()* for change event instances should behave as it does for signal event instances, except that the event should correspond to a *ChangeEventOccurrence*$^{(syn)}$ instead of a *SignalEventOccurrence*$^{(syn)}$.

Figure 5-7 depicts a class diagram of the new semantic elements required to capture the semantics of an accept event action triggered by a change event.

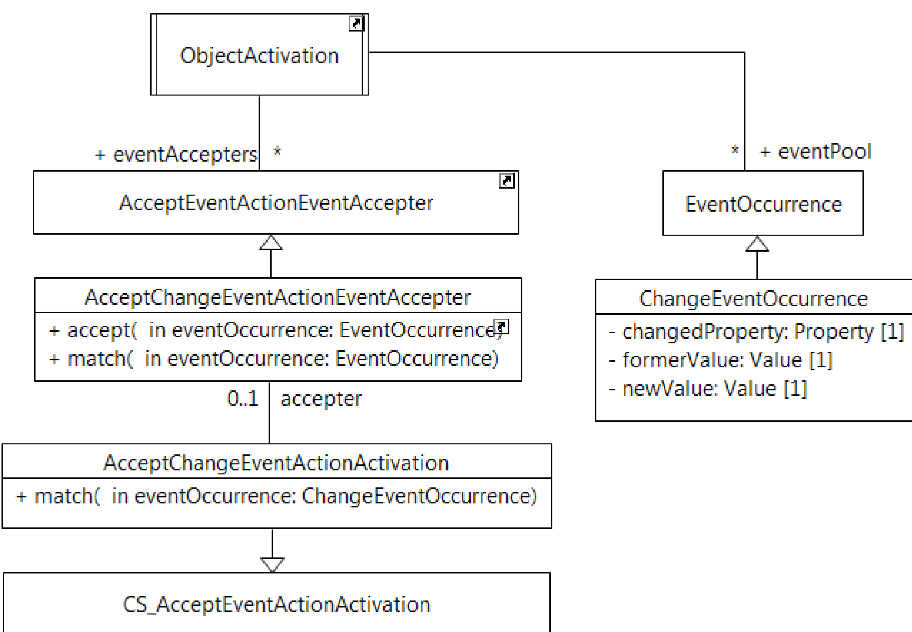

Figure 5-7. Semantic elements which capture semantics of change events

5.2.3. Adapting fUML execution semantics to FMI API

The execution semantics of untimed UML models for reactive systems corresponds to the execution cycle of reactive synchronous MoC introduced in Chapter 1. The behavior is activated at a discrete set of instants which corresponds to events arrival instants. No notion of simulated time exists. Thanks to the extension introduced in the previous section, this MoC is now sup-

ported. The execution of the FMUs, on the contrary, continuously depends on time. This difference raises an untimed vs timed semantics issue, as well as a discrete event vs continuous time issue.
This section focuses on these issues and proposes adaptation of semantics between execution semantics of UML models and semantics of FMI. It is organized into three subsections following the formalization of co-simulation. It refers to the extended semantic model introduced in the previous section and gives equivalent routines for each function defined in the formalization.

5.2.3.1.Instantiation and initialization

When the simulation is launched, the instances of the elements in the UML model are automatically created, initialized and then placed in the locus. The instantiation result in the locus of the model depicted in Figure 5-6 is an instance of the class *'Controller'* represented with an *Object$^{(sem)}$*, a representation of all its features and its classifier behavior.
The classifier behavior is started as soon as it is created. However, in FMI context, the behavior should start when the simulation effectively starts (i.e. at the first call of the *doStep()* function). The *ObjectActivation$^{(sem)}$* should be created but not started at this stage. Therefore, the *startBehavior()* operation of the *Object$^{(sem)}$* class must be overridden to support these new semantics.
Table 5-8 illustrates the equivalent routines in the fUML semantic model which correspond to the instantiation and initialization functions in the FMI API.

Table 5-8. fUML routines for instantiation and initialization of an untimed model of a reactive system

Formalization functions	fUML or PSCS semantic model
$inst_c()$	c.locus.instantiate();
$init_c()$	Features values are automatically initialized with values in the model during instantiation

5.2.3.2.Stepwise simulation and data propagation

As stated in chapter 3, the execution of active classes relies on Run-To-Completion semantics. When an event is dispatched from the event pool by calling the *dispatchNextEvent()* operation, the *match()* operation checks whether or not an *AcceptChangeEventActionEventAccepter$^{(sem)}$* waiting for that event exists. If so, the *accept()* operation propagates the control as far as possible until encountering a new blocking node. In the example given in Figure 5-6, the classifier behavior is supposed to wait for a change on the input port value. Once a change is detected, the control is propagated in the activity and, then, the object returns to wait for a future change on the input port value.
The absence of time information leads to an ambiguous situation regarding the instant at which the data must be propagated in the co-simulation model. Similar to an untimed model of a transformational system, two possibilities are identified:

- *The reaction to a received stimulus is instantaneous.* This means that the component takes zero time to execute and that the output must be immediately available and propagated in the co-simulation model. As a result, each time an UML component is met, the master

sets the value of its input ports, executes a *doStep$_c$()* of size zero on the component, and gets the value of its outputs before executing and advancing time in the FMUs.

- *The reaction to a received stimulus is differed.* This means that the component takes time to compute or, that the reaction is intentionally delayed because the environment is not waiting for an imminent response. However, the model does not provide exact information about the instant of this reaction. A default simulation step size (which is that used for the FMUs) is used in this case and data are propagated at the same time for all components. The master algorithm calls *doStep$_c$()* of size h_{FMU} on all components in the co-simulation model, then propagates data which corresponds to the traditional behavior of the master algorithm.

The instant at which the data is propagated is therefore a semantic variation point. We propose to consider the first alternative since it is closer to the properties of reactive systems.
The value of an observed port is set only if the new value is different from the old value. If so, a new *ChangeEventOccurrence$^{(sem)}$* is created and added to the event pool of the *ObjectActivation$^{(sem)}$*.

Table 5-9. fUML routines for stepwise simulation and data propagation of an untimed model of a reactive system

Formalization functions	fUML or PSCS semantic model

doStep$_c$(h)	if (firstSimulationStep) then
	    c.actionActivation.startBehavior();
	end if;
	if (c.objectActivation.eventPool.size() > 0) then
	    c.actionActivation.dispatchNextEvent();
	else
	    do nothing;
	end if;

set$_c$(inPort,value)	if (c.inPort.oldValue != value) then
	    c.setFeatureValue(inPort,value);
	    if (inPort.observed) then
	        evt=new changeEventOccurrence(inPort,c.inPort.oldValue, value);
	        c.objectActivation.eventPool.add(evt);
	    endif;
	else
	    doNothing;
	endif;

get$_c$(outPort)	c.getFeatureValue(inPort);

5.2.3.3. Termination

The behavior of reactive systems does not terminate only if a problem occurs or the termination is enforced. Therefore, the behavior of an active class is supposed to run infinitely. The execution of its behavior (the classifier behavior) is handled by the *ObjectActivation$^{(sem)}$* class. The

termination of the latter may be forced by calling the operation *stop(),* which will terminate all classifier behavior executions. At the end of the co-simulation, all instances in the locus are automatically destroyed.

Table 5-10. fUML routines for termination of an untimed model of a reactive system

Formalization functions	fUML or PSCS semantic model
$terminate_c()$	c.objectActivation.stop();

5.2.4. Pseudocode of the master algorithm

The pseudocode of the MA we propose for the co-simulation of untimed UML models of reactive systems in FMI context is the same as given in Figure 5-2 and with the same assumptions. However, the rollback functionality cannot have been provided for this scenario. In fact, the outputs produced by reactive systems depends on the current inputs and may also depend on a previous state of the system. Unfortunately, the semantics of saving and restoring the state of an UML model execution (all the locus and the current position in the activity) are not yet supported in the current implementation. This capacity is very important in co-simulation. However, since it requires some effort and time to implement it, it is planned as an extension for this work.

5.2.5. Experience on a representative example

5.2.5.1.Definition of the simulation scenario

The representative example consists of the definition and the simulation of a co-simulation scenario composed of an FMU, the so-called 'Inc' (the same used in the section 5.1.4), and an untimed UML model of a reactive system, so-called '*Controller*' presented in section 5.2.1.2. The '*inc*' component in the Figure 5-8 is an instance of the imported FMU 'Inc'. For this scenario, we will use a default step size *'h=1'* instead of '*h=0,5'* since we are interested, for reactive components, by the instants at which the value of the counter changes. The counter, in fact, is incremented each one unit of time.

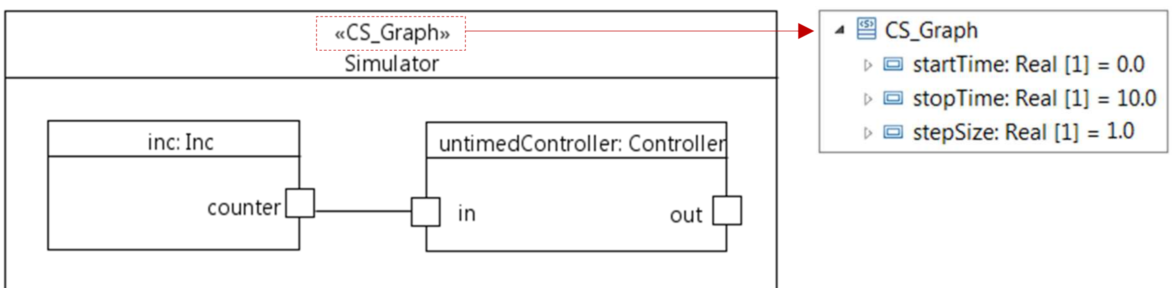

Figure 5-8. Co-simulation graph connecting an imported FMU to an untimed model of a reactive system

The value of the 'counter' is propagated to the input 'in' of the *'untimedController'*, an instance of the 'Controller' class defined in 5.2.1.2. The expected behavior of the *'untimedController'* component is an instantaneous reaction when the value of the counter reaches the *'treshold=4'*. The reaction consists in swithching the value of the output *'out'* from *'false'* to *'true'*. That is,

if the threshold is reached at time '$t_{in}=4$' then the 'untimedController' should react at '$t_{out}=t_{in}=4$'.
The co-simulation scenario will be simulated from '$t_{start}=0$' to '$t_{stop}=10$' with a default simulation step size '$h_{master}=1$' as indicated in the 'CS_Graph' stereotype. The simulation result are given in the next subsection.

5.2.5.2.Simulation of the co-simulation scenario

Figure 5-9 and Figure 5-10 depict simulation results of the co-simulation scenario defined in the previous subsection using respectively the basic master (as defined in the FMI standard in section 2.1.2) and the advanced master algorithm we proposed in section 5.2.4.

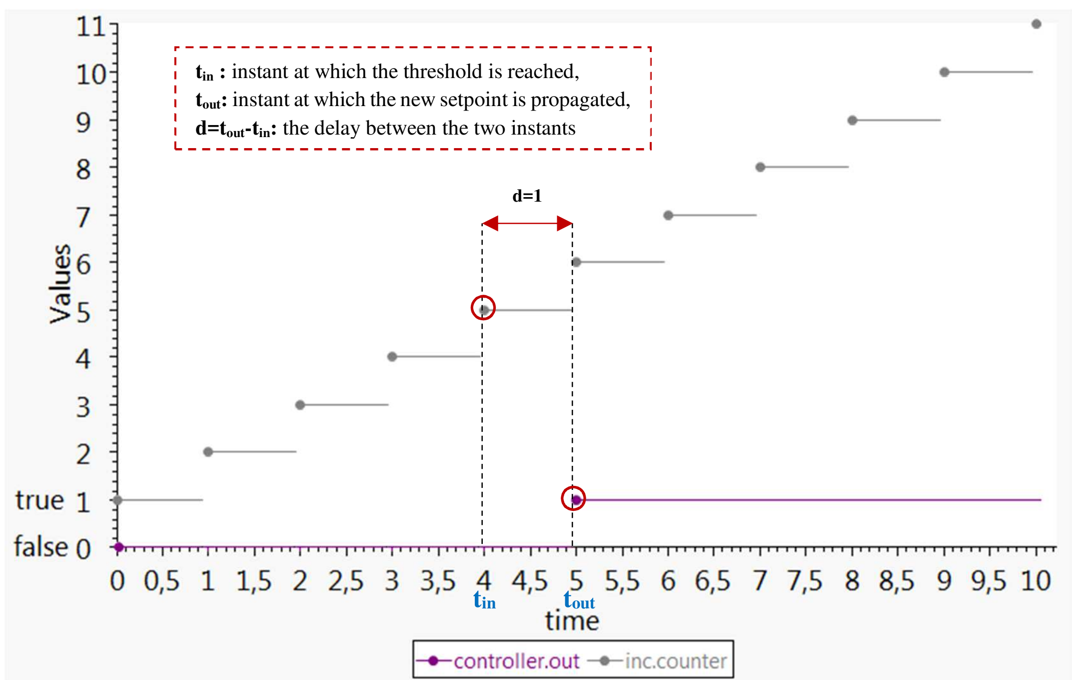

Figure 5-9. Co-simulation results of an untimed model of a reactive system - basic master

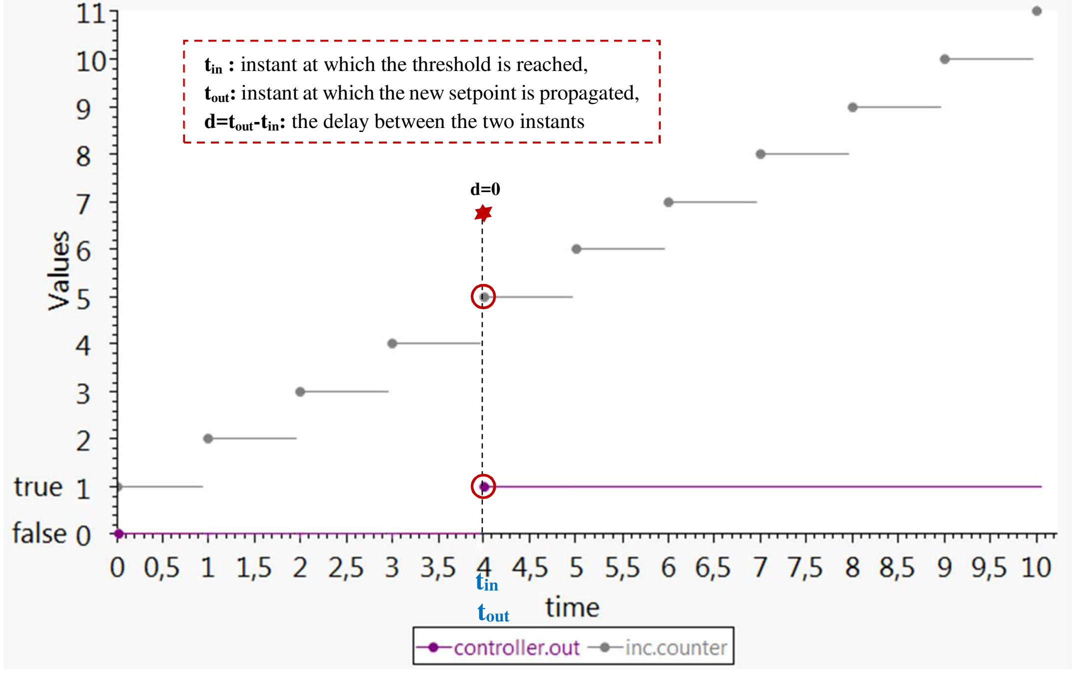

Figure 5-10. Co-simulation results of an untimed model of a reactive system - advanced master

Using the basic master algorithm given in section 2.1.2 (where the simulation step size $h_{master}=h_{FMU}>0$ for all components), we notice a delay between the instant at which the inputs arrive at the 'untimedController' component ($t_{in}=4$) and the instant at which this latter react to the change of the input value ($t_{out}=5$) (Figure 5-9).

Figure 5-10 demonstrates that this delay does not exist when using the master algorithm we proposed in section 5.2.4. The reaction of the 'untimedController' is in fact produced at the same time as the input value changes ($t_{in}=t_{out}=4$) which corresponds to the expected behavior.

5.3. Conclusion

In this chapter, we focused on the integration of untimed UML models in FMI-based co-simulation. For each kind of systems identified in the classification of section 3.2.1 (transformational and reactive systems), we provided rules for their modeling with UML in the context of the FMI standard, an adaptation between the execution semantics of UML models and the FMI API, and a master algorithm for the orchestration and synchronization of FMUs and UML components. The adaptation we proposed tackles, in particular, the semantic gap between untimed semantics of fUML* and timed semantics of FMI.
In the next chapter, we will deal with timed UML models where the behaviors described in this chapter are refined for the introduction of time information. As done for this chapter, we will provide rules for modeling this information in UML models, an adaptation between timed execution of UML models and FMI API, and master algorithms for synchronization of FMUs and UML components.

Chapter 6: Integration of timed UML models in FMI-based co-simulation

Outline

6.1. Timed models of transformational systems
6.1.1. Modeling rules for integration in FMI co-simulation
6.1.1.1. Model Structure and behavior
6.1.1.2. Applied stereotypes
6.1.2. Adapting fUML semantics to FMI API
6.1.2.1. Instantiation and initialization
6.1.2.2. Stepwise simulation and data propagation
6.1.2.3. Termination
6.1.3. Pseudocode of the master algorithm
6.1.4. Experience on a representative example
6.1.4.1. Definition of the simulation scenario
6.1.4.2. The simulation of the co-simulation scenario
6.2. Timed models of reactive systems
6.2.1. Modeling rules for integration in FMI-based co-simulation
6.2.1.1. Model Structure and behavior
6.2.1.2. Applied stereotypes
6.2.2. Extension of fUML semantics
6.2.2.1. The DE scheduler
6.2.2.2. The fUML extension
6.2.3. Adapting fUML execution semantics to FMI API
6.2.3.1. Instantiation and initialization
6.2.3.2. Stepwise simulation and data propagation
6.2.3.3. Termination
6.2.4. Pseudocode of the master algorithm
6.2.5. Experience on a representative example
6.2.5.1. Definition of the simulation scenario
6.2.5.2. The simulation of the co-simulation scenario
6.3. Conclusion

Time is a major concern when executing models for simulation purpose. The absence of time information in the system model considerably affects the simulations correctness when the goal is to verify the workflow duration of a system, or the correctness of its behavior when placed in a time-driven environment. Specifically, in the context of CPSs co-simulation, the computations are part of the system and are connected to physical components that continuously evolve in time. Their co-simulation should account for time properties of these components in order to produce correct results.

In the previous chapter, we identified a set of rules for the modeling of the structure and the behavior of transformational and reactive systems. However, the models we specified were untimed. This chapter focuses on timed UML models. We first identify important time information to model for each kind of systems by referring to their properties. Then we refer to UML syntax to identify a minimal subset to specify timed behaviors.

6.1. Timed models of transformational systems

6.1.1. Modeling rules for interation in FMI co-simulation

6.1.1.1. Model structure and behavior

A transformational system always executes the same behavior. This behavior, as stated in the previous chapter, is invoked, executed and then terminated at each simulation step. The produced outputs, in fact, only depend on the received inputs and do not consider the previous states of the system. These properties lead to the following conclusions:

A timed model of a transformational system does neither maintain information about the elapsed simulation time nor maintain information regarding the instants at which the simulation starts and finishes.

The outputs are produced once at the end of the activity execution. Therefore, in timed simulation, we essentially need to specify the duration of the entire activity.

As a result, the only important information we need to specify in a timed model for a transformational system is **how much time** computations take to run.

Table 6-1 exposes two alternatives to model this information in a timed UML model of the transformational system.

Table 6-1. Time modeling in UML models of transformational systems

Time information to model	Model requirement	UML concept
How much time computations take to execute?	The behavior must specify durations of computation nodes in the activity. *Or* The model must expose information about the execution duration of the whole behavior	C8: DurationConstraint$^{(syn)}$ on Action$^{(syn)}$ for computation actions. *Or* A Property(syn) 'stepSize' in the « CS_Timed» stereotype.

The first solution to model time on UML models consists in specifying the duration of the computations on the actions composing the activity. UML standard proposes a model of time which allows to represent time and durations, as well as actions to observe the passing of time. This alternative requires an extension of fUML* syntax and semantics as well as a control entity which is responsible for the execution of timed actions (i.e, pausing and resuming the execution at the right time) and for the time advancement locally in the component. In timed simulations, the control entity maintains information about the simulation start time and stop time as well as

the current simulation time. However, as stated previously, a model of a transformational system is not constrained to consider any notion of local and global time. It does not maintain information about the elapsed simulation time and the instants at which the simulation starts and finishes. This solution is therefore not well suited for the simulation of timed UML model of transformational systems. It will be explained further in the section 6.2.1 for time representation in reactive systems models.

The second solution consists in specifying the duration of the computations by adding an attribute ‘stepSize’ in the stereotype « CS_Timed». This stereotype will be applied to the class representing the transformational system (refer to section 6.1.1.2 of this chapter). The attribute ‘stepSize’ represents the duration of the whole behavior. Since the behavior is executed at each simulation step, this information corresponds to the step size of the model in the context of FMI. This solution does not require further extensions to the fUML* syntactic subset. The master can directly access the information before invoking the behavior of the component and compute the suitable step size to use. However, this solution does not ensure exact simulation results when there is more than one execution path (e.g. existence of alternatives or loops which depend on the inputs). Consider for example the behavior in the Figure 6-1. According to the value of the input ‘in’ (true or false), two execution paths are possible (‘compute1’ or ‘compute2’). Suppose that ‘compute1’ takes 4 units of time and that ‘compute2’ takes 5 units of time. In such cases, the ‘step size’ cannot always indicate the exact execution duration of the behavior. A choice strategy can be defined for the specification of the ‘step size’. One can choose the best or the worst execution duration of the behavior, or also an average of the different execution durations.

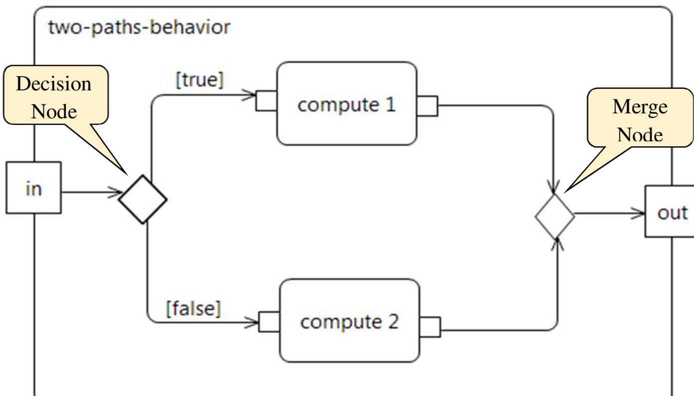

Figure 6-1. Two-path behavior

For the rest of the work, we choose the second solution. We suppose that the ‘step size’ of a UML model of a transformational system is the longer duration the component could take to compute.

6.1.1.2.Applied stereotypes

The FMI profile is extended with a new stereotype: the « CS_timed » stereotype. This stereotype is applied to all active classes whose behaviors are timed to the master in order to account for the delay between the invocation of the active class behavior and the production of the outputs. Table 6-2 recapitulates the stereotypes we must apply in a timed model of the transformational system.

Table 6-2. Stereotypes to apply for a timed model of transformational systems

Stereotype	Property	UML meta-class	Semantics
« CS_Timed »	StepSize	*Class*(syn)	Used to indicate that the class supports time semantics. The 'stepSize' property indicates to the master the amount of time taken by the component to produce a new output. It represents the step size of the component in the context of FMI.
« CS_Port »		*Port*(syn)	Identifies the ports which should be considered by the MA for data propagation.
« CS_Operation »		*Operation*(syn)	Applied to one, and only one, operation of the model. It enables the MST-engine to identify the operation that specifies the behavior of the system, therefore the operation to invoke for a simulation step.

The model given in the previous chapter for a simple transformational system is refined to introduce time. Figure 6-2 depicts a simple example of a timed model for a transformational system where both alternatives for time modeling are illustrated. For the rest of this section, we will consider the second alternative.

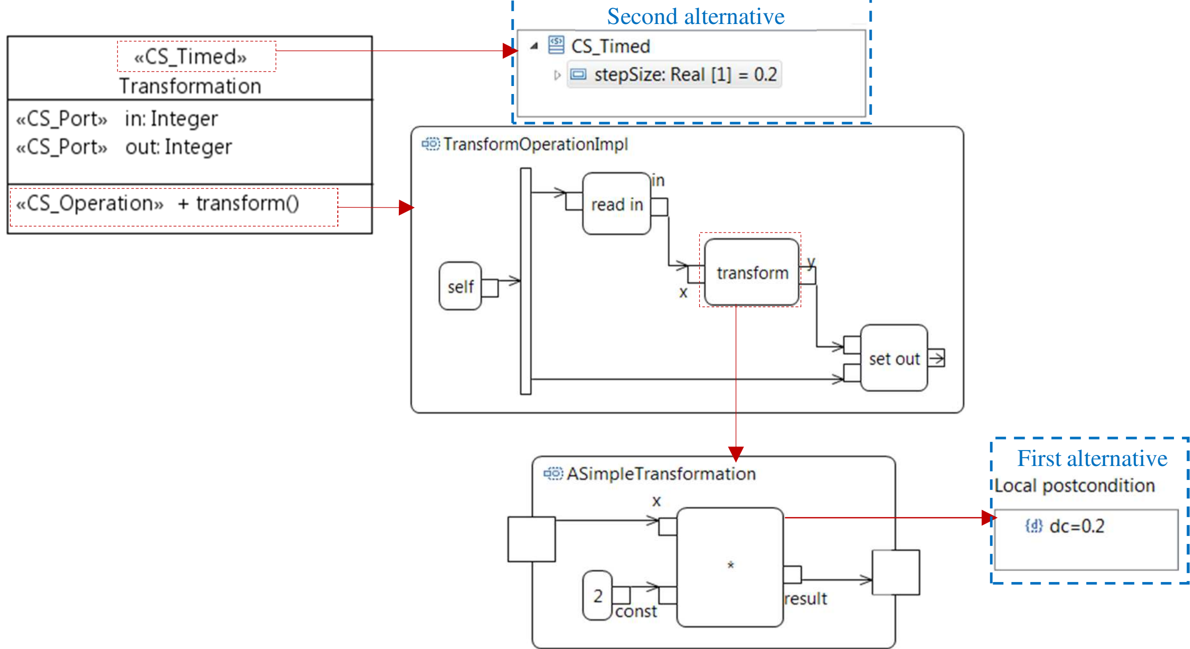

Figure 6-2. A timed model of a transformation system

6.1.2. Adapting fUML semantics to FMI API

The execution of timed UML models for transformational systems relies on the Data Flow MoC. In fact, when time is expressed using the second alternative, the execution of the behavior is still purely causal and do not consider any notion of simulated time. The time information, however, should be considered by the master to compute the suitable simulation step size.
This section focuses on how the master manage the integration of timed UML models with FMUs. Adaptation of semantics between execution semantics of UML models defined by fUML* (data flow) and semantics of FMI was already proposed in section 5.1.2. In this section we will demonstrate how the master will, in addition, account for time information expressed in UML models. It is organized into three subsections following the formalization of co-simulation given in section 2.1.1.2. It refers to the semantic model of UML models introduced in AnnexA and gives equivalent routines in the fUML* execution semantics for each function defined in the formalization.

6.1.2.1.Instantiation and Initialization

The instantiation and initialization routines are the same for the untimed models of transformational systems as illustrated in Table 6-3.

Table 6-3. fUML routines for instantiation and initialization of a timed model of a transformational system

Formalization functions	fUML or PSCS semantic model
instc()	c.locus.instantiate()
initc()	Features values are automatically initialized with values in the model during their instantiation

6.1.2.2.Stepwise simulation and data propagation

The master algorithm must account for the fact that the UML components are timed. As stated before, the duration of the activity executed when this component is invoked corresponds to the step size of that component. Transformational systems operate at their own rhythm, that is, no outputs could be retrieved before the end of the activity execution (the outputs are absent before the end of the activity execution). In FMI context, absent inputs (and therefore, absent outputs) are not allowed. An efficient master algorithm should not invoke functions related to data propagation between components (i.e, *getc()* and *setc()* functions), except at times when the values are present, and thus, should not perform a *doStep()* call with a step size smaller than the duration taken by the component to compute. Otherwise, a default value or an old value will be propagated which affect the correctness of the simulation results.
Consider a co-simulation scenario connecting an FMU to a timed UML model of a transformational system. Let $h_{UML}>=0$ be the step size of the UML component, h_{FMU} the simulation step size of the FMUs, and h_{MASTER} the step size chosen by the master for the co-simulation. For a simulation step starting at $t=t_c$:

- If $h_{UML}=0$ then $h_{MASTER}=0$ for the UML components and $h_{MASTER}=h_{FMU}$ for the FMUs. The outputs of the UML components are immediately propagated to the FMU when new data are available on its input ports as done for untimed UML models in Chapter 4. That is, they are produced and propagated at t_C. The outputs of the FMU are produced and propagated at t_C+h_{FMU}.

- If $h_{UML}>0$ then:
  - If $h_{UML}=h_{FMU}$ then $h_{MASTER}=h_{FMU}=h_{UML}$,
  - If $h_{UML}<h_{FMU}$ then two alternatives are possible:
    - $h_{MASTER}=h_{UML}$: the outputs of the UML component are computed and propagated at $t=tc+h_{MASTER}=tc+h_{UML}$.
    - $h_{MASTER}=h_{FMU}$: the outputs of the UML component are computed at $t=tc+h_{UML}$ and propagated at $t=tc+h_{MASTER}=tc+h_{FMU}$ with a delay of $h_{MASTER}-h_{UML}$.
  - If $h_{UML}>h_{FMU}$ then two alternatives are possible:
    - $h_{MASTER}=h_{UML}$: the outputs of the UML component are computed and propagated at $t=tc+h_{MASTER}=tc+h_{UML}$.
    - $h_{MASTER}=h_{FMU}$: the outputs of the UML component are not yet computed at $t=tc+h_{MASTER}=tc+h_{FMU}$. Since the FMI standard does not allow absent values, then default values must be specified for all UML component outputs.

We believe that the computational components execution is almost instantaneous but precision is important in case where the systems under design are time critical. We assume that $0<=h_{UML}<=h_{FMU}$. According to the analysis made above, $h_{MASTER}=h_{UML}$ which ensures the propagation of new output values without any delay.

Suppose now that the co-simulation model includes two timed UML components of transformational systems C1 and C2, and an FMU. Let $h_{UML1}>=0$ and $h_{UML2}>=0$ be the step sizes of C1 and C2 respectively, and h_{FMU} the step size of the FMU>0. Two situations are possible:

- $h_{UML1}=h_{UML2}$: This scenario is equivalent to the previous one,
- $h_{UML1}<>h_{UML2}$ then suppose that $h_{UML1}<h_{UML2}$:
  - $h_{MASTER}=max(h_{UML1},h_{UML2})=h_{UML2}$: the outputs of C2 are computed and propagated at $t=tc+h_{MATSER}$, wheareas the outputs of C1 are computed $t=tc+h_{UML1}$ and propagated with a delay of $h_{MASTER}-h_{UML1}$.
  - $h_{MASTER}=min(h_{UML1},h_{UML2})=h_{UML1}$: the outputs of C1 are computed and propagated at $t=tc+h_{MATSER}$ whereas the outputs of C2 are not yet computed. Since the FMI standard does not allow absent values, then default values must be specified for all UML component outputs.

  Both alternatives leads to non-precise simulation results. The presence of new values on outputs of the components with a little delay ensures at least that these values are really produced by the system, whereas default values are just predictions. Then, $h_{MASTER}=Max(\{h_C, c \in UT\})$ where UT is the set of timed UML components of transformational systems.

Table 6-4 illustrates the mapping between the formalization functions and fUML routines for stepwise simulation and data propagation of timed UML model of reactive systems.

Table 6-4.fUML routines for stepwise simulation and data propagation of a timed model of a transformational system

Formalization functions	fUML or PSCS semantic model routines
$doStep_c(h_{UML})$	c.dispatch(operationToExecute).execute();
$set_c(inPort,value)$	c.setFeatureValue(inPort,value)
$get_c(outPort)$	c.getFeatureValue(inPort)

6.1.2.3.Termination

The termination routines are the same as for the untimed models of transformational systems. They are illustrated in Table 6-5.

Table 6-5. fUML routines for termination of an untimed model of a transformational system

Formalization functions	fUML or PSCS semantic model
$terminate_c()$	do nothing.

6.1.3. Pseudocode of the master algorithm

```
/*Assumptions*/
On the co-simulation graph: no cycles exist in the co-simulation model, the graph connects a set of
FMUs with a timed UML model of a transformational system
On UML components: zero step size allowed,
On the FMUs: the step size proposed by the master algorithm is accepted by all FMUs,
/*Co-simulation parameters*/
tc: Current simulation time , tstart: Start simulation time ,  tstop: Stop simulation time
hC: the step size of the component being simulated
hMASTER: the co-simulation step size
X: U∪Y : set of ordered ports variables computed by "Variables-order" algorithm
C= F ∪ TU where TU: the set of timed UML components of transformational systems
/*Instantiate and initialize components c ∈ C */
for each component c ∈ C:
        instc();
        if c ∈ F
                initc(tstart, tstop);
        enf if;
/*Step wise simulation*/
hMASTER=Max({hC, c ∈ TU})
while (tc<tstop)
        for each input u ∈ X
                y = P(u);
                v = getc(y);
                setc(u,v);
                if (hC = 0)
                        doStepc(0);
                end if;
        end for
        for each c ∈ C
                if (hC > 0)
                        doStepc(hMASTER);
                end if;
        end for;
        tc=tc+ hMASTER;
end while;
/* Termination of the simulation*/
for each component c ∈ C:
        terminatec() ;
end simulation
```

Figure 6-3. Pseudocode of a master algorithm for a Co-simulation graph connecting FMUs with timed models of a transformational system.

The MST-engine is enriched with a new master algorithm depicted in Figure 6-3 which orchestrates a set of imported FMUs connected to one or more timed UML models of a transformational systems. In this algorithm, we suppose that the FMUs always accept the simulation step size proposed by the master and that UML models are allowed to have a zero simulation step size. The master waits for all UML components to finish their computations before propagating the new outputs

6.1.4. Experience on a representative example

6.1.4.1. Definition of the simulation scenario

The representative example consists of the definition and the simulation of a co-simulation scenario composed of an FMU, the so-called 'Inc' (the same used in the section 5.1.4), and an timed UML model of a transformational system, so-called '*Transformaiton'* presented in section 6.1.1.2. The '*inc'* part in Figure 6-4 represents the imported FMU 'Inc'. For this scenario, we will use the default step size *'h=0,5'*.

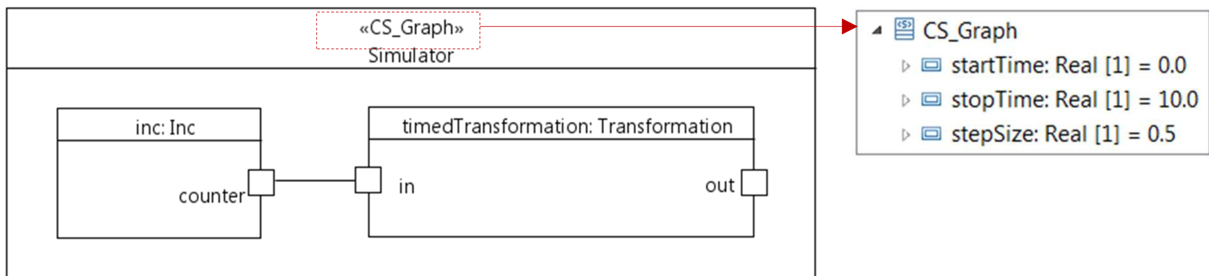

Figure 6-4.Co-simulation graph connecting an imported FMU to an timed model of a transformational system

The value of the 'counter' is propagated to the input 'in' of the *'timedTransformationr'*, which represents 'Transformation' class defined in section 6.1.1.2. The expected behavior of the *'timedController'* component is to output a result after '0,5' unit of time it receives values on its input port.

The co-simulation scenario will be simulated from 't_{start}=0' to 't_{stop}=10' with a default simulation step size 'h_{master}=0,5' as indicated in the 'CS_Graph' stereotype. The simulation results are given in the next subsection.

6.1.4.1. Simulation of the co-simulation scenario

Figure 6-5 and Figure 6-6 depict simulation results of the co-simulation scenario defined in the previous subsection using respectively, the basic master (defined in the FMI standard in section 2.1.2), and the advanced master algorithm proposed in section 5.1.3.

With a default step size 'h_{master}=h_{UML}=h_{FMU}=0,5' we obtain the same simulation results using both master algorithms.

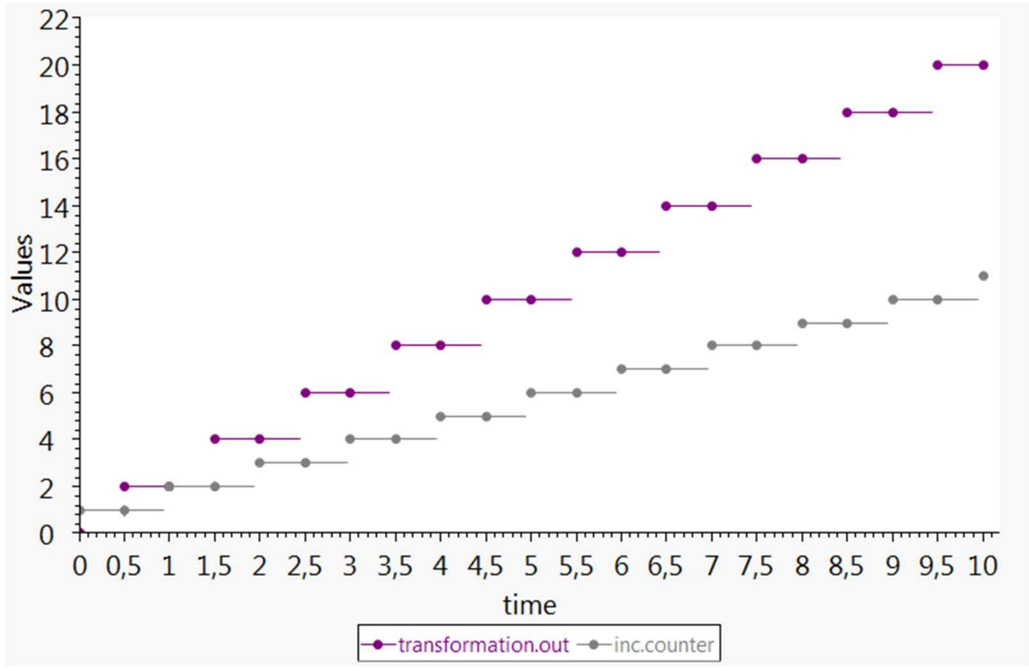

Figure 6-5. Co-simulation results of a timed model of a transformational system-Basic master

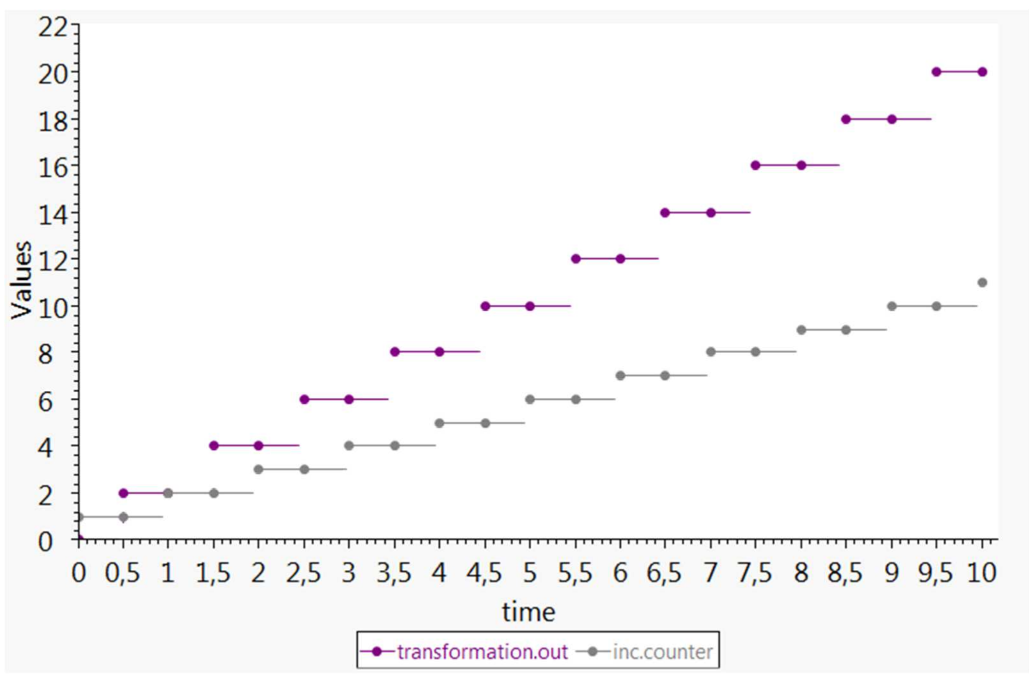

Figure 6-6. Co-simulation results of a timed model of a transformational system-Advanced master

6.2. Timed models of reactive systems

6.2.1. Modeling rules for integration in FMI-based co-simulation

6.2.1.1. Model structure and behavior

A reactive system outputs a reaction when some conditions become true. Controllers are typical examples of such systems. Here, we provide brief examples of reactive systems where modeling of time information becomes essential:

- A system may produce a reaction immediately (instantaneous response) or waits for some time before producing a reaction to a received stimulus (differed response). We take as an example, a system that controls the level of a liquid in a tank (similar to that described in Chapter 4). The controller receives information about the level of the liquid in the tank and should provide control commands to the plant based on this information:
  - o If the liquid approaches the maximum allowed level and the flow of liquid is high, then it must **immediately** order to open a valve to evacuate the tank.
  - o If the liquid approaches the maximum allowed level and the liquid flow is low, then it should order to open the valve **after some amount of time**.
- A system may need to output some commands **at a precise instant**. For example, in smart houses, it may need to configure the heater controller to activate the heater **at six p.m**. and deactivate it **at four a.m**. The heater controller could be further configured to stop the heater **for fifteen minutes** if the temperature reaches the maximum allowed value.
- A system that **periodically** executes some task. For example, the mailing system that updates the inbox every 5 minutes.

The syntactic UML elements related to the modeling of change events received on input ports of a reactive system was identified in the previous chapter in section 5.2.1.1. In this chapter, we focus on expressing time on the behaviors of reactive systems in order to model differing reactions to change events received on input ports and also, to model reactions to time events. For this, we need to identify how UML activities represent expressions written in bold. We distinguish two kinds of actions for the modeling of timed reactive systems activities:
- Computation actions which execute some operations. The execution can take time to execute (e.g, **for fifteen minutes**).
- Synchronization actions which wait for a precise instant to propagate the control (e.g, **at six p.m., after some amount of time** and also **immediately**). A timed behavior should be able to express time instants at which these actions should fire.

Table 6-6 recapitulates rules for timed behavior modeling of reactive systems.

Table 6-6. Time modeling with UML for transformational systems

Time information to model	Model requirement	UML concept
i.How much time computations take to execute?	The behavior must specify durations of computation nodes	C8: *DurationConstraint*$^{(syn)}$ on *Action*$^{(syn)}$ for computation actions.
ii.Time instants for synchronization of executions	The behavior must specify instants at which synchronization nodes must fire	C9: *AcceptEventAction*$^{(syn)}$ triggered by a *TimeEvent*$^{(syn)}$ for synchronization actions.

UML standard proposes a model of time which enables the representation of time in the applicative models. The latter comprises meta-classes to represent time and durations, as well as actions to observe the passing of time. It introduces numerous concepts of time modeling, but we identified a minimal subset of UML syntactic elements sufficient to express required time information.

i. When associated with an Action(syn), a *DurationConstraint*(syn) ***(C8)*** indicates that the execution duration of this action can take a duration between two boundaries defined as a *DurationInterval(syn)*. We note the duration interval '[dmin, dmax]' where 'dmin' and 'dmax' are of type *Duration(syn)* and specify the minimum and maximum action execution duration, respectively.
On actions, a *DurationConstraint*(syn) can be specified as a *localPreCondition* or a *localPostCondtion*. Local pre-conditions and local post-conditions are constraints that should hold when the execution starts and completes, respectively. They hold only at the point in the flow that they are specified. A *DurationConstraint*(syn) expressed on an action means that the duration at the end of the action execution must be between the minimum and the maximum durations defined in the duration interval '[dmin, dmax]'. Therefore, this constraint should be specified as a *localPostCondition* of the action. The choice of the duration taken during the execution of the behavior is a semantic variation point. One can always choose the minimum duration or the maximum duration as a strategy. A random choice of the duration between the two boundaries can also be the defined strategy in which case we need an observation to know which duration is finally taken by the action. Another alternative is to specify the same duration for both boundaries (dmin=dmax), which allows for the constraint of the duration of the action execution to an exact amount of time.
The *CallBehaviorAction(syn)* is a call action that invokes a behavior. The argument values are passed on the input parameters of the invoked behavior. In a synchronous call, a *CallBehaviorAction(syn)* waits until the execution of the invoked behavior completes and the values of output parameters of the behavior are placed on the result output pins. The invoked behavior represents the computation of the system. The choice to encapsulate the computations of the system in a *CallBehaviorAction(syn)* ensures that there is only one computation node in the activity that implements the operation for which a *DurationConstraint(syn)* will be specified.

ii. The *AcceptEventAction(syn)* is a particular kind of actions used to represent a synchronization point by waiting for the occurrence of a particular event. A *TimeEvent(syn)* ***(C9)*** specifies a point in time at which an event occurs. When associated to an *AcceptEventAction(syn)*, a time event specifies the instant at which the action should be effectively executed. As constrained by the fUML specification, this kind of actions is only allowed in behaviors associated to an active *Class(syn)* (i.e. in classifier behaviors). Time events are specified by an *Expression(syn)*. This may be absolute, in which case the actions occur at a date that is known from the beginning, or relative to some other point in time, when the action occurs sometime after the execution of a previously encountered node in the execution flow. An absolute time trigger is specified with the keyword 'at' followed by an expression that evaluates to a time value, such as '*at 7a.m.*'. A relative time trigger is specified with the keyword 'after' followed by an expression that evaluates to a time value, such as *'after 5 seconds*'.

6.2.1.2. Applied stereotypes

The « CS_timed » stereotype is applied to all computational components whose behaviors are timed. This stereotype was already introduced in section 6.1.1.2 but a new property, so-called 'NextEventTime', is added. Table 6-7 recapitulates all stereotype that should be applied to a timed model of a reactive system as well as their important properties, while reviewing their semantics.

Table 6-7. Stereotypes to apply for a timed model of reactive systems

Stereotype	Properties	UML meta-class	Semantics
« CS_timed »	NextEventTime	Class$^{(syn)}$	Used to indicate that the class supports time semantics. The 'nextEventTime' property will be used to indicate to the master the next time at which the component will produce new outputs.
« CS_port »		Port$^{(syn)}$	Identifies the ports which should be considered by the MA for data propagation.

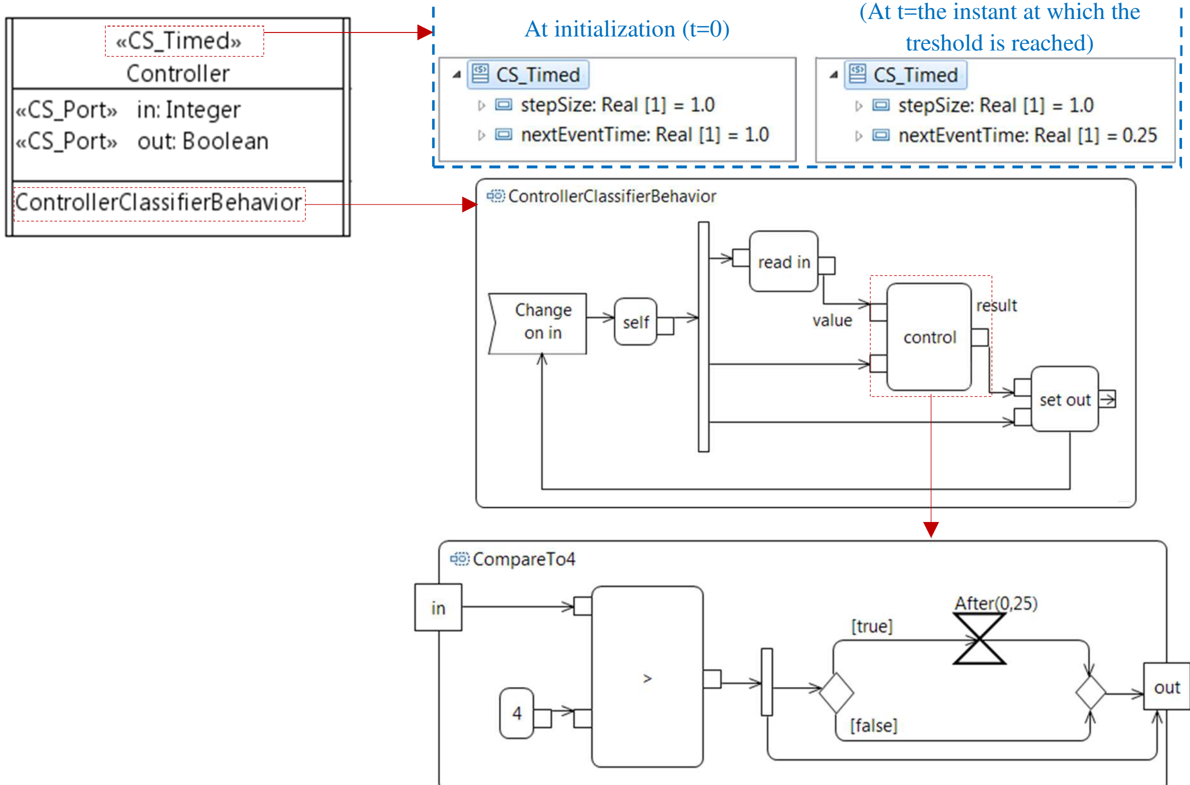

Figure 6-7. A simple example of a timed model of a reactive system

Figure 6-7 illustrates the use of an *AcceptEventAction*$^{(syn)}$, the so-called 'after(0,25)', triggered by a time event. The *TimeEvent*$^{(syn)}$ has a relative time stamp of 0,25 units of time. The 'after(0,25)' is a synchronization action which is used to delay the setting of the output by 0,25 units of time when the execution of the action '>' returns 'true' (i.e, when the input 'in' exceeds the value of four).

6.2.2. Extension of fUML semantics

Reactive systems receive events and produce reactions at discrete instants. The associated behaviors progress at events occurrences. Since the behavior is timed, each event occurrence may be given a time stamp corresponding to the instant at which it occurs. Events may be internal events (i.e, time events representing the completion of some processing or the end of waiting time) or external events (i.e, arrival of a new value on the input port of the system). A timed behavior of a reactive system is therefore the execution of a sequence of events. We recognize the DE MoC semantics.
As stated in Chapter 3, the fUML execution model is agnostic about time semantics (M2). It does not provide a control entity which appropriately schedules the execution of the various model elements in order to reflect the timing aspects (M3) . The construction of the DE MoC on fUML consists in overcoming these two shortcomings using the mechanisms introduced in the Chapter 3, namely the extensibility of the fUML semantic model (A1) and the control delegation (A2), as follows:

- Extending fUML with the UML syntactic elements identified previously in section 6.2.1.1., and with new semantics brought by these elements. New execution semantics shall provide the logic for interpreting time representation on activities.
- Providing a control entity, so-called DE scheduler, responsible for the scheduling of the model elements execution in order to reflect time aspects using the control delegation mechanism as explained in section 3.2.4.2.

In order to understand how the DE MoC can be constructed on the fUML semantic model, we refer to the principles of DE simulation. Section 6.2.2.1 sgives an overview of the key concepts of DE simulation, then represents the implementation of the DE scheduler in our framework and its interaction with the fUML semantic model. Section 6.2.2.2 focuses on the extension of the fUML syntax and semantics.

6.2.2.1.DE scheduler

Adiscrete event simulation engine must rely on one of the three DE simulation approaches (also called world-views) commonly used in the literature [27]: event-scheduling approach, activity-scanning approach, and process iteration approach. The event-scheduling world-view, so called Event-driven, focuses on events and is suited to the simulation of timed behaviors of reactive systems. The behavior is represented as a discrete chronological sequence of events. These events are arranged on a list called future event list (FEL) in a chronological time order as described in Figure 6-9. t_c is the current simulation time and E_{tc} is the event being executed. The execution of an event consists in updating the system state by pulling it from the event pool and executing actions that are associated with it if that exists. Once E_{tc} is executed, the first event to execute in the FEL is E_{tci}, referred to as the imminent event. The execution of an event

may generate new events which should be pushed in the FEL. For example, in Figure 6-9. Events order in the FEL the execution of E_{tci+1} generates an event E_{tk} where t_k is the time stamp of the event and $t_{i+1} < t_k < t_n$ which should be inserted at the right place in the FEL.

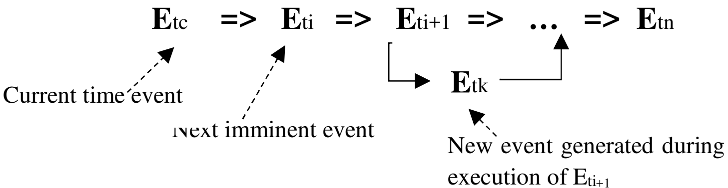

Figure 6-9. Events order in the FEL

1. Start of simulation
    a. Initialize clock to 0,
    b. Schedule initial events,
    c. Initialize state variables
2. While FEL not empty
    a. Remove the imminent event ($\mathbf{E_{ti}}$) from the FEL ($\mathbf{E_{ti}} \leftarrow \mathbf{E_{ti+1}}$)
    b. Advance the clock from $\mathbf{t_c}$ to $\mathbf{t_i}$ ($\mathbf{t_c} \leftarrow \mathbf{t_i}$)
    c. Execute $\mathbf{E_{ti}}$
    d. Generate future events $\mathbf{E_{tk}}$, if exist, and place their event notices on FEL in the correct position on the FEL according to time $\mathbf{t_k}$
3. End of simulation

Note that more than one event may occur at the same time (i.e. more than one imminent event E_{ti}) in which case the scheduler should pull all imminent events from FEL (step 2.a.) before advancing time then execute and generate future events of all of them.

Figure 6-8. Algorithm of discrete event simulation

The DE simulation relies on a variable time advance, that is, when an event is executed, the simulation clock is advanced to the time of the imminent event in the FEL. The DE simulator is responsible for advancing simulation time and guaranteeing that events are executed in the correct chronological order. Both tasks are based on the FEL. They are usually implemented in an event scheduling/time advance algorithm as illustrated in Figure 6-8. Algorithm of discrete event simulation.

The implementation of the DE scheduler is independent of the fUML semantic model. It is implemented using the control delegation mechanism using the aspect (A2). Figure 6-10 depicts an UML class diagram of the implementation of this scheduler in our framework. The latter (represented with the *DEScheduler* class) maintains a list of events, the *FEL*, ordered according to their relative time. It behaves as described in the scheduling algorithm presented previously in Figure 6-8. Algorithm of discrete event simulation. At the beginning of the simulation, the scheduler is initialized with the simulation parameters (i.e, start time and stop time of the simulation).

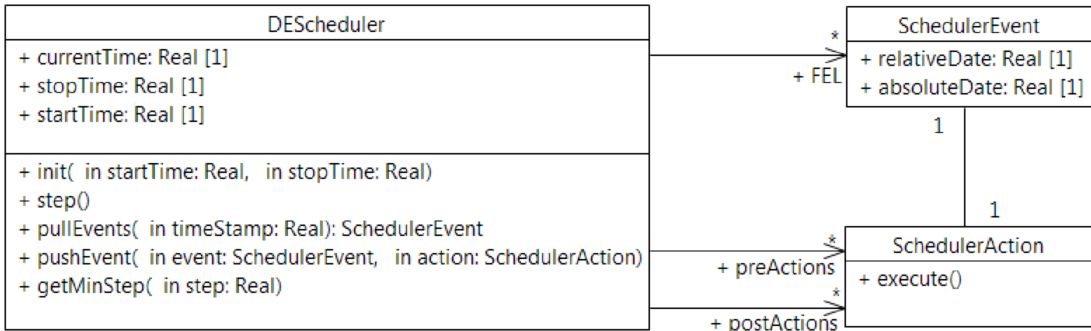

Figure 6-10. The DE scheduler model

A simulation step consists of pulling and executing the imminent events from the *FEL*, and updating the *currentTime*. The scheduler gives deterministic semantics to the execution of simultaneous events, that is, if there are two or more imminent events with the same time stamp, then all are pulled from the event list and executed at the same time. Each event is associated to a set of *SchedulerAction*. This set indicates the action to execute by the scheduler before (the *preRunAction*) or after (the *postRunAction*) the execution of an event.

The execution of timed activities is handled by the DE scheduler. It is independent from the fUML syntax and semantics, but it is in interaction with the timed activity nodes activations during the execution. The latter is responsible for the ordering of the events in the FEL and their execution by resuming the execution flow of the activity at the correct instants. Once a timed fUML *Action*$^{(syn)}$ is encountered, the execution should be suspended (the operation *suspend()* of the corresponding *ActionActivation*$^{(sem)}$) before sending offers to the next node in the activity. A *SchedulerEvent*, whose time stamp corresponds to the instant at which the execution of this action node will be resumed, is pushed in the *FEL*.

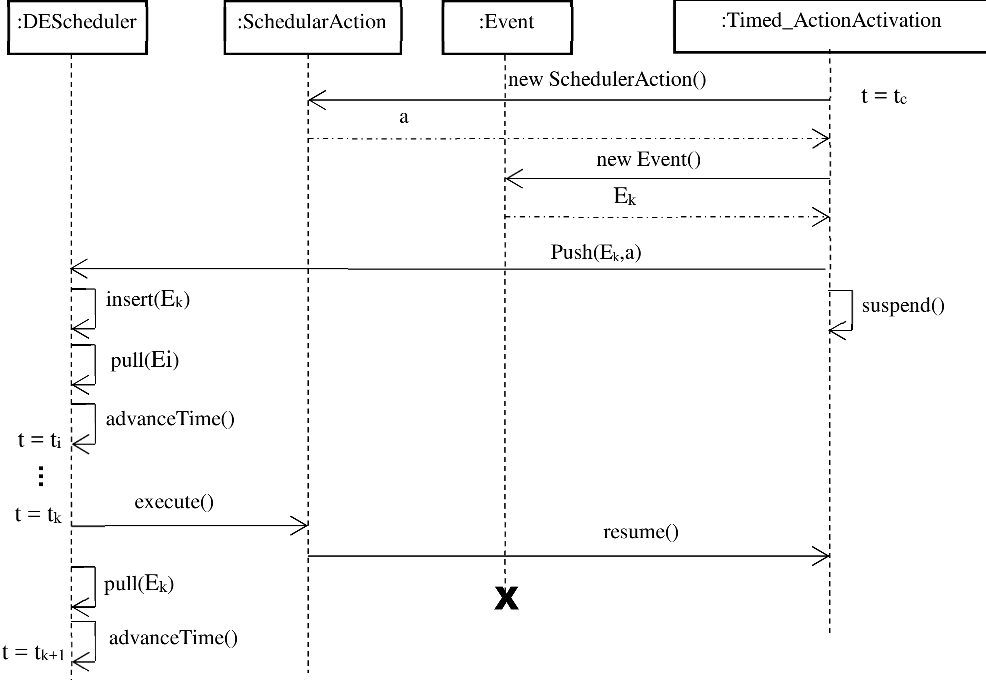

Figure 6-11. Interaction description of the DE scheduler with a semantic element capturing the execution semantics of a timed activity action

The *SchedulerAction* associated with this event consists in resuming the execution of this action node (the operation *resume()* of the corresponding *ActionActivation*$^{(sem)}$) when the event is pulled from the FEL. Figure 6-11 illustrates the interaction between the DE scheduler and the semantic visitor *Timed_ActionActivation*$^{(sem)}$. This semantic visitor implements the semantics of a timed *Action*$^{(sem)}$.

6.2.2.2. fUML extension

The extension of the fUML semantic model requires the definition of a new locus *Timed_Locus*$^{(sem)}$, new semantic visitors, as well as a new execution factory *Timed_ExecutionFactory*$^{(sem)}$ responsible of the instantiation of these visitors as follows:

- *Timed_Locus*$^{(sem)}$

This new locus extends the Locus(sem) of the fUML semantic model to account for objects representing the classes annotated with the “CS_Timed” stereotype and relying on DE MoC. These objects are represented in the locus by *Timed_Object*$^{(sem)}$ type.

- New semantic visitors

The new semantic visitors are in charge of transforming time information expressed on actions into time events in the FEL, as well as rerouting the control to the DE scheduler when a time event is scheduled. Listed here are the most important semantic visitors required for enabling the execution of timed behaviors:

- *Timed_Object*$^{(sem)}$

  This type represents an instance of an active *Class*$^{(syn)}$ to which CS_Timed stereotype is applied and for which new semantics are defined to account for time. It is a specialization of the *Object*$^{(sem)}$ semantic visitor. The behavior of a *Timed_Object*$^{(sem)}$ is controlled by a *Timed_ObjectActivation*$^{(sem)}$, a specialization of *ObjectActivation*$^{(sem)}$ as depicted in Figure 6-12. *Timed_ObjectActivation*$^{(sem)}$should ensure that the activity is not automatically started when a *Timed_Object*$^{(sem)}$ is instantiated. This task is performed by disabling the event dispatch loop and scheduling an event at t=0 in the FEL. The scheduler action associated to this event corresponds is responsible for resuming the activity execution and thus, for its starting.

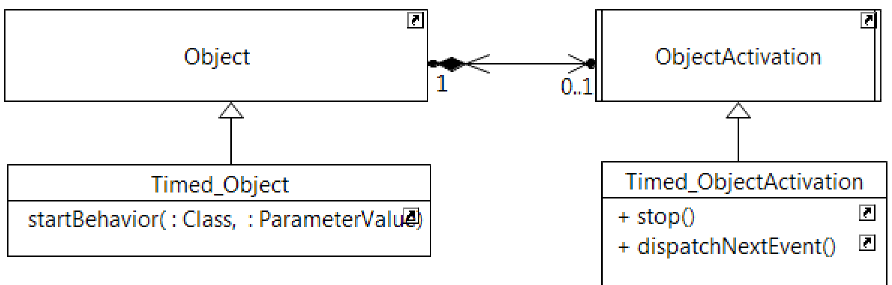

Figure 6-12. Extension of behavioral semantics for timed execution: Object and ObjectActivation

- *Timed_ActionActivation*$^{(sem)}$

  Time expressed on Action$^{(syn)}$ indicates relative or absolute instants at which events should be scheduled in the FEL. These instants correspond to the end of a processing or a wait action and therefore the instants at which the tokens should be propagated in the activity.

The operation responsible for tokens propagation is *sendOffers()*. The latter should be redefined to capture new semantics of timed actions execution, that is, to suspend the execution of the action node and to push a new event in the FEL with a time stamp corresponding to the instant at which the action should be resumed. The scheduler action associated to this event is in charge of resuming the execution of the *Action*$^{(syn)}$ and propagate data with *sendOffers()*.
The execution of *Action*$^{(syn)}$ nodes is handled by concrete actions activations which are specializations of *Timed_ActionActivation*$^{(sem)}$. Figure 6-13 depicts action activations we are interested in for the modeling of reactive systems behaviors, namely *Timed_CallBehaviorActionActivation*$^{(sem)}$ and *Timed_AcceptEventActionActivation*$^{(sem)}$.

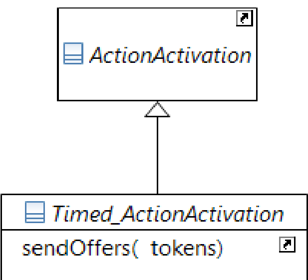

Figure 6-13. Extension of behavioral semantics: the activity nodes activations

- *Timed_ExecutionFactory*$^{(sem)}$

Timed_ExecutionFactory in Figure 6-14 is a new factory defined as a specialization of the *ExecutionFactoty*$^{(sem)}$ defined in the fUML semantic model. It is responsible for the instantiation of the new visitors introduced in this section, which capture time execution semantics.

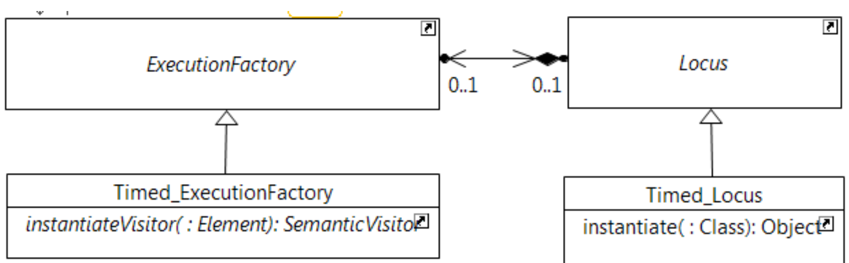

Figure 6-14. Extension of instantiation semantics: Locus and ExecutionFactory

6.2.3. Adapting fUML execution semantics to FMI API

The execution semantics of timed UML models for reactive systems corresponds to the semantics of the DE MoC introduced in 1.1.2.3. The behavior is activated at a discrete set of instants which corresponds to time events occurrences. The execution of the FMUs, on the contrary, continuously depends on time. This difference raises a continuous time vs discrete event semantics issue.

This section focuses on this issue and proposes adaptation of semantics between execution semantics of UML models and semantics of FMI. It is organized into three subsections following the formalization of co-simulation. It refers to the extended semantic model introduced in the previous section and gives equivalent routines for each function defined in the formalization.

6.2.3.1. Instantiation and initialization

The instantiation of timed models is handled by the new locus and execution factory: *Timed_locus*$^{(sem)}$ and *Timed_ExecutionFactory*$^{(sem)}$.

The initialization consists in initializing the instantiated objects with default values in the model as well as in initializing the DE scheduler with start simulation time and stop simulation time as illustrated in Table 6-8.

Table 6-8.fUML routines for instantiation and initialization of a timed model of a reactive system

Formalization functions	fUML semantic model
instc()	c.locus.instantiate();
initc()	Features values are automatically initialized with values in the model during instantiation DEScheduler.init(startTime, stopTime);

6.2.3.2. Stepwise simulation and data propagation

FMUs rely on CT MoC where outputs continuously change and can be retrieved at any instant during simulation. FMUs are usually executed with a fixed simulation step size (i.e. if the FMU does not require to rollback, the master usually invoke it using the same step size from the beginning to the end of the simulation). Timed UML component execution relies on the DE MoC where outputs change at discrete set of instants during simulation. Between two instants, the outputs do not change. The execution is handled by the DE scheduler and relies on a variable time advance. Time is advanced only when an event is executed. This raises a continuous time vs discrete event semantics gap. In order to integrate timed UML model in FMI-based co-simulation, the MST-Engine should synchronize the DE scheduler with the FMUs executions using sone of the following wrappers.

- Multiple Timestamps Wrapper

A call to *doStep()* at time t_c with a step size h corresponds to the execution of all the events scheduled between t_c and t_c+h in the FEL. At the end of the step, time advances to t_c+h. Let t_i be the time stamp of the imminent event in the FEL:

a. If (h<ti) then no event is scheduled between t_c and t_c+h and no new outputs are computed during the simulation step,

b. If (h>=ti) then, the DE scheduler executes events having relative time stamps between ti and h. Let t_l be the relative time stamp of the last event scheduled between t_c and t_c+h in the FEL:

b.1. If ($h>t_l$) then new outputs are computed at $t=t_c+t_l$ and propagated at $t=t_c+h$.

b.2. If ($h=t_l$) then the outputs are computed at $t=t_c+t_l$ and propagated $t=t_c+h=t_c+t_l$.

Note that in case **b.1.,** the outputs are computed at a time t_c+t_l prior to that of the data propagation t_c+h in the model which may affect the simulation results, and that, in both cases **b.1.** and **b.2.,** no outputs are propagated to the environment for events scheduled between t_c and t_c+h except the one scheduled at $t=t_c+t_l$. The MST-Engine misses events which also considerably affect the correctness of the simulation results. As a solution to that we propose the single time stamp wrapper.

- Single Timestamp Wrapper

Let t_i be the time stamp of the imminent event in the FEL. A call to *doStep()* at time t_c with a step size h corresponds to the execution of the events scheduled between t_c and t_c+h and having a timestamp equal to t_i. At the end of the step, time advances to t_c+h.

a. If $(h<t_i)$ then no event is scheduled between t_c and t_c+h then no new outputs are computed during the step,

b. If $(h>t_i)$ then new outputs are computed at $t=t_c+t_i$ and propagated in the model at $t=t_c+h$,

c. If $(h=t_i)$ then new outputs are computed at $t=t_c+t_i$ and propagated in the model at $t=t_c+h=t_c+t_i$

Note that in the case **b**, the outputs are computed at a time t_c+t_i prior to that of the data propagation in the model which may reduce the efficiency of the co-simulation by affecting the correctness of the results, and that, if other events are scheduled between t_c+t_i and t_c+h these events will be lost.

In order to avoid events missing, at each simulation step, the master proposes a simulation step size $h<=t_i$. This can be done by making accessible to the master the information about the time stamp of the next event in the FEL. At each simulation step, the master retrieves the time stamp of the next event in the FEL and computes a simulation step size accepted by all components in the co-simulation graph. This information is indicated in the property 'NextEventTime' of the stereotype 'CS-Timed' introduced in the Table 6-9.

Let $h_{UML}=t_i$ is the adequate step size for a timed UML model and h_{FMU} is the simulation step size used for the simulation of the FMUs, then $h_{MASTER}= \min(h_{FMU},t_i)$ is the simulation step size proposed by the master.

Table 6-9. fUML routines for stepwise simulation and data propagation of a timed model of a reactive system

Formalization functions	fUML semantic model

doStep$_c$(h)	if (firstSimulationStep) then
	        c.actionActivation.startBehavior();
	end if;
	if (DEScheduler.FEL.size() > 0) then
	        DEScheduler.step(h);
	else
	        do nothing;
	end if;

set$_c$(inPort,value)	if (c.inPort.oldValue != value) then
	        c.setFeatureValue(inPort,value,0);
	        if (inPort.observed) then
	            evt = new changeEventOccurrence(inPort, c.inPort.oldValue, value);
	            c.objectActivation.eventPool.add(evt);
	        endif;
	else
	        doNothing;
	endif;

get$_c$(outPort)	c.getFeatureValue(inPort,0);

6.2.3.3.Termination

The control of the activities execution is delegated to the DE scheduler. The latter is therefore responsible for their termination. At the initialization phase, the scheduler was initialized with the simulation parameters (start and stop time). A time event E_{stop} with a time stamp equal to the t_{stop} is pushed in the FEL. The scheduler action associated to E_{stop} corresponds to the termination of the activity execution. The termination of the activity execution is performed by calling the operation *stop()* which will terminate all classifier behavior executions.
At the end of the co-simulation, all instances in the locus are automatically destroyed.

Table 6-10. fUML routine for termination of a timed model of a reactive system

Formalization functions	fUML or PSCS semantic model
$terminate_c()$	DEScheduler.stop();

```
/*Assumptions*/
On the co-simulation graph: no cycles exist in the co-simulation model, the graph connects a set of FMUs
with timed UML models of reactive systems
On UML components: zero step size allowed,
On the FMUs: the step size proposed by the master algorithm is accepted by all FMUs,
/*Co-simulation parameters*/
tc: Current simulation time
tstart: Start simulation time
tstop: Stop simulation time
hC: the step size of the component being simulated
hMASTER: the co-simulation step size
ti: the next event time in the FEL of the component being simulated
X: U∪Y : set of ordered ports variables computed by “Variables-order” algorithm
C= F ∪ TU where TU: the set of timed UML components of reactive systems
/*Instantiate and initialize components c ∈ C */
for each component c ∈ C:
        inst_c();
        init_c(tstart, tstop);
/*Step wise simulation*/
while (tc<tstop)
        for each input u ∈ X
                y = P(u);
                v = get_c(y);
                set_c(u,v);
                if (hC = 0)
                        doStep_c(0);
                end if;
        end for;
        hMASTER=min({hC, c ∈ F},{ti_c, c∈ TU});
        for each c ∈ C
                if (hC > 0)
                        doStep_c(hMASTER);
                end if;
       end for;
        tc=tc+ hMASTER;
end while;
/* Termination of the simulation*/
for each component c ∈ C:
        terminate_c() ;
end simulation;
```

Figure 6-15. Pseudocode of a master algorithm for a Co-simulation graph connecting FMUs with a timed model of a reactive system.

6.2.4. Pseudocode of the master algorithm

Figure 6-15 depicts the master algorithm we propose for the integration of timed UML models of reactive systems.

6.2.5. Experience on a representative example

6.2.5.1. Definition of the simulation scenario

The representative example consists of the definition and the simulation of a co-simulation scenario composed of FMU, the so-called 'Inc' (the same used in the section 5.1.4), and a timed UML model of a reactive system, so-called '*Controller'* presented in section 6.2.1.2.
The '*inc'* component in the Figure 6-16 is an instance of the imported FMU 'Inc'. For this scenario, we will use a default step size *'h=1'* since we are interested in, for reactive components, the instants at which the values of the inputs change. The counter (and therefore the input 'in'), in fact, is incremented each one unit of time.

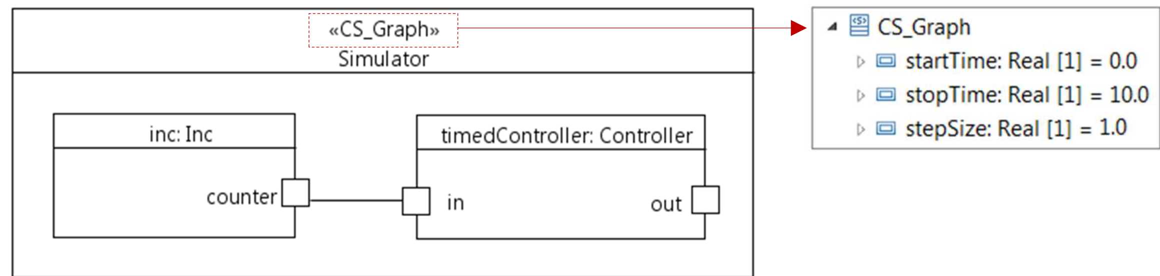

Figure 6-16. Co-simulation graph connecting an imported FMU to a timed model of a reactive system

The value of the 'counter' is propagated to the input 'in' of the *'timedController'*, which represents the 'Controller' class defined in 6.2.1.2. The default step size of the 'timedController' component is '$h_{UML}=1$' as indicated in the 'CS_FMU' stereotype in Figure 6-7.
This master should also take into account the internal time event of the component 'timedController'. This latter has a time event with a time stamp equal to 't_e=0,25' which is encountered when the value of the input 'in=true'. In this case, the reaction must be produced after 0.25 units of time instead of one.
The expected behavior of the *'timedController'* component, therefore, is to react to the input changes after an amount of time equal to the default step size (h=1) if 'in=false', and after an amount of time equal to the event time stamps when 'in=true'.
The co-simulation scenario will be simulated from 't_{start}=0' to 't_{stop}=10' with a default simulation step size 'h_{master}=1' as indicated in the 'CS_Graph' stereotype in Figure 6-16.

6.2.5.2. Simulation of the co-simulation scenario

Figure 6-18 and Figure 6-17 depict simulation results of this co-simulation scenario using the basic master as defined in the FMI standard (refer to section 2.1.2) and the advanced master algorithm we proposed in section 5.2.4 respectively.

The value of the counter reaches the threshold at 't=4'. At this instant 't=4', when executing a *dostep* on the 'timedController', the value returned by the control action is 'true'. In this case, the 'timedController is expected to produce a new setpoint at 't=4,25'. However, using the basic master algorithm given in section 2.1.2, all calls to doStep use the default step size 'h=1'. As a

result, there is a delay ‘d=0,75’ between the instant at which the new setpoint should be produced and the instant at which the new setpoint is actually produced. This delay is due to the ignorance of the master to the time event occurrence. Figure 6-17 demonstrates that this delay does not exist when using the master algorithm we proposed in section 5.2.4. The reaction of the ‘timedController’ is in fact produced at ‘t=4,25’.

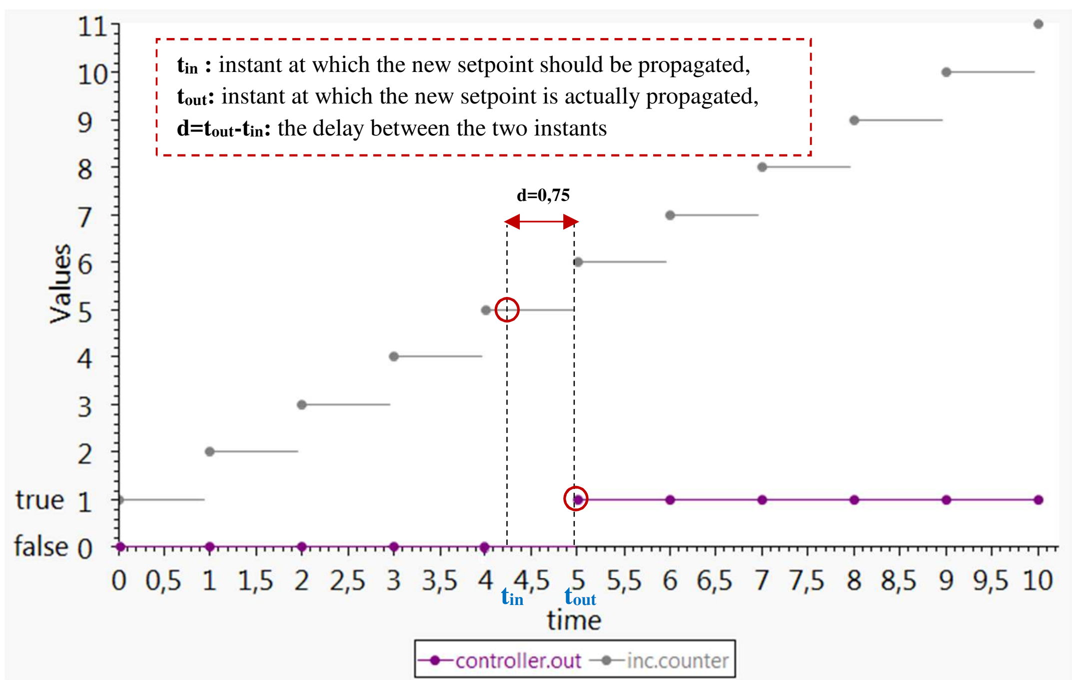

Figure 6-18.Co-simulation results of a timed model of a reactive system-Basic master

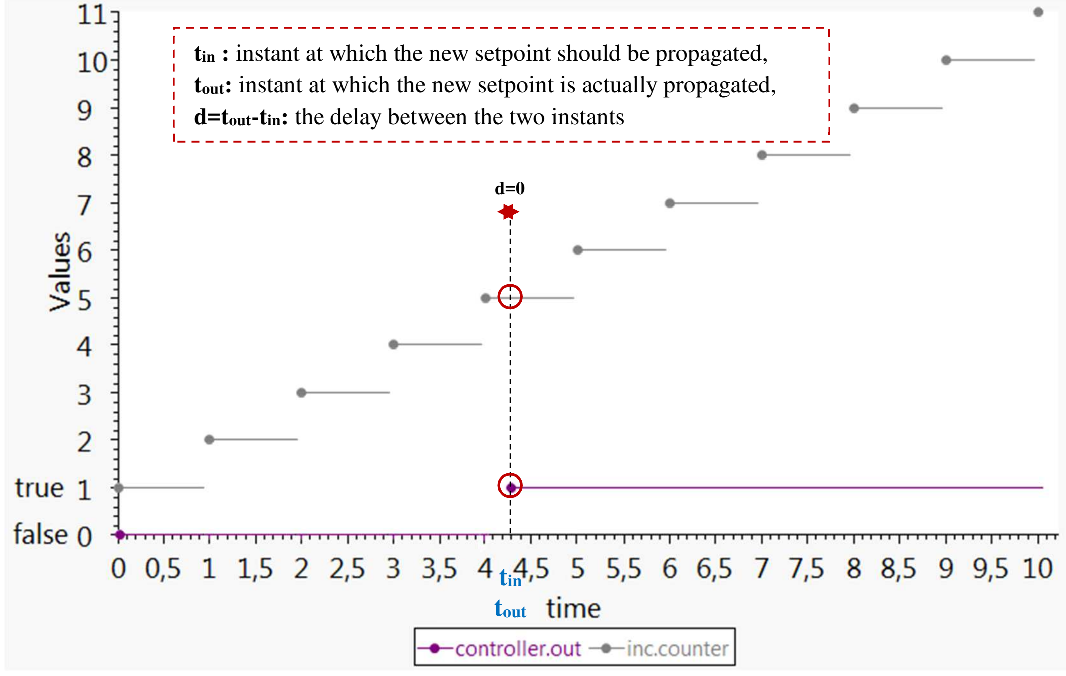

Figure 6-17. Co-simulation results of a timed model of a reactive system-Advanced master

6.3. Conclusion

In this chapter, we focused on the integration of timed UML models in FMI-based co-simulation. For each kind of systems, transformational and reactive systems, we provided rules for their modeling with UML in the context of the FMI standard, an adaptation between the execution semantics of UML models and the FMI API, and a master algorithm for the orchestration and synchronization of FMUs and UML components. The adaptation we proposed tackles, in particular, the semantic gap between DE semantics of fUML* and CT semantics of FMI.

PART III: EXPERIMENTS

Chapter 7: The Case Study: Energy auto-consumption management in smart energy building

Outline

7.1. Context
7.2. Specification of the case study
 7.2.1. The ‘ElectricLoad’
 7.2.2. The ‘ESS’
 7.2.3. The ‘ElectricityGrid’
 7.2.4. The ‘PV’
 7.2.5. The ‘ControlSelfConsumption’
7.3. Modeling of the case study in Papyrus
 7.3.1. Modeling of FMUs in Papyrus
 7.3.2. Modeling of ‘SelfConsumptionControl’ component in Papyrus
  7.3.2.1. Basic Self-Consumption Control
  7.3.2.2. Advanced Self-Consumption Control
7.4. Simulation of the case study in Papyrus/Moka
 7.4.1. The basic control scenario
  7.4.1.1. Definition of the co-simulation scenario
  7.4.1.2. Simulation results
 7.4.2. The advanced control scenario
  7.4.2.1. Definition of the co-simulation scenario
  7.4.2.2. Simulation results
 7.4.3. Interpretation of the simulation results

In this chapter, we aim at the validation of our approach. The validation is done by comparing co-simulation results of the case study using our approach in Papyrus/Moka with co-simulation results of the case study in Simulink. We will demonstrate how the approach we propose achieves better results. The chapter is organized as follows: Sections 7.1 and 7.2 introduces the case study we will use for our experimentations. Section 7.3 tackles the modeling of the case study in Papyrus following the rules defined in Chapters 4, 5 and 6, and finally, section 7.4 exposes the simulation results in Papyrus with an evaluation against the simulation results in Simulink.

7.1. Context

Smart grid is a concept involving an electricity grid that delivers electric energy using communications, control, and computer technology for lower cost and superior reliability [15].

The overall load in smart grids is not stable, as electricity demand varies significantly over the course of a day. High energy consumption could generate grid disruption and black-out which are very costly for energy providers and very uncomfortable for users. As a consequence, smart grids need smart consumption management system. Smart buildings are connected with energy management devices over communication networks to better monitor energy consumption and production.

Reducing peak demand and overall consumption are two of the most significant strategies of smart consumption management. Furthermore, real-time information transmitted over communication networks allows power outage anticipation, as well as service perturbation detection. By rapidly detecting and analyzing data coming from the distribution network, the smart grid can take corrective actions, so as to restore power stability when needed. Mathematical algorithms have been designed to predict power consumption increases, so that corrective actions can be taken.

In this context, the purpose of the system under study is the verification of the auto-consumption management strategy in a smart energy building. The case study is provided by the CEA/Liten whose goal is the definition of a UML-based language for the definition of new control strategies[26] for the system under design and their verification. As a first step, we need to validate the co-simulation approach we proposed in Papyrus since it will be used later as the simulation environment for the verification of new defined strategies. The purpose of the experimentations is to perform this preliminary validation.

The system under study is called 'Energy Auto-consumption' and was originally designed as a Matlab/Simulink model. The validation of the approach consists in modeling and simulating already defined control strategies in Papyrus, and in comparing the obtained simulation results with the simulation results in Simulink. We begin with the specification of the case study in the following section.

7.2. Specification of the case study

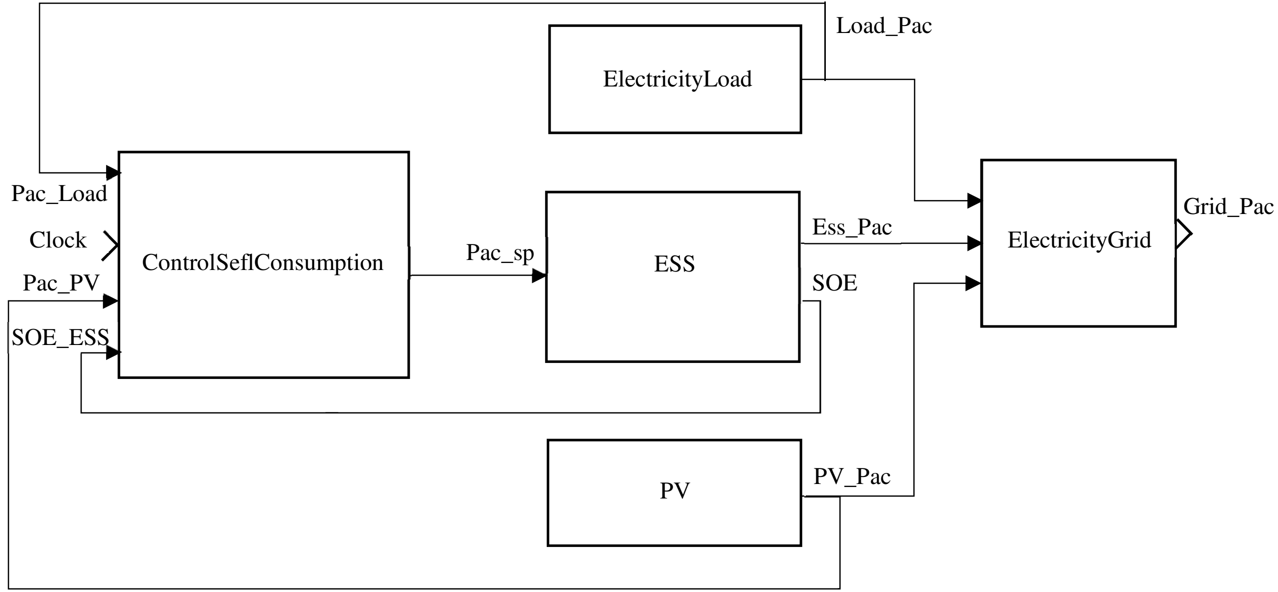

Figure 7-1. The 'Energy Auto-consumption' system

[26] A control strategy is an algorithm that computes a control signal to be followed by the system for the regulation of a system functionality

The 'Energy Auto-consumption' system is composed of five components as depicted in Figure 7-1. They are specified in the following subsections.

7.2.1. The 'ElectricLoad'

This component simulates the energy consumption of a building.

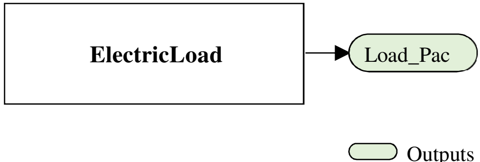

Figure 7-2. The 'ElectricLoad' component structure

- Output:
    - Load_PAC: is the amount of energy consumed by the building at a given instant.

7.2.2. The 'ESS'

This component simulates the operation of a storage unit. It is responsible for injecting energy in the electrical network when the power in this later decreases.

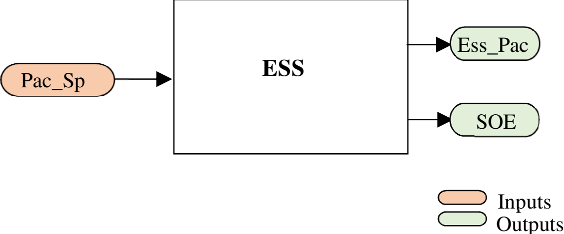

Figure 7-3. The 'ESS' component structure

- Inputs
    - PAC_Sp: received from the 'ControlSelfConsumption' component and indicates whether the storage unit should be charged or discharge.
- Output:
    - ESS_PAC: the energy injected in the electrical network.
    - SOE: the state of charge of the storage unit.

7.2.3. The 'ElectricityGrid'

This component simulates the electric network to which the storage unit and the building are connected. It is responsible for delivering electricity from producers to consumers.

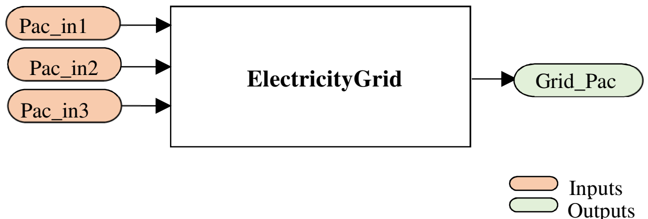

Figure 7-4. The 'ElectricityGrid' component structure

7.2.4. The ‘PV’

This component simulates the photovoltaic unit. It produces energy and injects it into the electrical network.

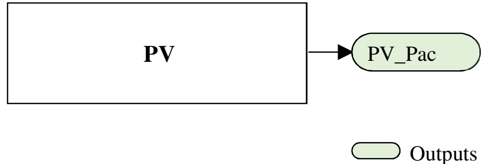

Figure 7-5. The ‘PV’ component structure

- Output:
  - PV_PAC: is the amount of energy produced by the photovoltaic panels.

7.2.5. The ‘ControlSelfConsumption’

This component encapsulates an energy consumption control logic. It monitors the ESS component based on information about the energy production and consumption of the whole system. Two variants of control are proposed: a basic self-consumption control and an advanced self-consumption control.

7.2.5.1.Basic Self-consumption Control

a. Structure

The basic self-consumption control receives as inputs information about the energy consumption of the building (received from the ‘ElectricLoad’ component on the input ‘Pac_load’), and information about the energy production (received from the ‘PV’ component on the ‘Pac_PV’ input). Then, it simply computes the output ‘ESS_Pas_Sp’ which will be communicated to the‘ESS’ component.

Figure 7-6. The basic ‘ControlSelfConsumption’ component structure

- Inputs
  - PAC_LOAD: received from the ‘ElectricalLoad’ component. It represents the instantaneous power consumed by the building.
  - PAC_PV: received from the ‘PV’ component. It represents the instantaneous power produced by the photovoltaic unit.
- Output:
  - ESS_PAC_sp: the control signal to be delivered to the environment of this component. It indicates instantaneous charged or discharged power.

b. Behavior

The basic self-consumption control behavior simply computes the difference of the energy consumed by the building and the energy produced by the photovoltaic panels. This information

indicates whether the grid is a consumer or a producer, and is used by the 'ESS' component to determine whether the storage unit should deliver (discharge) or not deliver (charge) power to

```
/*Inputs*/
PAC_Load: instantaneous power consumed by the building,
PAC_PV: instantaneous power produced by the photovoltaic unit,

/*Outputs*/
ESS_PAC_sp: instantaneous charged or discharged power.

/*The control strategy algorithm*/
        ESS_PAC_sp = PAC_Load – PAC_PV;
```

Figure 7-7. The basic 'ControlSelfConsumption' component behavior

the electrical network. Figure 7-7 is the specification of the control logic of the basic 'controlSelfConsumption' component.

7.2.5.2.Advanced Self-Consumption Control

a. Structure

The advanced 'ControlSelfConsumption' component requires more inputs from the other components compared to the basic control. It has four inputs, three parameters and one output as depicted in Figure 7-8.

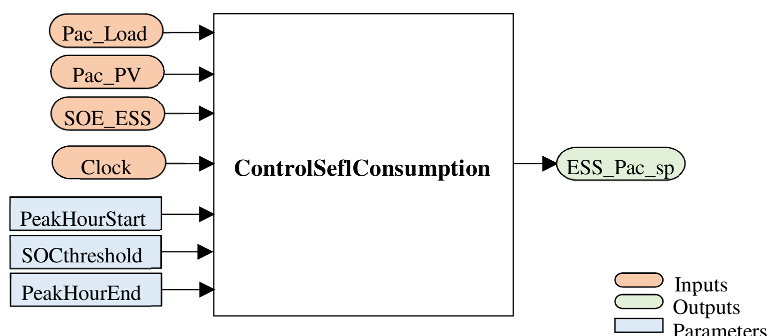

Figure 7-8. The 'ControlSelfConsumption' component structure in Simulink

- Inputs
  - PAC_LOAD: received from the 'ElectricalLoad' component. It represents the instantaneous power consumed by the building.
  - PAC_PV: received from the 'PV' component. It represents the instantaneous power produced by the photovoltaic unit.
  - SOE_ESS: received from the 'ESS' component. It indicates the state of charge of the storage unit.
  - Clock: the current time of the day.
- Parameters
  - SOC_THRESHOLD: the threshold from which the discharge is no longer authorized.
  - PEAK_HOUR_START: the lower bound of the interval 'peak hours'.

- PEAK_HOUR_END: upper bound of the interval 'peak hours'.

- Output:
    - ESS_PAC_sp: the control signal to be delivered to the ESS component. It indicates instantaneous charged or discharged power.

The control signal computed by the advanced 'ControlSelfConsumption' component depends not only on the inputs received from its environment (the components connected to its inputs), but also on the period during which the system operates. The controller requires information about the current time to check whether the system operates, or does not operate, in peak hours represented by the Clock input in Figure 7-8. When it is information about simulation, the time is determined with reference to the simulation time. This information may be provided by a component responsible for indicating whether the system operates in the 'peak hours' period or in 'off-peak hours' based on the current simulation time.

In Papyrus, we replace the Clock input (Figure 7-8) with PeackHours input (in Figure 7-10). The value of this latter is provided by the 'PeakHoursIndicator' component shown in Figure 7-9. It indicates whether the current time is in the Peak Hours period or not based on the parameters 'PeakHourStart' and 'PeakHourEnd'.

The 'PeakHoursIndicator' component has two parameters and one output as follows:

- Paramaters:
    - PeakHourStart: the start time of the peak hours period.
    - PeakHourEnd: the stop time of the peak hours period.

- Output
    - PeakHours: delivered to the ControlSelfConsumption components to give information about the period (peak hours or off-peak hours) of the system operation.

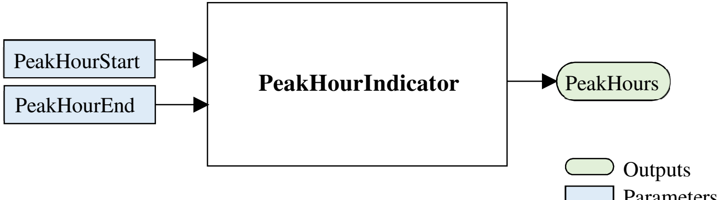

Figure 7-9. PeakHourIndicator component Structure

The output of the 'PeakHoursIndicator' component is given as an input to the control 'SelfConsumptionControl'. Some changes in the structure of this latter are therefore required (compared

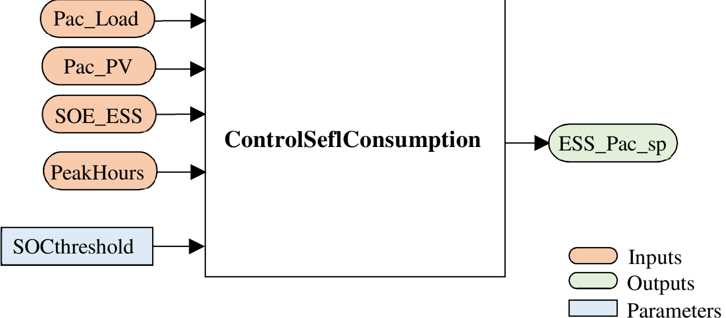

Figure 7-10. ControlSelfConsumption component structure in Papyrus

to the structure represented in Figure 7-8). The new structure of the advanced 'ControlSelfConsumption' is depicted in Figure 7-10.

b. Behavior

The control logic monitors the ESS component. It determines whether the storage unit should be charged or discharged while accounting for the state of the system as well as the current period of the day (i.e, peak hours or off-peak hours).

Figure 7-11 specifies the logic of the 'PeakHoursIndicator' component with a simple algorithm.

```
/*Parameters*/
peakHourStart: the start hour of the peak hours period
peakHourEnd: the stop time of the peak hours period
/*Outputs*/
peakHours: the verdict
/*Global Variables */
hour: the current simulation time
/*Initializations*/
peakHours = false

/*The logic algorithm*/
When (hour == peakHourStart)
      peakHours = true;
end when;
When (hour == peakHourEnd)
      peakHours =  false;
```

Figure 7-11. The 'PeakHoursIndicator' component behavior

The conditions for the charge and the discharge of the storage unit are as follows:

- The charge condition: the ESS should be charged when the current time does not correspond to peak hours' period and the power consumed by the building does not exceed the power produced by the photovoltaic unit.
- The discharge conditions:
  - o During peak period, the storage unit delivers the power available on the power grid.
  - o During off-peak period, the storage unit delivers the available power only if the power consumed by the building (the 'Pac_Load' input) exceeds the power produced by the photovoltaic unit (the 'Pac_pV' input) and the charge rate of the storage unit remains greater than the limit imposed by the control.

Figure 7-12 is the specification of the control logic of the 'ControlSelfConsumption' component conforming to these conditions.

```
/*Parameters*/
SOC_Threshold: the threshold from which the discharge is no longer authorized
/*Inputs*/
PeakHours = indicates whether the system operates or not in peak hour period
PAC_Load: instantaneous power consumed by the building,
PAC_PV: instantaneous power produced by the photovoltaic unit,
SOE_ESS: state of charge of the storage unit,
/*Outputs*/
ESS_PAC_sp: instantaneous charged or discharged power.

/*The control strategy algorithm*/
if (! PeakHours) then          // off-peak hours
      if (PAC_Load > PAC_PV) then          //discharge
           if (SOE_ESS > SOC_Threshold) then
                   SOE_PAC_sp = PAC_Load – PAC_PV;          //possible to discharge
           else  //case charge
                   ESS_PAC_sp = 0;          //discharge forbidden
           end if;
else        // peak hours
       ESS_PAC_sp = PAC_Load – PAC_PV;
end if;
```

Figure 7-12. The advanced ‘ControlSelfConsumption’ component behavior

In the following section, we focus on the modeling of the case study in Papyrus. As stated previously in section 7.1, we are interested in the modeling of control strategies with UML and their verification in the co-simulation environment we proposed in the contribution part. For this, we will particularly focus on the modeling of the ‘control self-consumption’ component with UML with respect to the modeling rules we defined in Chapter 5 and Chapter 6, and on its co-simulation with the other components using the master algorithms we proposed in Chapter 5 and 6.

7.3. Modeling of the case study in Papyrus

In the co-simulation scenario we propose, all components, except the ‘self-ConsumptionControl’, are imported FMUs and rely on the CT MoC. In this section, we will focus on the modeling of the different components in Papyrus.

7.3.1. Modeling of FMUs in Papyrus

The components ‘ElectricLoad’, ‘ESS’, ‘ElectricityGrid’ and ‘PV’ were originally designed with Matlab/Simulink and rely on CT MoC. They are exported as FMU for co-simulation (in Simulink), then imported into Papyrus. The FMUs are represented as UML classes annotated with stereotypes from the Co-Simulation profile as explained in Chapter 4.
Figure 7-16, Figure 7-14, Figure 7-13 and Figure 7-15 represent the result of the import of the FMUs ‘ESS’, ‘Grid_3inputPac’, ‘Load’ and ‘PV’s respectively.

ESS
«CS_FMU» ESS
<Generalization> Fmu2Proxy
SOCini : Real
chargingEfficiency : Real
chargingPmax : Real
dischargingEfficiency : Real
dischargingPmax : Real
nomEnergy : Real
Pac_sp : Real
Pac : Real
SOE : Real
«OutputDependency» <Dependency> SOCini
«OutputDependency» <Dependency> SOCini
«OutputDependency» <Dependency> SOCini
«OutputDependency» <Dependency> SOCini

«CS_FMU»
ESS
«Parameter» SOCini: Real
«Parameter» chargingEfficiency: Real
«Parameter» chargingPmax: Real
«Parameter» dischargingEfficiency: Real
«Parameter» dischargingPmax: Real
«Parameter» nomEnergy: Real
«CS_Port» Pac_sp: Real
«CS_Port» Pac: Real
«CS_Port» dischargingPmax: Real
«CS_Port» chargingPmax: Real
«CS_Port» SOE: Real

Figure 7-15. The 'ESS' FMU imported in Papyrus

Grid_3inputsPac
«CS_FMU» Grid_3inputsPac
<Generalization> Fmu2Proxy
Gain_Gain : Real
Pac_in1 : Real
Pac_in2 : Real
Pac_in3 : Real
Pac_grid : Real
«OutputDependency» <Dependency> Gain_Gain
«OutputDependency» <Dependency> Pac_in1
«OutputDependency» <Dependency> Pac_in2

«CS_FMU»
Grid_3inputsPac
«Parameter» Gain_Gain: Real
«CS_Port» Pac_in1: Real
«CS_Port» Pac_in2: Real
«CS_Port» Pac_in3: Real
«CS_Port» Pac_grid: Real

Figure 7-13. The 'Grid_3inputsPac' FMU imported in Papyrus

Figure 7-14. The 'Load' FMU imported in Papyrus

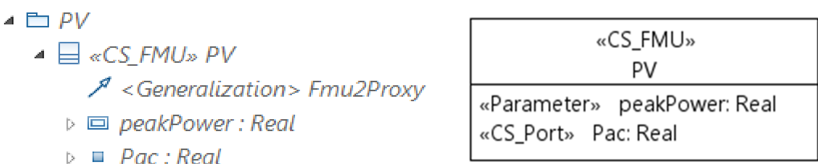

Figure 7-16. The 'PV' FMU imported in Papyrus

7.3.2. Modeling of 'SelfConsumptionControl' component in Papyrus

7.3.2.1. Basic Self-Consumption Control

The behavior of the basic 'controlSelfConsumption' is stateless and does not depend on time. It can be specified as an untimed model of a transformational system. We apply the modeling rules defined in section 5.1.1.

The basic 'Self-ConsumptionControl' is represented with a passive class to which the "CS_untimed" stereotype is applied (Figure 7-17). The class owns two input ports 'Pac_Load' and 'Pac_PV', one output port 'Pac_PV', and an operation 'controlConsumption' annotated with the "CS_Operation".

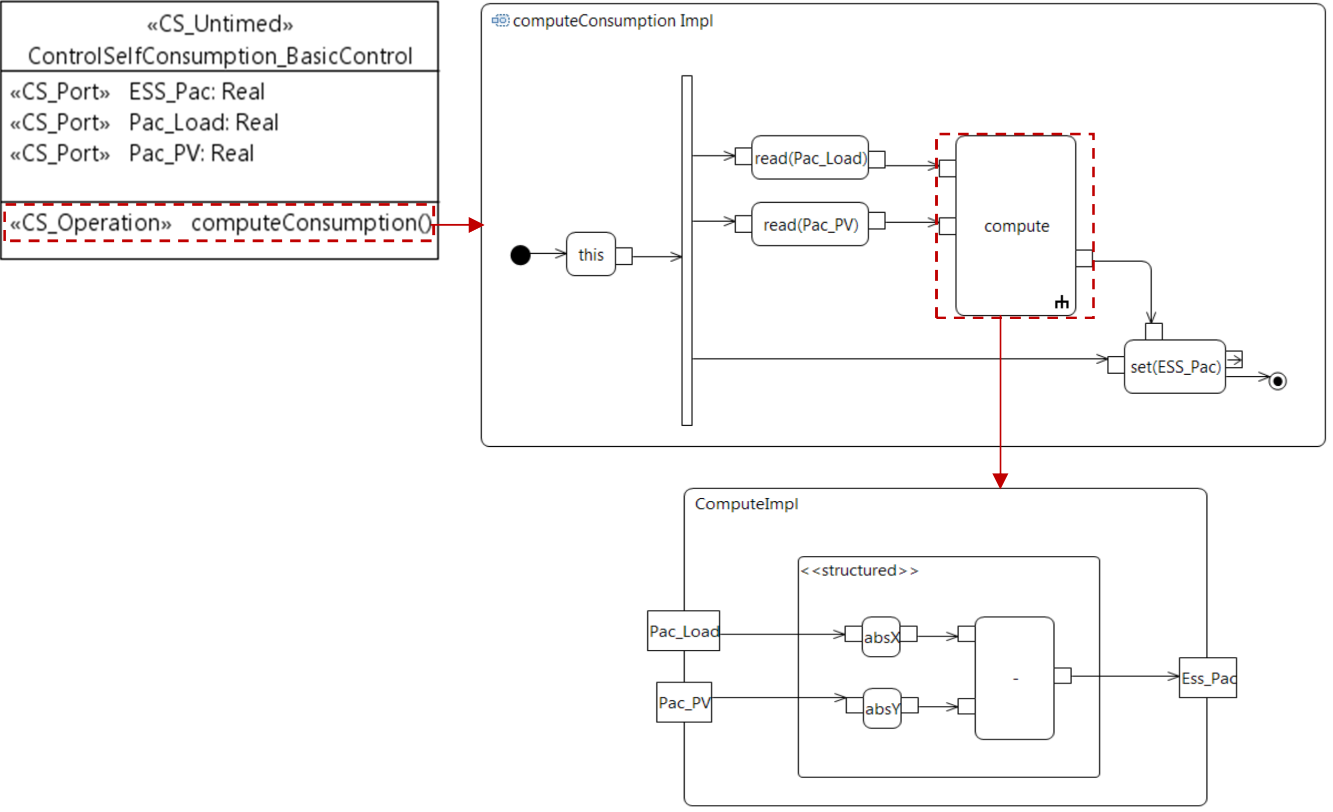

Figure 7-17.UML Model of the basic 'SelfConsumptionControl' - Structure and Behavior

The operation 'controlConsumption' implements the logic represented in Figure 7-7. It represents the computations to execute at each simulation step size (when the component is invoked by the master). It is specified and implemented with the activity 'computeConsumption Impl'. This later reads the inputs 'Pac_Load' and 'Pac_PV' ('read(Pac_Load)' and 'read(Pac_PV)' actions in the activity). Then, it calculates the difference between the absolute values of these values (the 'compute Impl' activity). The output is finally set with the result ('set(ESS_Pac)' in the activity)

7.3.2.2.Advanced Self-Consumption Control

The advanced 'SelfConsumptionControl' component is composed of two components. The 'PeakHourIndicator' produces an output at discrete instants and is reactive to time events. The corresponding UML model is therefore a timed UML model for a reactive system as depicted in Figure 7-18. The 'PeakHourIndicator' owns one output port 'peakHour' and a classifier behavior implementing the logic represented in Figure 7-11.

The peak hours start at 7p.m and terminate at 9p.m. each day. For the first simulation day, these two instants are represented with *AcceptEventActions*$^{(syn)}$ triggered by time events 'at(19)' (triggered by an absolute time event) and 'after(2)' (triggered by a relative time event).

‘after(22)’ is an *AcceptEventActions*$^{(syn)}$ triggered by an absolute time event. It represents the start time of the peak hours for the next day.

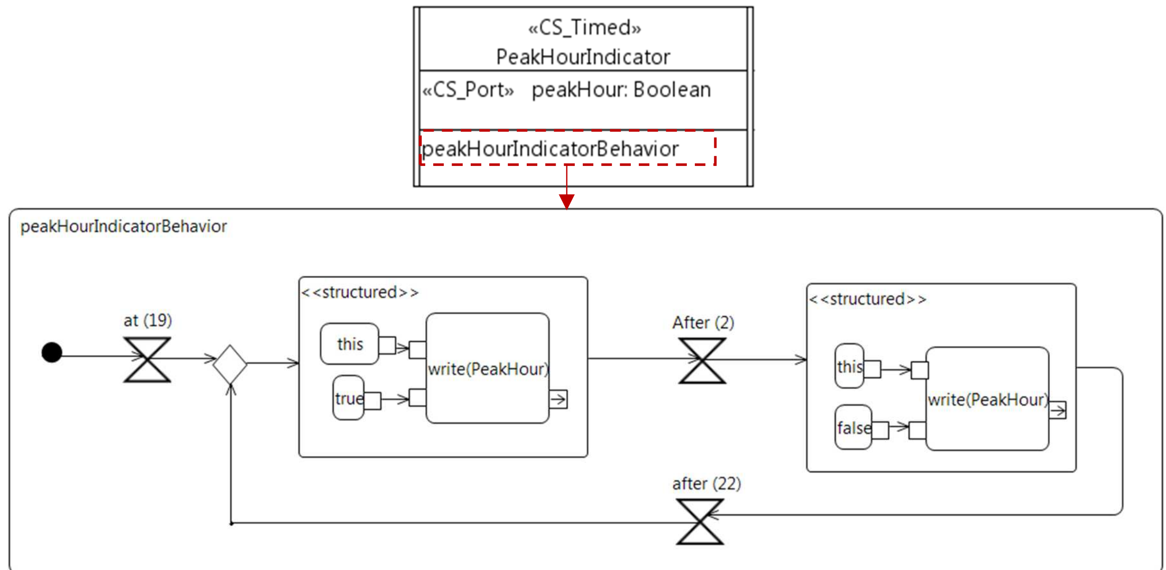

Figure 7-18. Modeling the Peak-up indicator component with UML

The behavior of the advanced ‘SelfConsumptionControl’ component is time independent and reactive to changes. The behavior specifying the control strategy reacts only if one of the inputs values changes. The behavior is time independent. The corresponding UML model is therefore an untimed UML model of a reactive system.

We apply the modeling rules defined in section 5.2.1. The advanced ‘ControlSelfConsump-tionn’ component is an active class to which the stereotype “CS_untimed” is applied. The class owns four input ports ‘ESS_Pac’, ‘Pac_Load’, ‘Pac_PV’ and ‘peakHour’, an output port ‘SOE_ESS’, and a classifier behavior which is executed once from the beginning to the end of he simulation. The classifier behavior is represented with the UML activities in Figure 7-19. It implements the control logic specified in Figure 7-12.

The activity ‘Advanced self-consumption control behavior’ reacts to the change of at least one input. This behavior is represented with an *AcceptEventAction*$^{(syn)}$ triggered by change events (for each input is defined a change event). Once a change is detected, the activity reads the input ports values (‘read(Pac_Load)’, ‘read(Pac_PV)’, ‘read(SOE_ESS)’ and ‘read(PeakHour’)). Then, it performs some computations implemented with the activity ‘controlConsumptionImpl’ as specified in the control logic of Figure 7-12. At the end of the computations, it sets the value of output port ‘ESS_Pac’ with the new state of charge of the storage unit (‘write(Ess_Pac)’).

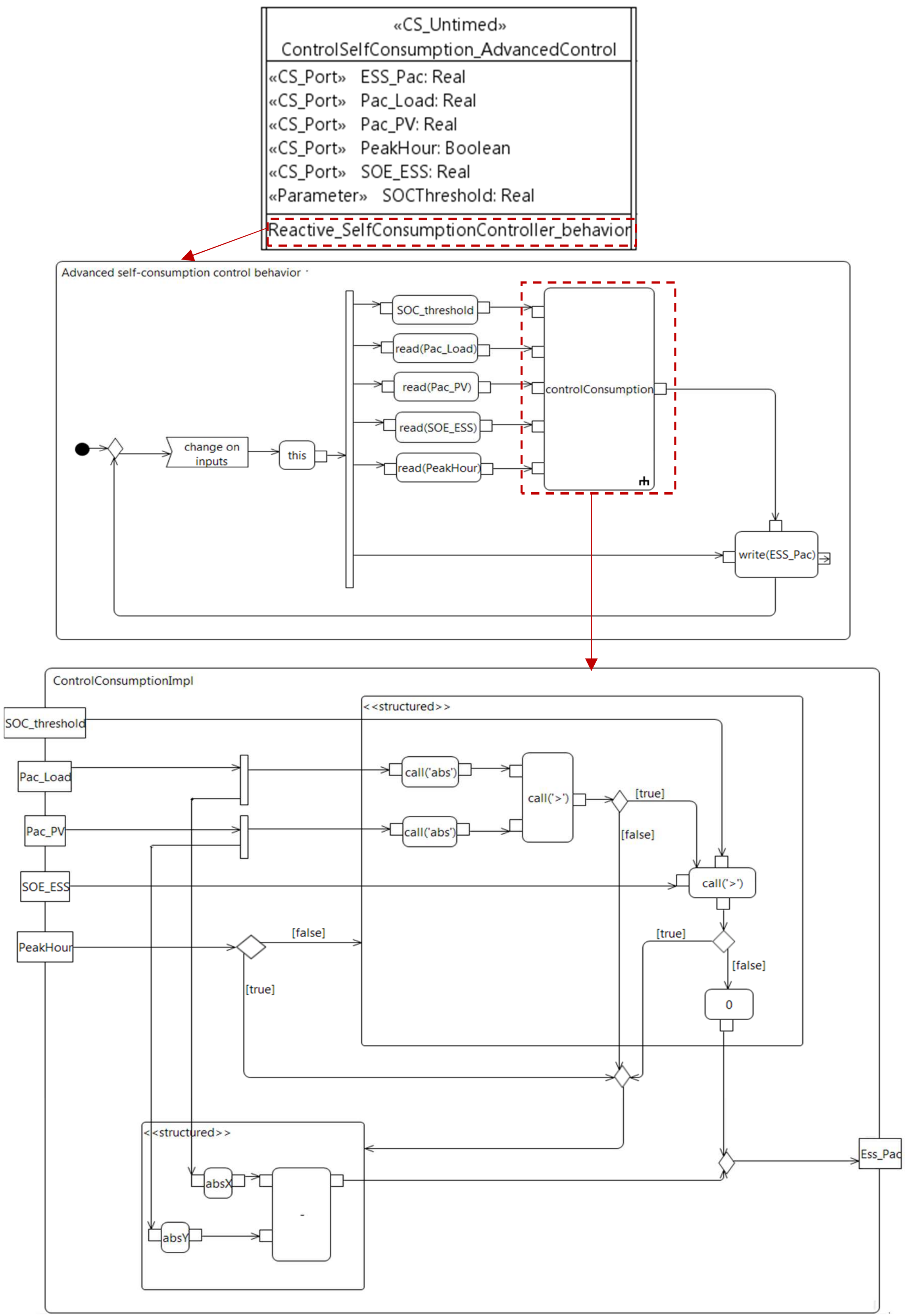

Figure 7-19.UML model of the advanced self-consumption control – Structure and Behavior

7.4. Simulation of the case study in Papyrus/Moka

The verification of smart grids models necessitates their simulation for long operation time (many days, or even weeks and months). A small simulation step size (for example h=10s) makes the simulation highly costly. For this reason, we propose to simulate the case study with a step size 'h=1hour'.
The goal of the simulation is to verify the operation of the storage unit (the output 'ESS_Pac' of the 'ESS' component) and the energy production/consumption in the electricity grid (the output Pac_Grid of the 'ElectricityGrid') for a given photovoltaic production and a given load consumption.

We expose simulation results of one operation day for two co-simulation scenarios: a basic control scenario using the basic 'SelfConsumptionControl' component (section 7.4.1) and an advanced control scenario (section 7.4.2) using the advanced 'SelfConsumptionControl' component.

7.4.1. The basic control scenario

7.4.1.1. Definition of the co-simulation scenario

The basic control scenario connects the imported FMUs 'Load', 'ESS', 'PV' and 'electricityGrid' to the basic 'SelfConsumptionControl' component specified in Figure 7-17.

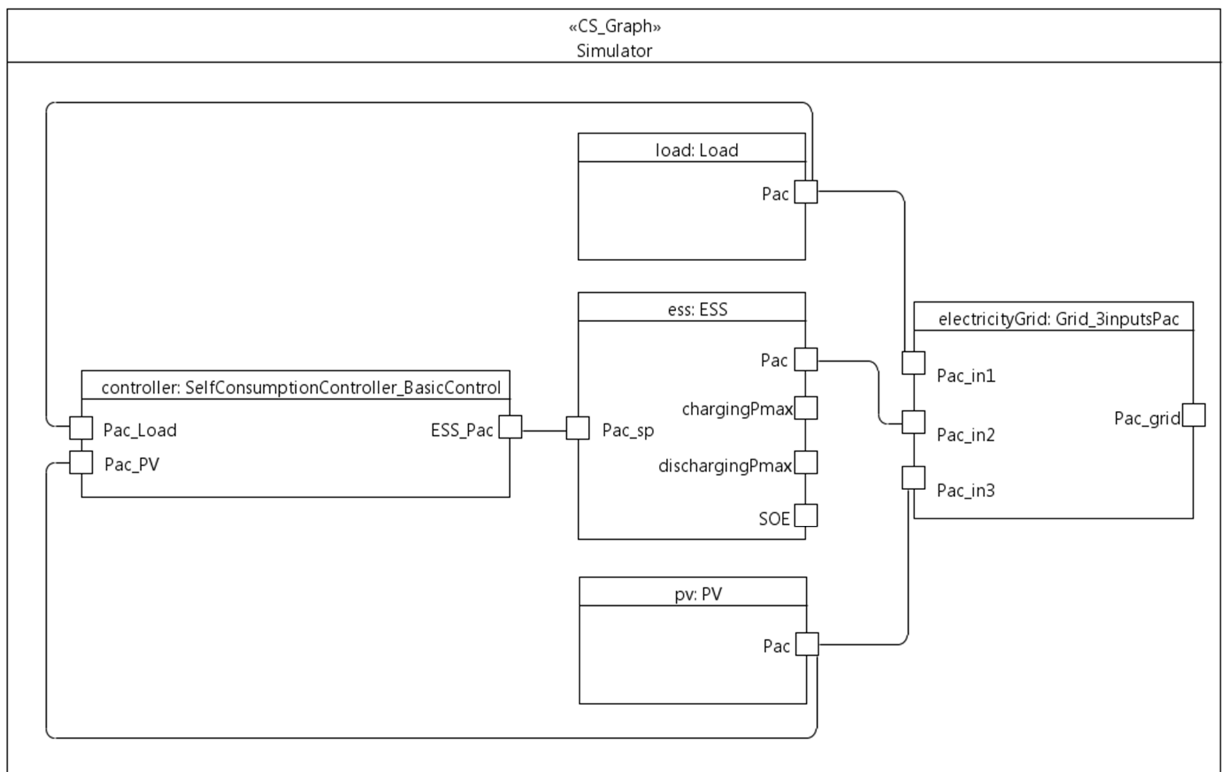

Figure 7-20. Co-simulation scenario connecting FMUs to the basic 'SelfConsumptionControl'

The co-simulation scenario of Figure 7-20 is simulated for one operation day from 't_{start}=0' to 't_{stop}=24' with a default simulation step size 'h_{master}=1' (1hour). The simulation results are given in the next subsection.

7.4.1.2.Simulation results

Figure 7-21 and Figure 7-22 depict simulation results of the case study in Simulink and in the co-simulation environment we proposed in Papyrus, respectively.

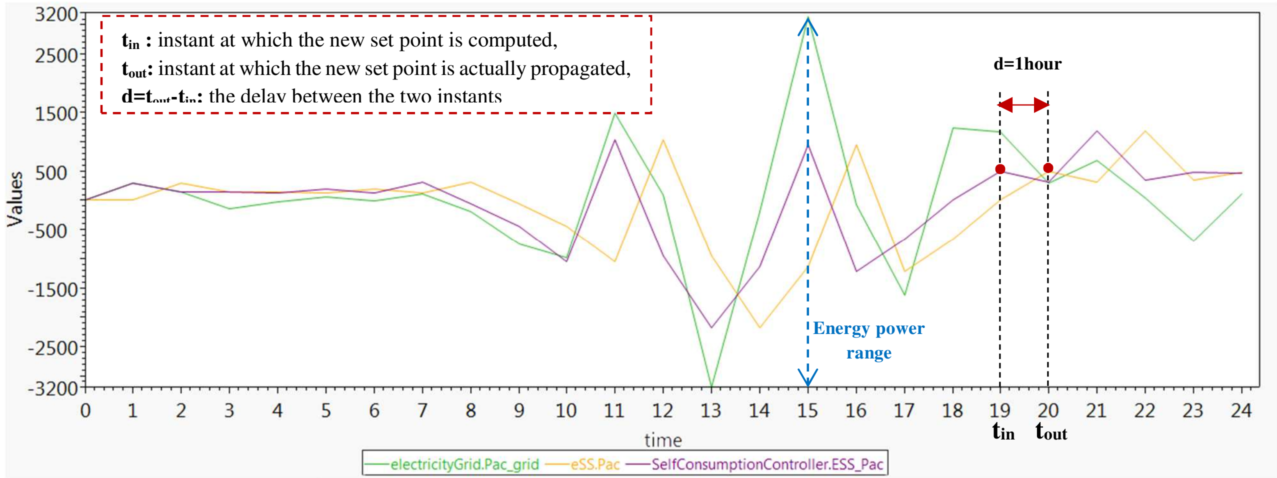

Figure 7-21. Simulation results of the basic control scenario in Simulink

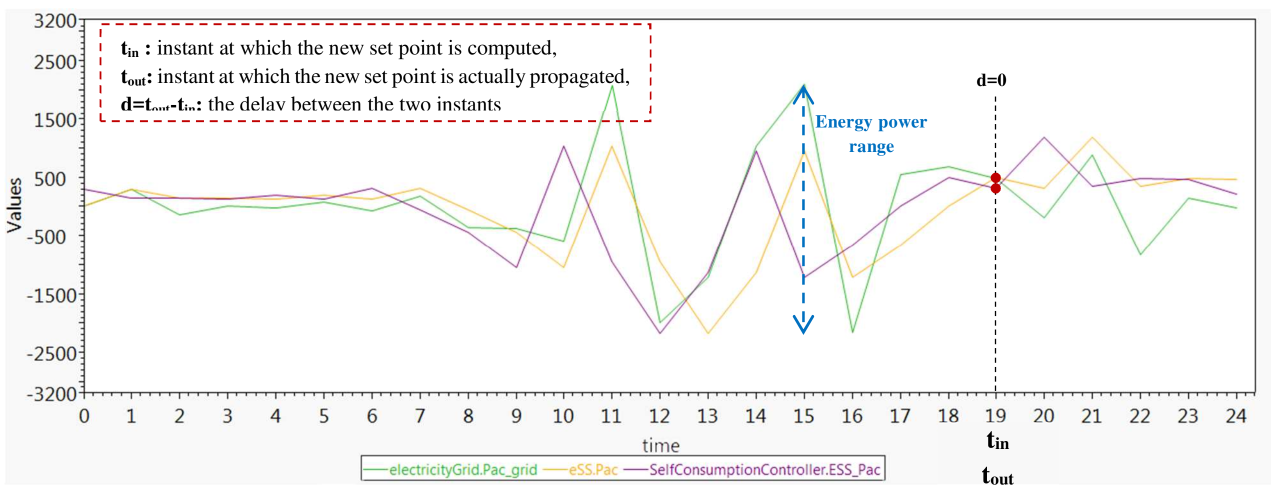

Figure 7-22. Simulation results of the basic control scenario in Papyrus

7.4.2. The advanced control scenario

7.4.2.1.Definition of the co-simulation scenario

The advanced control scenario is a co-simulation scenario which connects the imported FMUs 'Load', 'ESS', 'PV' and 'electricityGrid' to the advanced 'SelfConsumptionControl' component and the 'PeakHourIndicator component specified in Figure 7-19 and Figure 7-18.

The co-simulation scenario of Figure 7-23 is simulated for one operation day from 't_{start}=0' to 't_{stop}=24' with a default simulation step size 'h_{master}=1' (1hour). The co-simulation results are given in the next subsection.

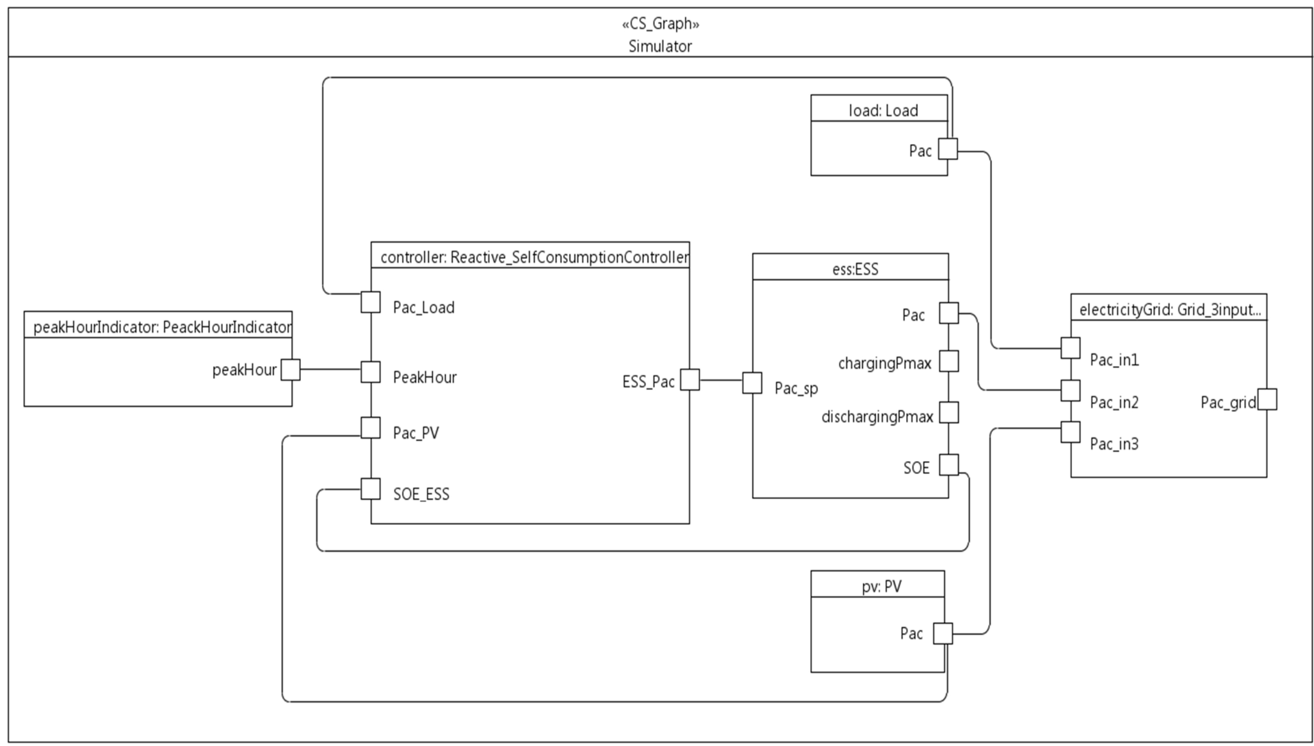

Figure 7-23. Co-simulation scenario connecting FMUs to the advanced 'SelfConsumptionControl'

7.4.2.2.Simulation results

Figure 7-24 and Figure 7-25 depict simulation results of the case study in Simulink and in the co-simulation environment we proposed in Papyrus, respectively.

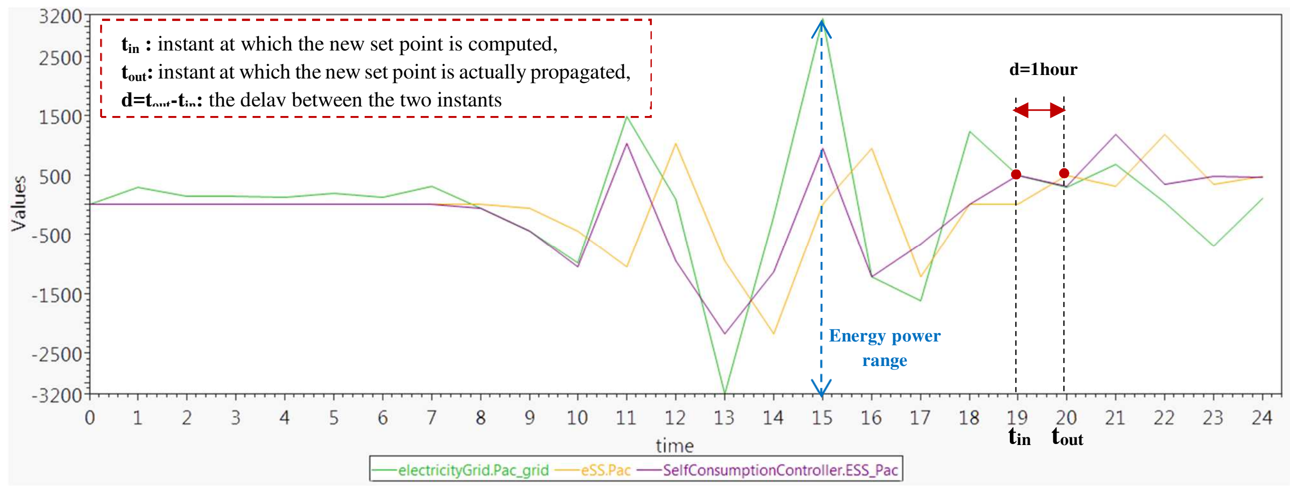

Figure 7-24. Simulation results of the advanced control scenario in Simulink

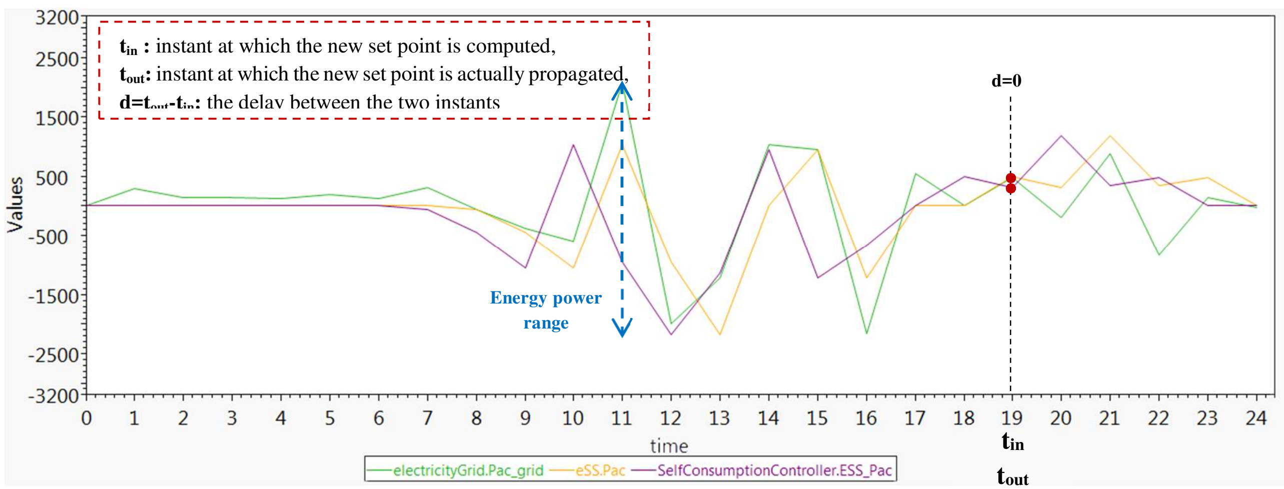

Figure 7-25. Simulation results of the advanced control scenario in Papyrus

7.4.3. Interpretation of the simulation results

The simulation results in Papyrus demonstrate that, using the master algorithm we proposed in the contribution part, we are able to:

- Get information about the energy power that should be delivered by the storage unit to the electricity network as well as the state of charge of the storage unit earlier. As illustrated in Figure 7-22 and Figure 7-21 (respectively Figure 7-24 and Figure 7-25) the new set point value (the output ‘Ess_Pac’) as well as the state of charge and the amount of energy to be injected in the network (the output ‘Pac’ of the ‘ESS’ component) are computed and propagated before one hour compared to the results in Simulink. This action ensures a better operation of the smart grid. In fact, we are able to avoid outages by anticipating energy demands (delivering the required energy power on time), and maintaining an acceptable state of charge by charging the storage unit as fast as possible when the level is under the required threshold.
- Minimize and maintain a more homogeneous rate of energy power distribution in the electricity grid. As illustrated in Figure 7-22 and Figure 7-21 (respectively Figure 7-24 and Figure 7-25), using the master algorithm we propose in Papyrus, the energy power range is smaller than when simulating in Simulink. This enables a better dimensioning and calibration of the system, and therefore, minimization of the system conception cost, a very important factor for energy providers.

7.5. Summary of the proposition validation

The objective of this chapter was to validate the approach we proposed: the integration of UML models in FMI-based co-simulation using the adaptation of semantics at master level. The approach was applied to two variants of a system managing the energy auto-consumption in smart grids, system including a simple control and system including an advanced control. Firstly, we modeled the case study in Papyrus. For this, we imported FMUs representing the physical part of the system and we modeled the cyber part with UML by referring to modeling rules defined in chapters 5 and 6. Secondly, we simulated the case study in Papyrus/Moka using the master algorithms we proposed in chapters 5 and 6. We demonstrated that the approach provides solutions to the Issue 1 and Issue 2 related to limitations of FMI for CPS domain (i.e, no support

to untimed semantics and time events). Finally, we compared the obtained results to the co-simulation results of the case study in Simulink. The comparison demonstrated that the proposed approach achieves better results. The master algorithm we propose enables to anticipate the demands of energy and to maintain a homogeneous rate of energy power distribution.

PART IV: CONCLUSION AND PERSPECTIVES

Conclusion and perspectives

In this chapter we summarize the motivation of this work including its context, we remind the main points of the proposition including its validation, and then we propose some follow-up possibilities.

❖ Summary of the thesis work

- **Context and motivations**

The verification of Cyber Physical Systems (CPS) requires the integration of heterogeneous models. These models differ in the way the components interact with their environment, execute their behavior and manage time and events. The main issue is determining the global behavior of the composed model where the coordination and the synchronization between the involved sub-models are required. Techniques for the verification of heterogeneous systems were found in the literature and evaluated for their applicability to CPS. The co-simulation is best suited for the simulation of CPS. FMI standard, in particular, was proposed as a standard for co-simulation. This standard is gaining popularity in the industry and is supported by many modeling and simulation tools.

Although FMI provides a standard interface for co-simulation of models from different languages/tools, it does not provide efficient solutions to cope with the heterogeneity of the involved MoCs. FMI was originally intended for co-simulation of physical processes, with limited support for other MoCs such as DE and Data Flow. These MoCs are commonly used to model the logic of cyber part of a CPS. We are particularly interested in UML, which is the reference standard for software modeling. Unfortunately, none of the present-day FMI-based co-simulation solutions consider UML models.

- **Approach and contribution**

The objective of this work was to propose an FMI-based co-simulation environment for CPS with integration of UML models. Three techniques for the integration of UML models in FMI-based co-simulation were evaluated for their feasibility and efficiency. We chose the adaptation of semantics at master level technique. This later consists of the simulation of co-simulation scenarios connecting black boxes FMUs with white boxes UML models, and requires adaptation of semantics between the FMI API and the execution semantics of UML models. This technique enables us to benefit from the FMI standard for co-simulation purposes, and to valorize the use of the UML language for the modeling of the software components of CPSs. In addition, we are not constrained by the inadequacies of the FMI standard in terms of its support for non-CT models.

UML models need to be executable for their integration in co-simulation approaches. We chose to base our work on fUML* standard which define precise execution semantics for a subset of

UML and provide an interesting and formal basis for the integration of UML models in FMI-based co-simulation. We identified different kinds of systems we would like to model and simulate with UML then evaluated the feasibility of their modeling and simulation with fUML*. These systems are classified according to two dimensions found in the literature: (a) systems can be transformational or reactive, (b) systems can be untimed or timed. As a result, we obtained four kinds of models we would like to integrate in FMI-based co-simulation:

- Untimed UML models for a transformational system,
- Timed UML models for a transformational system,
- Untimed UML models for a reactive system; and
- Timed UML models for a reactive system.

As a first step of the contribution, we set up a master simulation tool where we implemented the FMI standard in a UML tool. This latter proposes the definition of co-simulation scenarios connecting a set of imported FMUs and master algorithms for their simulation. Then, we proposed an incremental extension to this framework with the support of new co-simulation scenarios connecting black boxes FMUs with each kind of UML models we identified.

For each kind of UML models:

- We identified a set of rules to model it, namely a set of UML syntactic elements and annotations to expose important information, and potential extensions to fUML* in cases where execution semantics of the required UML syntactic elements are not defined,
- We proposed a master algorithm for each co-simulation scenario. This latter is responsible for the orchestration and synchronization of the simulations of the involved components. It takes into account the dependencies between the involved components and the MoCs they rely on. The proposed algorithms are based on adaptation of untimed semantics of fUML* (and its potential extensions) to timed semantics of FMI, and adaptation of the discrete semantics of fUML* (and its potential extensions) to continuous time semantics of FMI. Based on these adaptations, the master algorithms are both able to propagate data between components and trigger them at the correct points of time.

- **Validation of the approach**

The proposed approach was applied to representative examples of transformational and reactive systems in the contribution part, as well as, to an energy system of a smart grid in the experimentation part. We used the modeling rules we defined in our approach for the specification of the structure and the logic of the software component with UML. Then we used suitable master algorithms for their co-simulation with the rest of the system components. The simulation results demonstrated that, using our approach, we are able to better synchronize the involved components while considering untimed behaviors, instantaneous reactions, and detecting time events. In particular, in the energy case study, we managed to improve the operation of the energy auto-consumption control by propagating new control set points earlier.

❖ Perspectives

▪ Perspective related to UML models capabilities

The support of rollback by the components involved in a co-simulation scenario improves the efficiency of the master algorithm. This latter may propose to the involved components to remake a simulation step with a different simulation step size in case one (or more) component does not manage to simulate the whole step. This provides another way, to the master, to choose the suitable simulation step size for a better synchronization of the components.
The master algorithms we proposed do not consider the support of rollback by the involved components. For instance, these later can be enriched with the rollback functionality for co-simulation scenario connecting FMUs supporting rollback to UML models of transformational systems since they are stateless.

The definition of rollback capability semantics for UML models (in particular reactive behaviors) is therefore an interesting way to improve the integration of UML models in FMI-based co-simulation.

▪ Perspective related to supported UML models

The Precise Semantics of UML State Machines (PSSM) specification is an OMG standard that extends fUML. The PSSM specification extends the syntactic set of fUML with a subset of the abstract syntax of state machines as given in the UML specification, as well as the fUML semantic model in order to specify the execution semantics of the state machine abstract syntax subset.
We proposed an approach that enables the integration of UML models whose behaviors are specified with UML activities. This later can be extended for the support of UML models whose behaviors are specified with UML state machines. The extension would rely on similar modeling rules and adaptation of semantics between FMI API and PSSM semantic model.
This extension would valorize the use of UML for the specification of computational components and would enlarge the scope of UML models we can use in co-simulation scenarios.

PART IV: ANNEXES

ANNEX A: foundational UML (fUML) and PSCS for UML models execution: Syntax and Semantics

Outline

A.1. The syntax
A.2. The semantics
A.2.1. Behavioral semantics
A.2.1.1. *Object*$^{(sem)}$ semantic visitor
A.2.1.2. Execution$^{(sem)}$ and ActivityNodeActivation$^{(sem)}$ visitor
A.2.2. Instantiation semantics

The Object Management Group (OMG) proposes standards (fUML*: fUML, PSCS) that define precise semantics for a foundational subset of UML elements. This annex gives an overview of the key syntactic (section A.1) **Figure *A-1***and semantic (section A.2) elements of fUML*.

Two mentions will be used this chapter:
- *(syn) mention indicates that the element is a syntactic element*
- *(sem) mention indicates that the element is a semantic element*

A.1. The syntax

Foundational UML (fUML) is an OMG standard that formalizes precise execution semantics for a subset of UML abstract syntax. This subset is restricted to classes for structure modeling and, to activities and actions for behavior modeling. Abstract syntax elements considered by fUML for structure and behavior modeling are listed in Table A-1.

Table A-1. UML syntactic subset considered by fUML

Structure	Activity	Actions
Class	ControlFlow	CallBehaviorAction
Association	ObjectFlow	CallOperationAction
Property	MergeNode	SendSignalAction
Operation	DecisionNode	AcceptEventAction
Reception	ForkNode	ReadExtentAction
LiteralBoolean	JoinNode	ReadIsClassifiedObjectAction
LiteralInteger	InitialNode	ReclassifyObjectAction
LiteralUnlimitedNatural	FinalNode	ReduceAction

LiteralString	FlowFinalNode	StartClassifierBehaviorAction
InstanceValue	StructuredActivityNode	StartObjectBehaviorAction
DataType	LoopNode	AddStructuralFeatureValueAction
Signal	ConditionalNode	ClearAssociationAction
Trigger	ExpansionRegion	ClearStructuralFeatureAction
EventOccurence	ExpansionNode	CreateLinkAction
InstanceSpecification	ActivityParameterNode	CreateObjectAction
	InputPin	DestroyLinkAction
	OutputPin	DestroyObjectAction
	Activity	ReadLinkAction
	OpaqueBehavior	ReadSelfAction
	FunctionBehavior	ReadStructuralFeatureValueAction
		RemoveStructuralFeatureValueAction
		TestIdentityAction
		ValueSpecificationAction

fUML* defines execution semantics for elements listed in Table A-1. The next subsection concentrates on that.

A.2. The semantics

The fUML semantic model (right-hand side of Figure A-1) is designed following the visitor design pattern. Each executable element of the fUML abstract syntax (left-hand side of the figure) is associated with a semantic element which implements its execution semantics, so-called semantic visitor. Once instantiated, the semantic visitors constitute an interpreter for a given model.

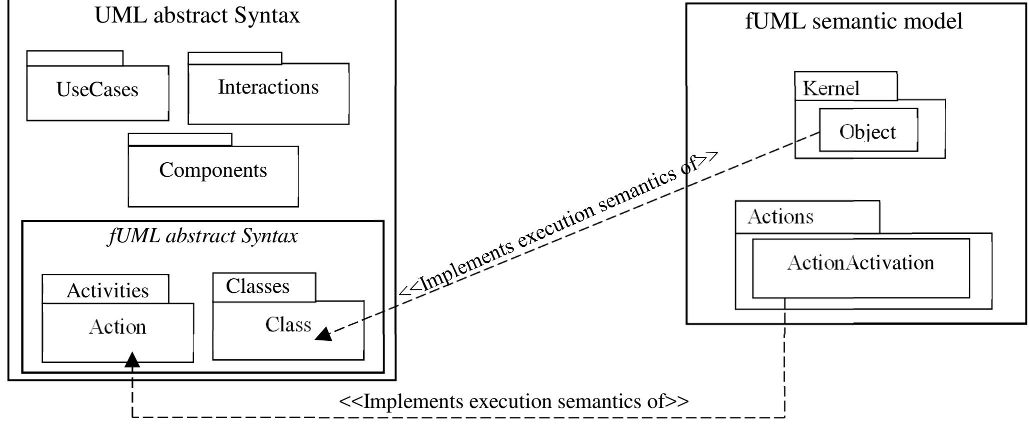

Figure A-1. Syntax and semantics of fUML

fUML introduces three abstract semantic visitors which represent the basis of the fUML semantic model: *ActivityNodeActivation*$_{(sem)}$, *Value*$_{(sem)}$, and *Evaluation*$_{(sem)}$ as depicted in Figure A-2.

- The *Value*$^{(sem)}$ semantic visitor is used to represent values on the runtime of fUML where a value is an instance of one or more classifiers. A value is always representable using a *ValueSpecification*$^{(syn)}$. *Value*$^{(sem)}$ specializes the *SemanticVisitor*$^{(sem)}$ class to allow the

representation of two primary elements: the *Object*$^{(sem)}$ and the *Execution*$^{(sem)}$ semantic visitors.

- The *ActivityNodeActivation*$^{(sem)}$ represents an abstract basis for the definition of execution semantics of activities nodes. Each concrete activation visitor (defined as a specialization of the *ActivityNodeActivation*$_{(sem)}$ semantic visitor) is used to model the semantics of a specific kind of activity node within the execution of an *Activity*$^{(syn)}$.
- The *Evaluation*$^{(sem)}$ is used to evaluate a specific kind of *ValueSpecification*$^{(syn)}$.

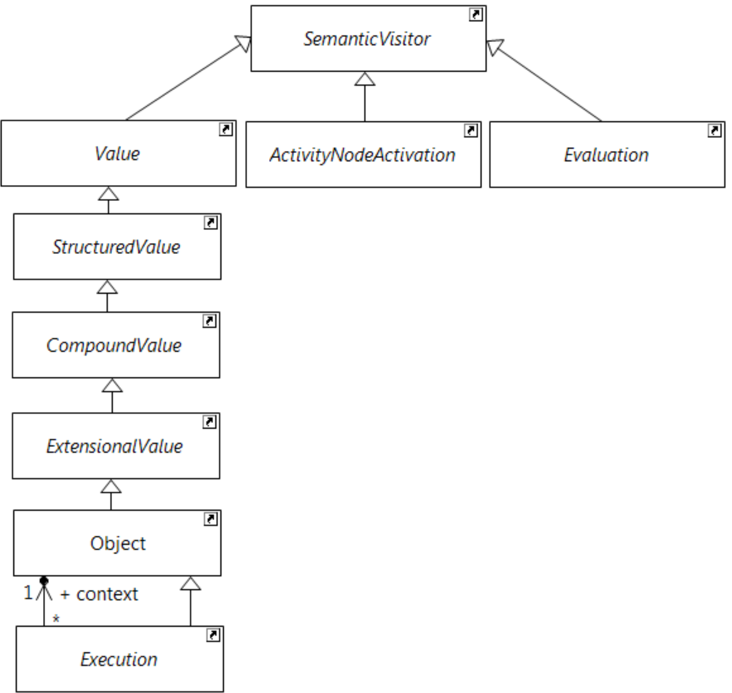

Figure A-2. Extract of semantic visitors considered by fUML

fUML defines three categories of semantic elements: those which capture structural semantics, those which capture behavioral semantics and, those which capture instantiation semantics. All these elements are defined as extensions for one of the abstract semantic visitors enumerated previously.

Structural semantics are captured by a set of semantic visitors which are extensions of the visitor *Value*$^{(sem)}$. In the context of this work, we are particularly interested in behavioral and instantiation semantics. The rest of this section gives further details about the most important semantic elements that are related to our work and contributions.

A.2.1. Behavioral semantics

These subsection focus on the behavioral semantics of fUML semantic model. The key elements which define behavioral semantics in the fUML semantic model are *Object*$^{(sem)}$, and the concrete semantic visitors which are derived from *Execution*$^{(sem)}$ and the *ActivityNodeActivation*$^{(sem)}$.

A.2.1.1. *Object*$^{(sem)}$ semantic visitor

Object$^{(sem)}$ represents the boundary between the structural and behavioral semantics. It may represent the instance of a passive or an active *Class*$^{(syn)}$ and provides operations which implement execution semantics of behaviors specified in each kind of classes as activities.

- Semantics for active classes

The operation *startBehavior()* of the class *Object*$^{(sem)}$ captures the execution semantics of active classes. An active *Class*$^{(syn)}$ is necessarily associated with a *Behavior*$^{(syn)}$ which represents its classifier behavior. Once instantiated, an active class starts its classifier behavior. The execution of this latter is controlled by the class *ObjectActivation*$^{(sem)}$ in the semantic model. The object activation invokes the classifier behavior for the execution of its behavior by sending an *InvocationEventOccurence*$^{(syn)}$. The classifier behavior accepts the invocation event occurrence and creates an *Execution*$^{(sem)}$ for the behavior associated to it. If the behavior is specified using an activity, then the created execution is an *ActivityExecution*$^{(sem)}$.
The operation *send()* in the O*bjectActivation*$^{(sem)}$ class allows an instance of an active *Class*$^{(syn)}$ to receive events. When an *EventOccurrence*$^{(syn)}$ is received, it is placed in the event pool of the O*bjectActivation*$^{(sem)}$ associated with the *Object*$^{(sem)}$. The order in which the events occurrences are consumed is a semantic variation point. fUML proposes a FIFO strategy which consists in consuming the first event occurrence in the pool, verifying whether the classifier behavior of the object is waiting an instance of the received event. If so, the event occurrence is consumed and the control is propagated to the next node in the activity and, if not the instance of the event is lost. fUML relies on run to completion (RTC) semantics where the principle is to propagate the control as much as possible in the classifier behavior when an event occurrence is received. fUML defines an event dispatch loop that waits for the arrival of an *EventOccurrence*$^{(syn)}$, when this happens a single event occurrence is dispatched from the event pool and, once this is completed, the dispatch loop returns to waiting for another *EventOccurrence*$^{(syn)}$ to arrive. The semantics of event dispatching are defined in the method *dipatchNextEvent()* of the O*bjectActivation*$^{(sem)}$ class.

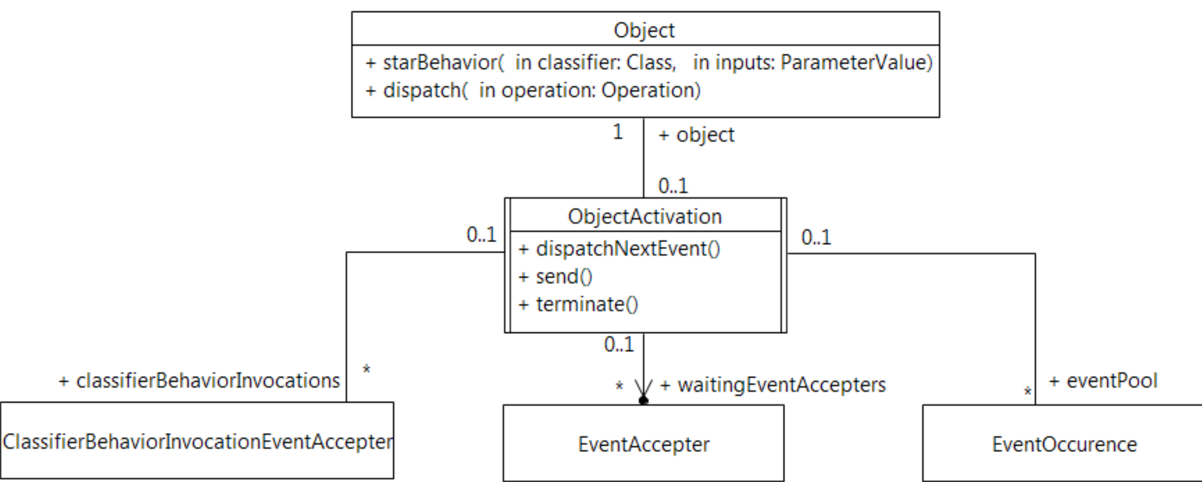

Figure A-3. Behavioral semantic elements related to the Object visitor

- Semantics for passive classes

The operation *dispatch()* of the class *Object*$^{(sem)}$ allows an instance of a passive *Class*$^{(syn)}$ to receive operation calls. The operation call is run synchronously in the context of the object that made the call. The strategy which identifies the actual operation to execute, in case of polymorphic definition of the operation, is a semantic variation point in fUML and should be defined in a class which extends *DispatchStrategy*$^{(sem)}$ class in the semantic model. The result of a call to the *dispatch()* operation is an *ActivityExecution*$^{(sem)}$.

A.2.1.2. Execution$^{(sem)}$ and ActivityNodeActivation$^{(sem)}$ visitor

Execution$^{(sem)}$ is a particular semantic visitors. It coordinates the execution of a set of elements specifying a *Behavior$^{(syn)}$*. There is an execution visitor class corresponding to each concrete subclass of *Behavior$^{(syn)}$* included in the fUML subset. In particular, the semantic visitor *ActivityExecution$^{(sem)}$* captures execution semantics of a behavior specified with an *Activity$^{(syn)}$*. It is associated with a group, so-called *ActivityNodeActivationGroup$^{(sem)}$*, which encapsulates the semantics of all node of the activity as depicted in Figure A-4.

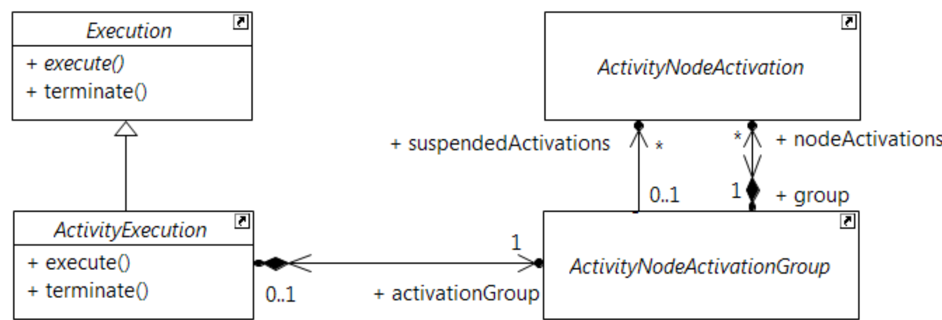

Figure A-4. The *ActivityExecution$^{(sem)}$* and *ActivityNodeActivation$^{(sem)}$* visitor

The *ActivityExecution$^{(sem)}$* class creates activity edge instances for all activity edges, activity node activations for all activity nodes and makes offers to all nodes with no incoming edges. The execution semantics of activities are captured by the semantic visitors which extend the abstract semantic visitor *ActivitynodeActivation$^{(sem)}$*.
The activations are divided into three categories: the *ActionActivation$^{(sem)}$*, the *ControlNodeActivation$^{(sem)}$* and, the *ObjectNodeActivation$^{(sem)}$* as depicted in Figure A-5. The execution of an activity node consists in the execution of the sequence of operations *receiveOffer()→fire()→sendOffers()* defined in its corresponding semantic visitor. This sequence of operations defines the semantics of tokens propagation in the activity. The operation *sendOffers()* is responsible for the propagation of a set of tokens to the outgoing edgesof the activity node. The operations *suspend()* and *resume()* of the *ActivityNodeActivation$^{(sem)}$* allow to respectively suspend and resume the execution of an activity node.
An execution terminates when either all node activations are complete, or an activity final node is executed.
Figure A-5 focus particularly on semantic visitors which extend the *ActionActivation$^{(sem)}$* visitor. Each sub-class of the *ActionActivation$^{(sem)}$* captures execution semantics of a particular action node and implements computations related to the execution semantics of the corresponding *Action$^{(syn)}$* in the method *doAction()*. For example the *CallBehavioActionActivation$^{(sem)}$* captures the execution semantics of the syntactic element *CallBehaviorAction$^{(syn)}$* which allows to specify a call to a behavior of a given object instance in a model.

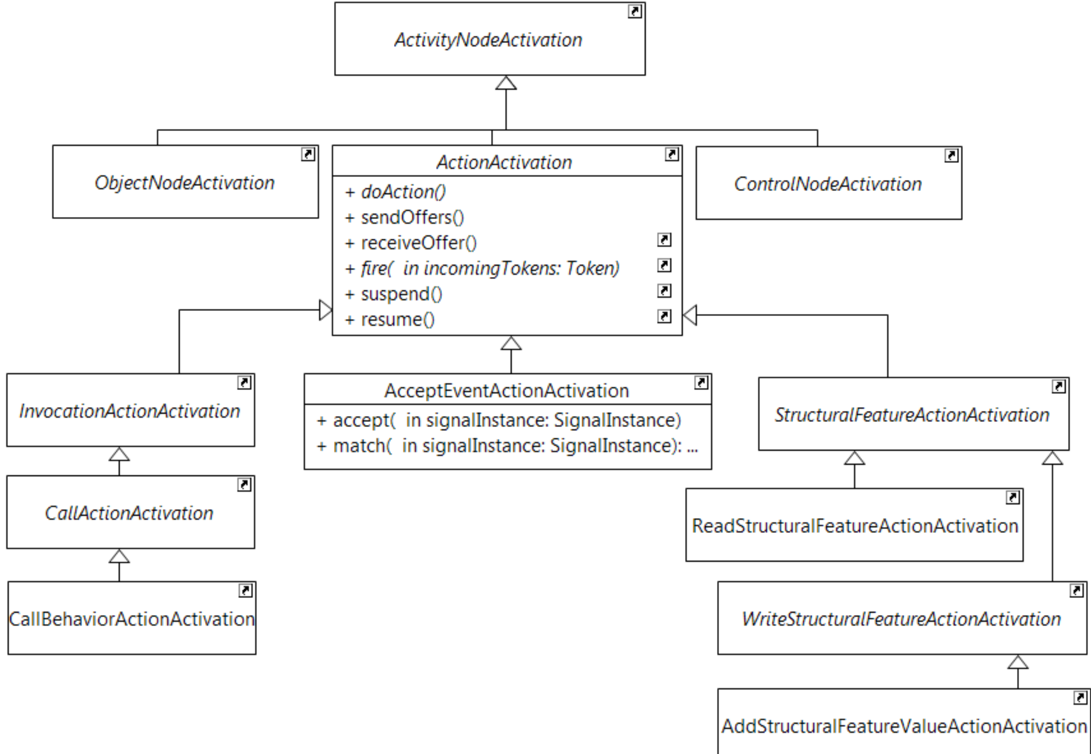

Figure A-5. Semantic visitor of activity nodes

The *AcceptEventActionActivation$^{(sem)}$* is an action activation for an *AcceptEventAction$^{(syn)}$*. The *AcceptEventAction$^{(syn)}$* is a particular action that waits for the occurrence of an event meeting specified condition. fUML provides execution semantics for *AcceptEventAction$^{(syn)}$* triggered by *SignalEventOccurrence$^{(syn)}$*. In the semantic model, each *acceptEventActionActivation$^{(sem)}$* is associated to an *AcceptEventActionEventAccepter$^{(sem)}$*. This latter handles reception of signal event occurrences on the behalf of that specific accept event action activation. For this, it defines two operations *match()* and *accept()*. The *match()* operation checks whether a given *SignalEventOccurrence$^{(syn)}$* matches the trigger specified for a given *AcceptEventAction$^{(syn)}$*. If so, the *accept()* operation is responsible for forwarding the *SignalEventOccurrence$^{(syn)}$* to the corresponding action activation which will enable control propagation to continue the execution of the following activity nodes.

A.2.2. Instantiation semantics

The instantiation of semantic visitors is handled by two specific classes in the semantic model which are: the *Locus$^{(sem)}$* and the *ExecutionFactory$^{(sem)}$* depicted in Figure A-6.

The class *Locus$^{(sem)}$* defines an operation *instantiate()* responsible for the representation of the executable syntactic elements specified in an applicative model. The operation takes a *Class$^{(syn)}$*as parameter which can be a *Type$^{(syn)}$* or a *Behavior$^{(syn)}$*. The result is respectively an *Object$^{(sem)}$* or an *Execution$^{(sem)}$* (which is a specialization of *Object$^{(sem)}$*). The locus is the virtual memory of fUML in which are stored all created visitors.

The instantiation of Visitors which capture execution semantics of behavioral elements is actually handled by the *ExecutionFactory$^{(sem)}$* class. The instantiation logic is captured by the operation *instantiateVisitor()* of the sub-classes *ExecutionFactoryL1$^{(sem)}$*, *ExecutionFactoryL2$^{(sem)}$*

and *ExecutionFactoryL3*$^{(sem)}$ (corresponding to the three conformance levels L1, L2 and L3 defined in UML specification [30]).

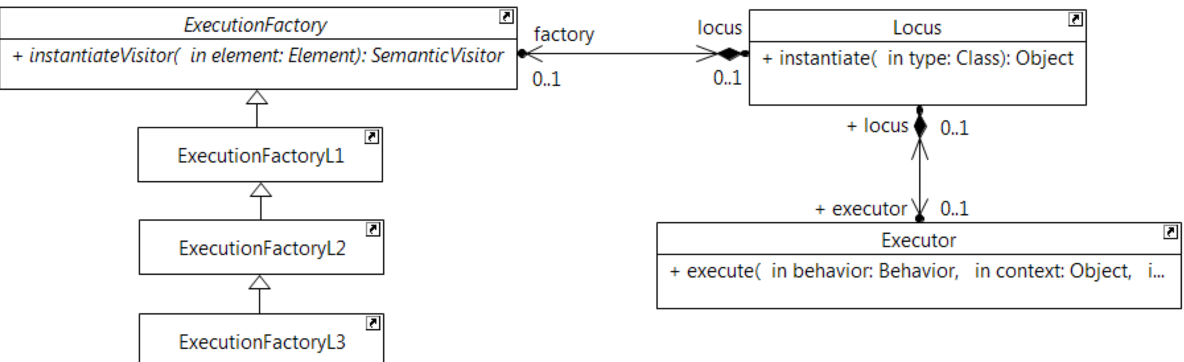

Figure A-6. The Locus, the executor, and the Execution Factory in fUML

In the previous subsections we presented the key elements of the semantic model of fUML. We have first identified the semantic visitors which capture the behavioral semantics of UML activities execution. Then we determined the semantics of the visitors' instantiation. Here is a recapitulation of semantic elements defined in the semantic model of fUML organized into three categories:

C-1 Visitors which define semantics for structure: they are specializations of the Value abstract visitor and define how the structural elements of a model are represented during the execution. The most important one is the *Object*$^{(sem)}$ semantic visitor

C-2 Visitors which define the semantics of behaviors: they are specializations of the *ActivityNodeActivation*$^{(sem)}$ abstract visitor and implement the execution semantics of activity nodes considered by the syntax subset of fUML.

C-3 The classes which define the rules of the instantiation and coordination between the semantic visitors

(a) *Execution*$^{(sem)}$ is a particular visitor used for the coordination for visitors defined for a set of activity nodes specifying an activity,

(b) *Locus*$^{(sem)}$ defines the rules of instantiation for the *Classifier*$^{(syn)}$ and allows to keep a trace for values created at the execution,

(c) *Executor*$^{(sem)}$ defines the entry point of an execution in the semantic model,

(d) *ExecutionFactory*$^{(sem)}$ handles the creation of visitors defined as specialization to *Execution*$^{(sem)}$ and *ActivityNodeActivation*$^{(sem)}$.

ANNEX B: FMI for co-simulation Standard

Outline

B.1. The FMU content
B.1.1. Structure (XMl file)
B.1.1.1. ‘CoSimulation’ element
B.1.1.2. ‘DefaultExperiment’ element
B.1.1.3. ‘ModelVariables/ScalarVariable’ element
B.1.1.4. ‘ModelStructure/Outputs’ element
B.1.2. Dynamics (DLLs/C-functions)
B.1.2.1. Instantiation and initialization
B.1.2.2. Stepwise simulation and data propagation
B.1.2.2. Termination
B.2. The Master Algorithm
B.2.1. Procedures calls order
B.2.2. Pseudocode of the master algorithm

The content of this appendix supplements information given in chapter 2 about the FMI for co-simulation standard, in particular about the content of an FMU (section B.1) as well as the master algorithm (section B.2). We focus on details we believe required for a better comprehension of this work. For a deep understanding of the standard, refer to the standard specification [17].

B.1. FMU content

A component which implements the FMI is called Functional Mockup Unit (FMU). It consists of one zip-file with extension “*.fmu” containing:

- An XML-file describing the variables of the FMU that are exposed to the environment in which the FMU shall be used (the structure), as well as other model information,
- A set of C-functions to setup and run the FMUs in a co-simulation environment (the dynamics). These C-functions can either be provided in source and/or binary form. An FMU for co-simulation embeds the solver responsible for the resolution of the equations described
- Further data can be included in the FMU zip-file (a model icon, documentation files, maps and tables needed by the model).

B.1.1. Structure (XML file)

The XML-file is defined by an XML-schema file called “fmiModelDescription.xsd”. In Figure B-1, the complete XML schema definition is shown.

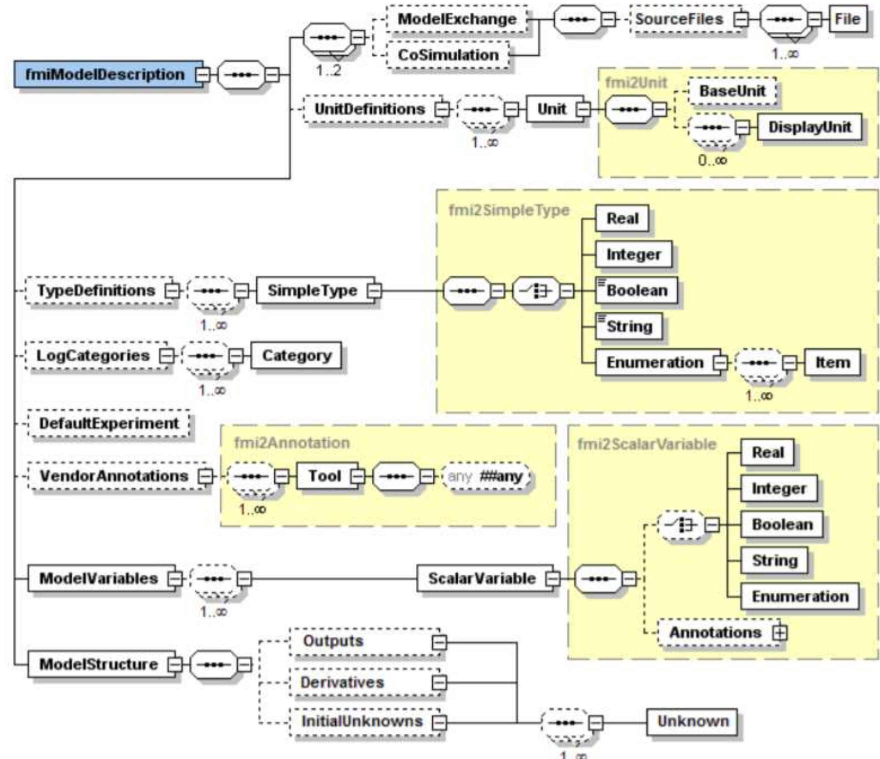

Figure B-1. XML schema of the FMI standard (version 2.0)

Each element in the XML schema has attributes in which information is introduced. The elements 'CoSimulation', 'DefaultExperiments', 'ModelVariables/ScalarVariable' and 'ModelStructure/Outputs/Unknown' are the most important ones.

B.1.1.1. 'CoSimulation' element

The XML file of an FMU intended for co-simulation must contain this element. If so, the FMU includes the model and the simulation engine, or a communication to a tool that provides the model and the simulation engine, and the environment provides the master algorithm to run coupled FMU co-simulation slaves together. The element enables to introduce information about the capabilities of the FMU. For example, its capability to rollback a simulation step is expressed in the attribute 'canGetAndSetFMUState'. Figure B-2 taken from [17] illustrates all attributes of this element together with a description of each one.

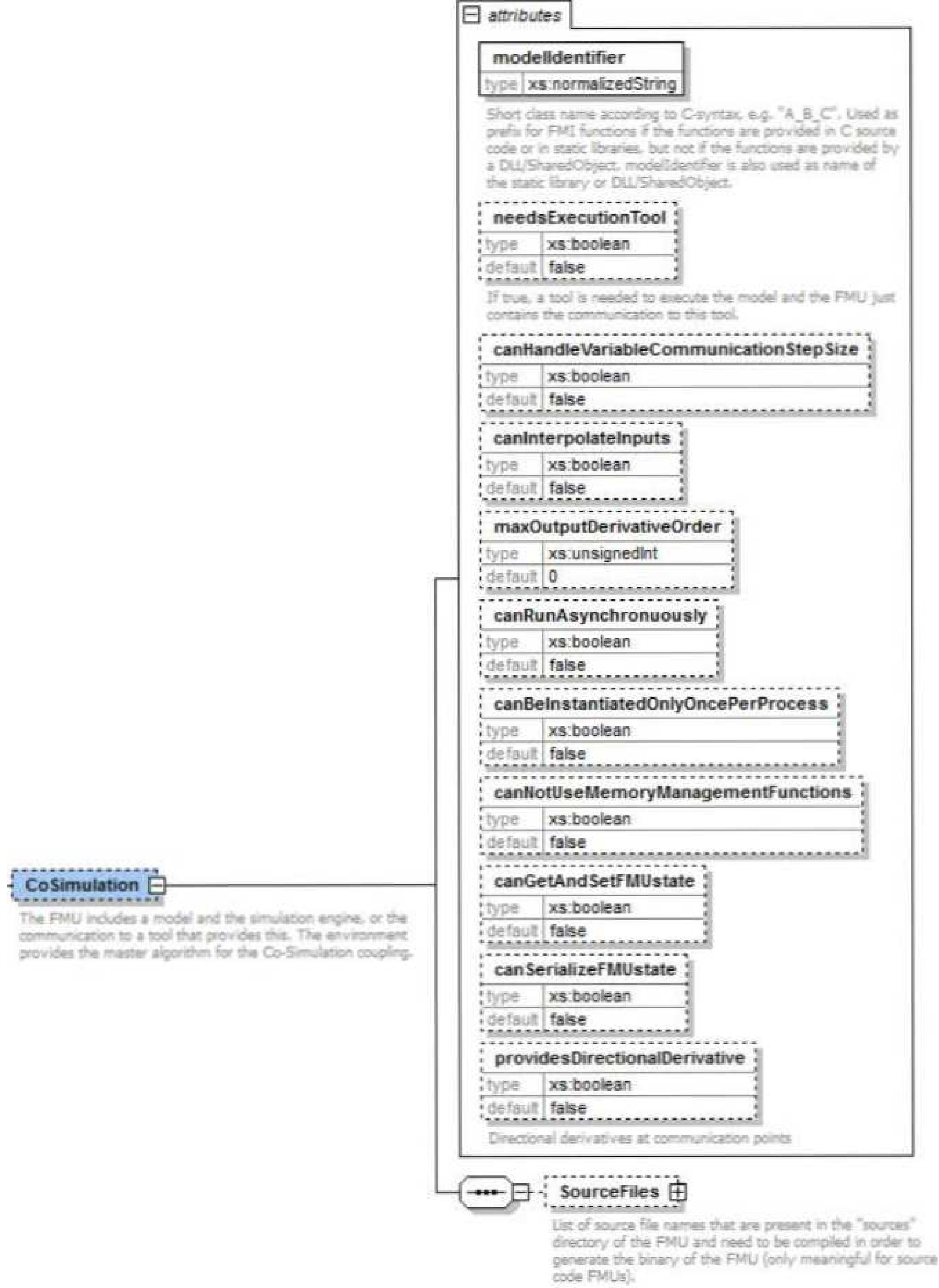

Figure B-2. The attributes associated with the 'CoSimulation' element in the XML schema

B.1.1.2. 'DefaultExperiment' element

An FMU can provide the default settings used by the solver in the original simulation tool (the tool in which the FMU was designed and simulated), such as stop time and stop time of the simulation, and relative tolerance. The FMU can also provide a default simulation step size. This later can be considered as the preferred (most suited) step size of the FMU for its simulation. Figure B-3 illustrates the attributes of the 'DefaultExperiment' attribute.

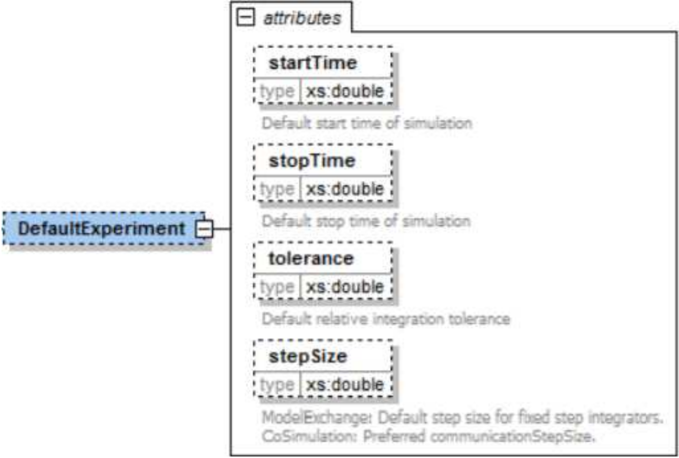

Figure B-3. Attributes of the 'DefaultExperiment' element in the XML schema

B.1.1.3. 'ModelVariables/ScalarVariable' element

'ModelVariables' element consists of an ordered set of 'ScalarVariable' elements (see Figure B-4 above). A 'ScalarVariable' represents a variable of primitive type, like a real or integer variable. The attributes of the 'ScalarVariable' element are illustrated in Figure B-4 together which a brief description of each attribute. This element allows to define the component inputs, outputs, parameters, local variables and time dependent variables. The causality of the variable is defined by the attribute 'causality'.

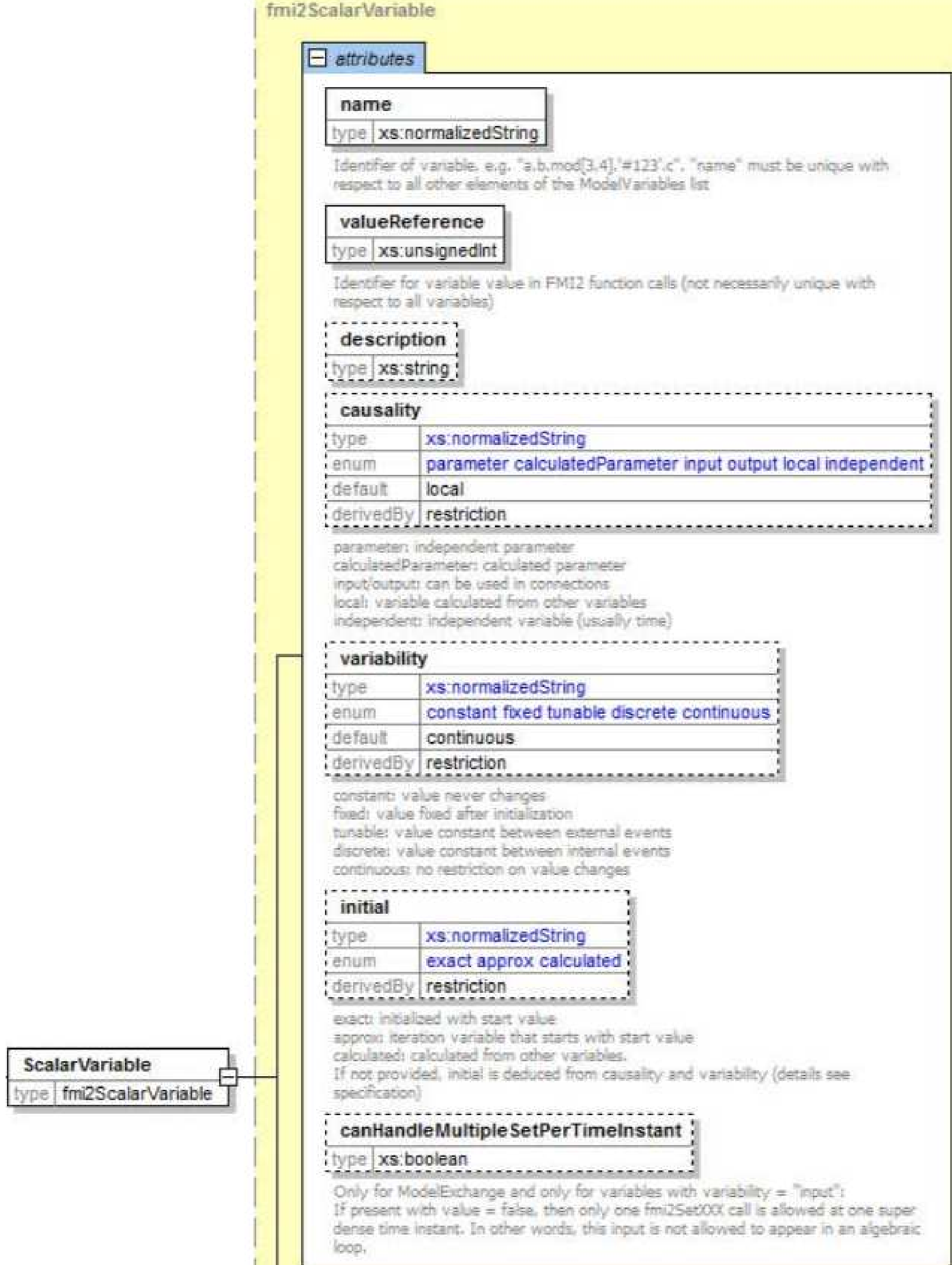

Figure B-4. Attributes of 'ScalarVariable' element in the XML schema

B.1.1.4. ‘ModelStructure/Outputs’ element

The ‘ModelStructure’ defines the structure of the model. Especially, the ordered lists of outputs, continuous-time states and initial unknowns (the unknowns during Initialization Mode) are defined here (see Figure B-6). It allows also to optionally define the dependency of the unkowns from the knowns. For example, the I/O dependency information is expressed in the ‘Outputs’ element. Figure B-6 illustrates the attributes related to the ‘Outputs’ element together with a description of each of them.

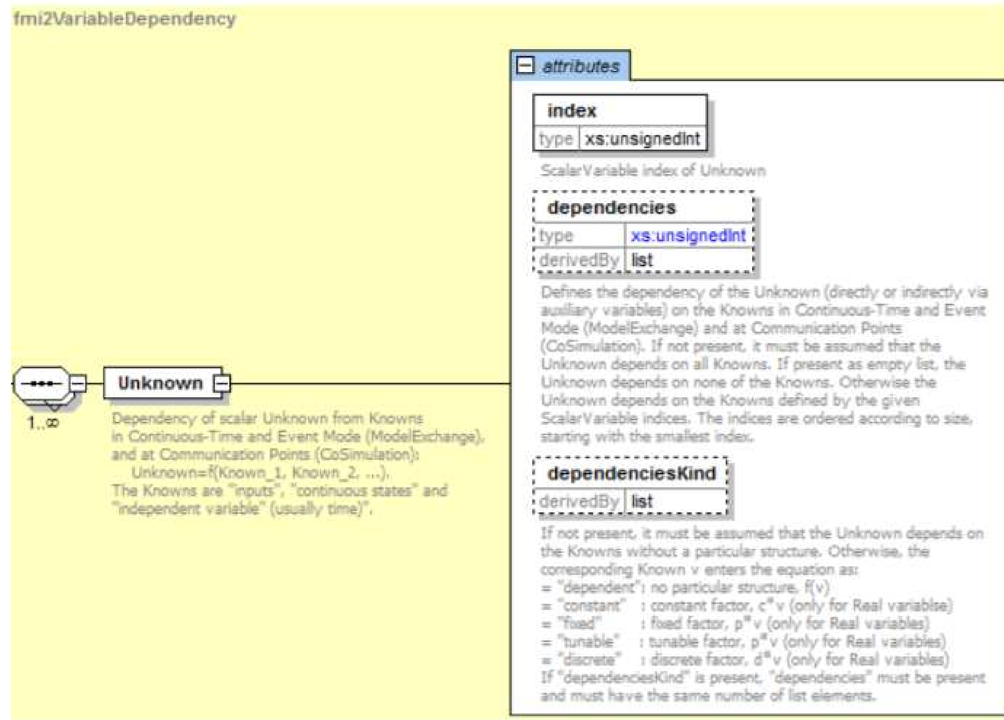

Figure B-6.'Unknown' element attributes in the XML schema

The XML file contained in the FMU should comply with this XML schema. Figure B-5 depicts an example of a "modelDescription.xml" file of the FMU *'TankPI.TankPIPlant'*.

```xml
<fmiModelDescription
  fmiVersion="2.0"
  modelName="TankPI.TankPIPlant"
  guid="{287d1a94-5b90-49a2-8dd8-f9541778062d}">
  <CoSimulation
    modelIdentifier="TankPI_TankPIPlant"
    canHandleVariableCommunicationStepSize="true"
    canInterpolateInputs="true"
    maxOutputDerivativeOrder="1"
    canBeInstantiatedOnlyOncePerProcess="true"
    canGetAndSetFMUstate="true"
    providesDirectionalDerivative="true"/>
  <DefaultExperiment startTime="0.0"  stopTime="250.0"  tolerance="0.0001"/>
  <ModelVariables>
    <!-- Index for next variable = 1 -->
    <ScalarVariable
      name="source.qOut.lflow"  valueReference="637534208"  variability="discrete">
      <Real  unit="m3/s"/>
    </ScalarVariable>


    <!-- Index for next variable = 11 -->
    <ScalarVariable   name="tank.h"  valueReference="33554432"  initial="exact">
      <Real  unit="m"  start="0.0"/>
    </ScalarVariable>


    <!-- Index for next variable = 14 -->
    <ScalarVariable   name="outValue"   valueReference="335544320"   causality="output">
      <Real   unit="m"/>
    </ScalarVariable>
  </ModelVariables>
  <ModelStructure>
    <Outputs>
      <Unknown index="14"  dependencies="11"  dependenciesKind="dependent"/>
    </Outputs>
  </ModelStructure>
</fmiModelDescription>
```

Figure B-5. Example of a model description xml file of an FMU for co-simulation

This FMU can handle variable communication step size, can interpolate inputs, can provide directional derivatives, and supports rollback as indicated in <coSimulation> element. It is preferable to simulate this FMU from time *'t=0'* to time *'t=250'* with a step size *'h=0,0001'* as indicated in <defaultExperiment> element. For simplification, not all variables of the model are shown in the figure. The model has an output 'outValue' which depends on the parameter 'tank.h' as indicated in outputs of the <modelStructure> element.

B.1.2. Dynamics (DLL/C-functions)

This section is organized in a way to conform the section 2.1.1.2. It introduces the functions responsible for the instantiation and initialization on an FMU, the stepwise simulation on an FMU, and the termination of an FMU as defined in the FMI API.

B.1.2.1. Instantiation and initialization

The function 'fmi2Instantiate' (Figure B-7) returns a new instance of an FMU. If a null pointer is returned, then instantiation failed. This function must be called successful before a simulation run starts.

```
fmi2Component fmi2Instantiate(fmi2String  instanceName,
                              fmi2Type    fmuType,
                              fmi2String  fmuGUID,
                              fmi2String  fmuResourceLocation,
                              const fmi2CallbackFunctions* functions,
                              fmi2Boolean                  visible,
                              fmi2Boolean                  loggingOn);
```

Figure B-7. 'fmi2Iinstantiate' function

After the instantiation, the FMU should be informed about the simulation parameters chosen by the master for the co-simulation using the function '*fmi2Instantiate*' (Figure B-8) The master can choose to use the parameters defined in the FMU (if information are available), or to propose other parameters. This function can be called after the 'fmi2Instantiate' ad before 'fmi2EnterInitializationMode'

```
fmi2Status fmi2SetupExperiment(fmi2Component c,
                               fmi2Boolean   toleranceDefined,
                               fmi2Real      tolerance,
                               fmi2Real      startTime,
                               fmi2Boolean   stopTimeDefined,
                               fmi2Real      stopTime);
```

Figure B-8. 'fmi2SetupExperiments' function

The FMU can now enter the initialization mode where the actual value of the FMU variables can be get and some variables values can be set (refer to Figure B-14 below for the list of variables we can set at initialization mode). The FMU is informed to enter the initialization mode with the function 'fmi2EnterInitializationMode', and to exit the initialization mode with the function 'fmi2ExitInitializationMode' (Figure B-9).

```
fmi2Status fmi2EnterInitializationMode(fmi2Component c);
```

```
fmi2Status fmi2ExitInitializationMode(fmi2Component c);
```

Figure B-9. 'fmi2EnterInitializationMode' and 'fmi2ExitInitializationMode'

B.1.2.2. Stepwise simulation and data propagation

After the instantiation and initialization of the FMU, the master is allowed to perform stepwise simulation and to propagate data from one FMU to another.

The values of the FMU variables defined in the XML file are get using the following functions:

```
fmi2Status fmi2GetReal   (fmi2Component c, const fmi2ValueReference vr[],
                          size_t nvr, fmi2Real value[]);
fmi2Status fmi2GetInteger(fmi2Component c, const fmi2ValueReference vr[],
                          size_t nvr, fmi2Integer value[]);
fmi2Status fmi2GetBoolean(fmi2Component c, const fmi2ValueReference vr[],
                          size_t nvr, fmi2Boolean value[]);
fmi2Status fmi2GetString (fmi2Component c, const fmi2ValueReference vr[],
                          size_t nvr, fmi2String value[]);
```

Figure B-10. 'fmi2GetXXX' function

It is also possible to set the values of certain variables at particular instants in time using the following functions:

```
fmi2Status fmi2SetReal   (fmi2Component c, const fmi2ValueReference vr[],
                          size_t nvr, const fmi2Real value[]);
fmi2Status fmi2SetInteger(fmi2Component c, const fmi2ValueReference vr[],
                          size_t nvr, const fmi2Integer value[]);
fmi2Status fmi2SetBoolean(fmi2Component c, const fmi2ValueReference vr[],
                          size_t nvr, const fmi2Boolean value[]);
fmi2Status fmi2SetString (fmi2Component c, const fmi2ValueReference vr[],
                          size_t nvr, const fmi2String value[]);
```

Figure B-11.'fmi2SetXXX' function

The simulation is performed from the start simulation time to the stop simulation time defined in the 'fmi2SetupExperiments' function by calling the 'fmi2DoStep' funtion on the FMU. The master should precise the current simulation tme 'currentCommunication pointn' as well as the current step size 'communicationStepSize'.

```
fmi2Status fmi2DoStep(fmi2Component c,
                      fmi2Real    currentCommunicationPoint,
                      fmi2Real    communicationStepSize,
                      fmi2Boolean noSetFMUStatePriorToCurrentPoint);
```

Figure B-12. 'fmi2DoStep' function

B.1.2.3. Termination

At the end of the simulation, the master should inform the FMU that the simulation run is terminated by calling the function ‘fmi2Terminate’ on the FMU. After calling this function, the final values of all variables can be inquired with the ‘*fmi2GetXXX’* functions.

```
fmi2Status fmi2Terminate(fmi2Component c);
```

Figure B-13. 'fmi2Terminate’ function

B.2. The master Algorithm

B.2.1. Procedures calls order

The FMI standard defines the FMU life cycle. It identifies four modes in which the FMU can be: Instantiated, Initialization mode, Slave Initialized, Terminated.
For each mode, the standard defines the functions which can be called on an FMU. Figure B-14 illustrates the life cycle of the FMU as well as the supported calling sequence

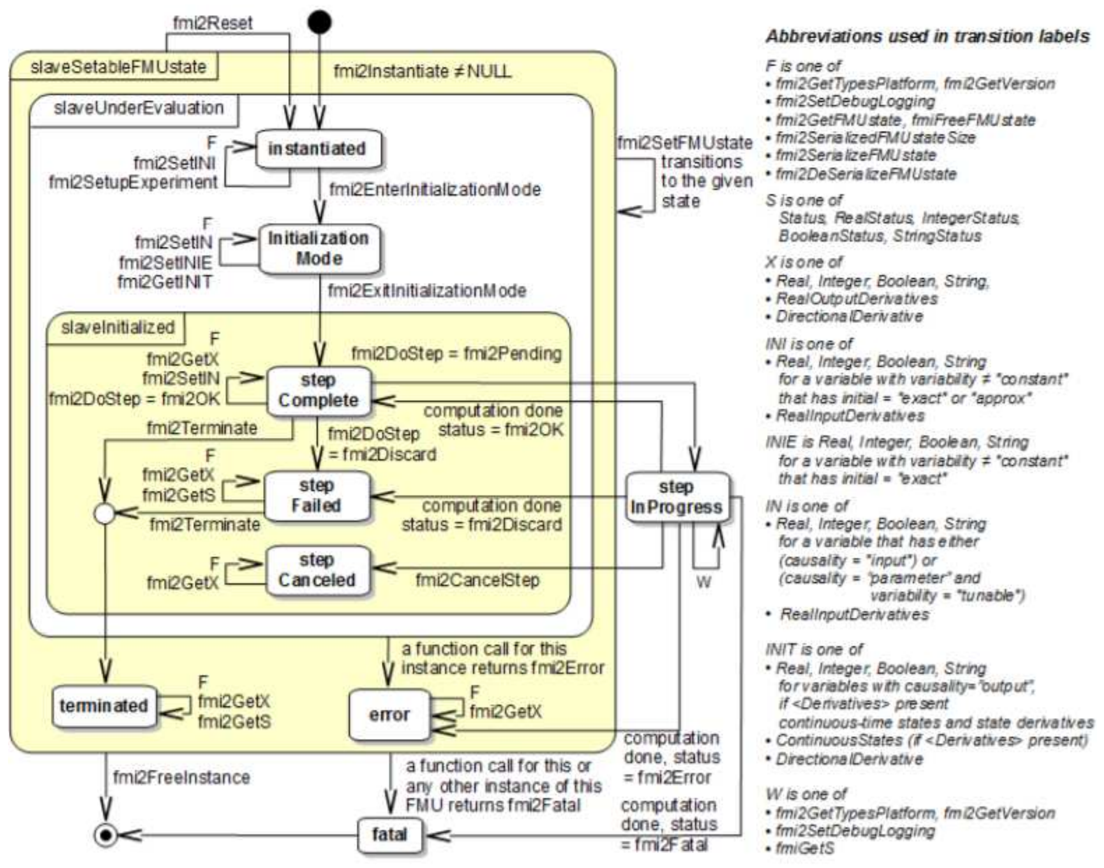

Figure B-14. Calling sequence of Co-Simulation C functions

B.2.2. Pseudocode of a basic master algorithm

The FMI standard provide a pseudocode of a master algorithm in order to sketch the typical calling sequence of the functions in a co-simulation environment.

The master orchestrates a co-simulation scenario composed of two FMUs ‘S1’ and ‘S2’ as depicted in Figure B-15.

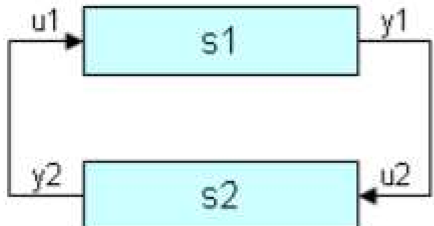

Figure B-15. Co-simulation scenarion composed of two FMUs

The FMUs support the minimum capabilities (no rollback, no variable step size). The pseudocode of the master algorithm is depicted in Figure B-16.

```
//////////////////////////
//Initialization sub-phase

//Set callback functions,
fmi2CallbackFunctions cbf;
cbf.logger = loggerFunction;  //logger function
cbf.allocateMemory = calloc;
cbf.freeMemory = free;
cbf.stepFinished = NULL;      //synchronous execution
cbf.componentEnvironment = NULL;

//Instantiate both slaves
fmi2Component s1 = s1_fmi2Instantiate("Tool1" , fmi2CoSimulation, GUID1, "",
                                      fmi2False, fmi2False, &cbf, fmi2True);
fmi2Component s2 = s2_fmi2Instantiate("Tool2" , fmi2CoSimulation, GUID2, "",
                                      fmi2False, fmi2False, &cbf, fmi2True);

if ((s1 == NULL) || (s2 == NULL))
      return FAILURE;

// Start and stop time
startTime = 0;
stopTime = 10;

//communication step size
h = 0.01;

// set all variable start values (of "ScalarVariable / <type> / start")
s1_fmi2SetReal/Integer/Boolean/String(s1, ...);
s2_fmi2SetReal/Integer/Boolean/String(s2, ...);

//Initialize slaves
s1_fmi2SetupExperiment(s1, fmi2False, 0.0, startTime, fmi2True, stopTime);
s2_fmi2SetupExperiment(s1, fmi2False, 0.0, startTime, fmi2True, stopTime);
s1_fmi2EnterInitializationMode(s1);
```

```
s2_fmi2EnterInitializationMode(s2);

   // set the input values at time = startTime
   s1_fmi2SetReal/Integer/Boolean/String(s1, ...);
   s2_fmi2SetReal/Integer/Boolean/String(s2, ...);
s1_fmi2ExitInitializationMode(s1);
s2_fmi2ExitInitializationMode(s2);


//////////////////////////
//Simulation sub-phase

      tc = startTime; //Current master time

      while ((tc < stopTime) && (status == fmi2OK))
      {
            //retrieve outputs
            s1_fmi2GetReal(s1, ..., 1, &y1);
            s2_fmi2GetReal(s2, ..., 1, &y2);
            //set inputs
            s1_fmi2SetReal(s1, ..., 1, &y2);
            s2_fmi2SetReal(s2, ..., 1, &y1);

            //call slave s1 and check status
            status = s1_fmi2DoStep(s1, tc, h, fmi2True);
            switch (status) {
            case fmi2Discard:
                  fmi2GetBooleanStatus(s1, fmi2Terminated, &boolVal);
                  if (boolVal == fmi2True)
                        printf("Slave s1 wants to terminate simulation.");
            case fmi2Error:
            case fmi2Fatal:
                  terminateSimulation = true;
                  break;
            }
            if (terminateSimulation)
                  break;

            //call slave s2 and check status as above
            status = s2_fmi2DoStep(s2, tc, h, fmi2True);
            ...

            //increment master time
            tc += h;
      }
      //////////////////////////
      //Shutdown sub-phase
      if ((status != fmi2Error) && (status != fmi2Fatal))
      {
            s1_fmi2Terminate(s1);
            s2_fmi2Terminate(s2);
      }


if (status != fmi2Fatal)
{
      s1_fmi2FreeInstance(s1);
      s2_fmi2FreeInstance(s2);
}
```

Figure B-16. Pseudocode of the master algorithm

ANNEX C: Papyrus/Moka support for FMI for co-simulation standard

Outline

C.1. Moka Overview
- C.1.1. Execution of models based on standards
- C.1.2. Interactive execution
- C.1.3. Extension for new execution semantics

C.2. Moka extended for the FMI standard
- C.2.1. Moka as a master for co-simulation
  - C.2.1.1. The import of an FMU for co-simulation in Papyrus/Moka
  - C.2.1.2. The definition of a co-simulation scenario in Papyrus/Moka
  - C.2.1.3. The simulation of co-simulation scenarios in Papyrus/Moka
- C.2.2. Moka as a slave for co-simulation

The work is tooled in the context of Papyrus which is an open source UML/SysML modeler based on the Eclipse platform[27]. It enables designers to describe very detailed models of their systems, and aims at providing an integrated environment for UML models and related profiles. Papyrus provides UML models execution by means of its additional component Moka. In this chapter, we aim at representing the functionalities of Moka in section C.1, as well as, the implementation of the framework of co-simulation we propose in chapter 0 related to the support of FMI-based co-simulation.

C.1. Moka Overview

Moka provides three important features: execution of models based on OMG standards, interactive execution, and extensibility of the framework for new execution semantics. This subsection introduces these functionalities.

C.1.1. Execution of models based on standards

Moka natively includes execution engines complying with the OMG standards fUML* (i.e. fUML and PSCS), by implementing the interpreter described in their specifications. This implementation supports the execution of UML subset defined in fUML* specifications. Each model constructed using the syntactic elements listed in annex A is executable.

C.1.2. Interactive execution

Moka is integrated with the Eclipse debug framework to provide control, observation and animation facilities over executions (Figure C-1). It is thereby possible to control execution of models(e.g., suspending/resuming executions after breakpoints have been encountered) as well as to observe states of executed models at runtime (e.g., emphasizing graphical views of model

[27] Refer to: eclipse.org/papyrus/

elements on which execution has suspended, retrieving and displaying any state information about the runtime manifestation of these model elements).

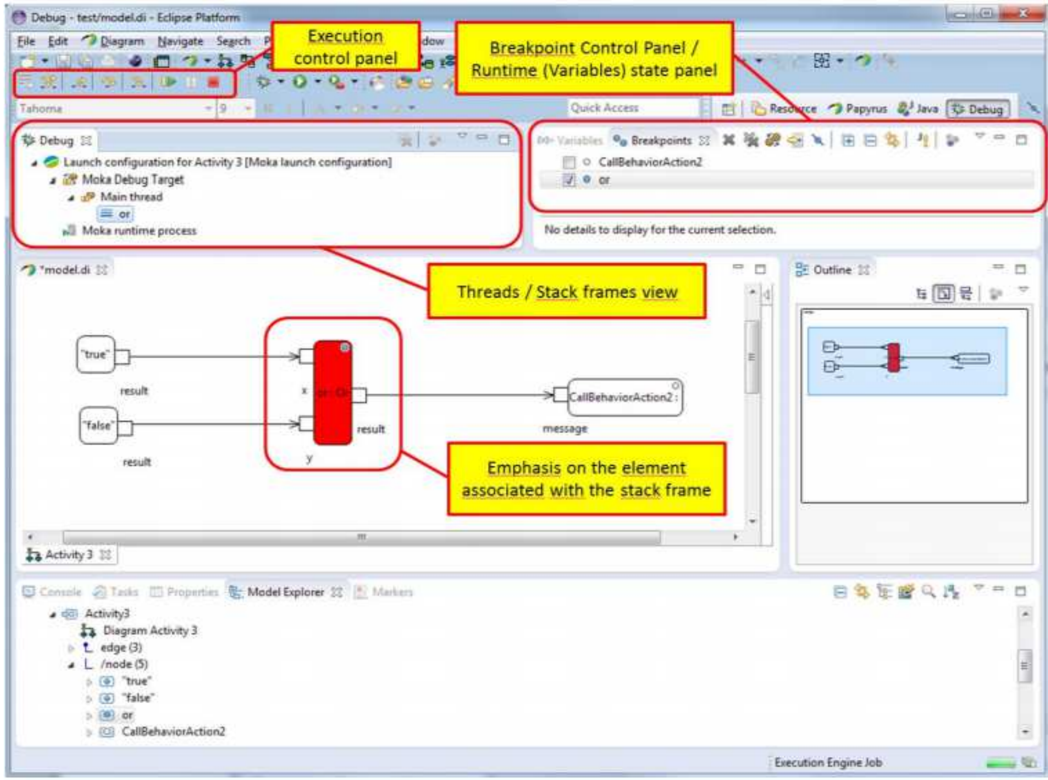

Figure C-1. Interactive execution in Papyrus/Moka

C.1.3. Extension for new execution semantics

Thanks to its architecture, Moka is a front-end for the integration of simulation tools and techniques. In fact, Moka can be easily extended to address new execution semantics. This can be done through extension points enabling registration of executable model libraries (e.g., new MoCs, trace libraries, etc.) or simply tool-level extensions of the execution engine. Domain or user specific customization of the modeling tool can be associated with a dedicated simulation engine in Moka. Moka is for example extended for simulation of business process models (Moka for BPMN) and for co-simulation of cyber-physical systems (Moka for FMI implemented in the context of this work).

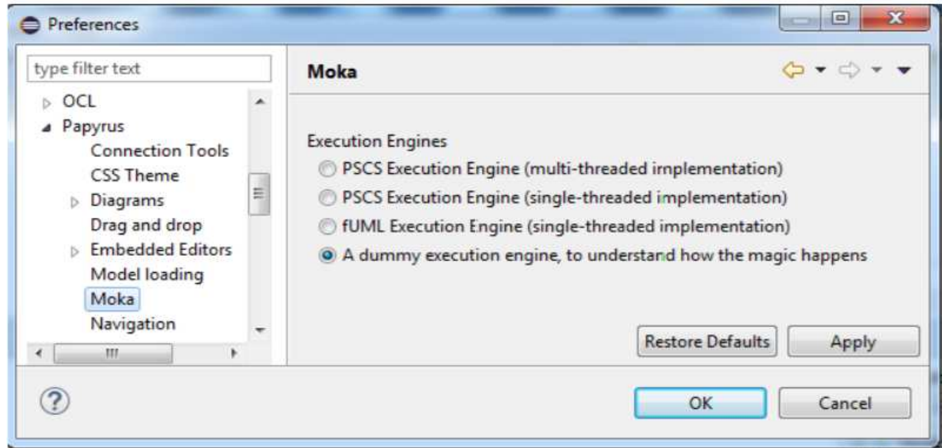

Figure C-2. The support of several execution engines in Moka

Figure C-2 illustrates the fact that Moka can integrate several execution engines.

C.2. Moka extended for the FMI standard

Moka is extended for the support of the FMI for co-simulation standard (Figure C-3). Moka can be used as: (a) a master for co-simulation where FMU for co-simulation can be imported, connected and simulated (b) a slave where the UML models are exported as FMUs for co-simulation compliant with the FMI specification.

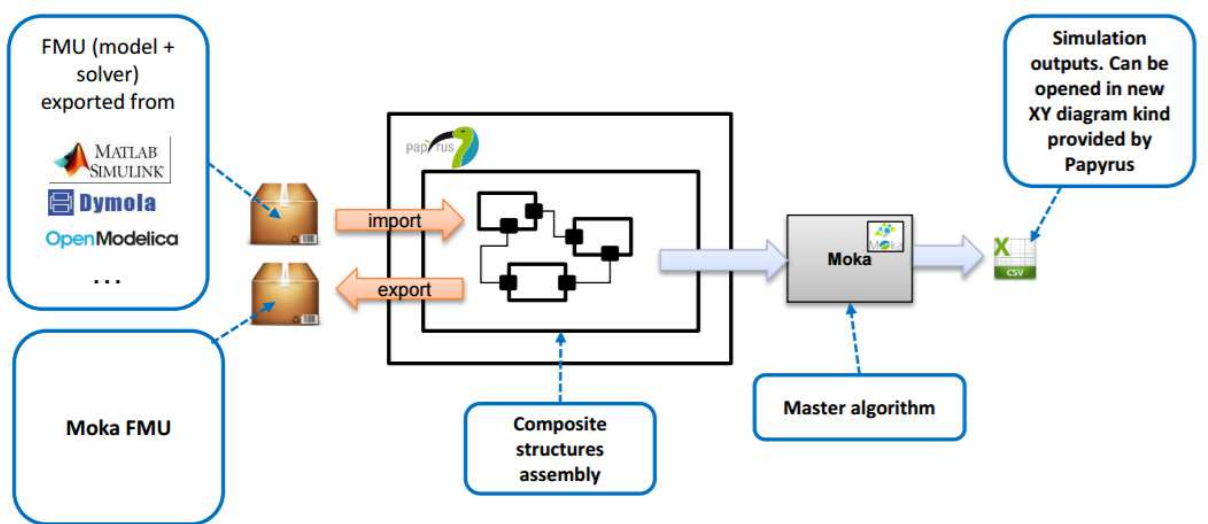

Figure C-3. Papyrus/Moka support for FMI for co-simulation

C.2.1. Moka as a master for co-simulation

Moka provides a FMI-based co-simulation environment for the modeling and simulation of CPS as described in chapter 4. It allows in particular: the import of FMUs for co-simulation, the definition of co-simulation scenarios and their simulation using basic and advanced master algorithms. The implementation of these features is part of this work. They were already explained in the chapter 4. This subsection represents the implementation steps of these features. It illustrates the result of this implementation and the steps for testing it with snapshots taken from Papyrus/Moka.

C.2.1.1. Import of an FMU for co-simulation in Papyrus/Moka

Moka allows the import of FMUs for co-simulation compliant with the version 2.0 of the FMI standard. The import is done by model transformation from FMU to an annotated UML model. (i.e. annotated with stereotypes from the co-simulation profile represented in chapte 4). Annotations add to UML models FMI specific concepts, and includes a direct link to the in-memory original FMU model.

The transformation of FMU to an annotated UML model is done with a QVTo transformation (Figure C-4). This transformation takes as an input the FMI meta-model, the UML meta-model and the co-simulation profile we presented in chapter 4 and produces an annotated UML model.

```
modeltype UMLTypes uses "http://www.eclipse.org/uml2/5.0.0/Types";
modeltype UML uses 'http://www.eclipse.org/uml2/5.0.0/UML';
modeltype FMI uses 'platform:/plugin/org.eclipse.papyrus.moka.fmi2/fmi2ModelDescription.xsd';
modeltype ecore uses ecore('http://www.eclipse.org/emf/2002/Ecore');
//modeltype fmiML uses FmiMLProfile('http:///schemas/FmiMLProfile/_1LNqQBwfEeWeGufsnalS6A/44');

transformation NewTransformation(in fmu : FMI, in umlTypes : UML, in fmiMLProfile : UML , in fmuProx-
yLibrary : UML, out output_model :UML);

/*
* fmi profile and its stereotypes
*/
property fmiProfile : UML::Profile = fmiMLProfile.rootObjects()![UML::Profile];

//fmu type (coSimulation or modelExchange)
property cs_stereotype : UML::Stereotype =
        fmiProfile.ownedStereotype![name = "CS_FMU"];

//fmu Port/locals/parameters/calculated parameters/independent
property port_stereotype : UML::Stereotype =
        fmiProfile.ownedStereotype![name = "Port"];
property parameter_stereotype : UML::Stereotype =
        fmiProfile.ownedStereotype![name = "Parameter"];
property calculatedParameter_stereotype : UML::Stereotype =
        fmiProfile.ownedStereotype![name = "CalculatedParameter"];
property local_stereotype : UML::Stereotype =
        fmiProfile.ownedStereotype![name = "Local"];
property independent_stereotype : UML::Stereotype =
        fmiProfile.ownedStereotype![name = "Independent"];

//fmu dependencies
property outputDependency_stereotype : UML::Stereotype =
        fmiProfile.ownedStereotype![name = "OutputDependency"];
property derivativeDependency_stereotype : UML::Stereotype =
        fmiProfile.ownedStereotype![name = "DerivativeDependency"];
property initialUnknownDependency_stereotype : UML::Stereotype =
        fmiProfile.ownedStereotype![name = "InitialUnknownDependency"];

//create a generalization for the generated class
property fmu2ProxyGeneralization  : UML::Generalization  = object Generalization{general  := fmuProx-
yLibrary.rootObjects()[Package]->any(true).getOwnedMembers()->selectOne(name="Fmu2Proxy").oclAs-
Type(Class)};

configuration property dllPath : String;
configuration property fmuLocation : String;

//variables for the FMU structure (class, ports and attrinbutes)
property globalPropertiesList : Sequence(UML::Property);

//queries
query getUmlClassifier(name : String) : Classifier{
        var classifiers : Collection(Classifier) := umlTypes.rootObjects()[Model].packagedElement->se-
lectByKind(Classifier);
        return classifiers->any(classifier : Classifier | classifier.name = name);
}
query InitialType::findLiteral() : EnumerationLiteral {
        var enumeration := findEnumeration("InitialType");
        return enumeration.ownedLiteral![name = self.toString()];
}

query VariabilityType::findLiteral() : EnumerationLiteral {
        var enumeration := findEnumeration("VariabilityType");
        return enumeration.ownedLiteral![name = self.toString()];
}

query DependenciesKindTypeItem::findLiteral() : EnumerationLiteral {
        var enumeration := findEnumeration("DependenciesKindType");
        return enumeration.ownedLiteral![name = self.toString()];
}

query findEnumeration(enumName : String) : Enumeration {
        return fmiMLProfile.objectsOfType(Enumeration)![name=enumName];
}
```

```
query findStereotype(fmiVariable : FMI::Fmi2ScalarVariable) : Stereotype {
	var stereotypeToApply : UML::Stereotype := local_stereotype;
	switch {
		case(fmiVariable.causality = CausalityType::local){
			stereotypeToApply := local_stereotype;
		};
		case(fmiVariable.causality = CausalityType::parameter){
			stereotypeToApply := parameter_stereotype;
		};
		case(fmiVariable.causality = CausalityType::calculatedParameter){
			stereotypeToApply := calculatedParameter_stereotype;
		};
		case(fmiVariable.causality = CausalityType::independent){
			stereotypeToApply := independent_stereotype;
		};
		case(fmiVariable.causality = CausalityType::input or fmiVariable.causality = Causali-
		tyType::output){
			stereotypeToApply := port_stereotype;
		};
	};
	return stereotypeToApply;
}
//helpers
helper setPropertyType (inout myProperty : UML::Property ,in fmiVariable : FMI::Fmi2ScalarVariable ){
	switch{
		case (fmiVariable.real <> null) {
			myProperty.type := getUmlClassifier("Real");
			myProperty.defaultValue := object UML::LiteralReal{value := fmiVaria-
ble.real.start->any(true)};
		}
		case(fmiVariable.integer <> null) {
			myProperty.type := getUmlClassifier("Integer");
			myProperty.defaultValue := object UML::LiteralInteger{value := fmiVariable.in-
teger.start->any(true)};
		}
		case(fmiVariable.boolean <> null) {
			myProperty.type := getUmlClassifier("Boolean");
			myProperty.defaultValue := object UML::LiteralBoolean{value := fmiVaria-
ble.boolean.start->any(true)};
		}
		case(fmiVariable.string <> null) {
			myProperty.type := getUmlClassifier("String");
			myProperty.defaultValue := object UML::LiteralString{value := fmiVaria-
ble.string.start->any(true)};
		}
		case(fmiVariable.enumeration <> null) {
			myProperty.type := getUmlClassifier("String");
			myProperty.defaultValue := object UML::LiteralString{value := fmiVariable.enu-
meration.start.toString()->any(true)};
		};
	};
}
helper setPropertyStereotypeValues (inout myProperty : UML::Property, in appliedStereotype :
UML::Stereotype, in fmiVariable :FMI::Fmi2ScalarVariable){
	if (fmiVariable.valueReference <> null){
			myProperty.setValue(appliedStereotype,"valueReference",fmiVariable.valueRefer-
ence);
	}endif;
	if (fmiVariable.initial <> null){
		myProperty.setValue(appliedStereotype,"initial",fmiVariable.initial.findLiteral());
	}endif;
	if (fmiVariable.description <> null){
		myProperty.setValue(appliedStereotype,"description",fmiVariable.description);
	}endif;
	if (fmiVariable.variability <> null){
		myProperty.setValue(appliedStereotype,"variability",fmiVariable.variability.findLit-
eral());
	}endif;
	if (fmiVariable.causality <> null){
		myProperty.setValue(appliedStereotype,"causality",fmiVariable.causality.toString());
	}endif;
```

```
        switch{
                case (fmiVariable.causality = CausalityType::input){
                        myProperty.setValue(appliedStereotype,"causality","in");
                };
                case (fmiVariable.causality = CausalityType::output){
                        myProperty.setValue(appliedStereotype,"causality","out");
                };
        };
}

//main function
main() {
   //map xml file to UML Package containing a class (the FMU) and list of dependencies
        var  targetPackage  :  UML::Package  :=  fmu.rootObjects()![FMI::FmiModelDescriptionType].map
map2UMLPackage();
}

mapping inout Package::addPackageImport(targetPackage : Package) : PackageImport{
        self.packageImport += result;
        result.importedPackage := targetPackage;
}
//mappings
mapping FMI::FmiModelDescriptionType :: map2UMLPackage() : UML::Package {
        result.applyProfile(fmiProfile);
        result.name := self.coSimulation.modelIdentifier->any(true);
        var classes := fmu.rootObjects()[FMI::FmiModelDescriptionType].map map2UMLClass(result);
        var dependencies := fmu.objectsOfType(FMI::ModelStructureType).map map2Dependencies(result);
}
mapping FMI::FmiModelDescriptionType :: map2UMLClass(inout mypackage : UML::Package) : UML::Class{
        mypackage.packagedElement += result;
        result.name := self.coSimulation.modelIdentifier->any(true);
        result.applyStereotype(cs_stereotype);
        result.ownedAttribute := self.modelVariables.scalarVariable.map map2UMLAttributes(result)
                                                 ->   union   (self.modelVariables.scalarVariable.   map
map2UMLPorts(result));
        setClassStereotypeValues(result, self);
        generalization := fmu2ProxyGeneralization;
}

mapping FMI::Fmi2ScalarVariable :: map2UMLAttributes(inout myClass : UML::Class) : UML::Property
when{
        self -> exists(s|s.causality <> CausalityType::input and s.causality <> CausalityType::output)
}
{
        myClass.ownedAttribute +=result;
        result.name := self.name;
        var stereotypeToApply := findStereotype(self);
        result.applyStereotype(stereotypeToApply);
        setPropertyStereotypeValues(result,stereotypeToApply,self);
        setPropertyType(result,self);
        globalPropertiesList += result;
}
when{
        self -> exists(s|s.causality = CausalityType::input or s.causality = CausalityType::output)
}
{
        myClass.ownedAttribute += result;
        result.name := self.name;
        result.applyStereotype(port_stereotype);
        setPropertyStereotypeValues(result,port_stereotype,self);
        setPropertyType(result,self);
        globalPropertiesList += result;
}

mapping   FMI::ModelStructureType   ::   map2Dependencies(inout   myPackage   :   UML::Package)   :   Se-
quence(UML::Dependency){
        init{
                result   :=   self.outputs.map   map2Dependencies(myPackage,outputDependency_stereotype)-
>asSequence()
                                -> union  (self.derivatives.map  map2Dependencies(myPackage,derivative-
Dependency_stereotype)->asSequence())
                                ->  union   (self.initialUnknowns.map   map2Dependencies(myPackage,ini-
tialUnknownDependency_stereotype)->asSequence());
        }
}
```

```
mapping FMI::Fmi2ScalarVariable :: map2UMLPorts(inout myClass : UML::Class) : UML::Port

mapping FMI::InitialUnknownsType :: map2Dependencies(inout myPackage : UML::Package, in sterestotype-
ToApply : UML::Stereotype) :  Sequence(UML::Dependency){
        init{
                result := self.unknown.map map2Dependencies(myPackage, sterestotypeToApply);
        }
}
mapping FMI::UnknownType1 :: map2Dependencies(inout myPackage : UML::Package, in sterestotypeToApply :
UML::Stereotype) : Sequence(UML::Dependency) {
        init{
                self.dependencies->forEach(dependency) {
                        result  +=  self.map  map2Dependency(myPackage,  sterestotypeToApply,  depend-
ency.toString().toInteger());
                };
        }
}
mapping FMI::UnknownType1 :: map2Dependency(inout myPackage : UML::Package, in sterestotypeToApply :
UML::Stereotype, in supplier_index : Integer) : UML::Dependency {
        myPackage.packagedElement += result;
        var client_index := self.index.toString().toInteger();
        result.client := globalPropertiesList->at(client_index);
        result.supplier += globalPropertiesList -> at(supplier_index);
        result.applyStereotype(sterestotypeToApply);
}
mapping FMI::UnknownType :: map2Dependencies(inout myPackage : UML::Package, in sterestotypeToApply :
UML::Stereotype) : Sequence(UML::Dependency) {
        init{
                self.dependencies->forEach(dependency) {
                        result  +=  self.map  map2Dependency(myPackage,  sterestotypeToApply,  depend-
ency.toString().toInteger());
                };
        }
}
mapping FMI::UnknownType :: map2Dependency(inout myPackage : UML::Package, in sterestotypeToApply :
UML::Stereotype, in supplier_index : Integer) : UML::Dependency {
        myPackage.packagedElement += result;
        var client_index := self.index.toString().toInteger();
        result.client := globalPropertiesList->at(client_index);
        result.supplier += globalPropertiesList -> at(supplier_index);
        result.applyStereotype(sterestotypeToApply);
}

mapping FMI::UnknownType1 :: map2Dependency(inout myPackage : UML::Package, in sterestotypeToApply :
UML::Stereotype, in supplier_index : Integer) : UML::Dependency {

        myPackage.packagedElement += result;
        var client_index := self.index.toString().toInteger();
        result.client := globalPropertiesList->at(client_index);
        result.supplier += globalPropertiesList -> at(supplier_index);
        result.applyStereotype(sterestotypeToApply);
}

mapping FMI::UnknownType :: map2Dependencies(inout myPackage : UML::Package, in sterestotypeToApply :
UML::Stereotype) : Sequence(UML::Dependency) {
        init{
                self.dependencies->forEach(dependency) {
                        result  +=  self.map  map2Dependency(myPackage,  sterestotypeToApply,  depend-
ency.toString().toInteger());
                };
        }
}

mapping FMI::UnknownType :: map2Dependency(inout myPackage : UML::Package, in sterestotypeToApply :
UML::Stereotype, in supplier_index : Integer) : UML::Dependency {
        myPackage.packagedElement += result;
        var client_index := self.index.toString().toInteger();
        result.client := globalPropertiesList->at(client_index);
        result.supplier += globalPropertiesList -> at(supplier_index);
        result.applyStereotype(sterestotypeToApply);
}
```

Figure C-4. The QVTo transformation

Figure C-5 depicts the steps to follow in Papyrus for the import of an FMU for co-simulation in a Papyrus project.

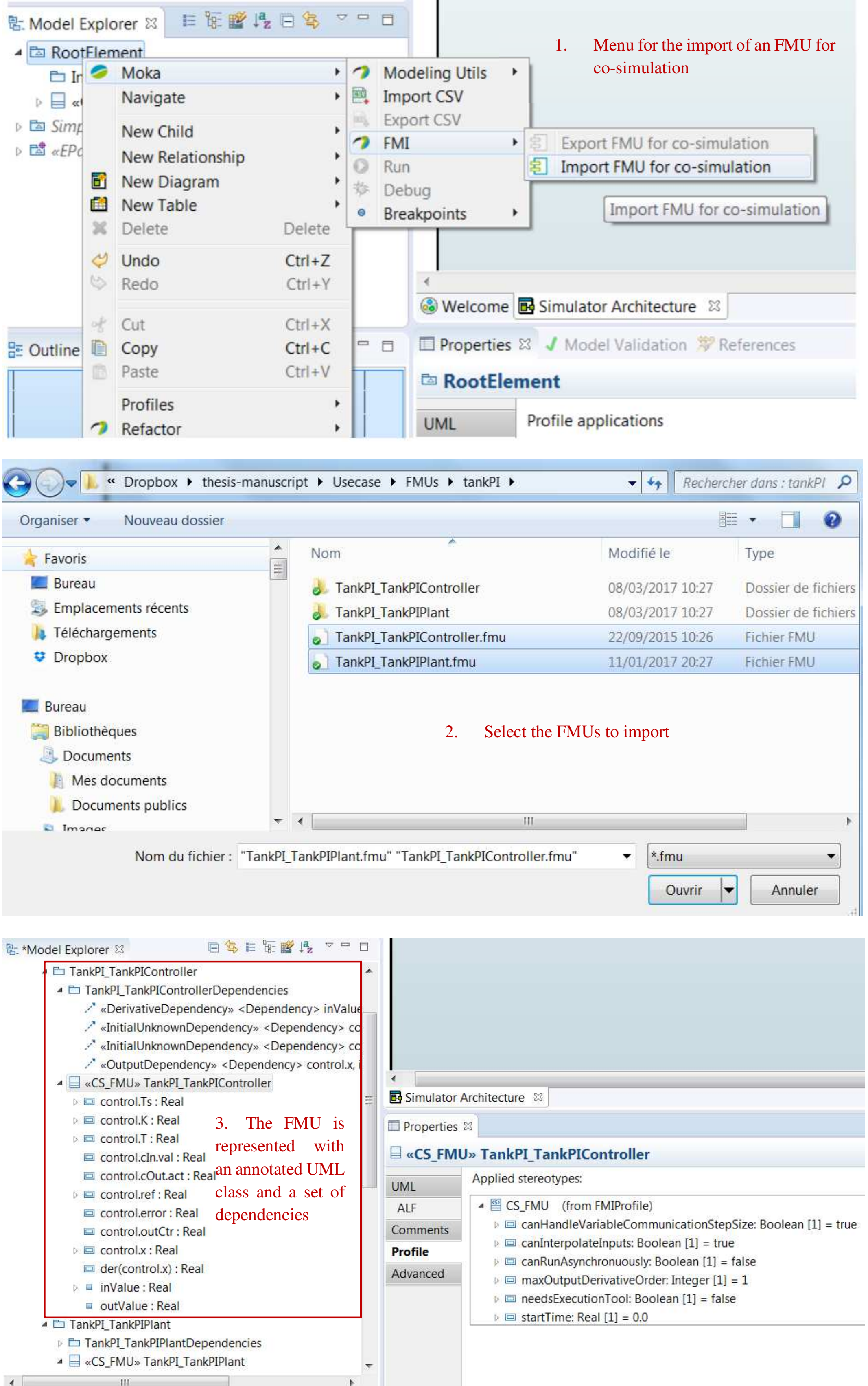

Figure C-5. The import of FMUs in Papyrus/Moka

C.2.1.2. Definition of a co-simulation scenario in Papyrus/Moka

The definition of a co-simulation scenarios consists in the assembly of imported FMUs in a composite class, and the configuration of the simulation parameters (start simulation time, stop simulation time and the step size). The connection of the imported FMUs in the simulator is done by a simple drag and drop of the classes representing the imported FMUs and their connection using connectors. The configuration of the simulation is done via the stereotype 'CS_Graph' applied to the Simulator as depicted in Figure C-6. No additional implementation is required for this task.

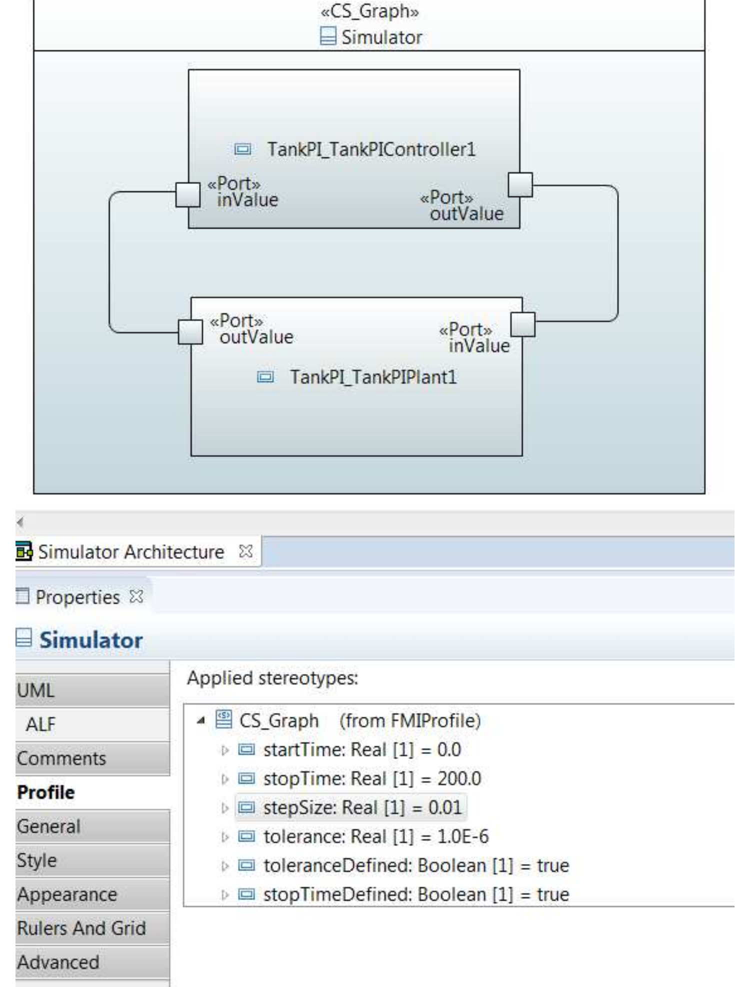

Figure C-6. The definition of co-simulation scenario in papyrus

C.2.1.3. Simulation of co-simulation scenarios in Papyrus/Moka

Moka provides a master algorithm specified by an executable UML model (Figure 4-9 of chapter 4), along with a dedicated model library. The current version of Papyrus/Moka supports the co-simulation of imported FMUs using the master algorithm represented in Figure 4-10 of chapter 4.

The simulation can be launched after the definition of a configuration run as illustrated in Figure C-7. An execution engine for co-simulation, the so-called 'org.eclipse.papyrus.moka.fuml.co-simulation', is defined as an extension to the fUML* execution engine (i.e. the interpreter implementing the semantics defined in fUML* specifications).

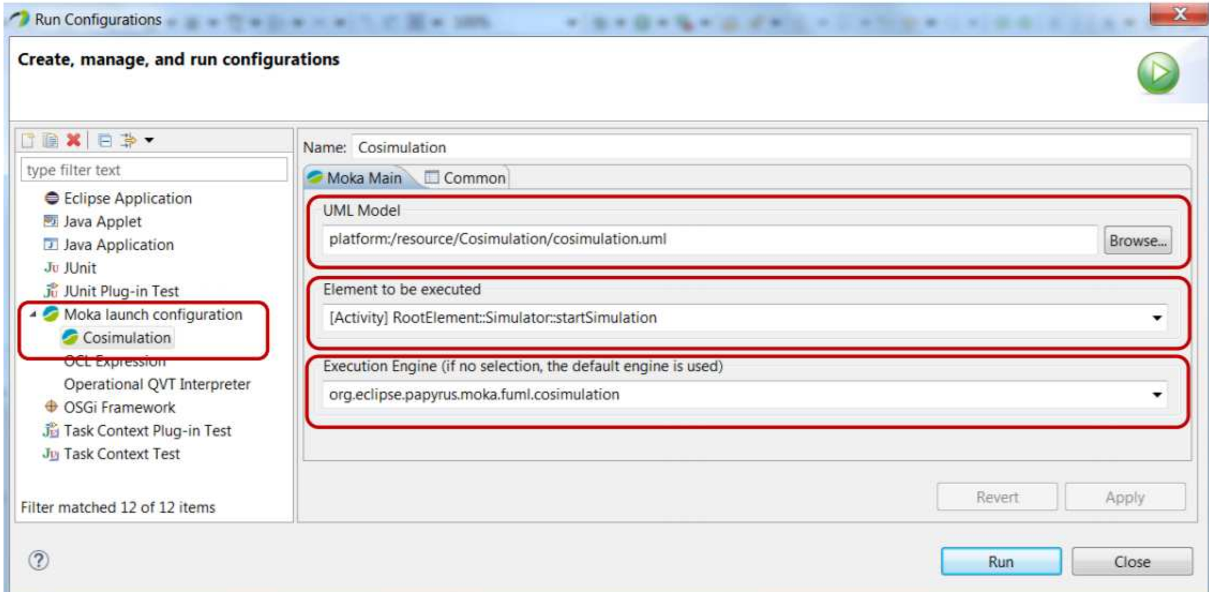

Figure C-7. The definition of a run configuration in papyrus/Moka

The simulation results are saved in a CSV file and visualized with XY charts integrated with papyrus as depicted in Figure 4-15 at the end of chapter 4.

C.2.2. Moka as a slave for co-simulation

In this case, Moka is a provider of FMUs for co-simulation. It enables to export FMUs from UML models by model transformation from FMU to UML models and by wrapping the UML execution semantics into the FMI API. For instance, there is some restrictions on the kind of supported model elements (i.e. only a subset of fUML* is supported) and on the capabilities of the exported FMUs (e.g. no support for rollback). Current works aims at enlarging the scope of the models we can export as FMUs for co-simulation to enable the use of UML models on other simulation tools.

This scenario is out of the scope of this work. Further information about the Papyrus tool support of FMI for both scenarios (i.e. master and slave) can be found in [28] ,[29] and [30].

[28] Refer to: modprod2017-tutorial-Papyrus-MOKA-FMI-Cosimulation
[29] Refer to: Papyrus-UserGuide for ModelExecution
[30] Refer to: youtube-Papyrus-chain

ANNEX D: Topological sort on directed graphs

Outline

D.1. Introduction to topological sort on directed graphs
D.2. Application to a co-simulation graph

In the section D.1. of this annex, we give general definition of topological sort, and the algorithm proposed by kahn for topological sort on directed graph. In the second section D.2, we illustrate the application of this algorithm for analysis of co-simulation scenarios.

D.1. Introduction to topological sort on directed graphs

A directed graph is a graph that is a set of vertices connected by edges, where the edges have a direction associated with them. A topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

The vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another. A topological ordering is possible if and only if the graph has no directed cycles. This latter is called directed acyclic graph (DAG). Any DAG has at least one topological ordering.

The Kahn's algorithm given in Figure D-1 is one of the algorithms used for topological sorting of directed graph.

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while S is non-empty do
   remove a node n from S
   add n to tail of L
   for each node m with an edge e from n to m do
      remove edge e from the graph
      if m has no other incoming edges then
         insert m into S
if graph has edges then
   return error (graph has at least one cycle)
else
   return L (a topologically sorted order)
```

Figure D-1. Kahn's algorithm for topological sort on directed graph

First, it finds a list of "start nodes" which have no incoming edges and insert them into a set S; at least one such node must exist in a non-empty acyclic graph.

Then:

- If the graph is a DAG, a solution will be contained in the list L (the solution is not necessarily unique).
- Otherwise, the graph must have at least one cycle and therefore a topological sorting is impossible

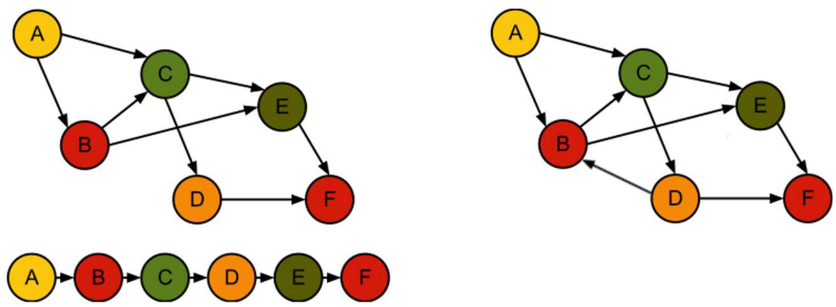

Figure D-3-a. Directed Acyclic graph **Figure D-3-b.** Directed cyclic graph

Figure D-3-a and Figure D-3-b depict two directed graphs composed of six vertices. The first one is acyclic. A topological order is then possible to make. The algorithm returns the list given below the dependency graph (L=A→B→C→D→E→F). The second one is a cyclic (cycle=B→C→D→B). A topological sorting is not possible to make.

D.2. Application to a co-simulation graph

A co-simulation graph includes a set of components. Each of them has ports through which data are propagated to the components connected to its outputs. We identify two kinds of dependencies between inputs and outputs ports: (a) external dependencies and (b) internal dependencies.

(a) An external dependency is expressed with a connector. A connection between an output ‘O’ to an input ‘I’ means that ‘I’ depends on ‘O’ and that the value of ‘I’ cannot be set before getting the value of ‘O’,
(b) An internal dependency is expressed with an UML dependency. If an I/O dependency exists between an output ‘O’ and an input ‘I’, that means the value of ‘O’ depends on the value of ‘I’.

In order to ensure the correctness of the propagated data, one should account for these dependencies. That is, we need to find a valid order in which the data are get/set from/to the FMUs ports. For this reason, a dependency graph is built such that: Vertices are ports whose value is get (if it is an output port) or set (if it is an input port), and edges are dependencies between two ports.

References

[1] IEEE standard for modeling and simulation (M&S) high level architecture (HLA)– framework and rules. *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*, pages 1–38, Aug 2010.

[2] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in event-b. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, 2010.

[3] Ahmad Alkhodre, Jean-Philippe Babau, and J.-J Schwarz. Modelling of real-time constraints using sdl for embedded systems design. 13:189 – 196, 09 2002.

[4] Henric Andersson, Erik Herzog, Gert Johansson, and Olof Johansson. Experience from introducing unified modeling language-systems modeling language at saab aerosystems. *Syst. Eng.*, 13(4):369–380, November 2010.

[5] M. U. Awais, P. Palensky, A. Elsheikh, E. Widl, and S. Matthias. The High Level Architecture rti as a master to the functional mock-up interface components. In *2013 International Conference on Computing, Networking and Communications (ICNC)*, pages 315–320, Jan 2013.

[6] Felice Balarin, Luciano Lavagno, Claudio Passerone, Alberto L. Sangiovanni-Vincentelli, Marco Sgroi, and Yosinori Watanabe. Modeling and designing heterogeneous systems. In *Concurrency and Hardware Design, Advances in Petri Nets*, pages 228–273, London, UK, UK, 2002. Springer-Verlag.

[7] Jens Bastian, Christoph ClauÃŸ, Susann Wolf, and Peter Schneider. P.: Master for co-simulation using fmi. In *8th International Modelica Conference*, 2011.

[8] Abderraouf Benyahia, Arnaud Cuccuru, Safouan Taha, François Terrier, Frédéric Boulanger, and Sébastien Gérard. Extending the standard execution model of UML for real-time systems. In *Distributed, Parallel and Biologically Inspired Systems - 7th IFIP TC 10 Working Conference, DIPES 2010 and 3rd IFIP TC 10 International Conference, BICC 2010, Held as Part of WCC 2010, Brisbane, Australia, September 20-23, 2010. Proceedings*, pages 43–54, 2010.

[9] Gérard Berry. The foundations of esterel, 1998.

[10] Frédéric Boulanger, Cécile Hardebolle, Christophe Jacquet, and Dominique Marcadet. Semantic adaptation for models of computations. In Benoit Caillaud, Josep Carmona, and Kunihiko Hiraishi, editors, *Proceedings of the 11th International Conference on Application of Concurrency to System Design*, pages 153–162. IEEE Computer Society, 2011.

[11] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. Execution Framework of the GEMOC Studio (Tool Demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, page 8, Amsterdam, Netherlands, October 2016.

[12] David Broman, Christopher Brooks, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. Determinate composition of fmus for co-simulation. In

Proceedings of the Eleventh ACM International Conference on Embedded Software, EMSOFT '13, pages 2:1–2:12, Piscataway, NJ, USA, 2013. IEEE Press.

[13] Joachim Denil, Bart Meyers, Paul De Meulenaere, and Hans Vangheluwe. Explicit semantic adaptation of hybrid formalisms for FMI co-simulation. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, part of the 2015 Spring Simulation Multiconference, SpringSim '15, Alexandria, VA, USA, April 12-15, 2015*, pages 99–106, 2015.

[14] Matthew Emerson and Janos Sztipanovits. Techniques for metamodel composition. In *OOPSLA 6th Workshop on Domain Specific Modeling*, pages 123–139, October 2006.

[15] J. Eyer, G. Corey, and Sandia National Laboratories. *Energy Storage for the Electricity Grid: Benefits and Market Potential Assessment Guide: a Study for the DOE Energy Storage Systems Program.* SAND (Series) (Albuquerque, N.M.). Sandia National Laboratories, 2010.

[16] Yishai A. Feldman, Lev Greenberg, and Eldad Palachi. Simulating rhapsody sysml blocks in hybrid models with fmi. In *Proceedings of the 10th International Modelica Conference; March 10-12; 2014; Lund; Sweden*, number 96, pages 43–52. Linköpings University Electronic Press; Linköpings universitet, 2014.

[17] FMI. Functional mock-up interface for model exchange and co-simulation, October 2013.

[18] Sahar Guermazi, Jérémie Tatibouet, Arnaud Cuccuru, Ed Seidewitz, Saadia Dhouib, and Sébastien Gérard. Executable modeling with fuml and alf in papyrus: Tooling and experiments. In *EXE@MoDELS*, 2015.

[19] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[20] Cécile Hardebolle and Frédéric Boulanger. Exploring multi-paradigm modeling techniques. *SIMULATION: Transactions of The Society for Modeling and Simulation International*, 85(11/12):688–708, November/December 2009.

[21] D. Harel and A. Pnueli. Logics and models of concurrent systems. chapter On the Development of Reactive Systems, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[22] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[23] Walid Hassairi, Moncef Bousselmi, Mohamed Abid, and Carlos Valderrama. Matlab/systemc for the new co-simulation environment by jpeg algorithm. In Vasilios N. Katsikis, editor, *MATLAB - A Fundamental Tool for Scientific Computing and Engineering Applications - Volume 2*, chapter 06. InTech, Rijeka, 2012.

[24] Jozef Hooman, Nataliya Mulyar, and Ladislau Posta. Coupling simulink and UML models. 2004.

[25] Edward A. Lee. Cyber physical systems: Design challenges. In *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, ISORC '08, pages 363–369, Washington, DC, USA, 2008. IEEE Computer Society.

[26] Anu Maria. Introduction to modeling and simulation. In *Proceedings of the 29th Conference on Winter Simulation*, WSC '97, pages 7–13, Washington, DC, USA, 1997. IEEE Computer Society.

[27] Richard E. Nance. History of programming languages—ii. chapter A History of Discrete Event Simulation Programming Languages, pages 369–427. ACM, New York, NY, USA, 1996.

[28] Seyed Hosein Attarzadeh Niaki and Ingo Sander. Co-simulation of embedded systems in a heterogeneous moc-based modeling framework. In *Industrial Embedded Systems*, pages 238–247, 2011.

[29] OMG. Uml profile for marte: Modeling and analysis of real-time embedded systems, http://www.omg.org/spec/marte/, 2011, 2011.

[30] OMG. Unified modeling language (uml), http://www.omg.org/spec/uml/2.5/, 2015, 2015.

[31] OMG. Query/view/transformation (qvt), http://www.omg.org/spec/qvt/, 2016, 2016.

[32] OMG. Semantics of a foundational subset for executable uml models(fuml), http://www.omg.org/spec/fuml/1.2.1./, 2016, 2016.

[33] Uwe Pohlmann, Wilhelm SchÃ¤fer, Hendrik Reddehase, Jens Röckemann, and Robert Wagner. Generating functional mockup units from software specifications. In *Proceedings of the 9th International MODELICA Conference; September 3-5; 2012; Munich; Germany*, number 76, pages 765–774. Linköpings University Electronic Press; Linköpings universitet, 2012.

[34] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[35] Ragunathan (Raj) Rajkumar, Insup Lee, Lui Sha, and John Stankovic. Cyber-physical systems: The next computing revolution. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 731–736, New York, NY, USA, 2010. ACM.

[36] Vitaly Savicks, Michael Butler, and John Colley. Co-simulating event-b and continuous models via fmi. In *Proceedings of the 2014 Summer Simulation Multiconference*, SummerSim '14, pages 37:1–37:8, San Diego, CA, USA, 2014. Society for Computer Simulation International.

[37] Tom Schierz, Martin Arnold, and Christoph Claud. Co-simulation with communication step size control in an fmi compatible master algorithm. In *Proceedings of the 9th International MODELICA Conference; September 3-5; 2012; Munich; Germany*, number 76, pages 205–214. Linköpings University Electronic Press; Linköpings universitet, 2012.

[38] J. Sztipanovits. Composition of cyber-physical systems. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, pages 3–6, March 2007.

[39] Philip Langer Tanja Mayerhofer. Moliz: a model execution framework for uml models. In *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards*, 2012.

[40] Jérémie Tatibouet, Arnaud Cuccuru, Sébastien Gérard, and François Terrier. Formalizing execution semantics of UML profiles with fuml models. In *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, pages 133–148, 2014.

[41] Jean-Philippe Tavella, Mathieu Caujolle, Charles Tan, Gilles Plessis, Mathieu Schumann, Stéphane Vialle, Cherifa Dad, Arnaud Cuccuru, and Sébastien Revol. Toward an Hybrid Co-simulation with the FMI-CS Standard, April 2016. Research Report.

[42] Stavros Tripakis and David Broman. Bridging the semantic gap between heterogeneous modeling formalisms and fmi. Technical Report UCB/EECS-2014-30, EECS Department, University of California, Berkeley, Apr 2014.

[43] Bert Van Acker, Joachim Denil, Hans Vangheluwe, and Paul De Meulenaere. Generation of an optimised master algorithm for fmi co-simulation. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, DEVS '15, pages 205–212, San Diego, CA, USA, 2015. Society for Computer Simulation International.

[44] K. Wan, D. Hughes, K. L. Man, and T. Krilavioius. Composition challenges and approaches for cyber physical systems. In *Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on*, pages 1–7, Nov 2010.

[45] Roel Wieringa. *Design Methods for Reactive Systems: Yourdan, Statemate, and the UML*. Morgan Kaufmann Publishers, Boston, 2003.

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

université PARIS-SACLAY

Titre : Co-simulation dirigée par les modèles des Systèmes Cyber-Physiques

Mots clés : CPS, Co-simulation, FMI, UML, fUML

Résumé : La conception des systèmes cyber-physiques (CPS) est réalisée à partir de plusieurs disciplines impliquant de multiples composants, physiques et autres cyber, interconnectés. La simulation d'un tel système nécessite une co-simulation des modèles associés à ces composants tout en assurant leur synchronisation. En particulier, FMI (Functional Mock-up Interface) est un standard de co-simulation très utilisé en industrie. Il offre une interface standard pour coupler plusieurs simulateurs dans un environnement de co-simulation, nommé « Master ». Celui-ci est chargé de fournir un algorithme pour une orchestration et une synchronisation efficaces des différents composants du système, nommés FMU (« Functional Mock-up Unit »). Cette norme s'impose de plus en plus dans l'industrie, et est supportée par de nombreux environnements de modélisation et de simulation. Cependant, FMI est initialement conçu pour la co-simulation des processus physiques, avec un support limité des formalismes à événements discrets qui est modèle de calcul et de communication largement utilisé dans les environnements de modélisation spécifiques au logiciel. En particulier, bien qu'UML soit un des langages de référence pour la modélisation de logiciels et soit très couramment utilisé dans l'industrie, aucune des solutions actuelles de co-simulation basées sur FMI ne permet de le prendre en considération.
La thèse défendue dans ce document est que l'ingénierie système en général bénéficierait énormément de l'intégration des modèles UML dans une approche de co-simulation basée sur la norme FMI. Cela permettra à un grand nombre de concepteurs logiciels d'évaluer le comportement de leurs composants logiciels dans un environnement simulé, et donc de les aider à faire les meilleurs choix de conception le plus tôt possible dans leur processus de développement. Cela pourrait également ouvrir de nouvelles perspectives intéressantes pour les ingénieurs système des CPS, en leur permettant d'envisager l'utilisation d'un langage largement utilisé pour la modélisation des composants logiciels de leurs systèmes. Dans ce contexte, l'objectif de cette thèse est de fournir un environnement de co-simulation pour les CPSs basé sur le standard FMI et qui prend en compte les modèles UML pour la partie logicielle. Nous mettons en place une approche de co-simulation où nous abordons différents types de composants caractérisant les composants logiciels d'un CPS. Notre contribution intervient à deux niveaux : localement au niveau des modèles UML, et globalement au niveau du « Master ». Localement, nous basons nos propositions sur les standards OMG, fUML (Semantics of a foundational subset for executable UML models) et PSCS (Precise Semantics of UML Composite Structures), qui définissent une sémantique d'exécution précise pour un sous-ensemble de UML. Pour chaque type de système, nous identifions un ensemble de règles pour le modéliser avec UML et les éventuelles extensions à fUML. Ensuite, au niveau global, nous proposons des algorithmes de « Master ». Ils assurent la synchronisation des composants du système en se basant sur une adaptation entre l'API FMI et la sémantique définie dans fUML, et sur un choix de pas de simulation adaptée à la nature de chaque composant. L'approche est illustrée par un cas d'utilisation du domaine des bâtiments intelligents, où l'objectif est d'évaluer différentes stratégies de contrôle (composants logiciels) pour l'optimisation de son autoconsommation en électricité.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

ÉCOLE DOCTORALE

Sciences et technologies de l'information et de la communication (STIC)

université PARIS-SACLAY

Title: Model-driven co-simulation of Cyber-Physical Systems

Keywords: CPS, Co-simulation, FMI, UML, fUML

Abstract: The design of cyber-physical systems (CPS) is realized from several disciplines involving multiple components, physical and other cyber, which are interconnected. The simulation of such a system requires a co-simulation of the models of these components and their synchronization. In particular, FMI (Functional Mock-up Interface) is a co-simulation standard widely used in industry. This latter is responsible for providing an algorithm with efficient orchestration and synchronization of the involved components, known as FMUs («Functional Mock-up Unit»). FMI standard is gaining popularity in the industry, and it is being supported by many modeling and simulation environments. However, FMI was originally intended for co-simulation of physical processes, with limited support for discrete event formalisms, even if this kind of formalism is commonly used to model the logic of software parts of a system. In particular, while UML is the reference standard for software modeling and is very commonly used in industry, none of the present-day FMI-based co-simulation solutions consider UML models.

Our thesis is that system engineering in general would greatly benefit from the consideration of UML in FMI-based co-simulation approach. It would indeed enable a significant number of software designers to evaluate the behavior of their software components in their simulated environment, as soon as possible in their development processes, and therefore make early and better design decisions. It would also open new interesting perspectives for CPS system engineers, by allowing them to consider a widely used modeling language for the software parts of their systems. In this context, the objective of this work is to define an FMI-based co-simulation environment for CPS with integration of UML models for the software part of the system. We set up a co-simulation approach where we address different kinds of systems characterizing the software part of CPS.

Our contribution is twofold: locally at the level of UML models, and globally at the master level. At the local level, we base our proposals on OMG standards fUML («Semantics of a foundational subset for executable UML Models») and PSCS («Precise Semantics of UML Composite Structures») which define precise execution semantics for a subset of UML. For each kind of system, we first identify a set of rules to model it with UML and potential extensions to fUML. Then, at the global level, we propose « Master » algorithms. They ensure the synchronization of the involved components based on an adaptation between the FMI API and the execution semantics defined in fUML, and on a choice of simulation step size adapted to the kind of each component. The approach is illustrated by a use case from the smart grids domain, where the objective is to evaluate different control strategies (software components) for the optimization of its electricity self-consumption.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France