An integrated language for the specification, simulation, formal analysis and enactment of discrete event systems

Oumar Maïga

▶ To cite this version:

Oumar Maïga. An integrated language for the specification, simulation, formal analysis and enactment of discrete event systems. Other [cs.OH]. Université Blaise Pascal - Clermont-Ferrand II, 2015. English. NNT : 2015CLF22662 . tel-01330780

HAL Id: tel-01330780

https://theses.hal.science/tel-01330780

Submitted on 13 Jun 2016

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.

N° d'ordre : D. U : 2662
E D S P I C: 738

UNIVERSITE BLAISE PASCAL - CLERMONT II
ECOLE DOCTORALE SCIENCES POUR L'INGENIEUR DE CLERMONT-FERRAND
&
UNIVERSITE DES SCIENCES, TECHNIQUES ET TECHNOLOGIES DE BAMAKO

Thèse

Présentée par

Oumar Y. MAÏGA

pour obtenir le grade de

DOCTEUR D'UNIVERSITÉ

SPECIALITE : Informatique

Titre de la thèse :

An Integrated Language for the Specification, Simulation, Formal Analysis and Enactment of Discrete Event Systems

Soutenue publiquement le 22/12/2015 devant le jury :

Prof. Jean-François Santucci	Rapporteur et examinateur
Prof. Claudia Frydman	Rapporteur et examinateur
Prof. Mamadou Kaba Traoré	Directeur de thèse
Prof. Ouaténi Diallo	Co-directeur de thèse
Prof. Moussa Lô	Examinateur
Dr. Jacqueline Konaté	Examinateur

// Abstract

This thesis proposes a methodology which integrates formal methods in the specification, design, verification and validation processes of complex, concurrent and distributed systems with discrete events perspectives. The methodology is based on the graphical language HILLS (High Level Language for System Specification) that we defined. HiLLS integrates software engineering and system theoretic views for the specification of systems. Precisely, HiLLS integrates concepts and notations from DEVS (Discrete Event System Specification), UML (Unified Modeling Language) and Object-Z. The objectives of HILLS include the definition of a highly communicable graphical concrete syntax and multiple semantic domains for simulation, prototyping, enactment and accessibility to formal analysis. Enactment refers to the process of creating an instance of system executing in real-clock time. HILLS allows hierarchical and modular construction of discrete event systems models while facilitating the modeling process due to the simple and rigorous description of the static, dynamic, structural and functional aspects of the models.

Simulation semantics is defined for HiLLS by establishing a semantic mapping between HiLLS and DEVS; in this way each HiLLS model can be simulated by a DEVS simulator. This approach allow DEVS users to use HiLLS as a modeling language in the modeling phase and use their own stand alone or distributed DEVS implementation package to simulate the models.

An enactment of HiLLS models is defined by adapting the observer design-pattern to their implementation.

The formal verification of HiLLS models is made by establishing morphisms between each level of abstraction of HILLS and a formal method adapted for the formal verification of the properties at this level. The formal models on which are made the formal verification are obtained from HILLS specifications by using the mapping functions. The three levels of abstraction of HILLS are: the Composite level, the Unitary level and the Traces level. These levels correspond respectively to the following levels of the system specification hierarchy proposed by Zeigler: CN (Coupled Network), IOS (Input Output System) and IORO (Input Output Relation Observation). We have established morphisms between the Composite level and CSP (Communicating Sequential Processes), between Unitary level and Z and we expect to use temporal logics like LTL, CTL and TCTL to express traces level properties. HiLLS allows the specification of both static and dynamic structure systems. In case of dynamic structure systems, the composite level integrates both sate-based and process-based properties. To handle at the same time state-based and process-based properties, morphism is established between the dynamic composite level and CSPZ (a combination of CSP and Z);

The verification and validation process combine simulation, model checking and theorem proving techniques in a common framework. The model checking and theorem proving of HILLS models are based on an integrated tooling framework composed of tools supporting the notations of the selected formal methods in the established morphisms.

We apply our methodology to modeling of the Alternating Bit Protocol (ABP) and the Automated Teller Machine (ATM).

The thesis has contributed to the following publications:

1. Ighoroje, U.B., Maïga, O., Traoré, M.K. The Formal Framework for the DEVS Driven Modeling Language. In Proceedings of European Modeling and Simulation Symposium (EMSS) 2011, Rome (Italy), September 2011.
2. Ighoroje, U.B., Maïga, O., Traoré, M.K.: The DEVS-driven modeling language: syntax and semantics definition by meta-modeling and graph transformation. In: Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium Article No. 49. Society for Computer Simulation International San Diego, CA, USA (2012).
3. Maïga O., Ighoroje U.B. and Traoré M.K. Intégration des Méthodes formelles dans la spécification et la vérification et validation des modèles de simulation. In Proceedings of MOSIM 2012, Bordeaux (France), June 2012.
4. Maïga O., Ighoroje U.B. and Traoré M.K.. DDML: A support for communication in Modeling and Simulation. 3rd IEEE Track on Collaborative Modeling and Simulation-CoMetS'12. Toulouse (France), June 2012.
5. Maïga O and Traoré M.K. Approche Formelle de vérification et validation des modèles de simulation. In : Journées Scientifiques de l'Ecole Doctorale des Sciences pour l'Ingénieur (EDSPI), Clermont-Ferrand (France), June 2013.
6. Maïga O. and Traoré M.K. An Integrated approach to the specification, simulation, formal analysis and enactment of discrete event systems. AUSTECH 2015, Abuja, Nigeria, October 2015.
7. Aliyu H. O., Maïga O., Abdoul-Wahab H. I. and Traoré M.K. Introducing HiLLS: High Level Language for System Specification. AUSTECH 2015, Abuja, Nigeria, October 2015. ***Best Presentation Award***.
8. Maïga O., Aliyu H. O. and Traoré M.K. A New Approach to Modeling Dynamic Structure Systems. Accepted for publication at European Simulation and Modeling Conference (ESM'2015), Leicester, United Kingdom, October 2015.
9. Aliyu H. O., Maïga O. and Traoré M.K. A Framework for Discrete Event Systems enactment. Accepted for publication at European Simulation and Modeling Conference (ESM'2015), Leicester, United Kingdom, October 2015.

Papers under review:

1. Aliyu H. O., Maïga O. and Traoré M.K. AnnoGram4MD: A Language for Annotating Grammars for High Quality Metamodel Derivation. Submitted to the 28th International Conference on Software Engineering, Austin,TX, May 14-22, 2016
2. Aliyu H. O., Maïga O., and Traoré M.K. Yet another Visual Language for DEVS. International Journal of Modeling and Simulation and Scientific Computing.

Keywords: HiLLS, System Theory, Software Engineering, Modeling and Simulation, formal methods, Verification and Validation, Enactment

Résumé

Titre: Un Langage Intégré pour la Spécification, Simulation, Analyse Formelle et En-action des Systèmes à évènements discrets.

Cette thèse propose une méthodologie qui intègre les méthodes formelles dans la spécification, la conception, la vérification et la validation des systèmes complexes concurrents et distribués avec une perspective à événements discrets. La méthodologie est basée sur le langage graphique HILLS (High Level Language for System Specification) que nous avons défini. HiLLS intègre des concepts de génie logiciel et de théorie des systèmes pour une spécification des systèmes. Précisément, HiLLS intègre des concepts et notations de DEVS (Discrete Event System Specification), UML (Unified Modeling Language) et Object-Z. Les objectifs de HILLS incluent la définition d'une syntaxe concrète graphique qui facilite la communicabilité des modèles et plusieurs domaines sémantiques pour la simulation, le prototypage, l'enaction et l'accessibilité à l'analyse formelle. L'Enaction se définit par le processus de création d'une instance du système qui s'exécute en temps réel (par opposition au temps virtuel utilisé en simulation). HiLLS permet la construction hiérarchique et modulaire des systèmes à événements discrets grâce à une description simple et rigoureuse des aspects statiques, dynamiques et fonctionnels des modèles.

La sémantique pour simulation de HiLLS est définie en établissant un morphisme sémantique entre HiLLS et DEVS; de cette façon chaque modèle HiLLS peut être simulé en utilisant un simulateur DEVS. Cette approche permet aux utilisateurs DEVS d'utiliser HiLLS comme un langage de spécification dans la phase de modélisation et d'utiliser leurs propres implémentations locales ou distribuées de DEVS en phase de simulation.

L'enactment des modèles HiLLS est basé sur une adaptation du patron de conception Observateur pour leur implémentation.

La vérification formelle est faite en établissant un morphisme entre chaque niveau d'abstraction de HiLLS et une méthode formelle adaptée pour la vérification formelle des propriétés à ce niveau. Les modèles formels sur lesquels sont faites les vérifications formelles sont obtenus à partir des spécifications HiLLS en utilisant des morphismes. Les trois niveaux d'abstraction de HiLLS sont : le niveau composite, le niveau unitaire et le niveau des traces. Ces niveaux correspondent respectivement aux trois niveaux suivants de la hiérarchie de spécification des systèmes proposée par Zeigler : CN (Coupled Network), IOS (Input Output System) et IORO (Input Output Relation Observation). Nous avons établi des morphismes entre le niveau Composite et CSP (Communicating Sequential Processes), entre le niveau unitaire et Z, et nous utilisons les logiques temporelles telles que LTL, CTL et TCTL pour exprimer les propriétés sur les traces. HiLLS permet à la fois la spécification des modèles à structures statiques et les modèles à structures variables. Dans le cas des systèmes à structures variables, le niveau composite intègre à la fois des propriétés basées sur les états et les processus. Pour prendre en compte ces deux aspects, un morphisme est défini entre le niveau Composite de HiLLS et CSPZ (une combinaison de CSP et Z).

Le processus de vérification et de validation combine la simulation, la vérification exhaustive de modèle (model checking) et la preuve de théorèmes (theorem proving) dans un Framework

commun. La vérification exhaustive et la preuve de théorèmes sur les modèles HiLLS sont basées sur les outils associés aux méthodes formelles sélectionnées dans les morphismes.

Nous appliquons la méthodologie de modélisation de HiLLS à la modélisation du Alternating Bit Protocol (ABP) et à celle d'un guichet automatique de dépôt de billet (Automated Teller Machine) (ATM).

Cette thèse a contribué aux publications suivantes:

1. Ighoroje, U.B., Maïga, O., Traoré, M.K. The Formal Framework for the DEVS Driven Modeling Language. In Proceedings of European Modeling and Simulation Symposium (EMSS) 2011, Rome (Italy), September 2011.
2. Ighoroje, U.B., Maïga, O., Traoré, M.K.: The DEVS-driven modeling language: syntax and semantics definition by meta-modeling and graph transformation. In: Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium Article No. 49. Society for Computer Simulation International San Diego, CA, USA (2012).
3. Maïga O., Ighoroje U.B. and Traoré M.K. Intégration des Méthodes formelles dans la spécification et la vérification et validation des modèles de simulation. In Proceedings of MOSIM 2012, Bordeaux (France), June 2012.
4. Maïga O., Ighoroje U.B. and Traoré M.K.. DDML: A support for communication in Modeling and Simulation. 3rd IEEE Track on Collaborative Modeling and Simulation-CoMetS'12. Toulouse (France), June 2012.
5. Maïga O and Traoré M.K. Approche Formelle de vérification et validation des modèles de simulation. In : Journées Scientifiques de l'Ecole Doctorale des Sciences pour l'Ingénieur (EDSPI), Clermont-Ferrand (France), June 2013.
6. Maïga O. and Traoré M.K. An Integrated approach to the specification, simulation, formal analysis and enactment of discrete event systems. AUSTECH 2015, Abuja, Nigeria, October 2015.
7. Aliyu H. O., Maïga O., Abdoul-Wahab H. I. and Traoré M.K. Introducing HiLLS: High Level Language for System Specification. AUSTECH 2015, Abuja, Nigeria, October 2015. ***Prix de la Meilleure Présentation***.
8. Maïga O., Aliyu H. O. and Traoré M.K. A New Approach to Modeling Dynamic Structure Systems. Accepted for publication at European Simulation and Modeling Conference (ESM'2015), Leicester, United Kingdom, October 2015.
9. Aliyu H. O., Maïga O. and Traoré M.K. A Framework for Discrete Event Systems enactment. Accepted for publication at European Simulation and Modeling Conference (ESM'2015), Leicester, United Kingdom, October 2015.

Articles en cours d'évaluation :

1. Aliyu H. O., Maïga O. and Traoré M.K. AnnoGram4MD: A Language for Annotating Grammars for High Quality Metamodel Derivation. Submitted to the 28th International Conference on Software Engineering, Austin,TX, May 14-22, 2016
2. Aliyu H. O., Maïga O. and Traoré M.K. Yet another Visual Language for DEVS. International Journal of Modeling and Simulation and Scientific Computing.

Mots clés: HiLLS, Théorie des Systèmes, Génie logiciel, Modélisation et Simulation, Méthodes Formelles, Vérification et Validation, Enaction

Acknowledgment/Remerciements

Je tiens tout particulièrement à exprimer ma plus profonde gratitude à Pr Mamadou Kaba Traoré de l'Université Blaise Pascal de Clermont, pour avoir dirigé ces travaux et m'avoir soutenu dans toutes les phases. Je remercie également mon Co-directeur de thèse Pr Ouaténi Diallo pour son accompagnement et ses conseils précieux.

Mes remerciements vont aux mes membres du GReP (Groupe de Recherche du Pôle) avec mention spéciale à Hamzat Aliyu pour sa collaboration et les nuits blanches à travailler sur HiLLS. Mes remerciements vont également à l'endroit de Bright Ighoroje et Hamidou Togo pour leurs apports et observations au développement de la thèse.

Je remercie le Chef de D.E.R de mathématiques et d'informatique Dr Yaya Koné pour son suivi dans les démarches administratives ainsi que toute l'administration de la Faculté des Sciences et Techniques (FST) de l'Université de Sciences, des Techniques et Technologie de Bamako (USTTB). Mes remerciements vont également à Pr Françoise Paladian, directrice de l'Ecole Doctorale des Sciences pour l'Ingénieur (EDSPI) de Clermont-Ferrand ainsi qu'à toute son équipe pour leur soutien administratif.

Je remercie Mme Jacqueline Girard, directrice du Pôle Universitaire et Technologique de Vichy (PUTV) et toute son équipe avec une mention spéciale pour Isabelle Vairet. Je remercie toute l'équipe du LIMOS (Laboratoire d'Informatique de Modélisation et d'Optimisation des Systèmes) de Clermont-Ferrand pour leur accueil pendant toute la durée de la thèse.

Mes remerciements vont à tout le staff de l'AUF (Agence Universitaire de la Francophonie), plus particulièrement Michel Namar et Birama Seyba Traoré pour leur soutien et accompagnement.

Mes remerciements vont également au coordinateur du PFF (Programme de Formation des Formateurs) pour sa disponibilités et son accompagnement pendant la formation doctorale et pour les supports accordés pour les conférences scientifiques.

Je tiens enfin, à remercier ma famille et mes amis pour leurs encouragements, et particulièrement Bintou pour avoir supporté mes multiples longues absences et m'avoir soutenu pendant tout le projet.

Table of Content

List of figures 10

List of abbreviations 13

I. General introduction 14

I.1 Context 15

I.2 Objectives 17

I.3 Outline 18

II. State of the art 19

II.1 Introduction 20

II.2 Modeling and Simulation 20

II.2.1 Simulation life Cycle 20

II.2.2 Verification and Validation 21

II.2.3 DEVS-Based Modeling and Simulation Framework 23

II.3 Formal Methods 46

II.3.1 Z language 48

II.3.2 Object-Z 49

II.3.3 CSP 52

II.3.4 Formal Verification and Validation of DEVS models 53

II.4 System Design 57

II.4.1 Unified Modeling Language 57

II.4.2 Object-Oriented design patterns 59

II.4.3 Enactment of discrete event systems 60

II.5 Language engineering 62

II.5.1. Definition of a Language 62

II.5.2 Visual Languages 63

II.5.3 Integrating Heterogeneous languages 65

II.5.4 Language integration techniques 69

II.5.5 Integrated languages for system specification 71

II.6 Conclusion 73

III. The High Level Language for Systems Specification 75

III.1 Introduction 76

III.2 Informal Presentation of HiLLS 77
III.3 HiLLS Abstract Syntax 80
III.3.1 Object-Z concepts Metamodel 81
III.3.2 System Theory concepts from DEVS 83
III.3.3 Object Orientation Concepts from EMOF 86
III.3.4 HiLLS Complete Metamodel 87
III.4 HiLLS Concrete Syntax 92
III.5 Queuing Network example 94
III.6 Set-theoretic semantics of HiLLS 98
III.6.1 Formalization 99
III.6.2 Closure under coupling of HILLS 109
III.7 HiLLS Usability Assessment 112
III.8 Conclusion 113
IV. Translational Semantics of HiLLS 115
IV.1 Introduction 116
IV.2 Operational Semantics for Simulation 117
IV.2.1 Semantic mapping to DEVS 118
IV.2.2 Semantic mapping to DSDEVS 122
IV.3 Operational Semantics for Enactment 124
IV.3.1 Enactment 124
IV.3.2 Enactment Methodology 124
IV.3.3 Metamodel of the framework 125
IV.3.4 Enactment protocol 126
IV.3.5 Implementation 127
IV.3.5 Traffic Light Example 127
IV.4 Semantic mapping to Z 132
IV.5 Semantic mapping to CSP 139
IV.6 Tooling Framework 144
VI.7 Conclusion 145
V. Application 146
V.1 Introduction 147
V.2 Alternating Bit Protocol 148

V.2.1 HiLLS specification of ABP 149
V.2.2 Equivalent models of the ABP in DEVS 156
V.2.3 Z specification and Analysis of the ABP 159
V.3 Automated Teller Machine (ATM) 166
V.3.1 HiLLS specification of ATM 167
V.3.2 DSDEVS specification of ATM 179
V.4 Conclusion 184
VI. General conclusion 185
References 188

List of figures

Figure 1: Simulation Life Cycle 21
Figure 2. Taxanomy of Verification, Validation and Testing Techniques [Balci 1997] 23
Figure 3. M&S Entities and Relationships between them [Zeigler et al. 2000] 24
Figure 4. Experimental Frame and its Components [Zeigler 1983] 25
Figure 5. System at OF level 27
Figure 6. Example of transitions occurrences 30
Figure 7. Example of Coupled Model 31
Figure 8. Correspondence between Model Hierarchy and Simulation Tree 31
Figure 9. Sending of initialization message 32
Figure 10. Sending of done messages to parents 32
Figure 11. Sending star message 33
Figure 12. Simstudio components 39
Figure 13. the Stack specification in Z 49
Figure 14. Template of Object-Z class 50
Figure 15. A sample model in Object-Z 51
Figure 16. Observer design pattern 59
Figure 17. Command design pattern 60
Figure 18. Defining a language 62
Figure 19. Mapping between abstract and concrete syntaxes 64
Figure 20. HiLLS definition elements 76
Figure 21. Traffic Light and subordinate 77
Figure 22. Traffic Light 78
Figure 23. Behavior of SynchronizedTL 79
Figure 24. Behavior of OppositeTL 79
Figure 25. Crossroad control traffic lights 80
Figure 26. Object-Z Metamodel 82
Figure 27. Discrete Event (part) Metamodel 84
Figure 28. OCL Constraints of Discrete Event Metamodel 85
Figure 29. HiLLS Object Oriented part metamodel 87
Figure 30. Complete Metamodel integrating the different parts 89
Figure 31. OCL constraints 90
Figure 32. HiLLS Concrete Syntax 94
Figure 33. Queueing Network 95
Figure 34. HSystem of the Queue 97
Figure 35. Survey results 113
Figure 36. From HILLS to Formal Methods 116
Figure 37. HiLLS operational Semantics mappings 118
Figure 38. controlled Trafficlight 121
Figure 39. Observer pattern with asynchronous notification 125

Figure 40. Metamodel of enactment framework 126
Figure 41. Specification of traffic light system 127
Figure 42 Control unit of the traffic light system 129
Figure 43 Display unit of the traffic light system 130
Figure 44 Coupled traffic light system 131
Figure 45 Results of the enactment of the traffic light system 132
Figure 46. Translation of an internal transition to Z 135
Figure 47. Conditional internal transition to Z 136
Figure 48. External Transition translation to Z 137
Figure 49. Conditional external transition to Z 137
Figure 50. Confluent transition to Z 138
Figure 51. Confluent conditional transition to Z 139
Figure 52. Illustration of HILLS to CSP translation 142
Figure 53. Translation of HILLS composite model to CSP 143
Figure 54. Overview of HiLLS tooling Framework 145
Figure 56. HiLLS metamodel of the alternating bit protocol 150
Figure 57. Message HClass 150
Figure 58. HiLLS model of the Generator 151
Figure 59. HiLLS model of the Sender 152
Figure 60. Receiver HSystem 153
Figure 61. The HiLLS model of the Medium 154
Figure 62. The Model of accumulator 155
Figure 63. ABProtocol HSystem 155
Figure 64. couplings mapping 158
Figure 65. Message and Init 159
Figure 66.getHeader and setHeader 160
Figure 67. getContent and setContent 160
Figure 68. Receiver in Z 160
Figure 69. Show proof of Receiver State schema 161
Figure 70. Proof of Reciver state schema 161
Figure 71. proof by reduce of Receiver state schema 162
Figure 72. Init and waiting2sendingTransition 162
Figure 73. show Proof on waiting2sendingTransitionExt 163
Figure 74. sendint2waintingTransitionINT1 163
Figure 75. sending2waitingTransitionINT2 163
Figure 76. syntax error in sending2waitingTransitionINT2 164
Figure 77. Existence of initial state 164
Figure 78. unsuccessful proof of the theorem 165
Figure 79. proof of the theorem 165
Figure 80. Trajectories in Z 166
Figure 81. Domain Model of the CDM 167
Figure 82. HiLLS specification of the Cassette 169

Figure 83. HiLLS specification of the Bundle Acceptor 170
Figure 84. HiLLS specification of the Bill Checker 171
Figure 85. HiLLS specification of the Escrow 172
Figure 86. HiLLS' specification of Bill and Reject box 174
Figure 87. HiLLS specification of the Control board 176
Figure 88. HiLLS specification of the Cash Deposit Machine (CDM) 178
Figure 89. Block diagram illustrating the structure of the CDM 179

List of abbreviations

ABP	Alternating Bit Protocol
ATM	Automated Teller Machine
BPMN	Business Process Modeling Notation
CCS	Calculus of Communicating Systems
CDEVS	Classic DEVS
CDM	Cash Deposit Machine
CSP	Communicating Sequential Processes
CSPZ	CSP combined with Z
CTL	Computaion Tree Logic
DDML	DEVS Driven Modeling Language
DEVS	Discrete Event System Specification
DSDEVS	Dynamic Structure DEVS
DSL	Domain Specific Language
DSML	Domain Specific Modeling Language
ERD	Entity Relationship Diagrams
FDR	Failure Divergence Refinement
GPPL	General Purpose Programming Language
HiLLS	High Level Language for System Specification
LTS	Labelled Transition System
PDEVS	Parallel DEVS
SLC	Simulation Life Cycle
UML	Unified Modeling Language
USTTB	Université des Sciences, des Techniques et des Technologies de Bamako
SysML	System Modeling Language
ZCCS	Z combined with CCS

I. General introduction

I.1 Context

It is a known fact that software development projects have low level of success compared to projects in other domains like construction. Success in these domains is due to well-defined engineering methodologies and standards. To increase success in software and hardware systems development there is a need to define principles, languages, methodologies and tools that help projects managers, designers and developers in the different steps. Over the years multiple development methods and software life cycle models classified as classic approaches and agile methodologies have been proposed. Classical approaches (Waterfall model, incremental methods etc.) articulate around four main phases: specification, design, implementation, verification and validation. The differences between them reside in how steps follow each other, details of each step and how verification and validation is conducted through the different phases. A common critic of most of these approaches is that error is detected very late or interaction with the product is very late. While classical approaches follow rigorously a requirement document and a defined project plan, agile methodologies (eXtreme Programming, Scrum, Feature Driven Development, Adaptive Software Development) focus on the interaction with the customer, changes in requirements and proposition of an early version of the software. A list of recommendations to complement the deficiencies of software development can be to:

- Capture Requirement in a precise and unambiguous way
- Integrate Many languages in one or different phases
- Detect and correct errors in the early phases
- Conduct Verification and validation through all the phases
- Make an early version of the software available
- Use Formal methods in complement to simulation and testing
- Use Object-Orientation Principles in design and implementation
- Generate code from models

In modeling and analysis of complex systems, it is important that the abstract model should accurately capture the structural and behavioral aspects of the system to make verification and validation using techniques such as simulation and formal analysis effective and the implementation of the system possible. System development generally requires knowledge of the system domain, modeling methodology, and model analysis and execution techniques. Domain experts are concerned with system characteristics, problems and behavior. Modeling experts use mathematical formalisms, algorithms, and/or computer programs to develop abstractions of systems. These abstractions must be translated into some semantic domain to investigate system properties. Due to the difference in concerns and expertise between experts involved in the system development, it is required to utilize a framework that supports communication and cooperation between domain experts and experts in other domains like modeling and simulation and formal methods. What is needed to achieve this is an intermediate notation which is highly communicable, expressive, and low enough to reduce the complexity of code synthesis for simulation, prototyping, enactment and formal analysis. This representation should be able to express the structural and behavioral characteristics of complex systems without ambiguities. In the area of software, systems, business processes and data engineering some notables languages such as Unified Modeling Language TM (UML) [OMG 2010a], System Modeling Language TM (SysML) [OMG 2010b], Business Process Modeling Notation (BPMN) [OMG 2012] and Entity Relationship Diagrams (ERD) [Chen 1976] respectively have been used as specifications that provide easy visual constructs that facilitate cooperation between domain engineers and technical

experts. Employing similar constructs for real-time, concurrent and distributed systems modeling with discrete event perspective would facilitate the system development process; this is a part of the objectives of this thesis.

Over the years, several modeling techniques for dynamic systems have been developed. The Discrete Event System Specification (DEVS) [Zeigler 1976] formalism has emerged as a preferred formalism because other formalisms have been proven to have an equivalent DEVS representation [Vangheluwe 2000]. Although DEVS is specifically tailored to suit discrete event systems, it supports full range of dynamic system representation. In particular, a Differential Equation System Specification (DESS) can have an approximate Discrete Time System Specification (DTSS) by selection of a sufficiently small constant time interval (discretization). A DTSS model, in turn has an equivalent DEVS representation. Also, quantization of a DESS system can result in an approximate DEVS model. As such, approximate models of continuous and hybrid systems can be developed with DEVS. DEVS also promotes separation of concerns by separating the model, simulator, and experimental frame thereby facilitating system development. The DEVS simulation protocol is well defined. However, DEVS is semi-formal and it does not provide concrete guidelines to express system structure, behavior, and traces. The modeler is free to develop system models in ways that are most appealing. The absence of a common agreed concrete syntax for DEVS has led to different implementations such as DEVSJAVA [Sarjoughian and Zeigler 1998], SimBeans [Präehofer et al. 1999], James II [Himmelspach and Uhrmacher 2004], etc. that makes collaboration between modelers and communicability of models difficult. An Objective of this thesis is to define a new language that reuse DEVS and provide a concrete syntax for modeling systems and make models communicable and improve collaboration.

Simulation and testing have been used as traditional methods of verification and validation to assess the qualities of the models by exploring some of the possible situations and scenarios and comparing with system specification. The limitation of simulation is that it does not guarantee that a verified system is error free because only some part of the system behavior is explored. On the other hand, formal analysis involves using deductive verification, axioms, and proof rules to determine the correctness of systems or models. They derive static properties when simulation derives dynamic properties. Formal methods have been used in the specification, development and verification of software and hardware systems but its use with computer simulation has not been explored exhaustively. Formal methods can be a good compliment of simulation thereby contributing to the reliability and robustness of systems [Micheal 1997]. Mathematical proof of correctness is the most effective means of model verification and validation if its applicability is possible [Balci 1997]. Formal analysis techniques allow for exhaustive check for the conformity of the specification to requirements and rigorous proof of assertions about the system. Advances in Formal methods have increased the range of systems that this can be applied to. Hence, integrating formal methods with a well-established simulation technique like DEVS would enable to derive premises or logical consequences of the model and confirm that the abstract specification conforms to the operational specification [Traoré 2006]. A goal of this thesis is to complement simulation-based verification by formal analysis techniques to check the multiple aspects of systems properties.

The increasing complexity of hardware systems, software systems and embedded systems often requires the use of different languages for modeling their different aspects. In general a system

development requires many models that provide several complementary views on the system structure and behavior. Communication between experts is constrained by the disparateness of the descriptions of the various aspects of the system in their respective domains. Usually, an exhaustive study of a complex system is done by adopting one of the two following prevalent methods:

a. Creating different models of the same system with different formalisms to express the aspects of the system required by experts of different domains.
b. Transforming a model defined in a specific formalism into models of other languages to carry out the investigations.

These methods are not without serious drawbacks. In addition to the difficulty of mutual communication between experts, the use of (a) is haunted by an arduous task of updating the different versions of the specification to maintain consistency. The time and efforts required to specify different models in different languages, yet for the same system could dissuade one from embarking on that track. By taking benefits of the Model-Driven Engineering (MDE), method, (b) solves the problem of manually specifying multiple models of same system. However, it is still characterized by inconsistencies resulting from two sources:

- Difference in the capabilities of different formalisms to effectively create abstractions of certain aspects of a system.
- Misinterpretations of the source model by the target languages due to lack of precisely defined semantics for the source language.

To handle the presented challenges, a system development methodology should provide a language with a concrete syntax that promotes communicability of models and collaboration between experts. Since DEVS is proven to be a common denominator for discrete event systems modeling and simulation, it can be used as a base formalism in a system development methodology with a provided concrete syntax to solve the communication and collaboration issues. The development methodology should include other analysis techniques to complement simulation to cover different aspects of system behavior. It is also important to maintain consistency between the different views of the system during the development process; this is an aspect of system development addressed by the language proposed in this thesis.

I.2 Objectives

Our objective is the definition of a modeling language called HiLLS with support for a precise and consistent unification of the various aspects of a system in one model. This approach is expected to make the model adaptable to multiple analysis techniques. The objectives of HiLLS include:

- *High communicability*: define a graphical concrete syntax for HiLLS that is easy to learn and communicate and make sharing and discussing models between experts simple thereby facilitating their cooperation.
- *High expressive power:* define the abstract syntax of HiLLS by integrating DEVS and Object-Z to support the precise modeling of complex systems with discrete event perspective. HiLLS inherits the expressiveness and properties of DEVS and Object-Z.

- *Simulation applicability*: adopt DEVS as a semantic domain for HiLLS by defining a semantic mapping from HiLLS to DEVS so that HiLLS models can be simulated by using DEVS existing simulators.
- *Amenability to formal analysis*: make HiLLS models amenable to formal analysis so that the user can analyze static and dynamic properties of the model by generating the formal models and verification conditions from it to ease the verification process
- *Hierarchy and Modularity*: adopt modular and hierarchical structuring concepts from the DEVS formalism and object oriented structuring concepts from Object-Z and MOF (Meta Object Facility).
- *Availability of supporting tools*: provide supporting tools that facilitates the design of graphical models as well as automated code synthesis for simulation and integrated support for visualization and formal analysis.
- *Automated code synthesis*: Generate code from HiLLS models to reduce error in translation process.
- *Executability of models*: provide enactment semantics for the automatic generation of a real-time executable code of the system from the HiLLS models.
- *Maintainability of consistency between all the views:* use HiLLS specifications as front-ends from witch other views are generated to maintain consistency between the different views and reduce the task of updating them.

I.3 Outline

Chapter II explores the state of the art in the modeling and simulation domain in general and DEVS-based modeling and simulation framework in particular. We discuss formal methods with focus on Z, Object-Z and CSP formal notations. We present also Formal analysis of DEVS models. Chapter III introduces the HiLLS language and presents its objectives. It presents the abstract syntax, concrete syntax and the set-theoretic semantics of HiLLS. Chapters IV presents the operational and logical semantics of HiLLS. Application of HiLLS to the modeling and analysis of the ABP (Alternating Bit Protocol) and ATM (Automated Teller Machine) is discussed in chapter V. The general conclusion of the thesis is drawn in chapter VI.

II. State of the art

II.1 Introduction

Nowadays, the daily life of a man occurs in an environment made up of complex systems going from simple tools of entertainment to critical systems such as embedded systems on medical systems, individuals cars, trains, planes and materials whose failures can be fatal for him. The study of these systems is done through experiments of various kinds. The construction of physical models to test certain characteristics of the system are often very dangerous, expensive, and even impossible in much of cases. One of the methods making it possible to study systems consists in characterizing the system studied by a system of equations and the use of mathematical techniques to determine the analytical solution or approximate solution by using approximation techniques. These methods known as analytical methods present some limits, they are difficult to use in the study of some category of systems like manufacturing systems, military systems, etc.

Simulation exceeds the limits of the analytical and numerical methods in the study of complex dynamic systems. It is largely used in industry and academia. It covers all the scientific, economic and social fields. Simulation has been applied to aeronautics, urban and interurban transportation systems, demography; propagation of epidemics, etc. Simulation has several advantages such as the facilitation of the experimentation process by using less financial and data-processing resources. Simulation is used to check that our models give us coherent results compared to the data of the real system and in conformity with the formal and verifiable requirements of this last.

Different simulation approaches exist in the literature: discrete-time, continuous and discrete-event simulations. For a comprehensive view on simulation, reader can consult the following papers: [Kreutzer 1986], [MacDougall 1987] and [Fishwick 1995]. In the simulation area, the discrete-event simulation approach has emerged as well accepted system analysis technique. Some known formalisms like UML integrate discrete-event modeling techniques in the specification and analysis of systems. Our thesis concentrates on the discrete-event approach to systems modeling and analysis.

This chapter presents modeling and simulation in general and DEVS-based modeling and simulation in particular as a background for the operational semantics of HiLLS. The chapter also discusses formal methods with focus on Z, Object-Z and CSP formal notations, existing formal verification approaches of DEVS models and basic concepts of the formal languages used as logical semantic domain for HiLLS. Basic concepts of system design and language engineering are presented as background respectively for enactment methodology of HiLLS and the integration approach used to define HiLLS. Related work of integrated languages is also discussed.

II.2 Modeling and Simulation

II.2.1 Simulation life Cycle

Simulation process turns around three main activities: modeling, simulation and the verification and validation activities (Figure 1). The modeling phase consists of using a particular simulation modeling approach to create an abstraction of the real or proposed system of interest named the problem entity in [Sargent 2000]. The conceptual model is the simplified representation of the problem entity developed in the analysis and modeling phase for a particular study. The computer

programming and implementation phase consists to build the computerized model. Inferences about the problem entity are obtained by conducting computer experiments on the computerized model in the experimentation phase.

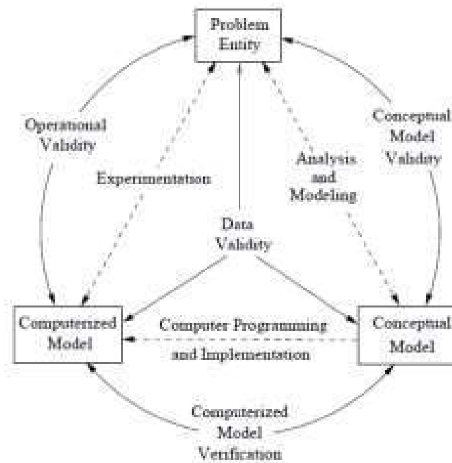

Figure 1: Simulation Life Cycle

The modeling and simulation process needs to ensure different kind of validity relation between the problem entity, the conceptual model and the computerized model. Conceptual model validity is defined as determining that the theories and assumptions underlying the conceptual model are correct and that the model representation of the problem entity is reasonable for the intended purpose of the model. Computerized model verification is defined as ensuring that the computer programming and implementation of the conceptual model is correct. Operational validity is defined as determining that the model's output behavior has sufficient accuracy for the model's intended purpose over the domain of the model's intended applicability. Data validity is defined as ensuring that the data necessary for model building, model evaluation and testing, and conducting the model experiments to solve the problem are adequate and correct".

II.2.2 Verification and Validation

Different definitions of model verification and validation exist in the literature with different formulations. The definition adopted in [Sargent 2000] is "ensuring that the computer program of the computerized model and its implementation are correct". In [Sargent 2000] Model validation is adopted to mean "substantiation that a computerized model within its domain of applicability possesses a satisfactory range of accuracy consistent with the intended application of the model". Validation is the process of comparing real system data and collected as simulation results with respect to experimentation conditions [Zeigler et al. 2000]. Different V&V techniques exist in the literature.

[Sargent 2000] gives a non-exhaustive list of techniques that can be applied in the V&V process:

- Animation (Displaying the operational behavior graphically and values of various performance measures through time),
- Comparison to Other Models (Various results of the simulation model being validated are compared to results of other models that have been validated),
- Face Validity (asking people knowledgeable about the system whether the model and/or its behavior are reasonable),

- Degenerate Tests (testing the degeneracy of the model behavior by appropriate selection of values of the input and internal parameters),
- Event Validity (comparing event occurrences in the model to that of the system),
- Extreme Condition Tests (testing how model react to unlikely combination of factors),
- Historical Data Validation (compare model to system by using the system's collected data),
- Internal Validity (analyzing the degree of variability in model behavior),
- Parameter Variability–Sensitivity Analysis (testing the effect of input and parameters variability upon the model's behavior and output),
- Multistage Validation (developing the model's assumptions, testing the assumptions and testing input-output relationship between the system and the model) etc.

A recommended procedure for some of these techniques in V&V is given by the author in another paper [Sargent 2001]. In model testing, the model is subjected to test data or test cases to determine if it functions properly. These techniques are independent from the modeling language used.
[Balci 1997] gives a list of principles and techniques in verification, validation and accreditation of simulation models that can help in assessing model credibility. Taxonomy of more than 77 V&V techniques is provided to assist simulation practitioners in selecting proper approaches for simulation model V&V (Figure 2). Taxonomy of 38 V&V techniques is presented for object-oriented simulation models. The cited principles turn around the application of V&V in the entire M&S lifecycle:

- The specification of simulation objectives and conditions
- Clear specification of the requirements,
- The nature of V&V results,
- Detection of error in M&S phases etc.

The techniques include those presented in [Sargent 2000]. Some techniques come from software engineering and others are specific to simulation field. The selection of V&V method may depend on model type, simulation type, problem domain, and M&S objectives.

Figure 2. Taxanomy of Verification, Validation and Testing Techniques [Balci 1997]

II.2.3 DEVS-Based Modeling and Simulation Framework

II.2.3.1 Entities of the Framework

The DEVS formalism defines a modeling and simulation framework composed of the following entities: the source system, the model, the simulator and the experimental frame (Figure 3).

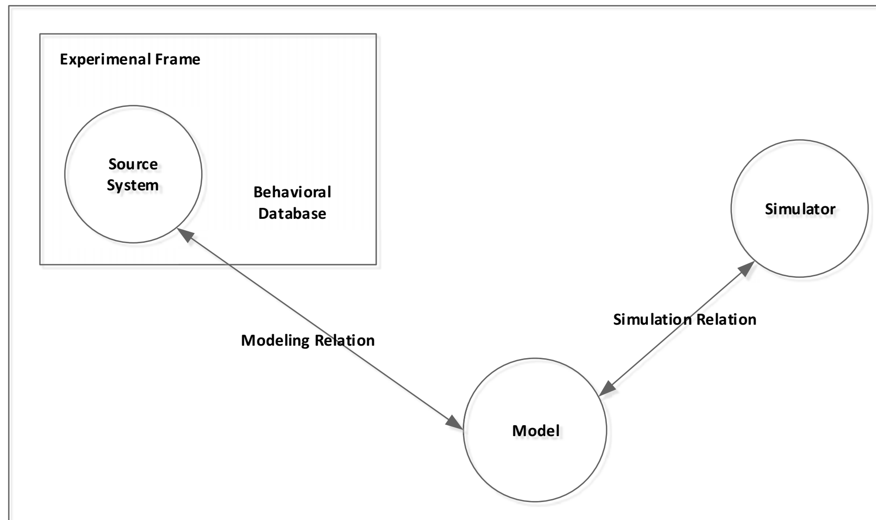

Figure 3. M&S Entities and Relationships between them [Zeigler et al. 2000]

II.2.3.1.1 Source System

The source system is real or virtual system concerned by the modeling and simulation process. It is the source of observables data that constitutes behavioral database. The term system in this thesis is used for any entity of the real world, phenomenon or process that can be represented in form computable by a machine.

II. 2.3.1.2 Model

The model is a simplified and abstract representation of the system. This representation can be physical, mathematic or logical. A model is used to study the structure and behavior of a system in a particular context. The results of the study are only meaningful in the specified context. The specification of models in the DEVS framework obeys to a hierarchy proposed by Zeigler [Zeigler et al. 2000].

II. 2.3.1.3 Simulator

A Simulator is an engine capable of reproducing the behavior of a model. The type of simulator used depends of the formalism used to create the associated model.

II. 2.3.1.4 Modeling and simulation relations

The existing relations between these entities are: the modeling relation between the system and the model and simulation relation between the model and the simulator. The modeling relation refers to the degree at which the Model faithfully represents the System and the simulation relation refers to correspondence between model design and its implementation.

II.2.3.1.5 Experimental Frame

The experimental frame (Figure 4) defines the limited set of circumstances under which the system is observed, modeled and studied. The operational formulation of the objectives of the experimentation is a part of the experimental frame specification in a context. There is no agreed framework for the factors that can characterize a context. There are similarities between the acts of specifying a model for a real system and that of specifying a context. However, there is a common agreement to recognize that a model of context must make explicit at least the underlying objectives, assumptions and constraints of the study.

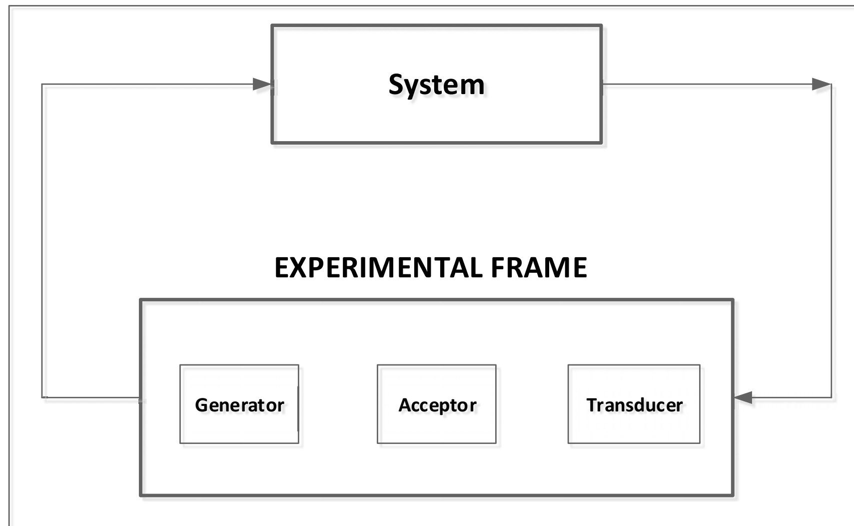

Figure 4. Experimental Frame and its Components [Zeigler 1983]

The set of experimentation circumstances defined by a frame consists of input, output, run control and summary variable sets. Constraints are imposed on the time segments of input and run control variables. Formally, the experimental frame is the following structure:
$EF = < T, I, O, C, \Omega_I, \Omega_C, SU >$, where:

- T is the time base
- I is the set of input variables, inputs are received from the model
- O is the set of output variables, outputs are sent to the model
- C is the set of run control variables which will be monitored to avoid situation that violate the frame constraints
- Ω_I is the set of input segments,
- Ω_C is the set of run control segments, i.e., constraints on the combinations of run control variables (including temporal constraints) which capture the domain operation required by the frame. Input/output behavior of a model in this frame is accepted as long as the run control constraints are not violated.
- SU is the set of summary mappings which are statistical and other aggregations of the input/output behavior into reduced and manageable spaces.

Different definitions of experimental frame and propositions for its structure and components have been proposed in the literature [Zeigler 1976] [Rozenblit 1985]. In most of the propositions, an experimental frame is composed of three type of components interconnected as shown in Figure 4. This structuration of the experimental frame allows the separation between a model and

the different contexts in which it can be experimented. So a model can be experimented using different experimental frames.

- **Generator:** Produces the input segments sent to a model. A generator is generally modeled in DEVS as a component without input port that generates outputs within constant or variable period of time.
- **Acceptor:** Continually tests the run control segments for satisfaction of the given constraints.
- **Transducer:** Collects the input/output data and computes the summary mappings. This is generally modeled by a DEVS model without output.

In [Rozenblit 1991] the basic experimental frame/model coupling results in the architecture where control of frame components is concentrated within the master experimental frame module whereas the simulators are responsible for execution of model component's dynamics. The centralized architecture involves a single experimental frame module directly linked to the global coordinator.

In [Traoré and Muzy 2005], authors advocated the separation between a model and its context, as a systematic part of the M&S process like the separation between a model and its simulator. This separation of concerns gives the following pairs: system/context pair, model/frame pair, and simulator/experimenter pair. In this approach experimenters adhere to the same message passing protocol as simulators during simulation.

II.2.3.2 System Specification Hierarchy

The systems specification hierarchy is the basis for a framework for modeling and simulation. We base our understanding of a system on Zeigler's hierarchy of system specification [Zeigler et al. 2000]. This framework employs a general concept of dynamical systems and identifies useful ways in which such systems can be specified. The proposed hierarchy is composed of five levels: OF, IORO, IOFO, IOS, and CN. This hierarchy is organized such that knowledge about the system is increased in higher levels; and knowledge of lower levels can be derived from higher levels. Modeling the structure of a system is done at the IOS and CN level. Knowledge of the behavior is given at the IORO level and this can be derived from IOFO knowledge. Usually, simulation algorithms are applied to the model to generate the behavior (IORO).

III. 2.3.2.1 Observation Frame (OF)

This level is concerned by how to stimulate the system with input, what variables to measure and how to observe them over a time base. It is the input output observation frame.
An observation frame is a structure of the form $=< X, Y, T >$, where X is the input interface , Y is the output interface and T is the time base.

At this level the system is seen a black box with input and output ports (Figure 5).

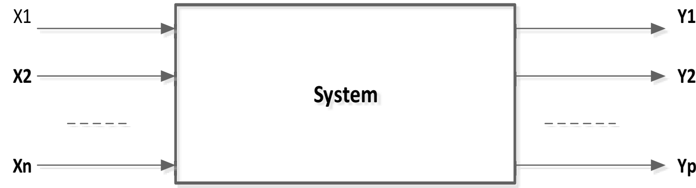

Figure 5. System at OF level

$X = X_1 \times X_2 \times \ldots \times X_n$
$Y = Y_1 \times Y_2 \times \ldots \times Y_p$

III.2.2.2.2 Input Output Relation Observation (IORO)

This level establishes a relation between input segments and output segments i.e. for each input segment $\omega \in (X, T)$ applied to the system it associates a corresponding output segment $\rho \in (Y, T)$ observed in the same time interval.
An IORO is a structure of the form $IORO =< X, Y, T, \Omega, R >$, where $< X, Y, T >$ is an Observation Frame, $\Omega = (X, T)$ is the set of input segments and R is the input/output relation that define the behavior of the system.

$R = \{(\omega, \rho) / dom(\omega) = dom(\rho)\} \subset \Omega \times (Y, T)$, with ω an input segment and ρ an output segment.

$dom(f)$ is the observation time interval of f.

III.2.3.2.3 Input Output Function Observation (IOFO)

This level adds the knowledge of the initial state and every input stimulus produces a unique output.

Given an $IORO =< X, Y, T, \Omega, R >$ with $R = \{(\omega, \rho) / dom(\omega) = dom(\rho)\} \subset \Omega \times (Y, T)$. For each (ω, ρ) we can find a function $f_i \in F = \{f_1, f_2, \ldots f_i, \ldots\}$ such that $\rho = f_i(\omega)$. The input/output relation R is partitioned to a set of functions $F = \{f_1, f_2, \ldots f_i, \ldots\}$.

An IOFO structure is defined as follows:

$IOFO =< X, Y, T, \Omega, F >$

$X, Y, T, et\ \Omega$ are the same as in IORO;

$f \in F \Rightarrow f \subseteq \Omega \times (Y, T)$ is a function, and if $\rho = f(\omega)$ then $dom(\rho) = dom(\omega)$.

Given an $IOFO =< X, Y, T, \Omega, F >$ one can derive an $IORO =< X, Y, T, \Omega, R >$, with $R = \bigcup_{f \in F} f$.

III.2.3.2.4 Input Output System (IOS)

The objective of this level is to determine how states are affected by inputs; given a state and an input what is the state after the input stimulus is over? What output event is generated by a state?

At IOFO level we know the initial state but not the other states (intermediary states and final states). IOFO is extended to IOS by defining the set of states and transitions.

An IOS is a structure of the form $IOS =< X, Y, T, \Omega, Q, \Delta, \Lambda >$, where $X, Y, T, et\ \Omega$ are the same as in IOFO; $Q$is the set of states; $\Delta: Q \times \Omega \rightarrow \mathrm{Q}$ is the global state transition function and $\Lambda: Q \times X \rightarrow Y$ $(or\ \Lambda: Q \rightarrow Y)$ is the output function.

III.2.3.2.5 Coupled Network (CN)

This level defines the components and how they are coupled together. The components can be specified at lower levels or can even be structure systems themselves-leading to hierarchical structure.

A CN level structure is of the following form:

$$CN =< T, X_N, \mathrm{Y_N}, D, \{\mathrm{M_d}/\mathrm{d} \in \mathrm{D}\}, \left\{I_d/d \in D \cup \{N\}, \left\{Z_d/d \in D \cup \{N\}\right\}\right\} >$$

where, X_N is the set of input;

$\mathrm{Y_N}$ is the set of output;

D is the set of components names;

$\forall d \in D, M_d$ is an IOS or a CN;

$\forall d \in D \cup \{N\}, I_d \subseteq D$ is the set of influencer of d,

$Z_d{:}\times_{i \in I_d} YX_i \rightarrow XY_d$ is coupling function

$YX_i = X_i\ if\ i = N$

$= Y_i\ if\ i \neq N$

$XY_d = Y_d\ if\ d = N$

$= X_d\ if\ d \neq N$

II.2.3.3 DEVS Modeling formalism

Models in DEVS are distinguished into atomic and coupled models [Zeigler et al. 2000]. Atomic models are component models that cannot be decomposed. The behavior of atomic models are specified in terms of input and output events, state space, transition functions from state to state, output functions and time advance function. Coupled models are hierarchical composition of atomic or coupled models. Two versions of DEVS exist: the original sequential version called Classic DEVS (CDEVS) [Zeigler 1976] and the parallel version called Parallel DEVS (PDEVS) [Chow 1996]. For the rest of the document the term DEVS refer to PDEVS.

II.2.3.3.1 Atomic DEVS

A DEVS atomic model is a structure of the form: $M =< X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta >$, where

$X = \{(p, v), p \in IPort \wedge v \in dom(p)\}$ is the set of inputs, $IPort$ is the set of input ports,

$Y = \{(q, v), q \in OPort \wedge v \in dom(q)\}$ is the set of outputs, $OPort$ is the set of output ports,

S is the state space,

$\delta_{int}: S \rightarrow S$ is the internal transition function

$\delta_{ext}: Q \times X^b \rightarrow S$ is the external transition function

$\delta_{conf}: S \times X^b \rightarrow S$ is the confluent transition function which resolves collision between internal and external events

$\lambda: S \rightarrow Y^b$ is the output function,

$ta: S \rightarrow \mathbb{R}^+ \cup \{+\infty\}$ is the time advance function.

Atomic DEVS provides a more appropriate way of modeling a system at IOS level

The semantics of an atomic model is as follows: at each time, the model is in some state s. If the lifetime of the state elapse i.e. $e = ta(s)$ then the model send and output $y = \lambda(s)$; if he receives no input within this simulation cycle he goes to the next state $s' = \delta_{int}(s)$(internal transition) with a new lifetime $ta(s')$ else the next state is $s' = \delta_{con}(s, x)$ (confluent transition, where x is the set of all events received by the model in this simulation cycle). If events are received before the time elapse $e < ta(s)$ then the next state is $s' = \delta_{ext}(s, e, x)$(external transition, x is the set of events received) with a new lifetime $ta(s')$. See [Zeigler et al 2000] for more details of simulation algorithms for DEVS.

In the example of the following figure (Figure 6), the model is initially in state s_0, after the expiration of the duration $t(s_0)$ and the reception of the input bag x_1, an output bag $y_1 = \lambda(s_0)$ is sent at the time instant t_1 and the model execute confluent transition with the new state defined by $s_1 = \delta_{con}(s_0, x_1)$ for a new duration equal to $ta(s_1)$. In this state no event is received before the total expiration of the duration (s_1) , this results in an output bag $y_2 = \lambda(s_1)$ and an internal transition with new state defined by $s_2 = \delta_{int}(s)$ for a new duration $ta(s_2)$. In state s_3, the model is stimulated by the input bag x_4 at the time instant t_4 before the expiration of its duration ($e_3 < ta(s_3)$); this results in an external transition with new state $s_4 = \delta_{ext}(s_3, e_3, x_4)$

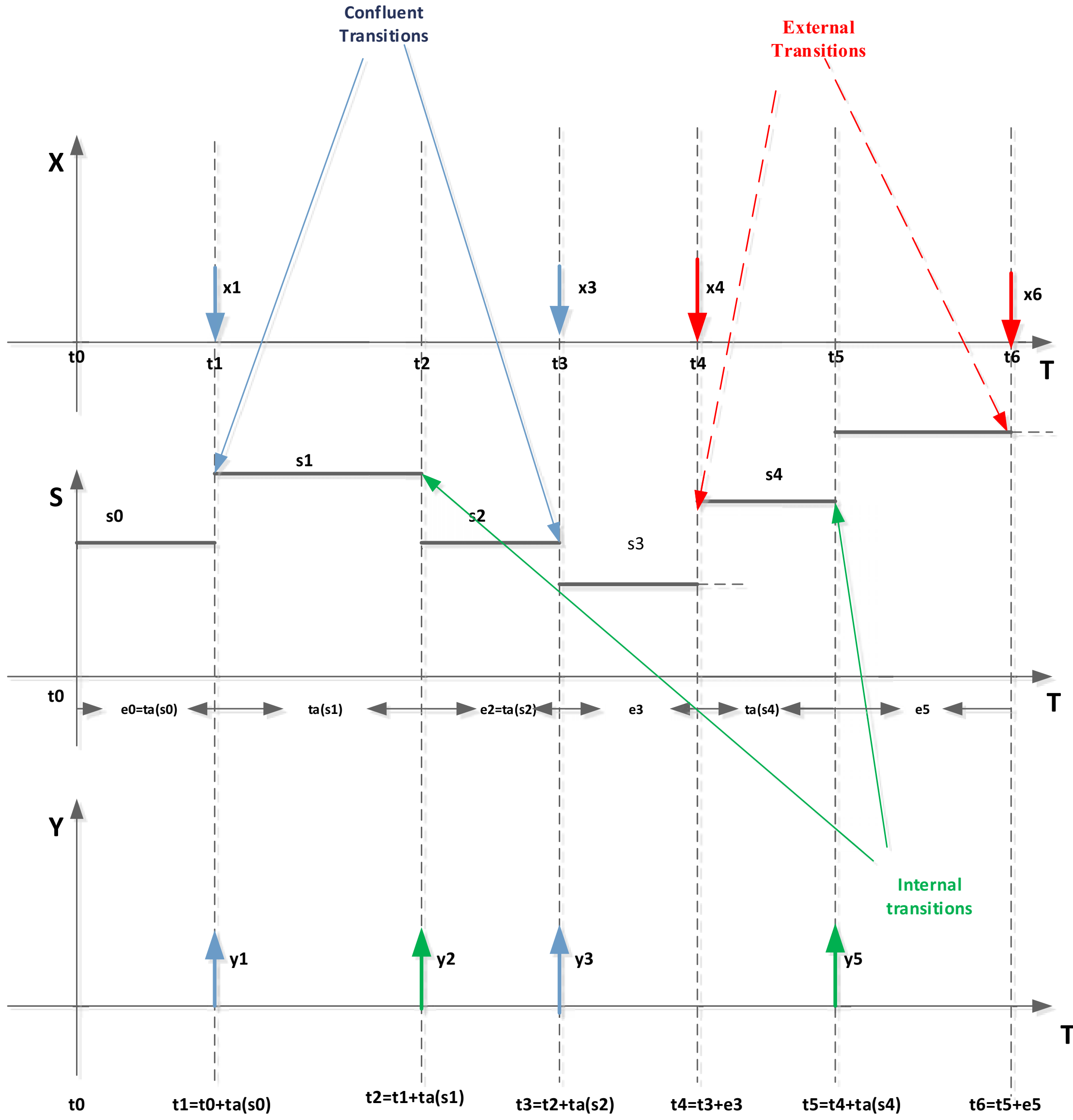

Figure 6. Example of transitions occurrences

II.2.3.3.2 Coupled DEVS

A coupled model is a structure of the form: $M =< X, Y, D, \{M_d\}_{d\in D}, EIC, EOC, IC >$, where

X and Y are the same as in the case of atomic model,

D is the set of names of models which compose M;

M_d is the model wich name is d;

$EIC \subseteq \{((M, ip_M), (d, ip_d))/ip_M \in IPorts_M, ip_d \in IPorts_d\}$ is the external input coupling relation

$EOC \subseteq \{((d, op_d), (M, op_M))/op_M \in OPorts_M, op_d \in OPorts_d\}$ is the external output coupling relation

$IC \subseteq \{((a, op_a), (b, ip_b))/op_a \in OPorts_a, ip_b \in IPorts_b\}$ is the internal coupling relation ;

Coupled DEVS provides basic concepts that appropriately capture the different aspects of a system at CN level.

A DEVS model is simulated by following a well-defined simulation protocol. We explain this protocol by considering the example of Figure 7.

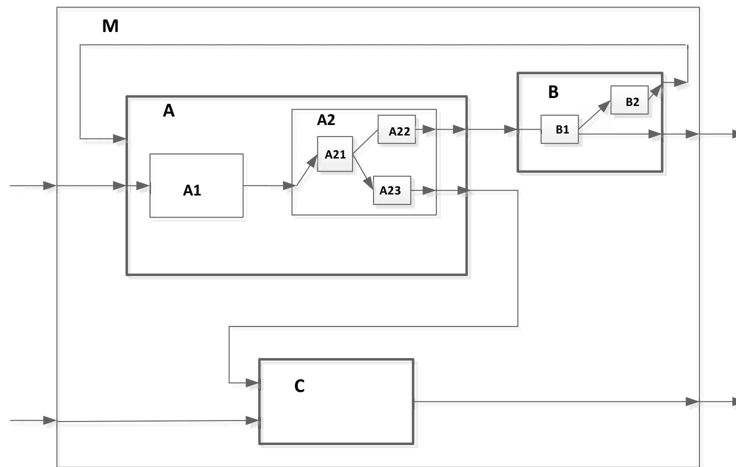

Figure 7. Example of Coupled Model

From each coupled model hierarchy corresponds a simulation tree; the corresponding simulation tree of the coupled model of Figure 7 is shown on Figure 8.

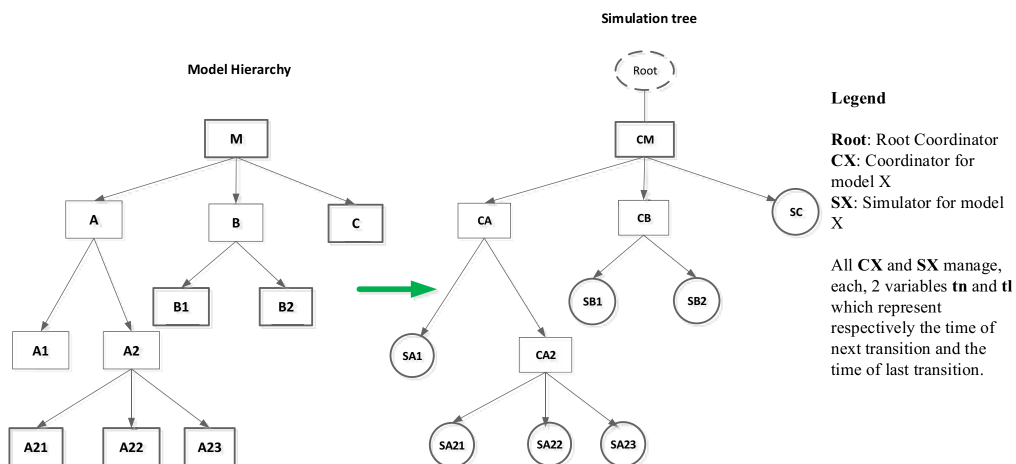

Figure 8. Correspondence between Model Hierarchy and Simulation Tree

The Root coordinator is responsible of starting the simulation at a time t by sending an initialization message (i,t) to its child CM. When CM receives the message (i,t), it forwards it to its children so that they can forward it also to their children in case of coordinators or update their tl and tn in case of simulators. The initialization process is shown on Figure 9 and Figure 10.

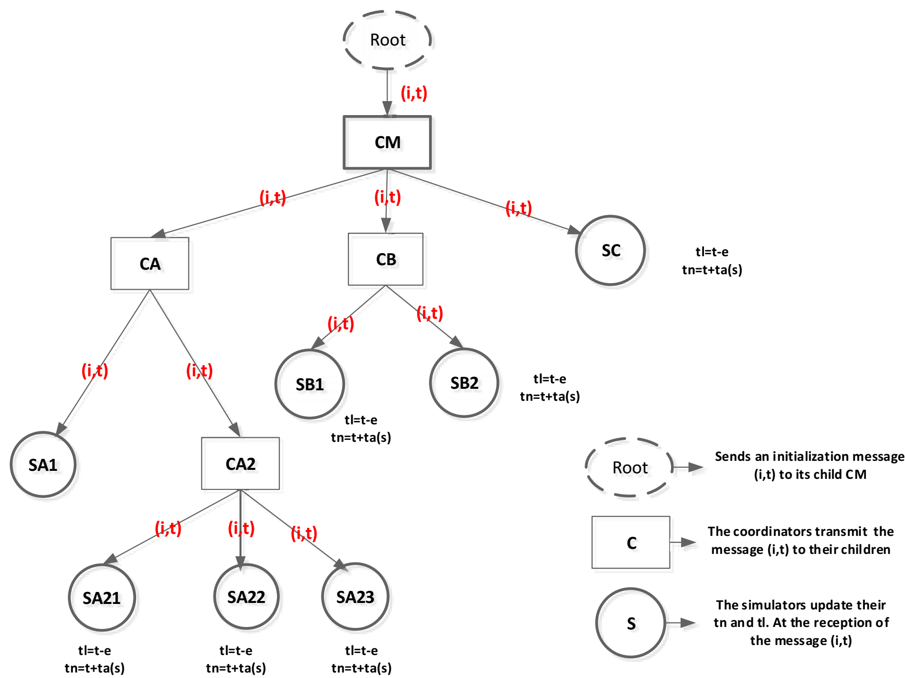

Figure 9. Sending of initialization message

After the updates of their tn and tl, each simulator sends a done message to its parent coordinator which also calculates its tn and tl and sends it to its parent. This process ends with the sending of the done message of the top-level coordinator to the root coordinator.

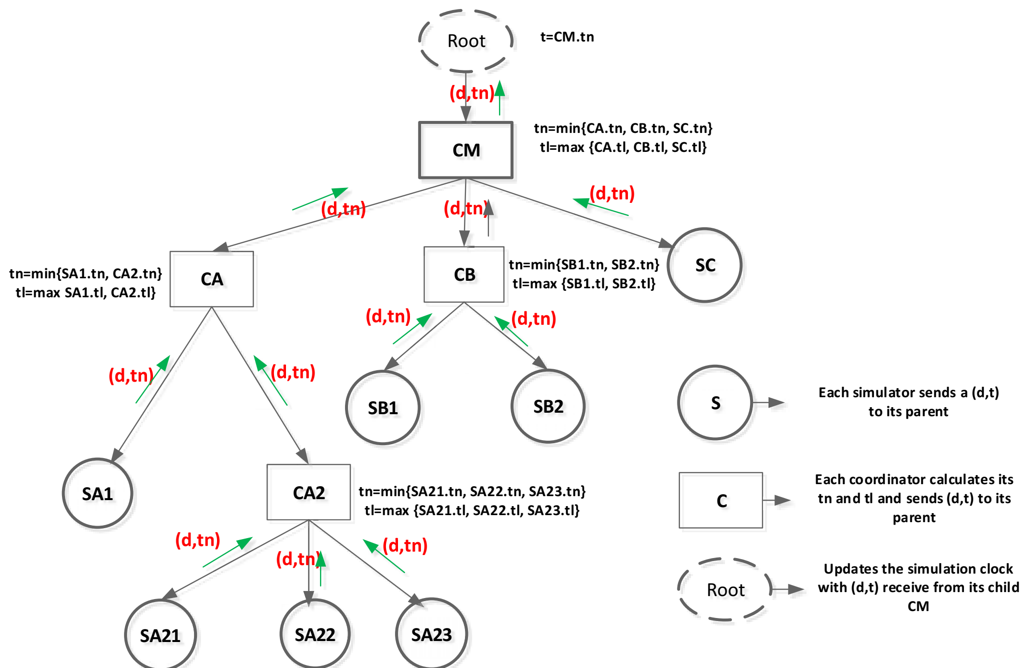

Figure 10. Sending of done messages to parents

The root coordinator sends a first message (*,t) message to its child CM (Figure 11) so that it can forward it to its children.

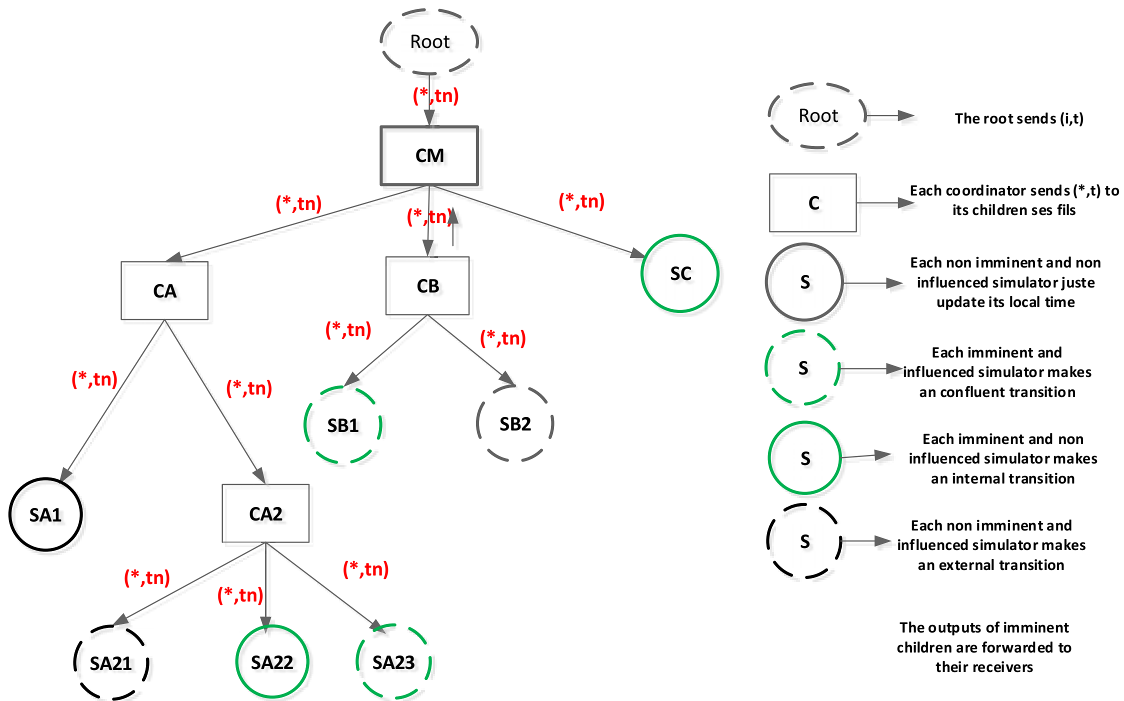

Figure 11. Sending star message

At the reception of the (*,t) message, imminent simulators send their outputs to their respective parents coordinators and all the simulators (including the imminent ones) wait for eventual inputs to decide which transition to perform. Each coordinator will send the received outputs as input to its children by analyzing coupling relations of the coupled model associated to it. Imminent components that are not influenced will make internal transition. Imminent and influenced components will make confluent transition. Non-imminent and influenced components will external transition. The cycle ends by sending of (d,tn) done message by all the components (including non-imminent and influenced components that have not performed any transition) to their parent coordinators like in the end of the initialization process.

An important property of the DEVS formalism is the closure under coupling property which ensures that any coupled model has an equivalent atomic model.

II.2.3.3.3 Modeling with DEVS

II.2.3.3.3.1 University Bus System

A bus shuttles between a downtown station and a university station, providing students and non-students with a transportation service. The growing number of users leads the university administration to set up a study to evaluate the performances of the bus system. We then consider a model for the UBS, with user identification possibilities. A user enters the model when he lines up in a station. He exits from the model when he gets off the bus at a station.

The university Bus System is a DEVS atomic model.

$UBS =< X, Y, S, \delta_{int},\ \delta_{ext}, \delta_{conf}, \lambda, ta >$

$X = \{Hello1, Hello2\}$ et $Y = \{Bye1, Bye2\}$

$Hello1, Hello2, Bye1, Bye2 \in \{(identity, statut) / \ statut \in \{"Student", "non-Student"\}\}$ and identity is a string.

$S = \{W_d, W_u, B, P, D, S'\}$

W_d is the ordered list of users waiting at the downtown station,

W_u is the ordered list of users waiting at the university station,

B is the ordered list of passengers in the bus,

$P \in \{1,2,3,4\}$ is the current position of the bus (1: downtown station, 2: university station, 3: from downtown to the university, 4: from the university to downtown),

D is the duration of the current action,

$S' \in \{"loading", "unload", "travel"\}$ is the current status of the bus,

τ is the current simulated time..

Internal transition function $\boldsymbol{\delta_{int}: S \rightarrow S}$

$\delta_{int}(W_d, W_u, B, P = 4, D, S = \text{travel}, \tau) =$

$$= (W_d, W_u, B, P = 1, D = 0, S = \text{unload}, \tau = \tau + D)\ if\ B \neq \emptyset$$

$$= (W_d, W_u, B, P = 1, D = \upsilon, S = \text{load}, \tau = \tau + D) if\ W_d \neq \emptyset\ and\ B = \emptyset$$

$$= (W_d, W_u, B, P = 3, D = \beta_1, S, \tau = \tau + D) si\ W_d = \emptyset\ etB = \emptyset$$

If the bus arrives at the university station (the travel time is elapsed, then an internal transition must occur), it must stop to let the passengers get off. If the bus is empty, the users waiting at the station can immediately start getting in it. If the bus was empty, and no one was waiting at the station, a stop is not required.

$\delta_{int}(W_d, W_u, B, P = 1, D, S = \text{unload}, \tau) =$

$$= (W_d, W_u, B = rest(B), P, D = \mu, S, \tau = \tau + D)\ if\ B \neq \emptyset$$

$$= (W_d, W_u, B, P, D = 0, S = \text{load}, \tau = \tau + D)\ if\ B = \emptyset$$

All the passengers in the bus must get off when the bus stops at the station. They do it, one by one. When all the passengers get off the bus, the users waiting at the station can start getting in it.

$\delta_{int}(W_d, W_u, B, P = 1, D, S = \text{load}, \tau) =$

$= (W_d, W_u, B, P = 3, D = \beta_1, S = "travel", \tau = \tau + D)\ if\ W_d = \emptyset\ or \|B\| = \alpha$

$= (W_d = rest(W_d), W_u, B = B + first(W_d), P, D = \mu, S, \tau = \tau + D) if\ W_d \neq \emptyset\ \wedge\ \|B\| < \alpha$

If there is no one waiting, the bus can start its travel to the other station. Also if the bus is full, the bus can start its travel to the other station. If not, users can get in it, one by one.

Similar specifications are done at the following lines for the case the downtown station is considered.

$\delta_{int}(W_d, W_u, B, P = 3, D, S = \text{travel}, \tau) =$

$$= (W_d, W_u, B \neq \emptyset, P = 2, D = 0, S = \text{unload}, \tau = \tau + D)\ if\ B \neq \emptyset$$

$$= (W_d, W_u, B, P = 1, D = \upsilon, S = \text{load}, \tau = \tau + D) if\ W_u \neq \emptyset\ and\ B = \emptyset$$

$$= (W_d, W_u, B, P = 3, D = \beta_2, S, \tau = \tau + D) if\ W_u = \emptyset\ and\ B = \emptyset$$

$\delta_{int}(W_d, W_u, B, P = 2, D, S = \text{unload}, \tau) =$

$$= (W_d, W_u, B = rest(B), P, D = \mu, S, \tau = \tau + D) if\ B \neq \emptyset$$

$$= (W_d, W_u, B, P, D = 0, S = \text{load}, \tau = \tau + D) if\ B = \emptyset$$

$\delta_{int}(W_d, W_u, B, P = 2, D, S = \text{load}, \tau) =$

$= (W_d, W_u, B, P = 3, D = \beta_1, S = "travel", \tau = \tau + D) if\ W_u = \emptyset\ or \|B\| = \alpha$

$= (W_d, W_u = rest(W_u), B = B + first(W_u), P, D = \mu, S, \tau = \tau + D) if\ W_u \neq \emptyset\ \wedge\ \|B\| < \alpha$

$\alpha \in \mathbb{N}$ is the capacity of the bus

$\beta_1 \in \mathbb{R}^+$ est le temps de Voyage de la gare du centre-ville jusqu'à la gare de l'université,

$\beta_2 \in \mathbb{R}^+$ is the travel time from downtown station to university station

$\mu \in \mathbb{R}^+$ is the time for a user to get in the bus

$\upsilon \in \mathbb{R}^+$ is the time for a user to get from the bus

$first(list)$ returns the first element of list

$rest(list)$ returns list, which has been reordered after its first element has been removed,

$\|B\|$ gives the number of element of list B,

$"list + element"$ performs an adding of “element” to “list”, and reorders “list”,

External transition function$\boldsymbol{\delta_{ext}: Q \times X \rightarrow S}$

$\delta_{ext}\big((W_d, W_u, B, P, D, S, \tau), e, (x_1, x_2)\big) =$

$$= (W_d + x_1, W_u, B, P, D = D - e, S, \tau = \tau + D) if\ x_1 \neq \emptyset\ and\ x_2 = \emptyset$$

$$= (W_d, W_u + x_2, B, P, D = D - e, S, \tau = \tau + D) if\ x_1 = \emptyset\ and\ x_2 \neq \emptyset$$

$$= (W_d + x_1, W_u + x_2, B, P, D = D - e, S, \tau = \tau + D) if\ x_1 \neq \emptyset\ and\ x_2 \neq \emptyset$$

A user can line up at the downtown station, at any time. Then, the time he enters the UBS is updated to the current time, and the time he enters the bus is set to +∞. Similar situation at the university station. Situation where two users line up simultaneously at the downtown and the university station.

Confluent transition function: $\delta_{conf}: S \times X^b \to S$
$\delta_{conf}(s, x) = \delta_{ext}(s, ta(s), x)\ \forall s \in S$

Output function $\lambda: S \to Y$

$\lambda(W_d, W_u, B \neq \emptyset, P = 1, D, S = "unload", \tau) = (Bye1 = first(B), Bye2 = \emptyset)$

$\lambda(W_d, W_u, B \neq \emptyset, P = 2, D, S = "unload", \tau) = (Bye1 = \emptyset, Bye2 = first(B))$

A passenger who gets off the bus at any station exits the system too.

$\lambda(W_d, W_u, B \neq \emptyset, P, D, S, \tau) = (Bye1 = \emptyset, Bye2 = \emptyset)$ in all other cases.

Time advance function $ta: S \to \mathbb{R}^+ \cup \{+\infty\}$

$ta(W_d, W_u, B, P, D, S, \tau) = D$

II.2.3.4 DEVS modeling Tools

Over the years, several modeling platforms and tools have been developed based on DEVS theory and its extensions. DEVS models have been constructed using programming languages or some XML-based storage structures. Most of these platform dependent methods take advantage of reusability in object oriented programming by extending and reusing classes of DEVS models. Some notable platform-dependent tools include:

- DEVS-Scheme [Zeigler 1990] : is a knowledge-based environment for modeling and simulation based on the Scheme functional language (a variation of Lisp)
- DEVS-C++ [Zeigler et al. 1996] is a DEVS-based modeling and simulation environment written in C++, which implements parallel execution and supports large-scale systems
- DEVSim++ [Kim 1994] is an object-oriented DEVS simulator implemented in C++ that defines basic classes that can be extended by users to define their own atomic and coupled DEVS components
- ADEVS [Nutaro 1999] provides a C++ library based on DEVS, which developers can use to build their own models, and supports integration with other simulation environments
- DEVSJAVA [Sarjoughian and Zeigler 1998] is a DEVS-based modeling and simulation environment that provides Java classes for users to implement their own models. It also supports PDEVS, DSDEVS, RT-DEVS and 2D/3D cellular automata models. Distributed simulation is also possible due to the integration of several distributed implementations of DEVS abstract simulators. DEVSJava is now a part of DEVS-Suite, which provides some graphical facilities for modeling and simulation activities [Kim et al. 2009].
- VLE [Quesnel et al. 2009] is a modeling library that in addition to the DEVS formalism, implements in C++ several modeling formalisms like Petri nets, 2D/3D cellular automata, Quantized State Systems, etc. and allow heterogeneous integration of models of these

formalisms in PDEVS coupled model and their simulation by PDEVS simulators. Coupled models can be created in VLE graphically but atomic models are defined textually in C++. VLE integrates particular ports called initialization and observation ports respectively used for initialization and observation of models. The observation ports are connected to EOV (Eyes of VLE) which present simulation results in different forms.

- CD++[Wainer 2009] provides a C++ library to specify models in CDEVS, PDEVS and Cell-DEVS, an extension integrating cellular automata and DEVS. DEVS models and Cell-DEVS models can be mixed in the same simulation in CD++. CD++ integrates multiple simulation algorithms implementations including parallel and distributed simulation strategies. CD++ also provides an Eclipse plugin allowing the edition of DEVS and Cell-DEVS models both textually and graphically, as well as the graphical visualization of Cell-DEVS simulation results.
- SimBeans [Präehofer et al. 1999] is a component-based software architecture based on Java and JavaBeans. The idea is to provide a set of layered components that can be used in model creation, output results analysis, and visualization using DEVS.
- PythonDEVS [Bolduc and Vangheluwe 2002] is a basic implementation of CDEVS in Python programming language.
- JDEVS [Filippi and Bisgambiglia 2004] is a DEVS modeling and simulation environment written in Java. It allows general-purpose, component-based, object-oriented, visual simulation of models
- James II [Himmelspach and Uhrmacher 2004] implements DEVS theory including PDEVS, PdynDEVS and cellular automata to model and simulate agent systems. The toolkit supports software-in-the-loop simulation to test agents in virtual environments. James II integrates sequential and multi-threaded implementations of various simulation algorithms.

While these tools have been very beneficial and expressive for constructing DEVS models, using them requires knowledge of the programming platform. To discuss the models built with these platform-based tools, the domain engineers would need to understand the programming language making them less communicable. Graphical modeling approaches are better in this regard since they are visual. They are easier to discuss and share with domain experts. Advances in software engineering have enabled modelers to use graphical models and perform automated code synthesis from the graphical models to the desired text-based modeling language. All these DEVS implementations support building models in a hierarchical and modular manner independently from simulators (according to the separation of concerns introduced by the DEVS modeling and simulation framework). This is a system oriented approach not possible in other popular discrete event simulation tools like GPSS [Schriber 1980], Simscript [Chao 1971], Simula [Dahl and Nygaard 2003], etc.

DEVSML 2.0 (DEVS Modeling Language) [Mittal and Douglass 2012] is a revised version of DEVSML [Mittal et al. 2007] based on Finite and Deterministic DEVS (FDDEVS) [Hwang and Zeigler 2009] and defined by EBNF grammar. The earlier version of DEVSML developed models in Java and used XML for interoperability. DEVSML is composed of a transparent simulation layer and a platform independent modeling layer. The simulation layer is realized earlier by DEVS/SOA. The proposed DEVSML 2.0 integrates to the simulation layer a transparent modeling framework with inclusion of domain specific languages and

transformations. The transformations include model-to-model transformation (transform a specific model to another model), model-to-DEVSML (transforms a model to DEVSML) and Model-to-DEVS (transforms a DSL to a specific DEVS platform to allow simulation). Another language called NLDEVS is proposed (Natural Language DEVS). A DEVSML specified model is executable by using DEVS.

SimStudio [Traoré 2008] is a plugin-based modeling and simulation platform based on the DEVS formalism. The use of DEVS as a base formalism makes it possible to couple heterogeneous models [Vanguluvwe 2000] and utilize the defined operational semantic. It aims to provide a complete M&S tool chain to assist the model developer from model design to results analysis. In order to incorporate the existing tools and make it possible to easily add new features, SimStudio uses a modular software architecture relying on plugins (Figure 12). While SimStudio has its native plugins for modeling, simulation, visualization, and management, other tools for modeling, visualization, simulation, and analysis can be plugged onto the platform. The platform consists of the following axis:

- *Modeling Axis*: consisting of graphical and textual tools for model design. The modeling modules provide their output specification in XML format as understandable by SimStudio. SimStudio adopts HILLS as its modeling language. The native modeling plugin for SimStudio is the HILLS Editor.
- *Simulation Kernel*: the core of SimStudio for automatically generating code for operational analysis, parallel and distributed simulation. The role of the simulation kernel is to automatically generate an operational solution from a model specification, then to handle its deployment and execution on various types of platforms (Java, C++, Python…), according to the user's choice. This can range from a simple execution offline or on a server to a distribution on a computing grid or cluster.
- *Analysis Axis*: integrating tools for formal analysis. The modules here would analyze both the model and simulation results and compare with system specification and desired properties. Existing tools for formal analysis (CTL, Z, CSP …) are plugged into SimStudio via its underlying XML-based structure.
- *Visualization Axis*: animation and visualization of the simulation results to facilitate interactive exploration of the system under study.
- *Management Axis*: provides services like user management, model storage and querying, web-based collaboration…

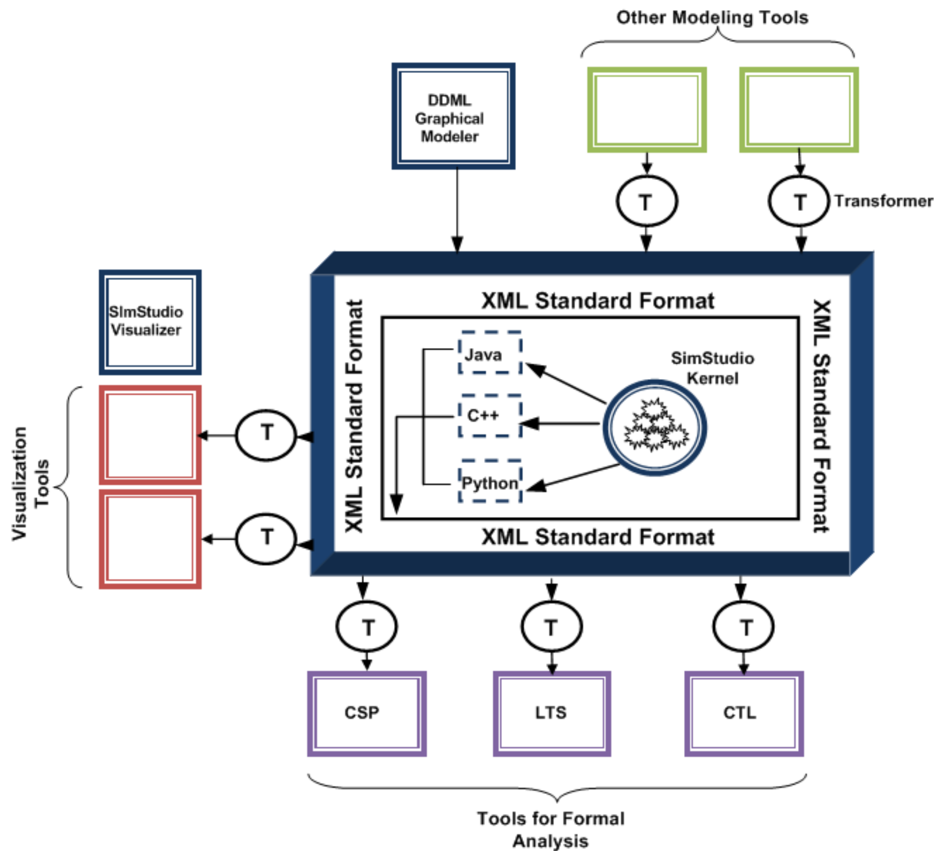

Figure 12. Simstudio components

SimStudio relies on the generic and platform-independent structure of XML to enable addition and integration of existing tools.

Related works address some visual notations and realizations for DEVS models. Some notable graphical approaches integrating UML to DEVS modeling include DEVS/UML [Money 2008], Executable UML with DEVS (eUDEVS) [Risco-Martin et al. 2009], and the object-oriented co-modeling methodology [Sung and Kim 2010]. Other approaches such as CD++ Builder [Matias et al. 2010], PowerDEVS [Pagliero et al. 2003], and The revised DEVS graph [Song and Kim 2010] are based on the definition of new languages.

- [Feng 2004] proposes Dcharts, formalism for modeling and simulation based design of reactive software systems. It is a UML-like statechart language but does not follow the UML standard.
- DEVS/UML provides a representation of DEVS models as UML state machines. A simulator has been proposed for such models. But all models representable by this approach are a subclass of DEVS (finite state assumptions) because of the difficulty of expressing DEVS concepts in UML state machines.
- eUDEVS approach is doing the opposite of DEVS/UML, it transforms UML models into DEVS models but the models obtained are in a restricted class of DEVS called FD-DEVS (Finite and Deterministic DEVS).
- In [Sung and Kim 2010] the authors present an approach using a subset of UML to support the process of object-oriented modeling based on DEVS. This approach uses

UML to model parts of the behavior of objects and DEVS to complete the missing discrete event information. They use the Use case diagrams, class diagrams and sequence diagrams of UML. The procedure to transform a given sequence diagram model in DEVS is adapted only to finite sets of states.

- In [Nikolaidou et al. 2008] SysML profile for DEVS is proposed as a standardized, graphical representation language of DEVS models stored in DEVSML, consequently transformed into executable code for existing DEVS Simulators, as DEVSJava and DEVSim++. SysML profiles are based on UML lightweight extension mechanisms. The authors argue that SysML is more suitable than UML for the graphical representation of DEVS models, since SysML language and especially block diagrams provide the natural representation of DEVS model decomposition. It is important to notice here that SysML is a UML profile. In the profile, DEVS model entities are defined as stereotypes of SysML entities, while constraints are used to restrict SysML semantics to DEVS formalism. A DEVS Coupled model is defined by SysML Block Definition Diagrams. Since SysML Block Definition Diagrams do not depict how components are interconnected, Internal Block Diagrams are used to describe coupling between subcomponents defined in the Block Definition Diagram. DEVS Atomic models are defined as stereotypes of SysML blocks. For DEVS Atomic models, a state machine diagram is used for the definition of internal transition function, output function and time advance function. An activity diagram is used for the definition of the external transition function. The reason of using two different kinds of diagrams to define internal and external transition functions is not clear in the paper. While the idea of the paper is interesting, the defined SysML profile for DEVS completely depends on SysML and inherits the same limitations and drawbacks.
- The revised DEVS Graph is a structured diagram form of the DEVS formalism (C-DEVS). DEVS Graph uses the concepts of ports and messages for structuring sequential events and it introduces the concepts of phase transition diagram to simply represent state transitions. A tool to facilitate model construction and enable transition from the graphical model to code would be a useful contribution and addition since this is not provided. It does not however provide a means to represent P-DEVS models; hence its modeling for parallel simulation is not practical.
- [Schulz et al. 2000] discusses the equivalence between the Discrete event system specification (DEVS) and statemate statecharts for embedded systems modeling. Statemate [Harel et al. 1990] is a successful commercial implementation of statecharts. Authors show that DEVS also can be used for modeling systems addressed by statecharts and more.
- [Zinoviev 2005] proposes a mapping of DEVS models onto UML models. In fact they map DEVS atomic and coupled models to UML state machines and components respectively. The way the transformed models will be simulated or analyzed is not precised. The mapping approach of DEVS atomic models state space to UML state machines states proves that this approach is limited to a subclass of DEVS.

Existing approaches including model transformations from DEVS to finite state machines and transformations of UML models into DEVS models do not address the general case because all the models obtained through the transformations are simply a subset of all models that can be represented by DEVS. HILLS is more expressive than existing approaches because it can specify

infinite systems by providing concepts that can represent them graphically in a finite manner. Our approach presents two advantages: it gives a complete methodological approach and the resulting visual language allows the automated generation of simulation code and accessibility to formal analysis; and the automated synthesis of code for enactment.

II.2.3.5 Verification and Validation of DEVS models

What kind of analysis one can do on DEVS models? The de-facto verification method of DEVS models is based on the analysis of simulation results. To achieve simulation of DEVS models, one needs a DEVS modeling tool that implements DEVS concepts and operational semantics. Analyzing DEVS models necessitate collecting simulation results, animating and visualizing the results and using statistical techniques to analyze simulation traces. Most of the presented DEVS tools allow only textual modeling in General Purpose Programming Language (GPPL) like Java or C++ and executing the models. It is not clear however how data is collected, or animation and visualization and analysis are done using these platform dependent DEVS modeling tools. Most of these do not provide any integrated means of verification and validation for DEVS models. From these tools it is not easy to get access to other analysis tools**.** There is a need to integrate different techniques such as simulation, model checking, symbolic reasoning and theorem proving to completely analyze system properties. In section II.3.4, we discuss DEVS models formal verification and validation approaches.

II.2.3.6 DEVS extensions

Many extended formalism from DEVS have been introduced:

- DEVS&DESS for combined continuous and discrete event systems,
- G-DEVS [Giambiasi et al. 2001] for the modeling of Discrete Event Systems where the trajectories are organized through piecewise polynomial segments,
- RT-DEVS [Cho and Kim 2001] for real-time Discrete Event Systems,
- Cell-DEVS [Wainer 2004]for cellular Discrete Event Systems,
- Fuzzy-DEVS [Kwon et al. 1996] for fuzzy Discrete Event Systems,
- ml-DEVS (Multi-Level DEVS) [Steiniger et al. 2012],
- Symbolic DEVS [Chi 1997],
- DSDEVS (Dynamic Structuring DEVS) [Barros 1995] and dynDEVS [Uhrmacher 2001] and rho-DEVS [Uhrmacher et 2006] for Discrete Event Systems changing their coupling structures dynamically,
- ALRT-DEVS (Action-Level Real-Time DEVS) [Gholami and Sarjoughian 2012][Sarjoughian and Gholami 2013],
- I-DEVS (Imprecise Real-Time for Embedded DEVS Modeling) [Moellami & Wainer 2011].

In addition, some subclasses such as SP-DEVS [Hwang and Cho 2004] and FD-DEVS [Hwang and Zeigler 2007] have been defined for achieving decidability of system properties. FRT-DEVS (Finite Real-Time DEVS) [Hwang 2012] is a subclass of RT-DEVS for making RT-DEVS properties formal verification decidable.

III.2.2.6.1 DEVS for dynamic structure systems

We present in this section, state of the art in dynamic structure systems modeling and simulation. We will focus DEVS-based approaches and particularly on DSDEVS because it is used as a semantic domain for HiLLS to allow simulation of HiLLS dynamic structure models.

Dynamic structure systems are characterized by the following on the structure and/or behavior of a system:

- Changes in the set of components: many systems are characterized by the dynamically changing number of their components. Existing components can be deleted and/or new components can appear during execution. To support this future a dynamic structure modeling and simulation framework should provide basic *addComponents()* and *deleteComponent()* operations and their semantics, i.e. define how these operations affect the structure and the behavior of the network that contains them and the other components.
- Dynamic couplings: some dynamic structure systems are essentially characterized by the variability of coupling links between components of the same network. To allow dynamic couplings, linking and unlinking operations should be provided. It will then be possible to link newly added to other components, or unlink existing components or redirecting outputs of some components to others.
- Dynamic interface: most systems assume static number of interaction ports but some systems undergo interface changes in time, i.e the number of input or output can change in time. Example of such kind of systems can be found in biology. Changes in the interfaces of a component can lead to changes in links between components at network level. *addPort()* and *removePort()* are the basic operations that can allow this kind of changes.
- Dynamic behaviour: refers to how a component can change its behavior depending on local or network situation. Changing behavior can lead to the redefinition of some or all the functions that govern the way the component reacts to external and internal events and the computation and nature of its outputs.

The major problem that arises when these operations are applied is how to ensure consistency between structure and behaviour? The consistency problem is related to the control of changes in components behaviors and their network structure. Different approaches have proposed different modeling approaches and controls scheme. We discuss in this section the existing approaches in the literature.

II.2.3.7.1 DSDEVS

Dynamic-structured DEVS (DSDEVS) [Barros 1997] has been proposed for over two decades to model dynamic-structured discrete event systems for simulation. It is a variant of DEVS with capability for modeling structural dynamics. It retains the specification of atomic DEVS while introducing a "network executive" model as part of the coupled model specification. Being the manager of structural changes, the network executive declares state variables that store the structural information of the system network containing it so that there is a one-to-one correspondence between the executive's instantaneous states and the network's structure. This concept was extended in [Barros 1997] to develop formalisms for modeling structural dynamics in Differential Equation, Discrete Time and Discrete Event Systems. As proposed in [Barros

1997] a discrete event dynamic structure system is a couple $DSDN = (\chi, M_\chi)$ where χ is the name of the network executive and M_χ is the dynamic of network. The structure of M_χ is of the form $M_\chi =< X_\chi, S_\chi, s_{0,\chi}, Y_\chi, \gamma, \Sigma^*, \delta_\chi, \lambda_\chi, \tau_\chi >$ where

- X_χ and Y_χ are the input and output interfaces of the system,
- S_χ is the set of states,
- $s_{0,\chi}$ is the initial state,
- Σ^* is the set of possible structures of the network,
- $\gamma: S_\chi \rightarrow \Sigma^*$ is the function which associate to each state of the network a corresponding structure. $\forall \Sigma \in \Sigma^* \wedge s_x \in S_\chi, \gamma(s_\chi) =< D, \{M_i\}_{i\in D}, \{I_i\}_{i\in D}, \{Z_i\}_{i\in D} >$
- $\delta_\chi: Q_\chi \times (X \cup \{\emptyset\}) \rightarrow S_\chi$ is the transition function,
- $\lambda_\chi: S_\chi \rightarrow Y$ is the output function and
- $\tau_\chi: S_\chi \rightarrow \mathbb{R}^+ \cup \{+\infty\}$ is the time advance function.

In DSDEVS only single central network executive is responsible of the management of structural changes in the network. This prevents ambiguity when different components require structural change. The single executive ensure also structure and behaviour consistencies. An Object Oriented implementation of the DSDEVS formalism is the DELTA Environment. DELTA has been implemented in Smalltalk [Barros 1995].

Based on Barros' work, [Hu et al. 2005] developed formalism with a few differences and additional concepts to simulate dynamic reconfigurations in component-based systems. In addition to the structural operations defined by Barros on individual components, they extended the same operations to the interfaces (and ports) of components. This approach would foster a more fine-grained analysis of complex systems. However, the authors did not provide a formal specification of the formalism; in all the examples provided, the concept was rather hardcoded in some java implementations of the DEVS simulation protocol.

II.2.3.7.2 dynDEVS

Another formalism based on DEVS which addressed the modeling of dynamic structure systems is dynDEVS [Uhrmacher 2001]. In contrary to DSDEVS which introduce the network executive for the specification of structural changes, dynDEVS introduces the transition functions, ρ_α and ρ_N at the level of atomic and coupled model definitions respectively. dynDEVS Atomic models are defined as follows: $dynDEVS =< X, Y, m_{init}, M(m_{init}) >$ where

- X,Y are the sets of inputs and outputs;
- $m_{init} \in M(m_{init})$ is the initial model;
- $M(m_{init})$ is the least set of the form $< S, s_{init}, \delta_{int}, \delta_{ext}, \lambda, ta, \rho_\alpha >$ where S is the set of state, $s_{init} \in S$ is the initial state, $\delta_{int}, \delta_{ext}, \lambda, ta$ are respectively internal transition, external transition, output and time advance functions with the same meaning as in DEVS and $\rho_\alpha: S \rightarrow M(m_{init})$ the model transition function which allows switching from different specifications of the same systems during simulation when it must respond to external and internal events and their implied state changes. To support continuity between models ρ_α preserves the values of variables common to two successive models and assigns “default initial” values to the “new” variables.

dynDEVS coupled is defined as follows: $dynDEVS =< X, Y, n_{init}, N(n_{init}) >$where

- X,Y are the sets of inputs and outputs;
- $n_{init} \in N(m_{init})$ is the initial configuration;
- $N(n_{init})$ is the least set of the form $< D, \rho_N, \{dynDEVS_i\}, C, Z_{i,j}, Select >$ where D is the set of components names, $\rho_N: S \rightarrow N(n_{init})$ the network transition function with $S = X_{d \in D \oplus m \in dynDEVS_d} S^m$, $\{dynDEVS_i\}$ is the set dynDEVS components, I_i is the set of influencers of i, $Z_{i,j}$. is the i-to-j output-input translation function; *Select* is the tie-breaking function. The function ρ_N preserves the state and structure of models and initializes the new components.

The particularity of dynDEVS is that all components can change structure and/or behaviour. This is powerful but can lead to inconsistencies and complex implementation.

II.2.3.7.3 Rho-DEVS

Rho-DEVS [Urmacher et al. 2006] is a new formalism that evolves from dynDEVS [Urmacher 2001]. Rho-DEVS introduces Dynamic ports and multi-couplings to handle changes in variable structure components interfaces and enabling or disabling certain communications between components in addition to the capability of specifying models which can adapt their own interaction structure and their own behaviour inherited from dynDEVS. In contrary to dynDEVS which is based on Classic DEVS and the assumption of a static set of ports, $\rho - DEVS$ is based on Parallel DEVS and manages a variable set of ports where special type of port contains structural change information. This information is produced by a special function. Multi-couplings bring some facilities for modelling special systems like cellular biological systems.

While DSDEVS has a sound formal background, model specification can be very laborious because modeler must list all the possible structures of the system as a part of the state space of the network executive (the manager of structural changes) which can grow exponentially depending on the number of components and dynamic couplings between them. As DSDEVS, dynDEVS modeling approach is not easy due to same reason and the complexity of specifying $S = X_{d \in D \oplus m \in dynDEVS_d} S^m$. Like DEVS itself, DSDEVS and dynDEVS doesn't have a concrete syntax and logical semantics. HiLLS proposes a simpler approach to modeling structural dynamics without need for a separate "executive" model to capture structural information: these informations are inherent in the system's configuration and appropriate structural changes occur naturally with state transitions. The ability to specify this aspect in a graphical language also makes our approach easier to use and accessible to a larger audience. We choose DSDEVS as our preferred formalism to a semantic domain of HiLLS for dynamic structure systems.

II.2.3.7.4 DYS-DEVS

A recent formalism DYS-DEVS for the specification of dynamic structure discrete event systems using single point encapsulated control functions is proposed in [Muzy and Zeigler 2014]. In this approach, at each time only one component (atomic component or coupled component) is responsible of structure changes. Two cases are possible: the component responsible of the structure changes can be the same every time (static single point) or can change during simulation (dynamic single point). DYS-DEVS distinguish between basic DYS-DEVS and network DYS-DEN models. A basic DYS-DEVS model is defined by $DYS - DEVS = (\mathcal{M}, S, \tau_b)$ where $\mathcal{M}$ is

the set of possible structures in form of classic DEVS atomic models ($DEVS = (X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta)$) of the component, S is the disjoint union the partial state sets of each possible structure and $\tau_b: \mathcal{M} \times S \rightarrow \mathcal{M} \times S$ is the structure transition function. A network model is defined by $DYS - DEN = (\aleph, S, \tau_\aleph)$ where $\aleph$ is the set of possible network structure, S is the disjoint union of state sets of possible network structure states and $\tau_\aleph: \aleph \times S \rightarrow \aleph \times S$ is the network structure transition function. DYS-DEVS and dynDEVS introduce similar structural changes control functions at atomic and network level. The difference between these functions resides in the fact that dynDEVS functions depends only on the current state while DYS-DEVS functions depend on the actual structure and the state. While the approach is formally defined, the modeling difficulty remains the same as with DSDEVS and dynDEVS for the same reasons.

II.2.3.7.5 Kiltera

[Posse 2008] proposed a visual modeling environment and code generators for DEVS and cellular systems. The author proposed also the Kiltera language based on process algebras (pi-calculus) and discrete-event modeling approaches. Kiltera is a language for the modeling and simulation of discrete-event dynamic structure systems. A Kiltera model is composed of concurrent and distributed processes communicating asynchronously via channels. Distributed modeling (site-dependant behavior) view and execution by implementing a variant of the time warp algorithm. The operational semantics of the Kiltera language is defined in terms of a labeled transition system and implemented simulation algorithms. Kiltera use the unicasting (i.e only one process react to a given event if many are listening to it) approach in the communication between processes by default. Multicasting is also possible by using a particular keyword. A limitation of this approach is that this multicasting is decided at the level of events (not easy and low level). Dynamic structure is modeled using process composition operators (choice, parallel, sequential), timeouts, sending of channels between processes, ending processes. Kiltera has concrete textual modeling syntax. The complexity of the language resides in the fact that it merges many concepts at the same level.

II.2.3.7.5 Other approaches to Dynamic Structure Systems Modeling and Simulation

Multimodels [Ören 1991] offer another possibility to represent dynamic structure models. The Multimodeling approach provide a framework for representing models containing several submodels, where only one model can be active at any time under some conditions. The structure changes in the multimodel result in change from one model to another due to the activation conditions. [Barros et al.1998] integrated the dynamic structure DEVS formalism with the multimodel paradigm by representing multimodels within the DSDEVS and DEVS formalisms so that they can be mapped to HLA/RTI distributed simulations.

[Santucci and De Gentili, 2009] proposed Dynamic variable structure modeling and simulation applied to the modeling and simulation of the Claude Levi-Strauss's mythical thought morphodynamics by merging DSDEVS concepts, the notion of structural supervisor of [Baati et al. 2007] and simulation algorithm of [Hui & Wainer, 2006]. The proposed implementation reuse classic-DEVS abstract simulator for simulation because of the target application.

DSDEVS-hybrid is a formalism for the modeling of hybrid variable structure systems proposed in [Pawletta et al. 2002]. DSDEVS-hybrid defines a variable structure system by using the

DSDEVS formal specifications, and parts of dynDEVS simulation algorithms. A coupled model noted is similar to coupled model in CDEVS extended with specifications and methods for continuous and variable structure systems and a specific state variable that store structural change information in the system. A toolbox of DSDEVS-hybrid for Matlab is described in [Deatcu and Pawletta 2009]. Another DEVS-based approach proposed by other authors for modeling variable structure hybrid systems is presented in [Chen et al. 1999]. [Hagendorf et al. 2009] proposed the extended Dynamic Structure DEVS (EDSDEVS) that combines CDEVS, PDEVS and DSDEVS in order to combines also the advantages of these approaches.

II.3 Formal Methods

A main challenge in the domain of complex dynamic systems modeling is to provide languages, methods and tools that will improve modeling process and helps to assess model integrity (i.e. to prove that the specified model satisfy the requirements of the systems). Formal methods are mathematical techniques applied to the specification, analysis, design and implementation of complex software and hardware [Clark and Wing 1996, Kuhn et al. 2003, Lamsweerde 2000]. The use of formal methods can greatly increase the understanding of a system through formal specification and formal verification. Formal specification is the process of describing a system and its desired properties, using a language with a mathematically-defined syntax and semantics. The kinds of system properties might include functional behavior, timing behavior, performance characteristics, or internal structure. Formal specification may be manipulated by automated tools for user confirmation through deductive theorem proving techniques. These automated methods can be used to generate concrete scenarios illustrating desired or undesired features about the specification. Z [Spivey 1992], CSP [Haore 1985], VDM [Dines et al.1978], B method [Abrial 1996], CTL [Emerson 1991] are some examples of formal methods. In the literature a distinction is made between formal specification formalisms and formal methods. Specification formalisms formally define formal abstract syntax and semantics for system specification. Formal methods are based on specification formalisms; they propose a concrete syntax and offer a methodology for models development. Most formal notation are used as formal specification formalism and formal method (examples are Z, VDM, Object-Z etc.).

Formal specification techniques can be classified into the following five main paradigms [Lamsweerde 2000]:

- History-based specification characterizes the maximal set of admissible histories over time. The properties depicting the behavior of the system are specified by temporal logic assertions.
- State-based specification characterizes the maximal set of admissible system states. The properties depicting the states of the system are specified by invariants constraining the system objects at any snapshot, and pre- and post-assertions constraining the application of system operations at any snapshot.
- Transition-based specification characterizes the required transitions from system state to state. The properties of interest are specified by a set of transition functions which give for input states and triggering events (eventually guarded by necessary precondition) the corresponding output states.
- Functional specification characterizes the system as a structured collection of mathematical functions that are grouped, either by types then defining algebraic

structures, or into logical theories (high-order functions). The properties of interest are specified as conditional equations that capture the effect of composing these functions.
- Operational specification characterizes the system as a structured collection of processes that can be executed by some more or less abstract machine.

Formal verification is based on two main techniques: model checking and theorem proving.
- Model checking is an automated technique to check the absence of errors (i.e., property violations) that requires a model of the system under consideration and a desired property and systematically checks whether or not the given model satisfies this property. Typical properties that can be checked are deadlock freedom, invariants, and request-response properties. It can be used to check partial specifications, then providing useful information about system's correctness even if the system is not completely specified. Also, it can produce counterexamples which typically are subtle errors in design. The problem of model checking is the state explosion problem. However, the improvements in model checking algorithms, data structures, and availability of faster computers and efficient representation of state transitions increase the size of systems that could be verified. As model checkers, we can cite SPIN [Holzmann 2003], SMV [McMillan 1992], UPPAAL [Larsen et al 1997], etc.
- Theorem proving is a technique where both the system and its desired properties are expressed as logic-based formulas, in terms of axioms and inference rules. Theorem proving is the process of finding a proof of a property from these axioms and rules, and possibly derived definitions and intermediate lemmas. In contrast to model checking, theorem proving can deal with infinite state spaces, but it usually results in a slow and often error-prone process. As theorem provers, we can cite Z/EVES [Saaltink 1999], FDR [FDR 2010], HOL [Gordon and Melham 1993], Isabelle [Paulson 1993], PVS [Owre et al. 1992] etc.

Formal methods have been used extensively in the specification of reactive systems. Examples worth of note include Statecharts [Harel 1998], Z, VDM, CSP… For systems with multiple static and dynamic aspects, extensions and combinations of formal methods have been proposed. For example, CSP has been extended to include timing aspects. This is called Timed CSP [Reed and Roscoe 1986]. The following are other combination of formal methods:
- Z and CSP [Fischer 2000] is an extension of Z for specifying CSP process. This integration is made regarding syntax and semantics. The tool used for verification of properties on the CSP-Z specification is FDR.
- ZCCS [Galloway and Stoddart 1997] is the combination of Z and Calculus of Communicating Systems (CCS) [Milner 1980]. A ZCSS specification is a CSS specification with data and axioms specified in Z
- CSP-OZ [Fischer 2000] is an integration of Object-Z and CSP. CSP-OZ defined a new semantics from those of CSP and Object-Z. Another combination of these languages CSP/Object-Z [Smith and Derrick 2001] adopts two different views for the same system using the two syntaxes. The object components are specified by the Object-Z and concurrency is specified by the CSP. The CSP part uses Object-Z classes. The link between both is defined by the links between operations and events and between classes and processes. These languages do not have tools to verify properties. Their refinement relations are based on those of Z and CSP.

- CSP2B [Butler 1999] and CSP||B [Schneider and H. Treharne 2002] are two variants combining CSP and B. For CSP2B, CSP-like specifications are associated with specifications to control the execution order of operations defined in the B machine associated with it. CBS||B maintains two dimensions: CSP and B parts. The CSP part acts as a coordinator of the operations of the concurrent B machines associated with it.

In the next section, we present the Z and CSP formal notations. Z and CSP are used a semantic domains for HiLLS that allow formal analysis of its models.

II.3.1 Z language

Z [Spivey 1992] is a formal notation introduced for abstract, declarative, non-executable requirements specifications free from implementation details. The language has formal semantics and has been standardized [ISO 2002]. Z is a typed language, consisting of three parts: a mathematical language, a schematic language and a refinement theory. The Z Formalism uses mathematical data types to model the data in a system. The notation of predicate logic allows the abstract description of the operations that cause state changes in the system. An important element in Z is the way of decomposing a specification into small pieces called schemas. Schemas are used to describe both the dynamic and static aspects of a system. It brings together data and operations. The schemas allow the reuse of existing schemas specifications with well-defined operations. Through the use of schema inclusion, it is possible to describe systems at the highest level of abstraction and increase the level of details in the specification at subsequent phases. In Z, the properties of operations, and their effects on the state of the system, can be explored and reasoned about formally using tools like Z/EVES [Saaltink 1997, Saaltink 1999]. The Z formal language is expressive and allows and unambiguous requirements specification by showing what has to be done, not how it should be done (free of implementation details).

In most case studies of system specification with Z, the methodology used is to treat the system as having an overall state schema. The operations are defined in terms of state changes on the overall state schema resulting from the composition of the different state schema. An operation is defined by specifying its signature, its precondition and postcondition. The signatures of the operation schemas include a primed and an unprimed version of the signature of the state schema and the input and output parameters. A change of state is a relation between the current state and future state of the system.

Several extensions have been proposed for the Z language. These extension include object oriented extensions like Object-Z [Smith 2000] [Duke et al. 1991], Z++ [Lano 1991], OOZE (Object Oriented Z Environment) [Alencar and Goguen 1991], MOOZ (Modular Object-Oriented Z) [Meira & Cavalcanti 1990], ZEST [Cusack & Rafsanjani 1992]) ([Stepney 1990]), process algebra extensions like ZCCS [Galloway and Stoddart 1997], CSPZ [Fischer 2000], etc. Some of these extensions also have been extended to include real-time features like TCOZ (Timed Communicating Object-Z) [Mahony and Dong 2000].

II.3.1.1 A simple specification of the stack data structure in Z

The stack (Figure 13) is characterized by a global constant *maxLenght* that models the maximum number of elements that can be stored in it. The state schema declares and constrains one state variable *contents* which is a sequence of integers to hold the items in the stack while the *INIT*

schema sets the state variable to a default value when invoked. The three operations *pop, push* and *top* manipulate the state schema; while the first two result in to changes of state by modifying the state variable *contents* as first indicated by the *schema inclusion* $\Delta Stack$ in their upper segments and their post-conditions as portrayed by the last predicates in their respective second segments, the last operation does not affect the value of any state variable when invoked and that is why it has no schema inclusion. The first predicate in the second segment of each of the operations specifies the pre-conditions for them to be successfully invoked. We assume that a basic knowledge of predicate logic is enough to understand the details of the sample model.

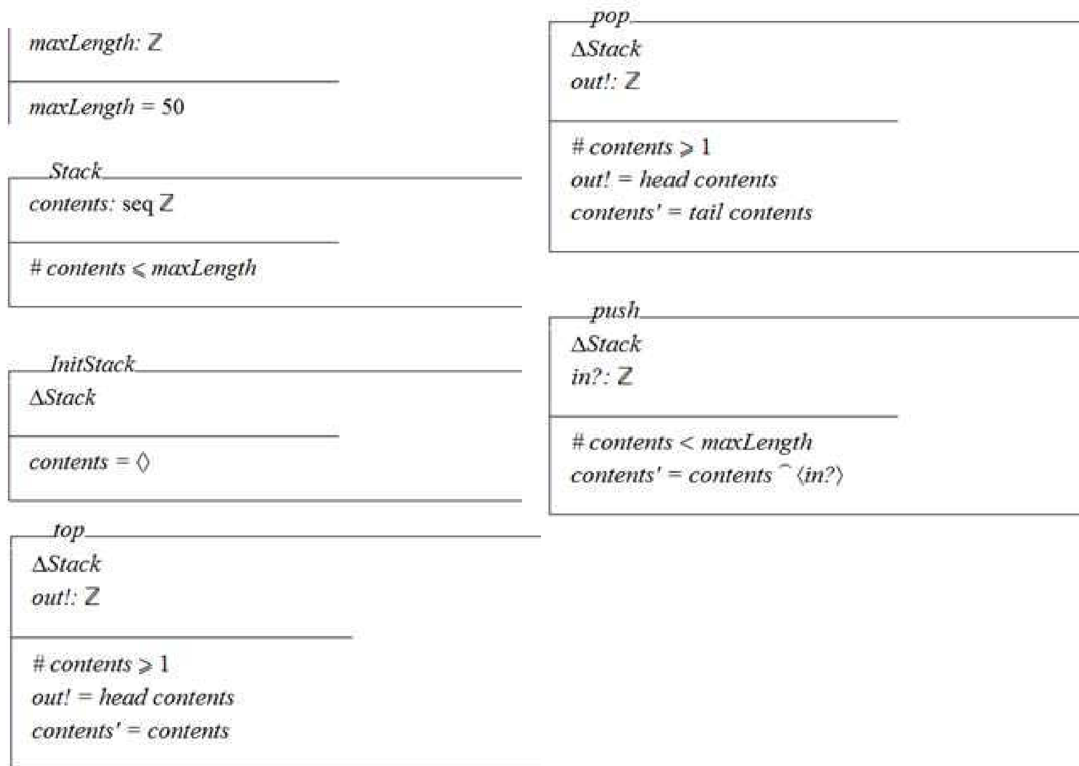

Figure 13. the Stack specification in Z

II.3.2 Object-Z

Object-Z is a conservative object oriented extension of Z. On top of the Z notion of schema, Object-Z introduces the concept of class schema which captures the object-oriented notion of a class by encapsulating a single state schema with all the operation schemas which are defined on its variables. Like in other object oriented languages, a class in Object-Z may inherit from other classes and have a visibility list restricting the way objects of the class may be used.

An Object-Z class has a unique name (Figure 14) as an identifier to differentiate it from other classes in the specification. In addition to the *class name*, the header may also specify some *generic parameters*.

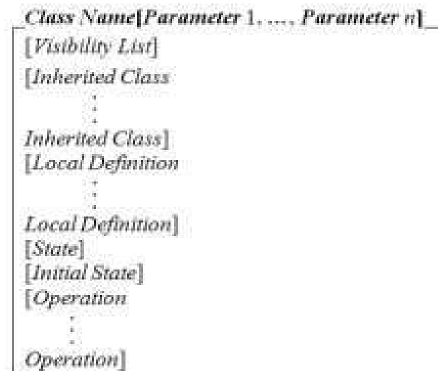

Figure 14. Template of Object-Z class

Since the class encapsulates its contents, the *visibility list* specifies the interface through which the elements of an object of the class may be accessed. i.e., a list of variables and operations that can be visible outside the class in similitude to public attributes and methods in Object-Orientation. An *Inherited Class* designates an existing class whose definition is imported for reuse. A *Local Definition* may be a local type or constant definition (usually specified in an *axiomatic schema*) or a *reference* to another class. A class may have a maximum of one *state schema* that defines its *state space* through the declaration of *state variables* and *invariants* (if any). This may be followed by a specification of the *initial state* and finally, the *operations* that use and/or manipulates the elements of the class. Further details about Object-Z's syntax and semantics can be found in [Smith 2000].

There are many specification formalisms for logical analysis; we have chosen Object-Z for three properties, two of which it inherits from Z (its base formalism).

1. Z is said to be considerably universal to be suitable for describing DESs for most kinds of logical analysis and
2. Z allows for separation of concerns. i.e., its syntax enable to decouple the specification of system properties from the requirements investigation logics and
3. Object-Orientation (which is peculiar to Object-Z) which enables to modularize a complex specification.

The state schema is nameless by convention and has its declarations partitioned by a Δ into primary and secondary variables. Secondary variables are implicitly available for change in any operation and are usually defined in terms of primary variables. Both constants and state variables can be object references. The operations are defined either as operation schemas or operation expressions.

An operation schema extends the notion of a schema in Z by adding to it a delta-list. The delta-list is a list of the primary variables which the operation may change when it is applied to an object of the class; all other primary variables remain unchanged. When two or more operations are combined to define a new operation, however, their delta-lists are united so that the new operation may change any variable which any of its constituent operations could have changed. In this way, delta-lists enable a more flexible calculus for combining operations than would be allowed by Z schemas alone.

Apart from conjunction ∧, other operation operators are the parallel operator ||, the choice operator [ ], the enrichment operator •, hiding and renaming. The parallel operator || is a binary operator introduced into Object-Z to allow specification of inter-object communication. The operator identifies and equates input variables in either operation with output variables in the other operation having the same base name. The identified input variables are hidden in the resulting operation; the output variables are not hidden and so may be equated with other input variables in subsequent parallel compositions.

The choice operator [] is a binary operator which allows the specification of nondeterministic choice between operations. The meaning of $op_1[]op_2$ is that either operation op_1 occurs or operation op_2 occurs but not both.

The enrichment operator • allows operations to be interpreted within the class' local environment enriched with the declarations and predicates of another operation or schema.

II.3.2.1 Stack specification in Object-Z

Figure 15 shows The Object-Z version of the stack data structure presented in II.3.1 to illustrate some main features of Object-Z. The Stack is now represented as a class in Object-Z. Contrary to Z, the state schema and the operations are defined within a single class. The visibility list of the class contains the three operations; *pop*, *push* and *top* and the *INIT* function meaning that these can be accessed in any instance of the Stack class just the same way public features are accessed in other Object-oriented paradigms. As in the Zversion, the class declares and initializes a global constant *maxLenght* that models the maximum number of elements that may be stored in any object of the class. The unnamed state schema of the stack class is the same as the definition of the state schema of the Z version. The *INIT* operation of the class plays the same role as the corresponding schema of the Z version; it is the constructor of the class. Similar to the use of schema inclusion in the Z version, the *delta* lists included in the operations *pop* and *push* indicate that they result in to changes of state by modifying the state variable *contents.* The *delta* lists simplify specifications. As in the Z version, no *delta* list is included in top operation since it does not affect the value of any state variable when invoked. Pre and post-conditions of the operations have the same meanings as in Z specification.

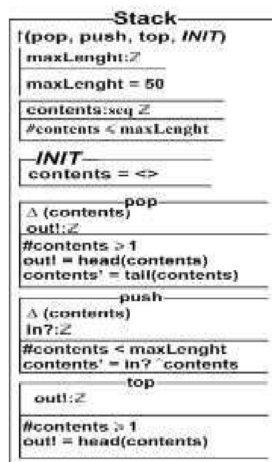

Figure 15. A sample model in Object-Z

II.3.3 CSP

CSP (Communicating Sequential Processes) [Hoare 1985] is a formal notation used for the specification and analysis of concurrent systems. CSP is a process algebra where basic elements are processes and events. In CSP, processes are independent entities that can communicate with each other. A process can execute events or actions. The way a process engages on events defines its behavior. Each process P is characterized by the set of events it can accept; this set is called the alphabet of the process and is denoted by $\alpha(P)$.

Formally, the syntax of CSP is defined by the following grammar:

- The basic process in CSP is STOP; it is the process that does nothing.
- To specify the behavior of a process, the prefix ($\rightarrow$) operator is used to defined explicitly the events that the process can execute and how it behaves after executing events. For example $a \rightarrow P$ defines a process that behaves like P after executing the event a. Specifying a complex behavior may require a sequence of prefix operator, use of parentheses and recursion.
- Input and output events: Communication between processes is realized through exchange of input and output events on channels. An input event is denoted by $c?v$ where c is the channel and v is the value received through it. An output event is denoted by $c!v$ where c is the channel and v is the value sent on it. For example $c?v \rightarrow P$ is a process that executes an event v received on channel c and behave like P. Similarly, $c!v \rightarrow Q$ is a process that sends an output event v on channel c and behaves like Q.
- Choice operator: There are several choice operators in CSP; the simpler is the operator denoted by $|$ that is applicable to processes of the form $a \rightarrow P$. For example, the process $a \rightarrow P|b \rightarrow Q$ is the process that behave like P if it executes event a or Q if it executes event b. $\boxdot$ and $\sqcap$ are other choice operator used in CSP called respectively internal and external choice operators. $P \boxdot Q$ is the process that can execute all the events that can be executed by P or Q and behaves like P if the executed event is initiated by another process and belongs to $\alpha(P)$ or Q if it belongs to $\alpha(Q)$. $P \sqcap Q$ is the process that behaves nonderministically as P or Q on internal events.
- Hidden events: In CSP, it is possible to hide some event from the alphabet of a process. If P is a process and A is a set of events then $P\backslash A$ is the process where elements of A are considered as internal events of the process.
- Parallel composition: The process $P_A \parallel_B Q$ is the parallel composition of P and Q. In this concurrent process, P can only execute events belonging to A and Q can execute only those in B. Events in $A \cap B$ are executed by P and Q synchronously.
- Interleaving: Interleaving is like a parallel composition without synchronization. In the interleaving process $P|||Q$, the constituent processes P and Q can execute events independently.

The semantics of CSP is based on three kinds of behaviors for expressing properties:

- Traces, i.e. finite sequences of events, for safety properties based on trace refinement;
- Stable failures, i.e. traces augmented with a set of events that cannot be executed, for liveness properties and deadlock freedom based stable failures refinement;

- Failures/divergences, i.e. stable failures augmented with traces ending in an infinite loop of internal events, for livelock-freedom based on failures/divergences refinement.

A divergence is a trace for which a process is not blocked and do not show any observable behavior, i.e. it executes internal actions in an infinite loop.
For verification purpose FDR computes all the traces, failures and divergences of processes making verification of infinite state systems impossible without using abstraction techniques.

II.3.4 Formal Verification and Validation of DEVS models

We present in this section the different techniques used in the literature for the verification and validation of DEVS models using formal methods.

Related works have addressed formal analysis of DEVS models. These proposals range from formal model-checking of sub-classes of DEVS, transformation of DEVS into formal methods for verification purposes, generation of traces from DEVS models for testing, or introducing clock constraints to DEVS to conform to some formal method. We present some notable works on this.

II.3.4.1 RTA-DEVS

Rational Time-Advance (RTA) DEVS is a subclass of C-DEVS that is realized by mapping the time advance to a rational number [Saadawi and Wainer 2010]. This work defines a transformation approach to obtain a Timed Automata (TA) [Alur 1999] that is behaviorally equivalent to RTA-DEVS. This restriction imposed on the elapsed time translates into having rational constraints in guards in the transformed TA model and ensures termination of reachability algorithms implemented in UPPAL, as irrational constants in TA guards render reachability analysis undecidable. The authors were able to show the behavioral equivalence of RTA-DEVS and TA by using timed weak bisimulation. Then following the conditions of this bisimulation, they construct a TA model for the basic behavior elements of RTA-DEVS, namely internal and external transitions. Then, they deduce the required constant values on the TA model to complete the bisimulation equivalence. This transformation, while beneficial for modeling the behavioral properties of an atomic DEVS model cannot be effectively applied to a system with many components. Earlier, they had proposed a technique for verification of DEVS models based on Model-checking [Saadawi and Wainer 2009]. The technique is to specify graphically DEVS models using Eclipse-CD + + [Mathias et al. 2010] and transforming these models into timed automata in UPPAAL model-checker [Larsen et al. 1997]. They illustrated their approach by a case study and compared their results using UPPAAL and the results of simulation in CD + +.

II.3.4.2 RT-DEVS

The Real-Time DEVS formalism (RT-DEVS) introduces a time advance function that maps each state to a range with maximum and minimum time values [Hong et al. 1997]. RT-DEVS was used to model a real-time system of train-gate-controller. It introduced an algorithm to build a timed reachability tree to be used for safety analysis. Further work on verifying RT-DEVS has been done using timed automata and UPPAAL [Circirelli et al. 2010], and a transformation from RT-DEVS to UPPAAL is shown. This transformation allows weak synchronization between

components of TA model as RT-DEVS semantics uses weak synchronization. The transformation given did not show formally timed behavior equivalence between RT-DEVS and TA models.

II.3.4.3 DEVS and LTS

Ernesto Posse worked on the development of an alternative theoretical foundation for DEVS [Posse 2008], not on the concept of I/O System within Zeigler's hierarchy of system specification, but based on structural operational semantics focusing on determinism and compositionality properties. The meaning of DEVS models is given in terms of labeled-transition systems. The advantage of this approach is that it allows us to reason about DEVS using existing tools for labeled-transition systems (LTS) and we can compare DEVS to other formalisms described in terms of labeled-transition systems. However, this is done only for Classic DEVS. Moreover, using only LTS would hide the structural properties and functional couplings of a model in the coupled network hierarchy.

II.3.4.4 FD-DEVS

The Finite and Deterministic DEVS (FD-DEVS) [Hwang and Zeigler 2009] is a class of DEVS defined by making the following assumptions: 1) the sets of events and states are finite; 2) the time advance is a mapping from states to non-negative rational numbers; and 3) an external input event can either reschedule or continue processing. The main restriction imposed here is that there can be no use of the time that has elapsed in a state to determine its transition to another state (a characteristic of the general DEVS formalism is that such elapsed time information can be employed in a unrestricted manner.) The network behavior of FD-DEVS is abstracted as a finite-vertex reachability graph, in the context of no restriction on the occurrences of external events. The authors developed an algorithm to do this abstraction. The key idea is that infinitely many instances of elapsed times of components in a FD-DEVS network can be abstracted by time zone equivalence. Based on the time zone abstraction technique, an algorithm to generate a finite-vertex reachability graph was introduced and their completeness and time complexity were investigated. However, an operational framework is not proposed.

II.3.4.5 DEVS and Semantic Composability Theory

The formal theory for semantic composability examines simulation composability using formal definitions and reasoning [Weisel et al. 2005]. Because DEVS and semantic composability theory appear to have certain topics in common, the authors raise the question of their relationship. They noted that a fundamental difference between the two theories is the requirement in composability theory for the computability of models. They show that composability theory and C-DTSS (and thus DTSS, DEVS…) are both sufficiently powerful to express all simulations that can run on a computer. An operational framework is also not proposed.

II.3.4.6 Z-DEVS

In [Traoré 2006], the author presented a method to make DEVS models amenable to formal analysis using the Z method. In this work, he defined Z-DEVS, a formalism that combines the DEVS well-established M&S paradigm with the Z formal paradigm. Specifying simulation models in Z-DEVS makes them accessible to formal analysis. Then, ambiguities and

inconsistency in requirements could be discovered early, when they can be corrected with much less expense than after code has been developed. Also, hidden properties can be revealed, using a theorem proving tool such as Z/EVES. The logic-based M&S framework that he proposed aims at building models of better quality and at increasing the understanding of key concepts such as VV&A, reuse and composability.

II.3.4.7 ϕDEVS

Another interesting work that has linked Z and DEVS is presented in [Trojet et al. 2009]. It has proposed a lightweight transformation approach of DEVS models to Z so that proofs can be performed on the resulting specification. The approach is based on their defined "constraints-based DEVS framework" noted $\phi DEVS$. $\phi DEVS$ allows to capture the behavior of a system using DEVS and its static constraints using predicate logic expressed in Z. This allows checking that the behavior of the system meets the specified constraints or not using Z/EVES theorem prover.

II.3.4.8 TC-DEVS

Time Constrained DEVS (TC-DEVS) is a class of DEVS that expands the DEVS atomic model definition with the introduction of multiple clocks incremented independently of other clocks [Dacharry and Giambiasi 2007]. Classic DEVS atomic models can be seen as having only one clock that keeps track of elapsed time in a state, and is reset on each transition. TC-DEVS also added clock constraints similar to TA (to function as guards on external and internal transitions). However, it allows clock constraints in states as invariants that contain clock differences. TC-DEVS is then transformed to an UPPAAL TA model. The paper however, did not explain a transformation of TC-DEVS state invariants to UPPAAL TA when the model has invariants with clock differences. Earlier, the authors had proposed linking DEVS and Timed Automata, in order to extend the design methodology for control systems, using DEVS to specify the low-level behavior of the control being designed, and Timed Automata to specify the high-level behavior or properties that the control should meet [Dacharry and Giambiasi 2005]. They introduce the concept of simulation relations between DEVS models and Timed Automata in order to verify that the control meets the specification of the concept. This concept is based on an analogous approach generally found in the Timed Automata theory where it is used for formal reasoning about the behavior of Timed Automata. Using this approach, it is possible to augment the design quality, increasing at the same time the understanding of a system by counting on validation and verification techniques consisting of not only simulation, but as well formal methods and model-checking. Again, Timed Automata theory can only expose the behavioral properties of a system.

II.3.4.9 DEVS and Temporal Logic

A method of verification of DEVS models in the environment DEVSim++ has also been previously proposed [Hong and Kim 2004]. The approach is to specify the model in DEVS (operational formalism) and use the temporal logic (TL formalism assertions) to specify the properties and time constraints of the system. The authors use a projection technique (external TLA assertions) to reduce the state space. The lifetimes of the states are not taken into account so the function of play time is not defined at the atomic model, but the temporal logic allows

expressing constraints on sequences of states. The technique used by the authors is very similar to the technique of model-checking using Buchi automata [Büchi 1960].

A transformation method of DEVS models in TLA+ has been earlier proposed [Cristia 2007]. The main conclusion of this work is that DEVS models describing discrete event systems can be easily translated into TLA+ specifications. This would be beneficial for DEVS since it lays the basis for a formal semantics of this powerful modeling language. Having a TLA+ specification of a DEVS model enables for formal verification of the model or to model-check it with the tools already available for TLA+ specifications. However, a generalization of the conversion of DEVS in TLA + has not been studied.

II.3.4.10 DEVS and Reachability Analysis

It has been shown that verification of general DEVS models through reachability analysis is not decidable [Hernandez and Giambiasi 2005]. The authors of this work based their deduction on building a DEVS simulation Turing machine. Since in Turing machines the halting problem is not decidable (i.e. with analysis only, we cannot know in which state a Turing machine would be), they concluded that this is also true for DEVS models: we cannot know if we reach a particular state starting from initial state, and hence reachability analysis for general DEVS is impossible. They argue that reachability analysis maybe possible only for restricted classes of DEVS. This result however was based on introducing state variables into DEVS formalism with infinite number of values.

II.3.4.11 DEVSPecl

DEVSPecL is a specification language for writing DEVS models in BNF format [Hong and Kim 2006]. This language is dedicated to modeling, simulation and analysis of discrete event models and it is based on a verification methodology that had been earlier proposed for the logical analysis of discrete event systems [Kim et al. 2001] without taking into accounts the temporal information and OpenDEVS [Thomas et al. 2006], whose characteristics are the preservation of model information, modeling, object-oriented and type-checking. Note that OpenDEVS is a proposal for standardized exchange format for DEVS that has not been implemented. The main objective of DEVSpecL is to provide a textual language for code generation to multiple target languages and tools for performance analysis by simulation and verification by testing. The testing technique adopted is a combinaison of white-box testing and black-box testing. DEVSpecL is textual and provides a test-based verification technique. At first sight our work and DEVSpecL appears to have same objectives; however there are fundamental differences between the two approaches. DEVSpecL is textual while ours is graphical. DEVSpecL adopts a test-based verification technique while ours offers formal verification techniques at different levels of abstraction.

Our work differs from all of the above. We use visual notations that have equivalent DEVS representation and the semantics are defined using formal methods. Therefore we can use both simulation and formal analysis to ensure the integrity of our models.

II.4 System Design

We present in this section system design with UML and some design patterns as background for our enactment semantics. We also present related works on the enactment of discrete event systems.

II.4.1 Unified Modeling Language

UML [Rumbaugh et al. 2004] is a standard object oriented notation for software systems design independent from specific software development process (e.g., XP, RUP), or technology (e.g., Java, .NET, embedded real-time). UML is customizable to target specific domains like hardware, embedded systems, real time systems, business rules, networks (MARTE, SPT, SysML). UML has several supporting tools from which diagrams can be saved and exchanged in XMI format; these tools can generate code, reverse engineer code, and perform impact analysis, refactoring, and complexity analysis.

UML has the concept of class, structured class and component. Structured class defines Port and Connector and provides means to describe a class as an aggregation of parts. There is no support for flow-oriented communications in UML structured classes. An important fact is that UML makes a clear distinction between structured classes and components. A component is a structured class that represents a modular part of a system that has required or provided interfaces and ports. Components can be assembled together by using assembly connectors to form larger components. Many UML standard stereotypes that apply to component exist: <<subsystem>>, <<process>>, <<service>>, <<specification>>, <<realization>>, <<Implement>>. Subsystem component represents a unit of hierarchical decomposition of a system. A system may have specification and realization elements. A specification component defines a domain of objects by only defining their interfaces. A realization component is a realization (implementation) of a specification component without attributes and methods. Provided interfaces are directly realized by the component itself or by realizing components or publicly provided by a port of the component. Required interfaces are designated by usage dependency from the component itself, or realizing component or required by a public port of the component. Implement component is similar to realization component; it has a dependency with a separate specification component. A process component is a transaction-based component. A service component is a stateless functional component.

In UML, behavioral specification is generally done by using activity diagrams, sequence diagrams, and state machine diagrams. It is known that these diagrams do not have formal semantics and no precise operational semantics is defined to make them executable. For example Activity diagrams semantics are described by informal text. Each existing tool for modeling activity diagrams implements its own operational semantics (which is generally domain specific). Some works addressed the formalization of the semantics of activity diagrams in Petri nets [Staines, 2010].

A more precise subset of UML called Foundational UML (fUML) [OMG 2013] is defined for specification of executable UML models. The subset fUML defines a semantics called base semantics.

II.4.1.1 Formal verification of UML models

It is a known fact that UML specifications are ambiguous and difficult or impossible to analyze formally. The adequate use of OCL constraints helps a lot to avoid ambiguities, but even then tool support is lacking. Some tools with limited capabilities exist for validating OCL and UML specifications [Gogolla et al. 2007].

The static semantics of UML is not completely defined and the dynamic semantics is mostly left to the decision of supporting tools. The implementation freedom in the dynamic semantics is intentional since the goal of UML is to provide a unified framework for various application domains with different needs. This freedom makes the validation problem of UML models dependent form semantic choices of available tools.

Facing the difficulty or impossibility of formal verification of arbitrary UML models, some proposes to limits the elements of models in some ways. Some authors propose to use a subset of UML [OMG 2013]. The problem of this approach is that a small subset may make the formal verification of models feasible or decidable, but at the same time it restricts the expressiveness of the language. In the opposite, a large subset is very expressive in modeling however, it is difficult or impossible to formally verify.

II.4.1.2 Code generation from UML models

Most UML tools allow code generation from UML diagrams. UML class diagrams are particularly used by the tools to generate code to GPLs like Java, C#, C++. The generated code is most of the time incomplete and requires additional coding in implementation phase which make synchronization of that code and the diagram between developers difficult. To allow generation complete code for methods, additional diagrams like activity diagrams and textual specifications are used to complement class diagrams. The way of combining UML diagrams, OCL and action semantics to deliver complete implementation code for a system is not part of UML standard leading to individual solutions that fail to consistently keep the link between the diagrams and the generated code.

II.4.1.3 Systems Modeling Language (SysML)

The Systems Modeling Language (SysML) is a standard graphical modeling language that customizes UML (by the profile mechanism) for the specification, analysis, design, verification and validation of systems including hardware, software, and processes. SysML is more expressive and flexible than UML for systems modeling. SysML reuses a subset of UML (activity diagrams, use-case diagrams, sequence diagrams, state diagrams, and package diagrams), and adapt class diagrams to be block definition diagrams and composite structure diagrams to be internal block definition diagrams. SysML adds also new diagrams (requirement diagrams, parametric diagrams). As UML, SysML is supported by a broad range of UML tools and its diagrams can also be exchanged between tools by using the standard XMI format. The advantages of SysML over UML for systems engineering is that SysML Requirement diagrams can be used to efficiently capture functional, performance, and interface requirements, whereas UML is subject to the limitations of Use Case Diagram to define high-level functional requirements. Also, SysML Parametric diagrams can be used to precisely define performance and quantitative constraints while UML provides no straightforward mechanism to capture this sort of essential performance and quantitative information.

We choose UML design pattern for enactment. Since UML is not completely formal, we use only a subset that is formalizable.

II.4.2 Object-Oriented design patterns

Design patterns in Object-Oriented modeling are documented solutions to some general problems that can be reused to build new models. In this subsection, we present the overviews of two design patterns from [Gamma et al. 1994] that are re used to define the metamodel of our enactment framework.

II.4.2.1 Observer design pattern

It is a behavioral pattern for establishing relationships between objects at runtime such that changes in the state of an object (referred to as subject) trigger some actions in another (the observer). It is defined by the Gang of Four [Gamma et al. 1994] as a pattern that "*define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.*"

Figure 16 shows an overview of the observer pattern. The basic idea is that the *Subject* maintains a list of references to some independent objects called the *Observers*. Whenever there is a change of state in the subject, all its observers must be notified by the invocation of the *update* method of each of them. Each observer (i.e., ConcreteObserver) must implement its update method to implement the corresponding actions to be taken whenever this happens. This pattern is widely used in Graphical User Interface (GUI) programming and it provides the underlying principle for the Model-View-Controller (MVC) architecture [Krasner & Pope 1988] so that all views are automatically updated whenever there is a change of state in the model.

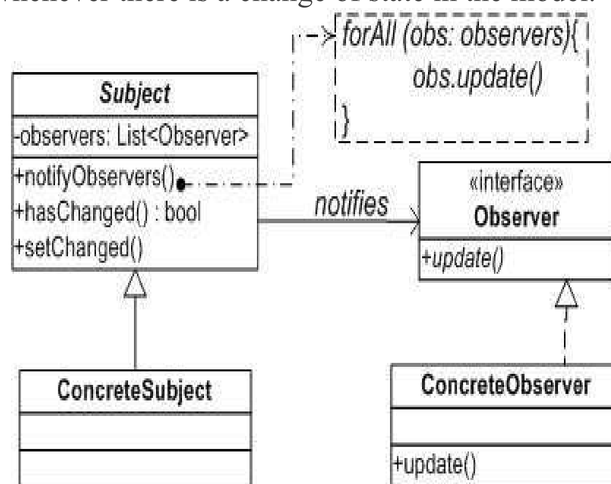

Figure 16. Observer design pattern

II.4.2.2 Command design pattern

The *command* design pattern is shown in Figure 17. A command in this context means a method call. The pattern provides a methodology to encapsulate a command in an object and issue it (the command) in such a way that the requested operation and the requesting object do not have to know each other.

From Figure 17, *Client* is the requesting object while the method *action*() of *Receiver* is the requested operation. *Client* creates the request command and delegates its execution to the *Invoker* which manages a queue of command threads. The invoker identifies the receiver of the request carried by each command in its queue and then executes the command. When its *execute()* method is invoked, the command delivers its request by invoking the appropriate *action()* method. This pattern provides a methodology for asynchronous(i.e., non-blocking) method call, sharing of a method call among multiple objects, saving method calls in a queue so that they are executed when the necessary conditions have been satisfied, etc., it has also been used to decouple clients from server methods in Asynchronous Remote Method invocation (ARMI) [Raje et al. 1997].

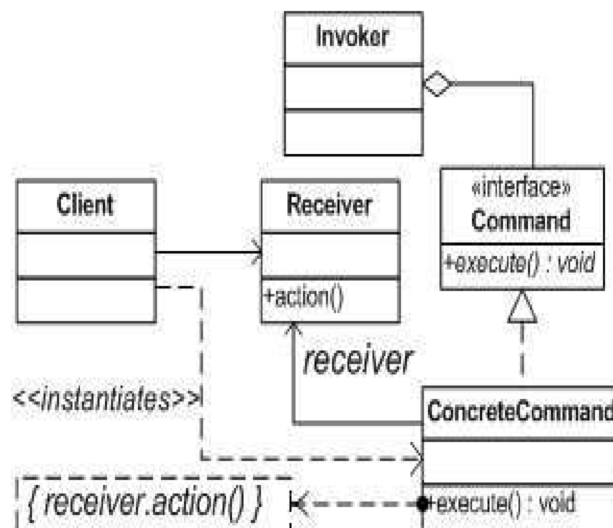

Figure 17. Command design pattern

II.4.3 Enactment of discrete event systems

Enactment refers to the prototyping of a system in real-time i.e. execution of a system using real-world clock.

PROTOB [Baldassari et al. 1989; Baldassari and Bruno 1991] is system development environment that integrates tools, for modeling, prototyping and implementation of distributed systems using an operational software life cycle paradigm. In PROTOB, systems are described with PROT nets, an Object-Oriented formalism that combines high-level features of timed Petri nets, and workflows to model event-driven distributed systems. PROT nets describes a system as consisting of interacting autonomous objects called "actors" where each actor is an instance of a class. The behaviour of a class is described in a Petri nets dialect as consisting of *places* (describing the states) and *transitions* through which places are connected with *arcs*. An active place has a queue of message-carrying *tokens* that are moved from places to places through transitions. Some places are designated for Input/Output operations to allow actors interact with one another. Message passing between actors is achieved by moving tokens between their I/O interfaces. According to the authors, operational program codes can be generated from PROT nets specifications for general purpose languages though it is not clear what the structures of such

codes look like. The similarity between PROTOB approach and the enactment framework presented in this thesis is that system description in both cases are based on some well-defined formalisms - Petri nets in PROTOB and DEVS in our framework. Interestingly, the approach proposed in this paper can arguably accommodate a broader category of DESs based on the fact that the underlying formalism, DEVS, has been proven to provide a common denominator for most DES formalism including Petri nets.

In [Hu & Zeigler 2004], the authors proposed an approach of Model Continuity to Support Software Development for Distributed Robotic Systems based on Modeling-Simulation-Execution methodology [Hu & Zeigler, 2002]. As defined by the authors, Model continuity refers to the ability to use the same model of a system throughout its design phases, provides an effective way to manage this development complexity and maintain consistency of the software. Model continuity is ensured by using the same model in modeling, simulation and execution phases. Real-Time DEVS and Dynamic Structure concepts are used in modeling phase in order to support the modeling of the robots sensors and actuators as activities and dynamic reconfiguration of robots. In the simulation phase, different DEVS simulator implementations (supporting different communications schemes from point to-point socket communication to advanced middleware such as CORBA) are used for the incremental verification of the model. The real-time execution is achieved by mapping the robot specifications into a real hardware execution environment controlled by DEVS real-time execution engine; it is however not clear what is the methodology used in building the said execution engine. The main similarity with this work and the one presented in this thesis is that system description is based on DEVS in both cases. The difference is in the fact that their enactment engine resides in hardware for enactment of robot systems while ours is a software enactment on any suitable computer system.

II.5 Language engineering

II.5.1. Definition of a Language

The general concept of language refers to the cognitive ability to learn and use complex communication systems, or to describe the set of rules that makes up these systems, or the set of elements that can be produced from those rules. All languages rely on the process of relating signs or sounds with particular meanings.

There are broadly three aspects of languages, which include language form, language meaning and language in context. Language semantics is concerned with how meaning is inferred from words and concepts. Pragmatics deals with how meaning is inferred from context. Natural languages are spoken or signed languages, they are distinguished from constructed and formal languages. Our concern in this section is about constructed languages implied in software engineering processes.

II.5.1.1 Language constituents

Formally a language, L is defined by $L = < A, C, S, M_{CA}, M_{AS} >$; with an abstract syntax, A, a concrete syntax, C, a semantic domain S, a mapping function between C and A, M_{CA}, and a semantic mapping function between A and S, M_{AS}, as shown in Figure 18. The abstract syntax is the definition of the conceptual elements, relationships between them, and the well-formed rules. It defines the set of syntactically correct models. The concrete syntax of a language can be textual or graphical notations. The semantics describe the meaning of the models, usually in terms of a mathematical model of computation or semantic domain.

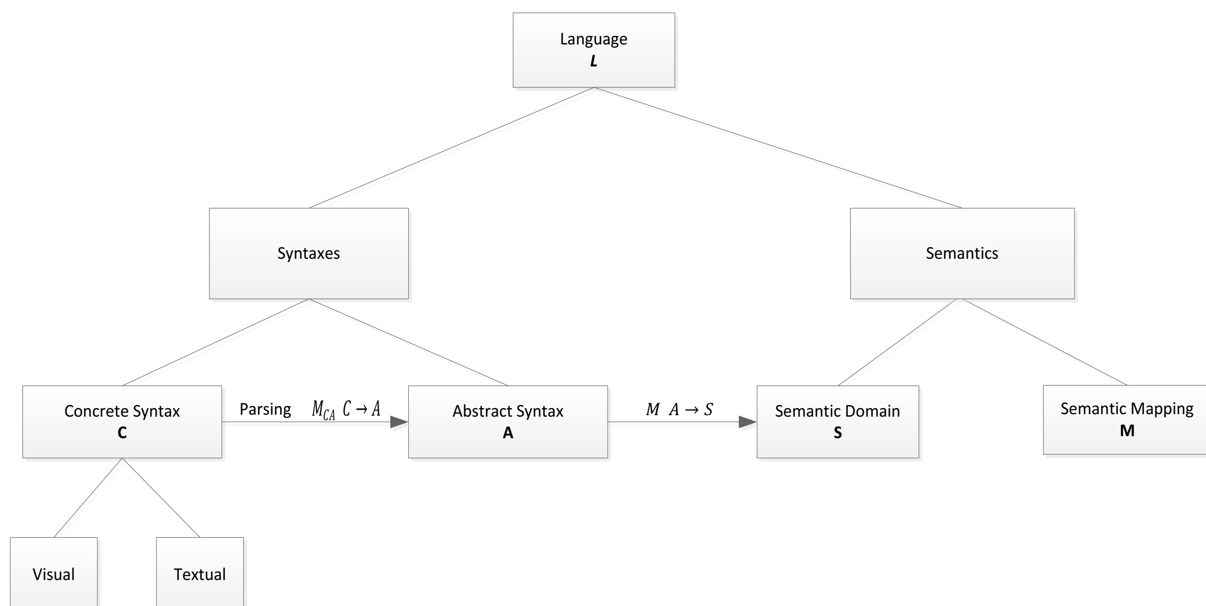

Figure 18. Defining a language

Different abstract syntax definition languages exist in the literature: ADL, MOF, Ecore, BNF, EBNF. Different frameworks have been developed for the definition of the abstract syntax of a

language depending on the language type (graphical or textual) and the abstract syntax definition language used. Examples are: Eclipse TMF (Textual Modeling Framework), Eclipse GMF (Graphical Modeling Framework), GME (Generic Modeling Framework).

II.5.1.2 Semantics

Semantics is defined as a precisely defined mapping of the elements of a language into a precisely defined domain of values. The mapping is termed the semantic mapping. The domain of values is termed the semantic domain. The semantic domain may consist of purely mathematical constructs, such as sets, functions, or algebras, or it may itself be a language, such as B, Object-Z, or even a subset of the source language.

Different styles of semantic exist:

- Axiomatic semantics: characterize a program by a set of satisfied properties by variables. It maps language constructs into logical theories, consisting of mathematical structures together with axioms defining their properties. It uses a declarative approach for description of the properties and their evolutions by constraints and operations. The Hoare's logic allow the specification of this kind of properties in the form {P}i{Q} which means if the logical formulas P (called pre-condition) is true before the execution of the instruction i, then the logical formulas Q (called post-condition) will be true after the execution. This kind of semantics is not operational but allows mathematical proof of programs properties. Axiomatic approaches support general reasoning and a comprehensive expression of language features, but at the cost of using elaborate formalisms for which support tools may not exist.
- Denotation semantics: associate to each expression e of the language, its denotation ⟦e⟧ which is a well-defined mathematical object that represents the computation represented by this expression. It describes in form of functions the effects of a program on a state without saying how this program is executed.
- Translational semantics: defines the semantics of a language by translating it to another language which has a well-defined semantics. This transformation allows the use of tools associated to target language to perform some actions (verification, simulation, animation etc.) on the models of the source language. It is necessary in this case to precise how the semantics of transformed models are interpreted and how results obtained in the target platform are interpreted in the source platform.
- Operational semantics: maps a language into structures of an abstract execution environment. It describes how every valid expression of the language is interpreted in terms of successive steps giving its value. It is composed of rules which describe the effects of the constructs of the language. It describes how a program is executed. An operational semantics allows the description of the evolution of a model during its execution. It allows staying in the same technological platform and manipulating the concepts of the formalism for simulation and animation.

We define for HiLLS, axiomatic and operational semantics by translating its models to languages with well-defined semantics like Z, CSP and DEVS for formal analysis and simulation.

II.5.2 Visual Languages

A visual language describes its concrete syntax as a set of graphical elements and the relations between them. Each graphical element is characterized by its attributes such as the form, the color, the dimensions, the position, etc. A visual language is said to be geometric if the positions of graphical elements are important. If the positions of elements are not important the language is connection based. A hybrid language is geometric and connection based.

For the definition of abstract syntax of a visual language, two approaches exist: the first is based on graph transformations which extends existing techniques for textual languages to graphical languages. The second approach is based on metamodeling. Metamodeling defines the abstract syntax of the language by a metamodel and a set of constraints to define the static semantics. These constraints are expressed in the metamodeling language itself (multiplicities and relationship constraints) or a constraint specification language such as OCL (Object Constraint Language).

The concrete syntax is represented by a graphical metamodel which is conform to the metamodel of graphical elements representations. The mapping between the abstract and the concrete syntax associates to each concept its representation (Figure 19). A constraint specification language such as OCL is usually needed for this mapping. The most used tools such as XMF (eXecutable Metamodeling Facility), GME (Generic Modeling Environment), Kermeta (Triskel Metamodeling Kernel) and Eclipse GMF (Graphical Modeling Framework) have their own metamodeling languages (Xcore, MOF, Ecore etc.) and graphical elements specification languages (GEF for GMF).

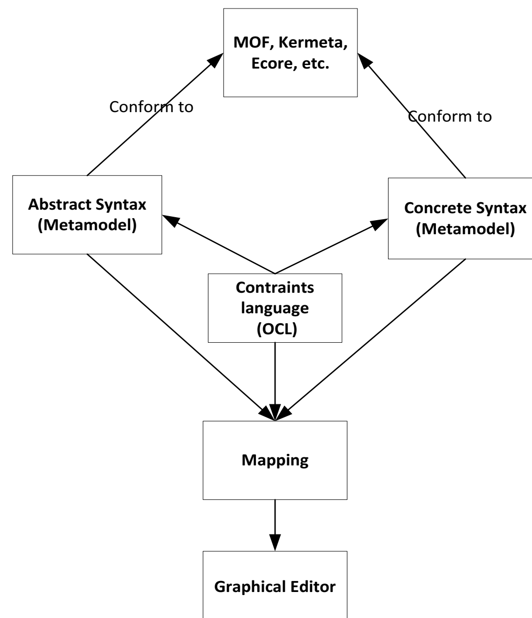

Figure 19. Mapping between abstract and concrete syntaxes

Many research efforts in the literature towards making visual modeling languages MDE-compliant have led to the provision of formal semantics for such languages and have shown that the use of mathematical tools and mechanisms is the most pragmatic approach to adopt. In the late 90's, [Spivey 1998] demonstrated that mathematical notations provide an essential means to precisely describe the must-have properties of a system, provide a reliable source of reference for

investigating such properties and requirements and promote a common understanding among the stakeholders in the design, development, use and optimization of the system.

[Lano and Biccaregui 1999] pointed out that there are certain ambiguous cases in the UML semantics that inhibit model transformation efforts involving UML models. They proposed a formal semantics for the UML using structured theories in a simple temporal logic to solve the problem. [Szlenk 2006] holds a similar view about the UML while acknowledging its universality in dealing with the intricacies of software systems via modeling. The paper provided a mathematical foundation based on lists and functions to express the conceptual UML class diagrams so as to remove the semantic ambiguities that make model verifications and transformations more difficult than the engineers could imagine.

In [Barret et.al. 2011], the lack of precise and common reference points for semantic definitions was identified as a great challenge to the MDE-based exploration of the potentials of many visual modeling languages. The authors maintained that the semantics of UML metamodel described with a mixture of the Object Constraint Language (OCL) and natural language is not amenable to rigorous formal analyses. By evaluating the various methods of defining semantics of modeling languages, they maintain that the use of some mathematical frameworks with well-defined meanings will help in automating the generation and analysis of supporting tools for the language.

II.5.3 Integrating Heterogeneous languages

II.5.3.1 Levels of Integration

In the literature, different notions designating the use of different language in the specification of a system have been discussed. Some notions have similar meanings and similar techniques to deal with syntactic and semantic relationships between the involved languages. Some techniques address only the transformation between languages (inter-model communication or cooperation) and neglect the syntactical integration level.

As defined earlier, each language has abstract syntax, concrete syntax and semantic domains. Integrating heterogeneous languages implies different cases:

- Common syntax, common semantics (eg., composing different models written in Simstudio)
- Common syntax, different semantics (eg., composing Statecharts and CSP models written in Java)
- Different syntaxes, common semantics (eg., composing different DEVS models written in different DEVS implementations like DEVSJAVA, PyDEVS, DEVS-Scheme, DEVS++ etc.)
- Different syntaxes, different semantics (eg., composing Petri Net and DEVS models)

[Groβe-Rhode 2004a] classified integration into the following levels:

- Syntactic integration: concerned by the integration of languages rather than integrating models. It defines syntactic transformations between languages.
- Methodological integration: unifying development activities of different methods.

- Ontologic integration: concerned by the explicit representation of concepts and their relationships.
- Formal semantic integration: defining the semantic relationships between the participant languages or models or defining a common semantic domain.

All of the different integration levels can be used together in a complete integration method as declared by the author.

II.5.3.2 Semantics of Integration

Different approaches to semantics of integration of heterogeneous specifications exist in the literature:

- Common semantic domain called system model: In this approach the different modeling languages are formalized into a common (formally defined) modeling notation called system model which serves as the semantic basis and for analyzing consistency of models. This approach has been adopted in many efforts towards the formalization of UML semantics [Lano 2009], [Groβe-Rhode 2004b].
- Model transformation approach: In this approach, model transformations and consistency issues are typically dealt with at the syntactic level of the modeling notation. [Boronat et al. 2009], [De Lara and Vangheluwe 2002]. All the models are translated to a common formalism syntax which is used to carry investigation about the system [Vangheluwe 2004].
- Multiformalism and Heterogeneous semantics: In this approach different models specified in different modeling languages are interrelated and executed as co-simulation [Ptolemy II], [Mosterman and Vangheluwe 2004], [Patel and Shukla 2004].

[Groβe-Rhode 2004b] discusses the semantic integration of heterogeneous specifications using the approach of a common reference model and the mathematical theory of transformation systems. The approach has applied to semantics of the integration of UML specifications (ex. Integration of class diagrams and state diagrams). [Lano 2009] applied similar approaches to provide semantics to UML specifications.

In [Boronat et al. 2009], a multimodeling language is defined of as a set of sublanguages and correct model transformations between some of the sublanguages. The abstract syntax of each language is defined as a Metamodel and associated OCL constraints in MOF (Meta-Object Facility). In their approach, the semantics of a multimodeling language is given in term of the semantics of it sublanguages semantics in the mathematical theory of institutions and connections between them. The purpose of the connections is to ensure semantic correctness of model transformations. A prerequisite of this approach is the use of the same metalanguage to define the abstract syntax of sublanguages.

II.5.3.2.1 Ptolemy II

[Patel and Shukla 2004] proposes an extension to SystemC [Bhasker 2002] which is basically based on a discrete-event model of computation for system-level modeling and simulation. It has been extended by adding three different models of computations for heterogeneous modeling and simulation of multi-domain hardware/software and embedded systems. The added models of

computation include CSP (Communicating Sequential Processes), SDF (Synchronous Data Flow), and FSM (Finite State Machine). SystemC extensions use similar principles in Ptolemy [Ptolemaeus 2014] which basically integrates the notion of variety of models of computation for different domains modeling and co-simulation (where all components are necessarily built in java, which is a low level approach). Ptolemy II integrates four distinct and complementary classes of syntaxes for multi-domain modeling: block diagrams (from UML), bubble-and-arc diagrams, imperative programs, and arithmetic expressions. An agreed common denominator to all these syntaxes is defined and is composed of a common type system and expression language. Block diagrams are used to express concurrent compositions of communicating components; bubble-and-arc diagrams are used to express sequencing of states or modes; imperative programs are used to express algorithms; and arithmetic expressions are used to express functional numeric computation. Ptolemy II also integrates a number of semantic domains that are operational in nature. Particular interoperability mechanisms are used to make the different execution semantics communicate and exchange data [Tripakis et al. 2013]. Ptolemy II offers a coherent approach to the integration of the heterogeneous models but the target purpose is simulation and some part of the specification use low level routines.

II.5.3.2.2 AtoM3

[De Lara and Vangheluwe 2002] proposes a Tool for Multi-formalism and Meta-Modelling (AToM^3) as a multi-formalism visual modelling environment. In the environment, a metamodel is created for each language used in the development process. Model transformation is defined between the different languages. The tool is used to describe formalisms commonly used in the simulation of dynamic systems, as well as to generate custom tools to process models expressed in the corresponding formalism. Metamodels of the following formalisms are available in the environment: Entity-Relationship, GPSS, Deterministic Finite state Automata, Non-Deterministic Finite state Automata, Petri Nets, Data Flow Diagrams and Structure Charts. These metamodels are the basis for the automated multi-paradigm modeling [Mosterman and Vangheluwe 2004]. In [Feng et al.2007], the authors use Dcharts [Feng] for system behavior specification and translate it for simulation using the simulation engine of AToM3 and model checking using FDR2. The mutli-formalism approach in AToM3 is based on metamodeling and model transformation to establish cooperation between the different formalisms. This approach doesn't address the integration of languages at syntactic level.

II.5.3.2.3 ForSyDe

ForSyDe (Formal System Design) is a system-level specification methodology [Sander and Jantsch 2004], [ForSyde 2014] implemented on top of the functional language Haskell. Initially based on a synchronous model of computation, it offers also the multi-paradigm modeling capability. The purposes of ForSyde are code synthesis (by using a refinement technique to generate implementation code to VHDL or C/C++ based on design transformation rules) and simulation of System on Chip, Hardware and Software systems. A limitation of this approach is the lack of object-oriented and component-based modeling features.

II.5.3.2.4 ModHel'X

ModHel'X [Hardebolle and Boulanger 2007] is a framework for heterogeneous modeling implemented in EMF by providing an extensible set of models of computation, multi-view

modeling and semantic adaptation mechanisms. ModeHel'X also has a simulation-based multiformalism approach like Ptolemy [Ptolemaeus 2004] and SystemC [Patel and Shukla 2004].

II.5.3.2.5 MOOSE

[Fishwick 1995] introduces Multimodeling as the process of creating multimodels (or hybrid models) by heterogeneous coupling of models of different types. Each language can completely specify a model (an independent module) structure and behavior. The purpose of this approach called MOOSE (Multimodeling Object Oriented Simulation Environment) mainly focuses on how the models of different types can communicate in a simulation. Later, an XML-based environment called RUBE is created to support the Multimodeling and simulation process with visualization and user interaction capabilities [Fishwick et al. 2003]. RUBE is composed of MXL (Multimodeling Exchange Language) and DXL (Dynamic Exchange Language). The methodology consists in transforming the heterogeneous models created in MXL into homogeneous DXL models.

II.5.3.2.6 DEVS/RAP KIB

[Sarjoughian 2005] proposed a multi-formalism modeling composability framework using the concept of Knowledge Interchange Broker for composing disparate modeling formalisms. The approach is applied to the composition of models of Discrete-Event System Specification and Reactive Action Planning formalisms (DEVS/RAP KIB). The composition used the model of a vehicle specified using the DEVS formalism and a model of the control agent of the vehicle specified using the Reactive Action Planning Formalism. The author advocated the composition of modeling formalisms instead of composing models but the syntactic level of the composition is not addressed. The proposed approach to multi-formalism composition is based on characterizing how two formalisms can interact with one another via Input and output mappings. This approach to multi-formalism composability is more oriented to how models communicate in execution phase and is not concerned by solving the problem at abstract syntax level of the composed formalisms. The approach is influenced by the purpose of the languages used which is simulation execution and analysis. The approach is similar approaches to co-simulation and interoperability approaches with the appropriate transformations of exchanged messages between models of different formalisms.

II.5.3.3 Integration beyond the operational level

Some frameworks use hierarchical composition of heterogeneous models e.g. ModHel'X, Ptolemy II, SystemC, RUBE. Some frameworks require models at the same level of the composition to use a unique model of computation. This is the case of ModHel'X. The execution of these hierarchical models is sequential in most of these frameworks. For example ModHel'X selects one component to observe at each step.

The look of all these approaches in detail, shows that each of them uses one or more of the following modelling approach:

- Component-based modeling formalism
- State-based modeling language
- Process Networks

- Event graphs
- Synchronous reactive modeling
- Finite-state machine
- Discrete-event modeling formalism
- Continuous systems modeling
- Data flow diagrams (synchronous data flow, dynamic data flow)
- Functional modeling language

Some of these modeling approaches are very similar. For example Data flow diagrams and Process Networks are all used for the specification of concurrent systems in Ptolemy II. Finite State machine, discrete event and synchronous reactive modeling are used for specifying behavior. The formalisms can be classified as structural, behavioral, temporal or functional specification formalism. While they address the integration problem at operational level (interoperability of different Models of computation), we propose to build a language as a coherent whole by integrating syntactically and semantically languages that complement each other and have strong basis and high expressiveness (OO paradigm, Object-Z, DEVS) and using different semantic domains (DEVS, Z, CSP) for different purposes (simulation, theorem proving, model checking). In this manner, HiLLS address structure, static semantics, behavior, and functional specification of systems. We are confident that the defined language is expressive enough to handle the class of systems addressed by these frameworks because the language is rooted in DEVS-based system theoretic concepts and DEVS has been proven to be the common denominator of discrete event formalisms [Vangheluwe 2004] and able to approximate continuous systems [Zeigler et al. 2000]. Ptolemy II also uses a discrete event formalism closely related to DEVS but the integration is not done at syntactic level.

II.5.4 Language integration techniques

Different mechanisms for integrating heterogeneous languages exist in the literature: embedding, aggregation, inheritance.

II.5.4.1 Language embedding

The embedding of heterogeneous languages defines a new language with a single abstract syntax tree. The abstract syntax tree of the new language consist of the elements of the participants languages.

II.5.4.2 Language aggregation

This mechanism aggregates a set of languages for specifying different views of a system. Aggregation of heterogeneous languages results in separate abstract syntax trees for each language. It establishes a relationship between languages in terms of references. Aggregation is generally used to combine behavioral and structural specification languages.

II.5.4.3 Language inheritance

This mechanism refines or extends an existing language by inheriting from existing concepts in language. It reuses existing concepts of the base language and extends it by new concepts.

II.5.4.4 Extension

A language L extends another base language B if L contains the concepts of B and additional concepts. The new concepts introduced by L may specialize (extend or restrict) concepts in B. A given language may define its proper extension mechanisms so that the resultant language can be compliant the standard of the base language and supported by related tools. For example UML has particular extension mechanisms.

II.5.4.5 Model Composition

Several techniques and tools for integrating metamodels, usually referred to as metamodel composition in the literature have been proposed. We provide brief descriptions of metamodel merge, interfacing and refinement , described by [Emerson and Sztipanovits 2006] and used in the MetaGME environment [Ledeczi et al. 2001]. We also discuss language extension and restriction proposed by [Erdweg et al. 2012]:

- *Metamodel merge*: The metamodel merge is used to integrate two metamodels that share some common abstractions of real world entities into a unified whole. It is synonymous to the MOF's Package Merge technique that recursively take the union of model elements matched by name and metatype in the two source packages with two exceptions: 1) Metamodel merge occurs at class level instead of package level and 2) common concepts do not necessarily have to match by name in metamodel merge. Once matching classes are identified, the two classes cease to exist but merge into a new class in the integrated metamodel; the new class encompasses all attributes and associations of the source classes.
- *Metamodel extension and restriction:* Metamodel extension is the mechanism of adding to the vocabulary of an existing language. There are two input metamodels: the base metamodel representing an independent and stand-alone language and extension which captures a set of concepts (that do not necessarily qualify to be an independent language) to be added to the base metamodel's vocabulary. One unique characteristic of this technique according to [Erdweg et al. 2012] is that it is a conservative integration, i.e. the participating language fragments must be reused as- is. Object-Orientation's *inheritance* is used as a means of realizing metamodel extension such that the *extension* class simply inherits the *base* class and provide additional concepts by means of attributes and/or associations.Metamodel restriction on the other hand is the opposite of extension. It involves the *restriction* of the use of certain concepts in the vocabulary of a language. According to [Erdweg et al. 2012], this may be achieved by specifying rules that rejects any model that contains the restricted constructs during validation.
- *Metamodel Interfacing*: Metamodel interfacing is employed to combine two metamodels describing the vocabularies of two distinct but related domains in order to explore the relationships between the two domains. It's implementation requires the creation of new classes and relations (that do not necessarily belong to either of the two source metamodels) which serve as the interface between the distinct metamodels through associations.
- *Metamodel refinement:* Class refinement is used to establish relationships between closely related (or in fact, same concepts) expressed at different levels of abstraction in two independent metamodels. Specifically, a hierarchical containment relationship is created between the two metamodel with the more abstract metamodel fragments as the container(s) of the more detailed descriptions provided by the other.

II.5.5 Integrated languages for system specification

We present here related work on integrated languages for system specification. We concentrate major works that are important for our work.

II.5.5.1 DEVS-based Real-Time System Design

[Kim et al. 2001] described real-time system design as a process that involves iterations of modeling, logical/behavioral analysis and simulation for performance evaluations until quality assurance properties are proven before finally enacting the implementation of the candidate system. The authors are of the opinion that seamless transitions between the development processes would be restrained if they were treated with different models. They proposed a unifying design methodology based on DEVS formalism to specify models of different phases of discrete-event systems with a common semantics. While the DEVS formalism itself was used to model the system in a general sense and to treat the simulation phase of the development, two extensions of the formalism, Communicating DEVS [Lee 2013] and Real-Time DEVS [Hong et al. 1997], were employed to handle the specific issues of the analysis and implementation phases respectively. While the framework suggests a sound integration of the system development processes, it would address a considerably smaller audience than it would if it were built behind some universal visual notations to enhance communication among experts outside the DEVS community. Also, its applications are limited to static-structured systems as no case of systems' structural dynamics is considered.

II.5.5.2 Clepsydra

Clepsydra Methodology [Ciacciac et al. 1995] promotes use of different languages for requirements and design specifications because the use of a unique language for both phases does not adequately fit all the required purposes [Hoare et al. 1990]. It combines the Z language for requirements specification and Larch language for design specifications. The Z language is chosen for its simpler semantics and its logical and abstract specifications. The Larch language is adopted by the authors because of its peculiar structure divided in LSL (Larch Shared Language) and LIL (Larch Interface Language) sections. A relationship is defined between design phases. Requirements and design specifications can be formally analyzed by using available tools for Z and Larch.

II.5.5.3 UML-B

Motivated by the need to enhance the use of formal methods in the industry, a team of researchers at the Southampton University developed the UML-B [Snook et al. 2004] [Snook & Butler 2008] to provide an environment for a refinement-based object-oriented behavioral modeling of complex systems. While UML is highly communicable and universal for object-oriented modeling, Event-B is known for its preciseness and amenability to rigorous formal reasoning with system models. Essentially, UML-B is a UML profile that provides visual concrete notations for B's modeling constructs to enhance the communication of systems and verification techniques among industrial partners. The semantics of the language is given in B to take full benefits of its existing tools for formal verifications. However, the language cannot be

extended to include simulation and/or prototyping since the underlying language, Event-B has no time base, an essential requirement of simulation and prototyping protocols.

II.5.5.4 ModelicaML

ModelicaML [Shamai et al. 2009]; [Pop et al. 2007] presents yet another innovative way to provide support for the specification, analysis and simulation of systems' functional requirements, structure and dynamic behavior within one platform. This was done by combining the UML/SysML and Modelica [Modelica 2012]. Modelica is an equation-based object-oriented modeling language (with sound formal semantics) to specify and simulate the dynamic behaviors of discrete- and continuous-time systems. The SysML on the other hand is a UML profile used for the specification, analysis, design, and verification of systems. [OMG 2012]. ModelicaML extends the profile with additional constructs that accommodate the specific artifacts of Modelica. The unification takes advantage of the communicability of the SysML to communicate Modelica models more easily with domain experts. However, [Shamai et al. 2009] also pointed out that ModelicaML takes benefit of the descriptive power of the graphical constructs of SysML at the expense of the loosely defined semantics inherited from the UML while they intend to leverage this drawback with the precise semantics of Modelica. This still brings us back to the not-yet-answered question: to what extent is the mapping from UML/SysML to Modelica when the source language is devoid of a generally accepted formal semantics? The application is also limited to static-structured systems.

II.5.5.5 µSZ

[Büssow et al. 1998] propose the µSZ language which results from the combination of several languages. µSZ is a language of specification and checking of reactive systems based on the combination of Z [Spivey 1992], flow diagrams and Statecharts [Harel 1998]. Statecharts is one of the most popular formalisms of specification of the reactive systems because of the availability of tools like STATEMATE [Harel et al.1990] supporting its specifications. The language uses Z notation to specify the functional part of the system whereas the Statecharts and the flow diagrams are used respectively to specify its dynamic and structural aspects. Several works were proposed for the formalization of the semantics of the statecharts which constitutes the heart of the notation µSZ. A formalization of µSZ by using a metamodel and Object-Z as metalanguage has been proposed by Geisler [Geisler et al. 2000]. This work shows that the Object-Z language can be used in the field of modeling and simulation.

II.5.5.6 MontiArcAutomaton

The MontiArcAutomaton Modeling Framework [Look et al.2013] [Ringert et al. 2013]integrates six independently developed modeling languages to model robotics applications: a component & connector architecture description language called MontiArc that uses the notion of components and connections between components via typed ports, automata that models the behavior of atomic components, I/O tables for specifying input and output relations, class diagrams for modeling the system's static structure, OCL for constraints specification, and a Java DSL for specifying guards on transitions. The framework is designed for the complete modeling of structural and behavioral characteristics of cyber-physical systems using the integrated languages. The goals of the framework include analysis, platform independent development, problem specific modeling, support for reuse and code generation to different target platforms like Java

and Phyton. The analysis of MontiArcAumaton models is done by generating code to Mona System [Henriksen et al. 1996] which have simulation and formal analysis (safety and equivalence checking) capabilities based on monadic second-order logic. All the languages used are implemented as languages in form of context-free grammars. The integration of the languages is done by aggregating the architecture description language and UML/P class diagrams, extending MonticArc by inheritance and embedding the Java DSL and the OCL language in the automata language using the Eclipse-based compositional Framework Monticore [Krahn et al. 2010] which offers embedding, aggregation, and inheritance integration mechanisms of heterogeneous languages. The framework used supports only the integration of textual languages.

Ultimately, most efforts towards unifying modeling constructs have yielded some highly communicable front-ends for some abstract and not-quite-friendly but precise and formally sound modeling formalisms. This has helped to make the benefits of these precise languages available to a larger audience. However, they are not immunized against the threat posed by the imprecise semantics of the graphical constructs. They also, in most cases, pay little attention to the verification of the correctness and completeness of specified models with respect to the language's syntax. Ensuring the quality of a system specification starts from making sure the model itself is a faithful instance of the language. It is when this is achieved that the dependability attributes of the specified system can be effectively investigated.

II.6 Conclusion

We presented in this chapter, the field of modeling and simulation in general and DEVS-based modeling and simulation framework in particular. DEVS has been shown to be a common denominator for modeling and simulation of discrete systems [Vangheluwe 2004]. We presented the existing DEVS implementations and discussed problems of communicability of their models and collaboration between experts. We pointed out that most DEVS implementations do not provide integrated means of verification and validation. Verification is generally performed during simulation execution which cannot cover all the scenarios and can leave some design errors undetected. There is a need to complement simulation with additional verification and validation techniques such as model checking and theorem proving. To make the formal analysis possible, we integrate DEVS and other languages, such as Object-Z, to allow the modeling of complementary views of the system in the modeling phase and make the models adaptable to different analysis methodologies at the verification and validation stage.

In defining the semantics formally, we realize that one formal method might not be suitable to fully capture all aspects of a system. While most of the already published works try to translate DEVS into one formal method (thereby focusing on only one aspect of a system), we make use of several formal methods to capture these different aspects depending on the power of the method chosen. Most of the works described above use a subset of DEVS by creating assumptions which could limit their implementation. However, we create a refinement of DEVS. We combine software engineering and system theoretic modeling views into our formalism. We also relax the assumption of finite state space. To solve the infinite state space problem, we create partitions of the infinite state space into classes of equivalence states (which we call configurations) using a finite set of variables.

The objective of HiLLS is not to avoid heterogeneity but integrate them in coherent manner at different levels. The differences of HiLLS with multiformalism approaches and integrated languages presented are that it integrates:

- A graphical concrete syntax that ease the modeling process and communicability of models. The graphical syntax uses similar notations to UML concrete syntax for classes and relationships. Some of the frameworks have graphical notation but the set of graphical element is too large (e.g. Ptolemy II).
- A formal object-oriented specification language based on predicate logic namely Object-Z. For accessibility of formal methods to non-expert users, we use Object-Z because it is based on the widely known Z notation and because predicate logic is accessible to computer scientists more than other logics.
- System theoretic concepts for formal component-based modeling that guarantees closure under coupling property. This is not proven for many other approaches.
- Different semantic domains for different purposes. The models for simulation and formal analysis are extracted from the HiLLS specifications.
    - Simulation execution is parallel. HiLLS can benefit from DEVS-PADS implementations because of the mapping to DEVS.
    - Model checking and theorem proving are possible
- A dynamic structure modeling approach that ease to modeling of such systems graphically and make them amenable to different kind of analysis techniques.
- Enactment semantics for real-time execution of systems.

The next chapter presents HiLLS and how it integrates concepts from different languages and formalisms.

III. The High Level Language for Systems Specification

III.1 Introduction

In general, the definition of a model specification language consists of syntax, semantics domain(s), syntax mapping and semantic mapping(s). The syntax is further divided into two: abstract syntax and concrete syntaxes. While the abstract syntax defines the set of well-formed models that can be specified with the language, concrete syntax describes the concrete notations we have chosen to represent the entities and relationships defined in the abstract syntax. The semantics of the language is the precise and detailed meanings of its concrete modeling constructs while semantic domain is the context from which such meanings are derived. Following the same paradigm of language definition, we describe the components of HiLLS in the tuple:

$HiLLS = \langle A, C, M_{AC}, \{S_i\}, \{M_{AS_i}\}\rangle$ such that:

- A is the abstract syntax
- C is the concrete syntax
- M_{AC} syntactic mapping between the abstract and concrete syntaxes
- S is the family of semantic domains of the language, i.e. *S = {Simulation, Logic, Enactment}*

For every semantic domain, s in S, there is a mapping function, m_{As} of the abstract syntax A to s.

Most language definitions provide one semantic domain, we define a family of semantic domains and semantic mappings for HiLLS to capture the multiple use cases mentioned earlier. Figure 18 illustrates the relationships between the components. We discuss the various components in details in the subsequent sections.

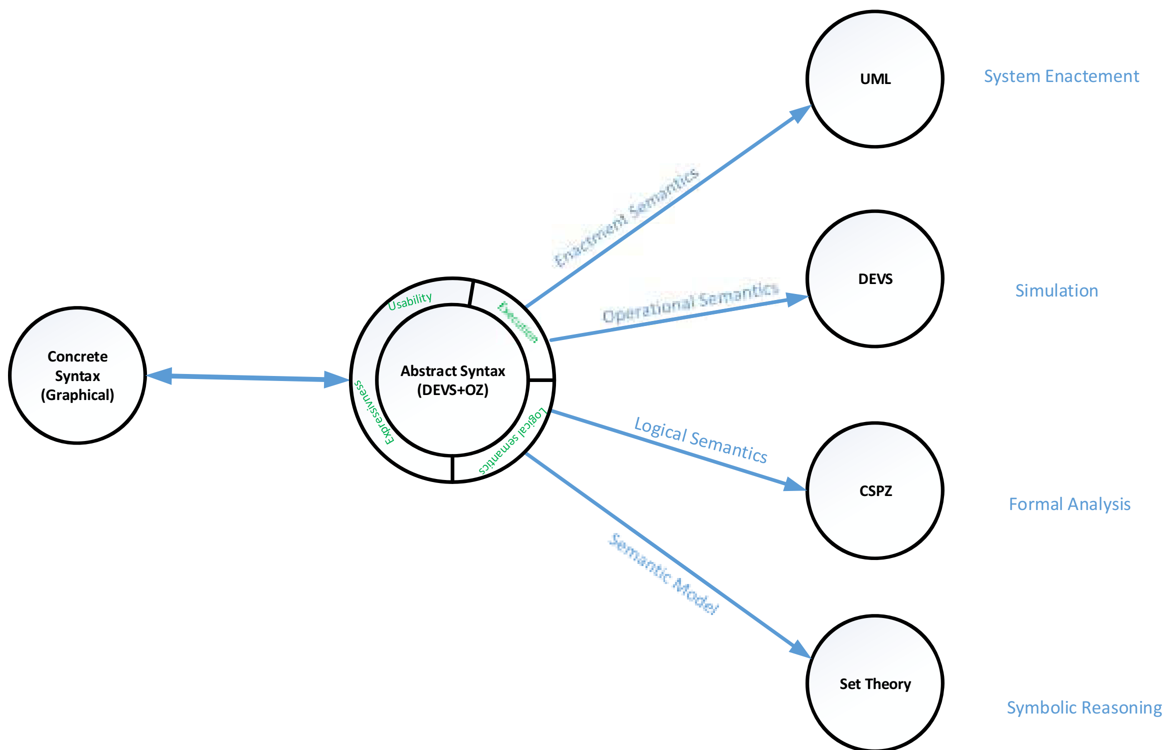

Figure 20. HiLLS definition elements

This chapter presents HiLLS syntaxes and set-theoretic semantics. It discusses how the abstract syntax of the language is built using the abstract syntaxes of constituent languages.

III.2 Informal Presentation of HiLLS

In this section we present a modeling example to introduce HiLLS in system modeling. We consider the modeling of simple traffic lights. HiLLS allow system modeling as extended classes with additional compartment for the discrete event behavior and two boxes at the left and the right containing input and output ports declarations respectively. The notation used for a system class is similar to that of UML class. HiLLS uses the concepts of inheritance and composition between systems with similar semantics to inheritance and composition in UML.

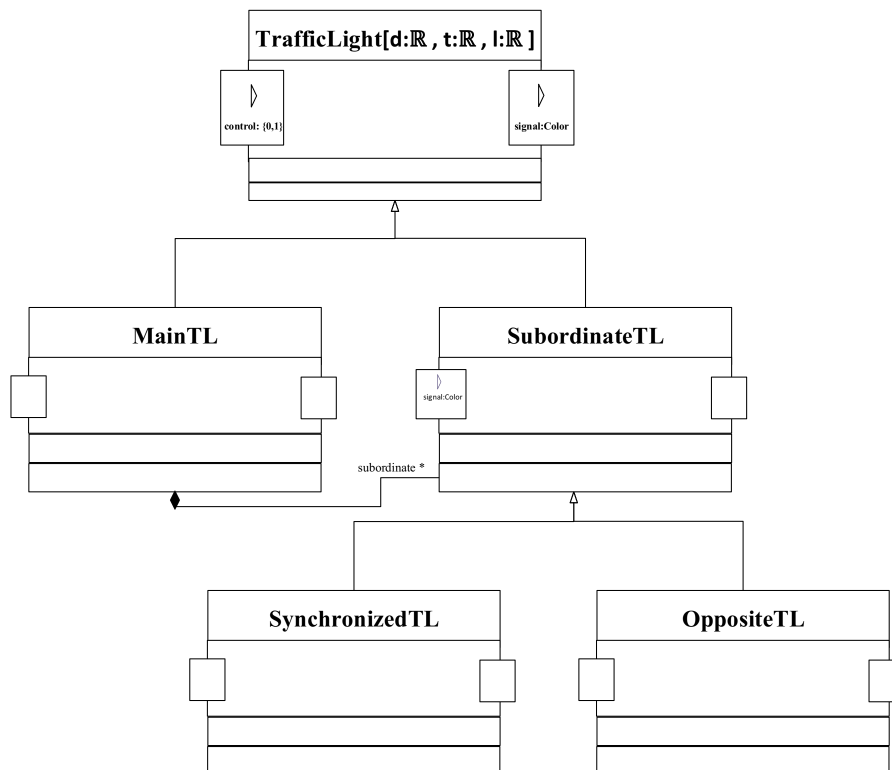

Figure 21. Traffic Light and subordinate

Traffic lights allow controlling circulation at crossroads. In Figure 21, we have an abstract traffic light class $TrafficLight[d{:}\,\mathbb{R}, t{:}\,\mathbb{R}, k{:}\,\mathbb{R}]$with three parameters that define the durations of its configurations and an output port *Signal* for sending the color of the light. We have a main traffic light *MainTL* and a subordinate traffic light *SubordinateTL* classes that inherit form $TrafficLight[d{:}\,\mathbb{R}, t{:}\,\mathbb{R}, k{:}\,\mathbb{R}]$. The *SubordinateTL* has two subclasses: *SynchronizedTL* and *OppositeTL*.

The various configurations (corresponding to the states) of the traffic light model are: *move*, *brake* and *stop.* These configurations are finite i.e., their times advances (duration) are finite (they are represented as rectangle with compartments). The duration of *move*, *brake* and *stop* are respectively determined by the value *d*, *t* and *k*. The behaviour of *MainTL* is described in Figure 22. MainTL is continuously and successively making internal transition from *move* to *brake*,

from *brake* to *stop* and from *stop* to *move* by sending respectively *Yellow*, *Red* and *Green* as output on port *Signal*. Note that the internal transitions are represented with labelled continuous lines between the configurations. The label represents here the output.

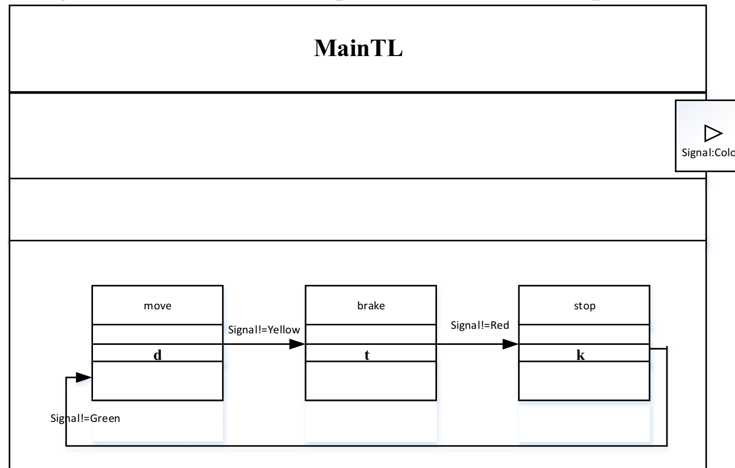

Figure 22. Traffic Light

The output of the traffic light informs the environment (road users) with the colors of the light displayed as outputs.

The SynchronizedTL and OppositeTL have an input port of type color. It has the following configurations: *move*, *toBrake*, *brake*, *toStop*, *stop* and *toMove*. *move*, *brake*, and *stop* are infinite configurations i.e., their time advances are infinite (note that they are also represented as rectangle with compartments like finite configurations but with additional vertical line in the right). *tomove*, *tobrake* and *tostop* are transient configurations i.e., their time advances are equal to zero (note that there are represented as circle). *The Synchronized TL* always goes to the configuration corresponding to the color of the received input. Precisely, it makes external transition to *tomove*, *tobrake* and *tostop* for the reception of *Green*, *Yellow* and *Red* respectively before making instantaneous internal transition to *move*, *brake*, and *stop*. The behavior of the SynchronizedTL is presented in Figure 23.

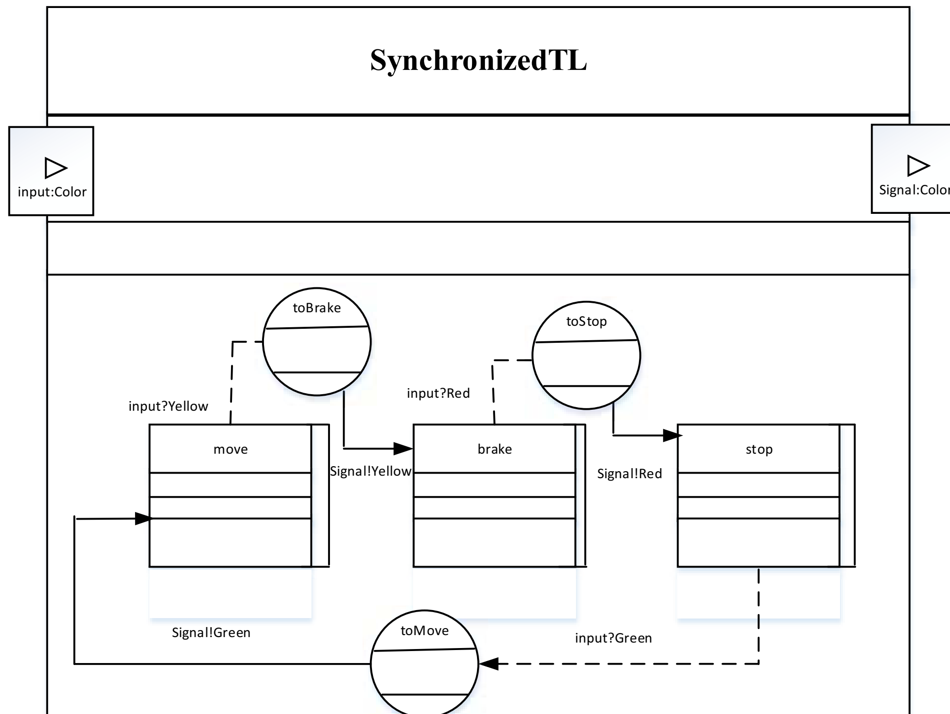

Figure 23. Behavior of SynchronizedTL

The behavior of the OppositeTL is presented in Figure 24.

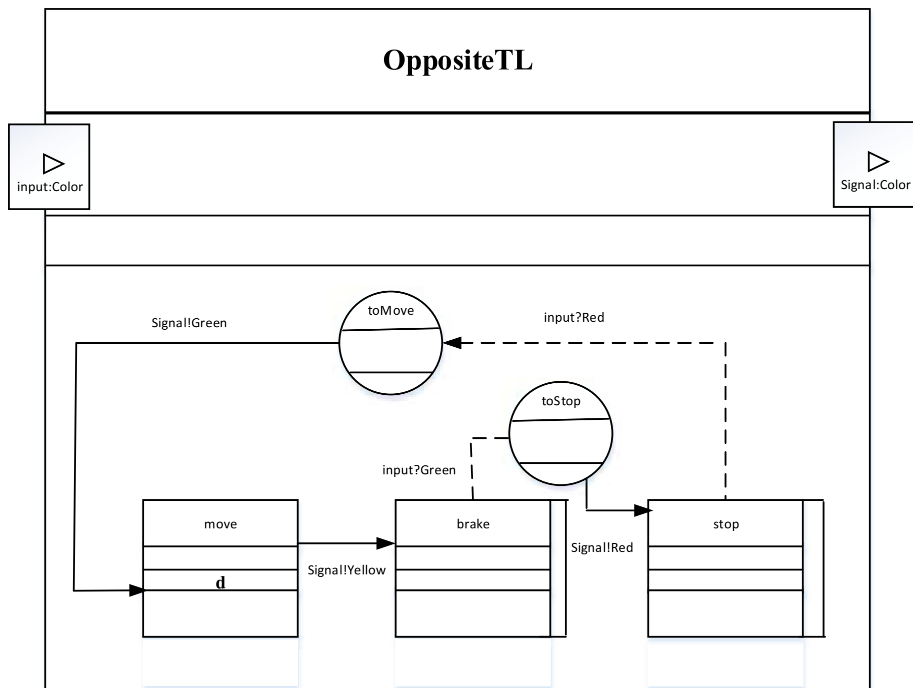

Figure 24. Behavior of OppositeTL

Traffic light system at a crossroad is composed of a *MainTL*, a *SyncronizedTL* and two *OppositeTL* as shown in Figure 25. The predicates shown in the schema box inside the system class specify the couplings between the main traffic light and the subordinate traffic lights. For example the coupling predicate *synchronized.input == main.signal* means that there an internal coupling between the output port *signal* of *main* and the input port *input* of *synchronized.* Note that we did not show here how the outputs of the subordinate traffic lights are used. To complete the specific additional components like Display units (without output port) can be connected to the subordinate TLs to receive and display their outputs.

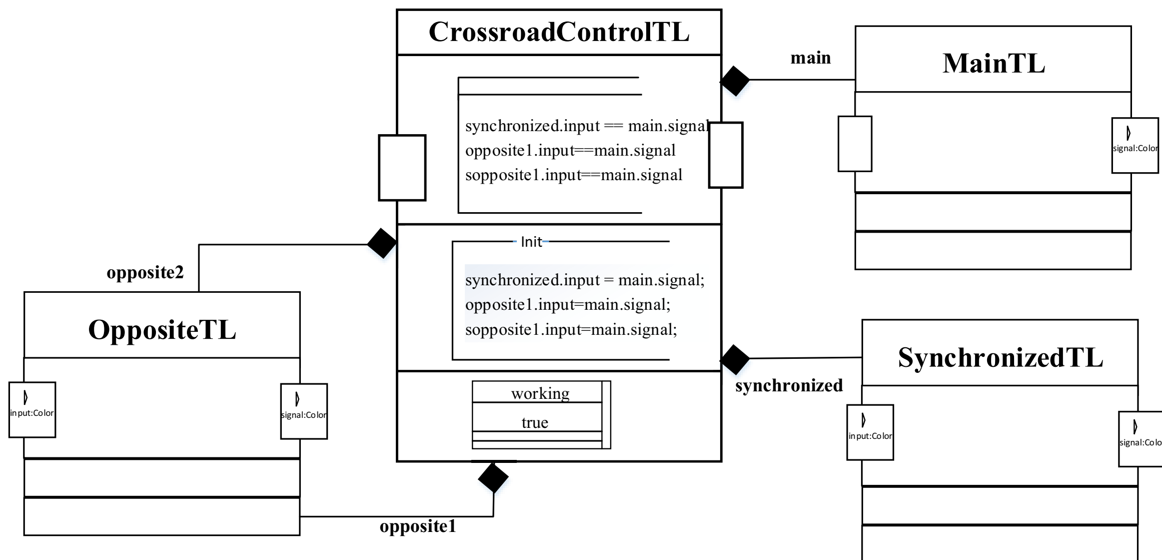

Figure 25. Crossroad control traffic lights

More details will be given on abstract and concrete syntaxes of HiLLS in next sections.

III.3 HiLLS Abstract Syntax

A specification of a system in system theory includes the specification of its structure and behavior. For system structure specification, important concepts are decomposition and composition. Decomposition defines how a system can be broken down into components systems [Zeigler et al. 2000]. Composition is how component systems may be coupled together to form a larger system. The closure under coupling property of the DEVS formalism guarantees that a composition of system is also a system. In order to decompose a system, one needs to divide a system into meaningful parts that can represent subcomponents in the system. Each part of the system must be characterized by well-defined attributes for the definition of its internal state and interfaces. Attributes domains can be defined by concepts and how they can be manipulated. System theory offers means to define systems in a hierarchical and modular manner but lacks some constructs to define new concepts (other than the concept of model and its transition functions) like the concepts of classes and their attributes. Object-Oriented modeling is known to be an excellent paradigm to define concepts and their relations and how these concepts can be manipulated. These concepts form the basis for the definition of objects internal state and behavior. While system theory focuses on system structure (the inner constitution of a system) and behavior (its outer manifestation), software engineering focuses on the static, dynamic, and functional aspects of a system. The static view is the time-independent view of a system. The

static aspects remain the same throughout the life cycle of the system. In particular, the static model describes the components of the system, the attributes (simple and complex), relationships between components, and the input and output ports of each components. The dynamic view captures the changes that the system undergoes with time and in response to events (internal or external). Dynamic modeling provides a view of a system in which control and sequencing are considered, either within a component (by means of state machines) or between components (by analysis of component interactions). The functional view is a high-level view of how several interactions work together to implement a system concern. It captures the functionality of the system – what the system can do. We believe that system theory paradigms (DEVS, RT-DEVS, DSDEVS) and software engineering paradigms (object orientation, formal methods) can complement each other to design and analyze complex systems. We integrate object orientation in DEVS-based system theory by using Object-Z [Smith 1992].

Object-Z is a conservative object oriented extension of the Z formal notation. The main concept in Object-Z is that of the class schema which captures the object-oriented notion of a class by encapsulating a single state schema with all the operation schemas which may affect its variables. We choose object-Z over other object oriented languages like UML for many reasons. The first reason is that Object-Z has a formal syntax and semantics and reasoning techniques developed for it contrary to UML which lacks precise and formal semantics. To avoid the impreciseness of UML, OCL (Object Constraints Language) [OMG 2010] is used to enforce the static semantics of UML models. OCL is not sufficient to express complex behavioral constraints needed for operations [Rusu and Lucanu 2011]. The second reason is that Object-Z is more expressive than UML class diagrams; this makes it possible to encapsulate the static semantics of model elements with the domain model of a system in terms of invariants defined in the corresponding Object-Z classes. Object-Z offers other facilities to express properties of a class temporal history similar to temporal logic formulas. So Object-Z can play the role of UML, OCL and linear temporal logic in a formal integrated manner.

The objective of this section is the detailed description of the HiLLS metamodel. We show how the HiLLS abstract syntax is defined from the abstract syntaxes (metamodels) of the selected languages (or fragments of languages). The first metamodel is concerned by pure object oriented concepts borrowed from EMOF [OMG 2015]. The concepts of this metamodel will play the role of interfaces between concepts from system theory and predicate logic. The second metamodel presents the abstract syntax of object-Z specification concepts. This part of the language brings possibilities to specify data and data transformations by using logical expressions and predicates. The third metamodel describes the system theory concepts of the language, i.e. concepts that are used to define the internal behavior of a system. The complete metamodel is finally presented; it shows how the different parts are integrated to form a precise and coherent whole. Formal transformation between HiLLS and the different semantic domains and selected languages will use this metamodel.

HiLLS combines Object-Z concepts with system theory concepts. Since Object-Z abstract syntax is defined as BNF grammar, our first problem is to translate this grammar to a formal Object-Z metamodel by adopting the metamodeling architecture of EMF/Ecore and extend it with system theory concepts.

III.3.1 Object-Z concepts Metamodel

The Object-Z expressions and predicates are used for the specification of data and operation bodies. It contributes also to the definition of the static part of a system (Figure 26).

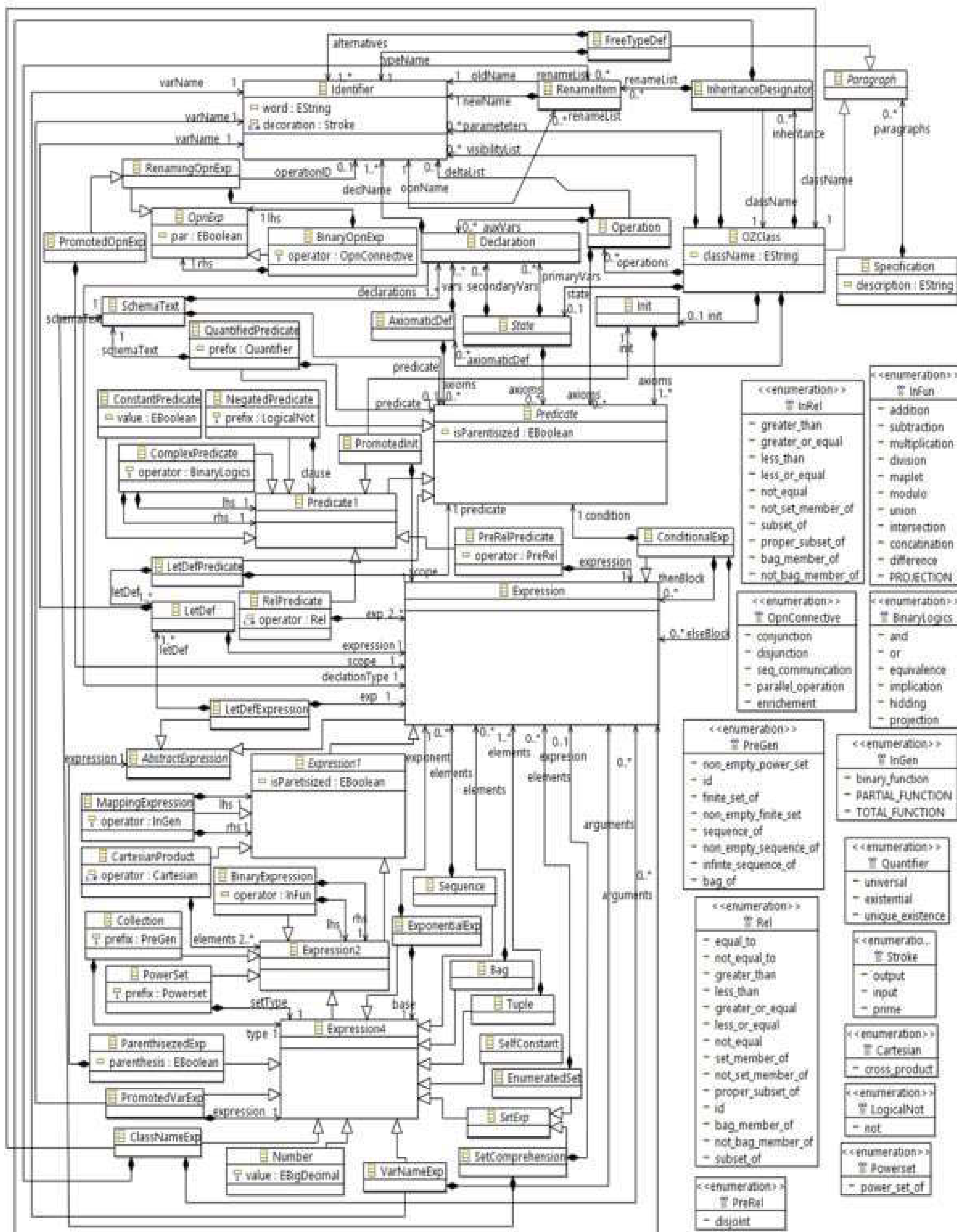

Figure 26. Object-Z Metamodel

Class *HClass* describes the Object-Z class which, by virtue of its inheriting the class *HClassifier*, may consist of a state schema (class *StateSchema*) that declares the state variables (class *Declaration*) and possible constraints on them, a specification of the initial state (class *Init*) which describes the object's starting state, operations (class *Operation*) that serves as the interfaces to communicate with the object and which may manipulate the state variables by means of predicates (class *Predicate*), local definitions (class *AxiomaticSchema*) to specify constants, global variables.

In addition to the amenability of Object-Z to formal analysis, the level of refinement provided by this segment of the meta-model helps to precisely and completely model systems' behavior in a generic form that can be refined to executable program code for the enactment of systems.

III.3.2 System Theory concepts from DEVS

HiLLS describes a complex system as an assembly of components (having autonomous behaviors) interacting with one another to produce the overall behaviors of the system. Thus, the overall behavior of the system is dependent on the characteristics of individual components and the way they influence one another. A component, being a system in itself may also contains its own components (sub-components); the smallest indivisible system, one without components is called the unitary system. This hierarchy of system-component composition typifies a tree structure with a system at the root and its components as direct children of the root node. By a depth-first traversal of the hierarchy tree, the compositions of individual components can be discovered recursively with all siblings of any node being components of their parent node and unitary systems constituting the leaves of the tree.

A system may have one *input interface* and/or one *output interface* through which it interacts with the environment by sending and/or receiving events. In HiLLS, we refer to all services and messages exchanged between a system and its environment as *events*. In either case, an interface contains one or more variable(s) representing the *port*(s) through which different kinds of events are received (for *input interface*) or sent out (for *output interface*). The *domain* of a port is a specification of the *kind of* events it may receive or produce; this is modeled as the Object-Z classifier in the metamodel to accommodate all primitives and user-defined classes of events. Thus, by implication, the domain of a port defines the set of events that may be received or provided through it. In HiLLS, system attributes comprise the state variables, ports and parameters. While *variables* and *ports* may take different values from their respective domains at different instants, parameters are defined with constant values that persist throughout the system's life cycle.

In HiLLS, a system serves as a logical boundary for all its components. Hence, peer components may interact and collaborate directly with each other but a component can interact with external systems only through the input and output ports of the parent system. Based on these requirements, we classify system couplings into three categories: *Internal Coupling, Input Coupling* and *Output Coupling*. In all cases, the coupling predicate facilitates the exchange of events between two ports. The concepts of *Internal Coupling, Input Coupling* and *Output Coupling* are adopted from the DEVS' External Input Coupling, Input Coupling and External Output Coupling (Zeigler, Praehofer & Kim 2000) respectively. While *Internal Coupling* is

established between two peer components of same system, *Input Coupling* is defined between the input of a system and that of one of its components to enable the component receive some external events and *Output Coupling*, by connecting the output of a component to that of its parent allows it (the component) to send events to some external system. In any case, the port producing the event (the influencer) is referred to as the *sender* while that which is influenced is called the *receiver*.

The discrete event part of the language is formalized by the following metamodel (Figure 27). This metamodel defines the concept of state-machine, configuration, event and transitions.

By associating certain behaviors and possibly some activities (set of actions that do not change the state of a system) to some unique sets of constraints on the system attributes, HiLLS allows the user to define *configurations* for the different circumstances of the system. Hence, the system's dynamics is described by the sequence of *transitions* between the configurations.

A discrete events model of a system can have an unimaginable size of state space. The size of the state space can even become infinite leading to a problem of state explosion. In order to represent a system with a large number of states, we use a finite number of state variables (after abstracting the key variables that can give a reasonable description) to partition the state space into a finite number of *Configurations*. A C*onfiguration* is a partition or subdivision of the state space into non-overlapping and nonempty subsets of states. As such, a configuration has the following properties:

- every state in the state space belongs to a particular *Configuration;*
- the sets of *Configurations* are mutually disjoint;

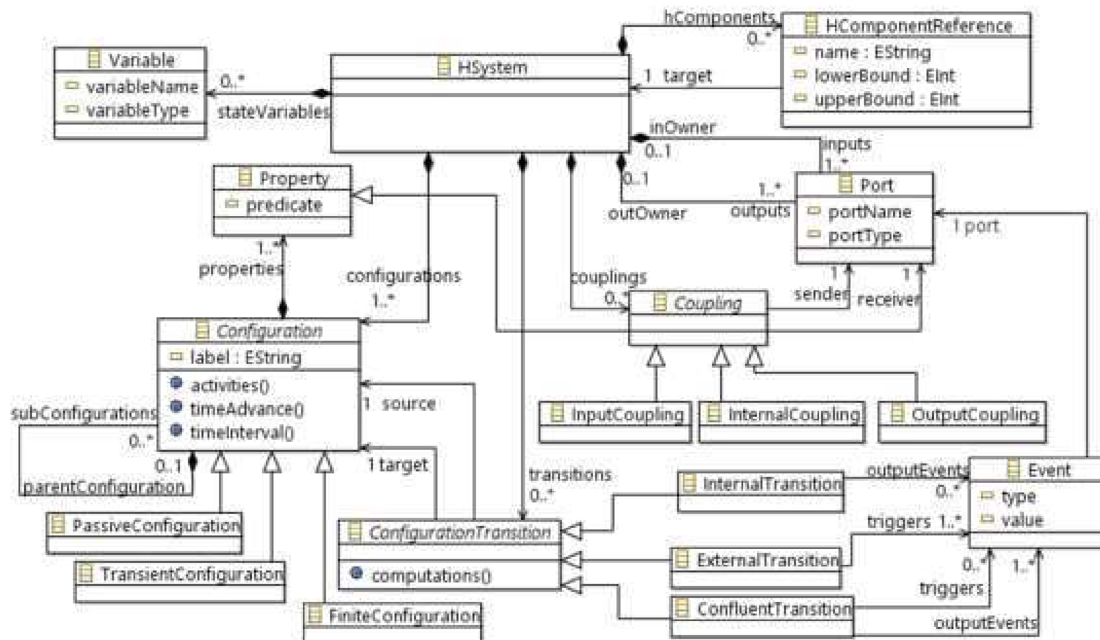

Figure 27. Discrete Event (part) Metamodel

The metamodel is accompagnied by the following constraints:

```
context Port
    inv owner_Constraint ('A port is used for either input or output and not both'):
        inOwner->isEmpty() xor outOwner->isEmpty()
    def:
        owner:System = if (inOwner->isEmpty()) then outOwner else inOwner endif

context Coupling
        inv No_feedback_coupling ('Coupling ports of same system is illegal'):
            sender.owner <> receiver.owner

context InputCoupling
    inv EIC_Constraints ('sender = input of container, receiver = output of a component'):
        sender.owner.components.target-> includes(receiver.owner) and
        sender.owner.inputs -> includes(sender) and --sender is an input port of its owner
        receiver.owner.inputs -> includes(receiver) --receiver is an input port of its owner

context OutputCoupling
    inv EOC_Constraints ('sender = input of a component, receiver = output of container'):
        receiver.owner.components.target->includes(sender.owner) and
        sender.owner.outputs -> includes(sender) and --sender is an output port of its owner
        receiver.owner.outputs -> includes(receiver)--receiver is an output port of its owner

context InternalCoupling
    inv IC_Constraints ('sender = output of a component, receiver = input of a component'):
        sender.owner.outputs -> includes(sender) and
        receiver.owner.inputs -> includes(receiver)
    inv peer_Constraint ('influencer and influencee must be components of same system'):
        sender.owner.hContainer = receiver.owner.hContainer
```

Figure 28. OCL Constraints of Discrete Event Metamodel

The states in a *Configuration* are "alike" (as a result of the configuration on state variables and similar transitions to states belonging to the same *Configuration*); thus they are related by an equivalence relation and a *Configuration* is an equivalence class of states; and a *Configuration* is defined by *Properties* and a *Property* is a cross product of *SubSets* of the domains of the state variables. The concept of configuration is introduced to facilitate the graphical representation of the discrete event behavior of models. The configurations are probably infinite sets of state; we use operations to determine the target state in a transition between configurations. For example; supposing we have a state space defined by two integer variable $S = \{(x, \mathbb{Z}), (y, \mathbb{Z})\}$ then we may have configurations like $c_1 = (x \leq 0 \wedge y \leq 0)$, $c_2 = (x \leq 0 \wedge y > 0)$, $c_3 = (x > 0 \wedge y \leq 0)$and $c_4 = (x > 0 \wedge y > 0)$, each defining a unique subset of the state space of interest to the modeler such that the union of all configurations must yield S.

A configuration can be *unitary* or *composite*. A composite configuration is composed of other configurations. Each *Configuration* has a *TimeAdvance* function that maps each member state to a real time advance. The lifetime ore time advance defines the maximum sojourn time of the system in a configuration before an internal transition event takes place. This value is generated by the *lifetime function* of the configuration which is a function of some system attributes. Based on its sojourn time, we classify a configuration into one of three categories: a *transient configuration* lives for a duration of zero time unit and most often has an instantaneous (or no) task associated to it. In contrast, a *passive configuration* is an abstraction of a circumstance in which a system remains indefinitely until an external influence overcomes its inertia. It is assigned an infinite lifetime. The third category is the *finite configuration* with a sojourn time greater than zero and less than positive infinity; we use this as an abstraction of states that reign for a limited period or to schedule a limited delay. HiLLS also allows the modeler to specify activities in configurations. An *activity* is a set of *operations* that do not result into a change of state which may be performed by the system during its sojourn in a configuration.

The structural and behavioral dynamics of a HiLLS' state machine is modeled by the successive transitions between its configurations. A transition is accompanied by a sequence of *computations* for the reconfiguration of state variables and possibly, the generation of output events. The computations may also involve the derivation of output events in some cases. A configuration transition may occur as a self-scheduled event, or an impulse-driven event or a combination of both. From the moment a *non-passive* (transient or finite) configuration is assumed, an *internal transition* event is scheduled to occur at the end of its (the configuration's) life time provided no external influence is received before this time. A *non-transient* (finite or passive) configuration undergoes an *external transition* if an event (impulse) is received at an input port before the end of its reign. The third category, *confluent transition,* which is peculiar to non-passive configurations is a combination of the first two - it occurs when an event is received just at the end of the tenure of the current configuration; thus, the received event is treated instantaneously followed by a transition to either the scheduled target configuration or another depending on the result(s) of processing the received event.

Transitions take place from a *source* to a *target* configuration. In the case where a transition has two or more candidate target Configurations, a sequence of pre-*conditions* is specified to help the system decide on the path to the appropriate target. Conditions are defined by predicates on system attributes; since predicates evaluate to true or false, a condition would always generate two paths (one for each of the truth values), each leading to a candidate target configuration or another condition.

III.3.3 Object Orientation Concepts from EMOF

This metamodel defines the traditional concepts of class, data types and attributes. The metamodel of the object oriented part of HiLLS is presented in Figure 29. It is composed of the following elements:

- **NamedElement:** It is the base class that describes any entity or relationship in a metamodel that has a unique name. Therefore, every metamodel element that could be identified by a name inherits a name attribute from this class.
- **Package:** A package is a classifier that is a logical collection of classifiers.
- **Class:** It describes a class which models an independent entity. A class may have some attributes (eAttributes) and/or references (eReferences) describing its structural features and it may inherit from some other classes by referring to them as superTypes.
- **Attribute:** It describes attributes which model the named data contained in a class. Every attribute has a type that is defined by a data type. The minimum and maximum numbers of possible occurrences of an attribute in a class are defined by lowerBound and upperBound respectively; this is known as the cardinality of the attribute.
- **Operation:** A class has 0 or more operations that define its behaviour. An operation has 0 or more parameters classified as input and output.
- **DataType:** It describes the named types of data held by attributes. These may be primitive types like integer, boolean, etc.
- **Enum:** It describes an Enumerator, simply called Enum which is a special kind of data type which explicitly enumerates a finite list of values, called literals that an attribute may possibly take.

* **EnumLiteral:** It models a named literal that is specified in an Enum. In addition to its name, a literal has a unique integer value associated to it which signifies its position in the list of literals defined by the containing Enum.
* **Reference:** It describes references which are named associations between classes in a metamodel. Thus, a reference of a model is an association from the class to another class (identified by eReferenceType). Like attributes, a reference also has a cardinality that is described by its lowerBound and upperBound. A reference with containement is a stronger type of association which implies ownership of the target of the association by the class from which it originates. Finally, a reference's Opposite is a corresponding reference that is navigable in the opposite direction.

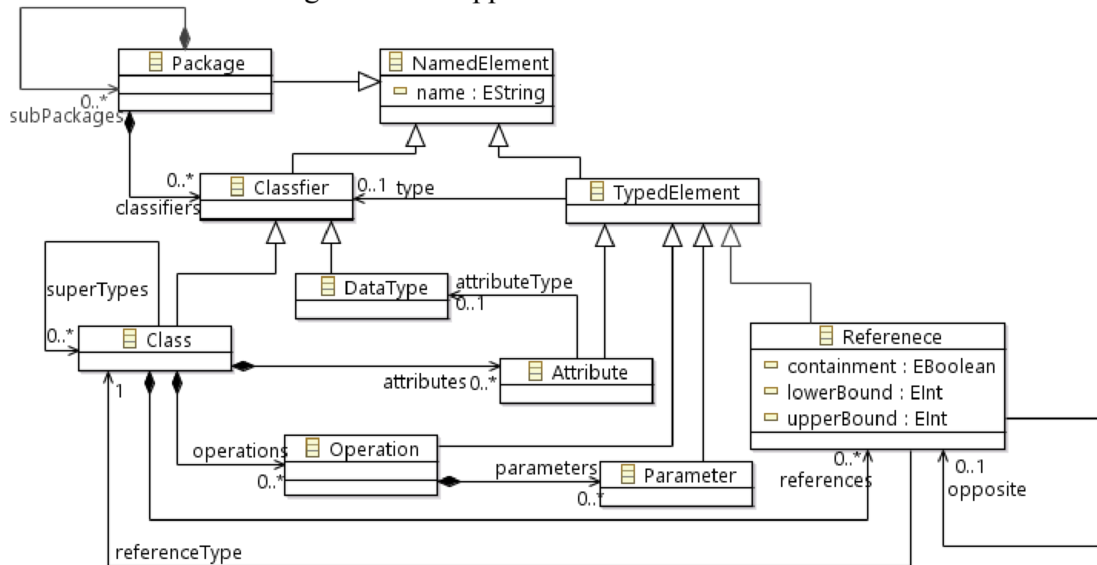

Figure 29. HiLLS Object Oriented part metamodel

Following the recommendation of the metamodel interfacing technique, we introduce an excerpt (Figure 29) from the Ecore metamodel [Budinsky et al. 2003] to be used as the interface between the DEVS/RT-DEVS and the Object-Z metamodels. The motivation for our choice of the interface segment is twofold: 1) there is significant overlap between the concepts it describes and those of the Object-Z that would enable them merge reasonably with; the DEVS/RT-DEVS metamodel can as well extend the object concept to describe systems if we consider a system as an object with autonomous behavior and I/O ports through which it may influence or be influenced by its environment; 2) the relationships between the classes provide a renowned pattern that could add more clarity to the overall metamodel.

III.3.4 HiLLS Complete Metamodel

Using the meta-model merge technique [Emerson and Sztipanovits 2006], the class *HClassifier* describes constructs that are common to the system-theoretic and software engineering parts of the HiLLS syntax while each of the two adds its peculiarities through meta-model extension [Emerson and Sztipanovits 2006]. *HClassifier* describes an object having state attributes, constants, operations as the interface for interacting with its environment and history.

HSystem extends *HClassifier* to describe a system that in addition to possessing the mentioned properties also has ports through which it may influence or be influenced by its environment, components and the processes that define its autonomous behavior.

In addition to facilitating logical reasoning, some Object-Z concepts such as predicate and expression are reused to precise the necessary details of certain abstract system-theoretic concepts, thanks to the meta-model refinement technique [Emerson and Sztipanovits 2006] [Khan and Risoldi 2012]. For example, *Predicate* in Object-Z is reused to refine the p*roperties, activities* and *sojourn Time* of *configurations* in the system-theoretic part. It also gives the precise details of the *computations* associated with *configuration transitions.* Similarly, *Expression* and *Declaration* provide the details of *events* and *ports* respectively. By virtue of this integration and the type system inherent in the details, formal methods can be used to investigate the logical correctness and consistencies of models.

The integrated metamodel is presented in Figure 30. By applying the metamodel merge technique, the pairs of classes - of the form (Interface, Object-Z) - (Attribute, Declaration), (Operation, Operation) and (OZClass, Class) fuse to form *HDeclaration, HOperation* and *HClass* respectively. Though the metamodel merge technique recommends that the resulting class from merging two or more classes should bear all attributes and relations of the merged classes, we considered it necessary to use our discretion to remove duplicate properties by retaining the ones at such levels of refinements to allow for the extraction of all merging participants. This may require further research to enhance the metamodel merge technique.

The DEVS metamodel integrates with the *interface* through the *inheritance* of *HClass* by *HSystem;* we can as well consider this to be integration by *metamodel extenxion* technique. Also to avoid duplicate of concepts, the *HClass' state* replaces the *stateSpace* of *HSystem* since the former provides a better refinement to satisfy all participating formalisms.

For the moment, we are still investigating the consistency issues surrounding having inheritance between *HSystems* or between *HClass* and *HSystem.* Therefore, we employ the *metamodel restriction* technique to prevent such specification.

The *class refinement* technique is used extensively to provide detailed descriptions of abstractly specified concepts in the DEVS metamodel through containment relationships between them and appropriate concepts in the Object-Z metamodel. Particulartly, class refinement underscores the following Class-containment reference pairs: (P*ort, portDecl), (ConfigurationTransition, computations), (Event, value), (Configuration, timeAdvance),(Configuration, activities)* and *(Configuration, timeInterval).*

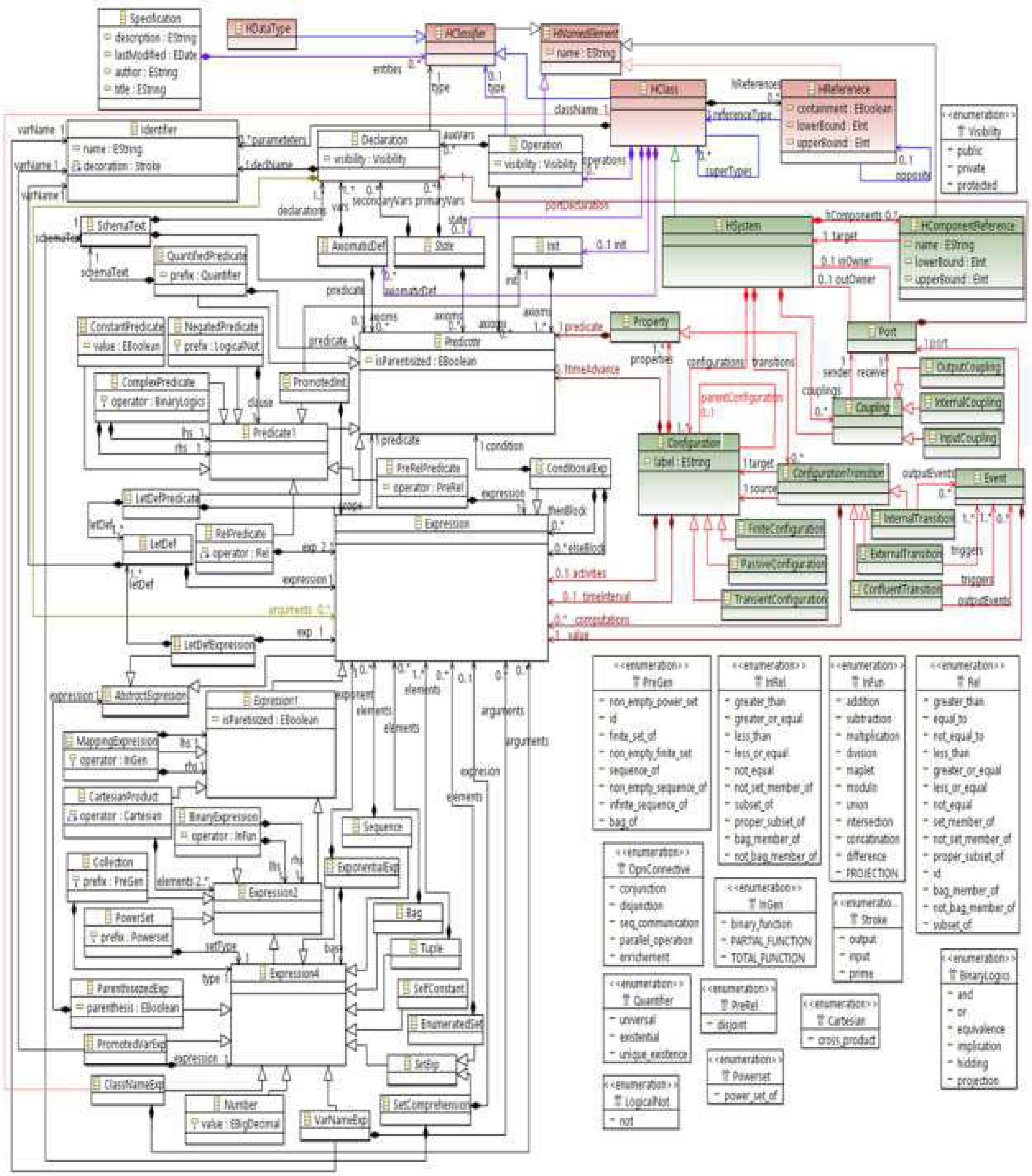

Figure 30. Complete Metamodel integrating the different parts

Some OCL constraints associated to complete metamodel are given in Figure 31.

```
import 'HiLLS.ecore'
package hills

context HSpecification
        inv unique_HSystem_name ('No two systems have identical names'):
                entities->forAll(ent1:HClassifier, ent2:HClassifier| ent1 <> ent2 implies ent1.name <> ent2.name)

context HSystem
        inv dedicated_ports_constraints ('A port is used exclusively as input or output and not for both.'):
                inputs->excludesAll(outputs)
        inv unique_configurations ('configurations must have unique names and properties'):
                configurations->forAll(config1:Configuration, config2:Configuration|config1<>config2 implies
                        config1.label <> config2.label and config1.properties<>config2.properties)
        inv sub_and_parent_configuration_relations (''):
                configurations->forAll(config1:Configuration,config2:Configuration|
                        config1.parentConfiguration = config2 implies config1.subConfigurations->includes(config2))

context Configuration
        inv configuration_classification_constraints('Configurations are classified by the values of their sojourn times'):
                self.oclIsTypeOf(TransientConfiguration) implies self.sojournTime.oclAsType(Real) = 0.0 and
                self.oclIsTypeOf(FiniteConfiguration) implies self.sojournTime.oclAsType(Real) > 0.0 and
                self.oclIsTypeOf(PassiveConfiguration) implies self.sojournTime.oclAsType(Real) = UnlimitedNatural

        inv isolated_configuration_constraint ('Isolated configuration is illegal'):
                HSystem.transitions->exists(source = self or target = self)
        inv nonPassive_Configurations_sonstraints ('A non-passive configuration must not be a final state'):
                not self.oclIsTypeOf(PassiveConfiguration) implies HSystem.transitions->exists(source = self)
        inv pi_and_miu_constraints:
                let pi:Predicate = self.parentConfiguration.sojournTime,
                        miu:Predicate = self.subConfigurations->select(label='ACTIVE_CONFIG').sojournTime
                inv
                self.sojournTime = pi implies self.parentConfiguration->notEmpty() and self.parentConfiguration.sojournTime <> miu and
self.sojournTime = miu implies self.subConfigurations->notEmpty() and self.subConfigurations->forAll(sojournTime <>pi)
context ConfigurationTransition
        inv PassiveConfig_InternalTrans_Constraint ('Internal and confluent transitions cannot originate from a passive cofiguration'):
                self.oclIsTypeOf(InternalTransition) or self.oclIsTypeOf(ConfluentTransition)
                implies not self.source.oclIsTypeOf(PassiveConfiguration)

context Port
        inv owner_Constraint ('A port is used for either input or output and not both'):
                inOwner->isEmpty() xor outOwner->isEmpty()
        def:
                owner:HSystem = if (inOwner->isEmpty()) then outOwner else inOwner endif

context Coupling
                inv No_feedback_coupling ('Coupling ports of same system is illegal'):
                        sender.owner <> receiver.owner

context InputCoupling
        inv EIC_Constraints ('EIC is a coupling from an input port of a container to an input port of its component'):
                sender.owner.hComponents.target-> includes(receiver.owner) and -

context OutputCoupling
        inv EOC_Constraints ('EOC is a coupling from an output port of a component to an output port of its cotainer'):
                receiver.owner.hComponents.target->includes(sender.owner) and -

context InternalCoupling
        inv IC_Constraints ('IC is a coupling from an output port of a system to an input port of its peer'):
                sender.owner.outputs -> includes(sender) and
                receiver.owner.inputs -> includes(receiver)
        inv peer_Constraint ('influencer and influencee must be components of same container system'):
                sender.owner.hContainer = receiver.owner.hContainer

endpackage
```

Figure 31. OCL constraints

Communications between components of a complex system are done via the ports in the input and output interfaces. System specification languages often assume static (permanent) couplings of these components. However, in reality, many systems are dynamic structure systems.

A dynamic-structured system is one whose set of components and/or the topology of their connections are time-varying (change dynamically). Static-structured systems on the other hand are characterized by a fixed number of interacting components and permanent links between the components that influence one another. Examples of such dynamic couplings exist in automatic

switching systems, relay systems, gear transmission systems and computing networks in which sessions (connections) are automatically established and destroyed between different devices, the ecosystem in which the number of components (plants and animals) changes as a result of growth, reproduction and death. Intuitively, we know that the pattern of interactions between the components would not remain the same. On the other hand, an automobile gear transmission system is one in which the set of components may be considered fixed but the linkages between them changes following some rules. Another situation where structural dynamics can be very useful is the modeling of redundancy in system design to ensure uninterrupted operations. In this case, an isolated component may be automatically loaded into operation in the event of the failure of the active component.

In HiLLS, structural dynamics is realized in configurations with coupling predicates, predicates dedicated to specifying the instantaneous relationships between two ports of different systems in an assembly.

The difference between our work and existing approaches presented in section II.2.5 for the modeling of dynamic structure systems is that we described ports as some special state variables declared in the interfaces for exchanging events with the environment. In addition to the instantaneous variations in the events held in a port, the source and/or destination of such event may also change with time; this concept practically translates into structural changes in the system. Thus, a coupling predicate (in a configuration) specifically defines a relationship between two ports, each belonging to different systems. Hence, by specifying the couplings between the components of a system in its configurations, we are able to manage its structural dynamics through transitions between the configurations.

Similar approaches exist in the literature that addressed how to integrate heterogeneous models and how to analyse them. Vallecillo [Vallecilo 2010] proposed a system modeling approach that uses metamodel integration techniques in the context of the RM-ODP [Linington et al. 2011]. The approach is to integrate metamodels of languages of different views of a system into a *global metamodel* having two-way correspondences with each *view metamodel* with the primary aim of maintaining consistencies between models of disparate system's views. One interesting strength of the approach is that stakeholders can simply deal with system elements in their respective views as no concrete syntax is provided to the global metamodel because it is considered that it would be too complex to be handled by users.

Interestingly, the trade-off is between creating many simple models of different views and creating just one somewhat more complex unified model. Our approach is different in that we propose to develop the unified metamodel into a complete language that replaces the many different languages while all views can be automatically generated with the help of MDE thereby reducing the required manpower - we can see from Figure 30 that the size and complexity of the unified metamodel is just about that of the largest of the participating metamodels. Moreover, use of the unified metamodel allows for reuse of some language constructs to enrich the specification of others.

Another approach to system specification and analysis that is interestingly related to ours has been proposed by Attiogbé [Attiogbé 2010]. In order to ensure consistencies between heterogeneous specifications for multifaceted analysis of complex systems, the author advocates

the specification of an abstract reference model from which specific models are derived for different analysis of system properties. The derivation of specific models may be done by translation of the reference model to the specific facet or by extension of the reference model. The idea also, is that observations from the results of specific analyses can be fed back in to the reference and transmitted to all facets. The idea was demonstrated with a case study that uses a specification in B [Abrial 1996] as a reference model that is checked using the B4free tool [CLEARSY 2009] and extended to derive a specific facet model for analysis with ProB [Leuschel and Butler 2008]. While this approach and that presented in this paper share some common intents, there are some fundamental differences in the methodologies they adopt, particularly in the area of the source and composition of the formalism for specifying the reference model. Firstly, the author proposed the specification of abstract mathematical model using First Order Logic (FOL) or some ad-hoc formalism while we propose to fully define a language that formally integrates concepts from formalisms to specify a unified model. Secondly, the case study provided also suggests that the reference model's usability is restricted to formal methods while we intend we intend derive equivalent models for formal methods, simulation and enactment from our unified model. In that way, the integrated formalisms (and techniques) complement one another not only through feedbacks to the unified model but also through reuse of constructs for refinement where necessary. Lastly, we have adopted a metamodel-based approach to formalism integration to considerably raise the level of abstraction of the unified model - thanks to model transformation techniques - as to bridge the gap between the formal techniques and domain experts.

[At-Ameur et al. 2004] presented a component Oriented approach to solve the problem of global requirements validation in the domain of embedded systems especially in avionics. The approach consists of specifying a central model representing the static structure and global constraints of the system and a set of views for specific analysis purposes. Each view is specified using a separate formal language suite for a given aspect of the system (for example timed automata) and associated tools (e.g., UPPAAL) to verify specific properties of the system. To ensure consistency, coherence criteria is specified between the different views by using a concept of institution presented by the authors. For each view, the formal language used is represented by a suitable institution and the synchronization between them is realized by synchronization relation between the corresponding institutions on the models and linked to the central model. The case study presented used UPPAAL and LUSTRE to validate the reconfiguration mechanism of a command and slaving subsystem, in charge of controlling three aerodynamic surfaces of an aircraft but it is not very clear how the synchronization is done at the operational level. The approach gives a sound synchronization technique between the views but the approach is constrained by required skills in the languages used for these views. The central model defined is not rich enough to allow the derivation of the models for the specific views.

III.4 HiLLS Concrete Syntax

In this section, we present the concrete syntax of HiLLS. The concepts and the relationships specified in the abstract syntax are drawn in the same style as the abstract syntax with the graphical notations replacing the corresponding elements. Elements that are represented in dashed boxes are realized within other graphical elements. HiLLS models possess properties like attributes, functions (or subroutines), and associations. Associations are rendered in HiLLS with the same notations as in the UML.

The notation of a HiLLS model (Figure 32 b) is similar to that of a UML class in structure and use with a few additional elements. It has the traditional UML Class compartments for name (label and template), attributes (state variables and system parameters; note that complex attributes are formed by composition), and functions. In addition to these, there is a fourth compartment which houses the state machine (consisting of configurations and transitions between them) for expressing the system's dynamic behavior. Unlike the conventional way of writing attribute declarations and function signatures directly in their respective compartments in the class diagram, HiLLS uses a schema in the attribute compartment to declare all simple state variables and a second schema in the same compartment to specify all system's parameters. Similarly, functions are defined in schemas placed in the third compartment. This allows the modeler to go a step beyond just writing the function signatures; the entire body of the function can be specified formally in a platform-independent form. The input and output interfaces of the model are denoted by rectangles attached to the left and right sides respectively of the attribute (second) compartment with inner circles denoting the ports. A port is labeled with its name and domain. Figure 32 a shows HiLLS class for which discrete event behaviour is not specified.

The notation for a finite configuration is a box with four compartments for label, properties, activities, and sub-configurations and a symbol, f_c that denotes its life time function (Figure 32 d). Passive configuration has a concrete symbol similar to that of finite configuration except that the rectangle has a vertical stripe attached to its right edge as an indication of its infinite life time. The transient configuration does not share much resemblance with the other two. It is denoted by a circle with three compartments for its label, properties and activity. Its shape implicitly defines its zero life time.

For e*xternal* transitions, we use a dashed arrow emanating from the top or bottom (and away from the edges) of the source configuration. An i*nternal* transition is a solid arrow drawn normally to the right side (not the corner/edge of a finite configuration) of the source configuration while the confluent transition is denoted by a dashed-dotted arrow originating from the top-right edge of a finite configuration or as a tangent to a transient configuration from the right. In any case, the arrow goes into the left corner (inflow) of a condition diamond or terminates as a normal to the left side of the target configuration. The condition diamond is a diamond shape containing a predicate which specifies a condition that is tested during a transition event to choose one of two paths to the appropriate target configuration (depending on the truth value of the condition). A transition arrow may enter the condition diamond through either of its horizontal corners and emanate from both the top and bottom corners to define two contrasting paths for the truth values of its predicate. Note that the two arrows emanating from a condition diamond must take the same form as that entering it. Transition operations are represented by labels of their respective arrows.

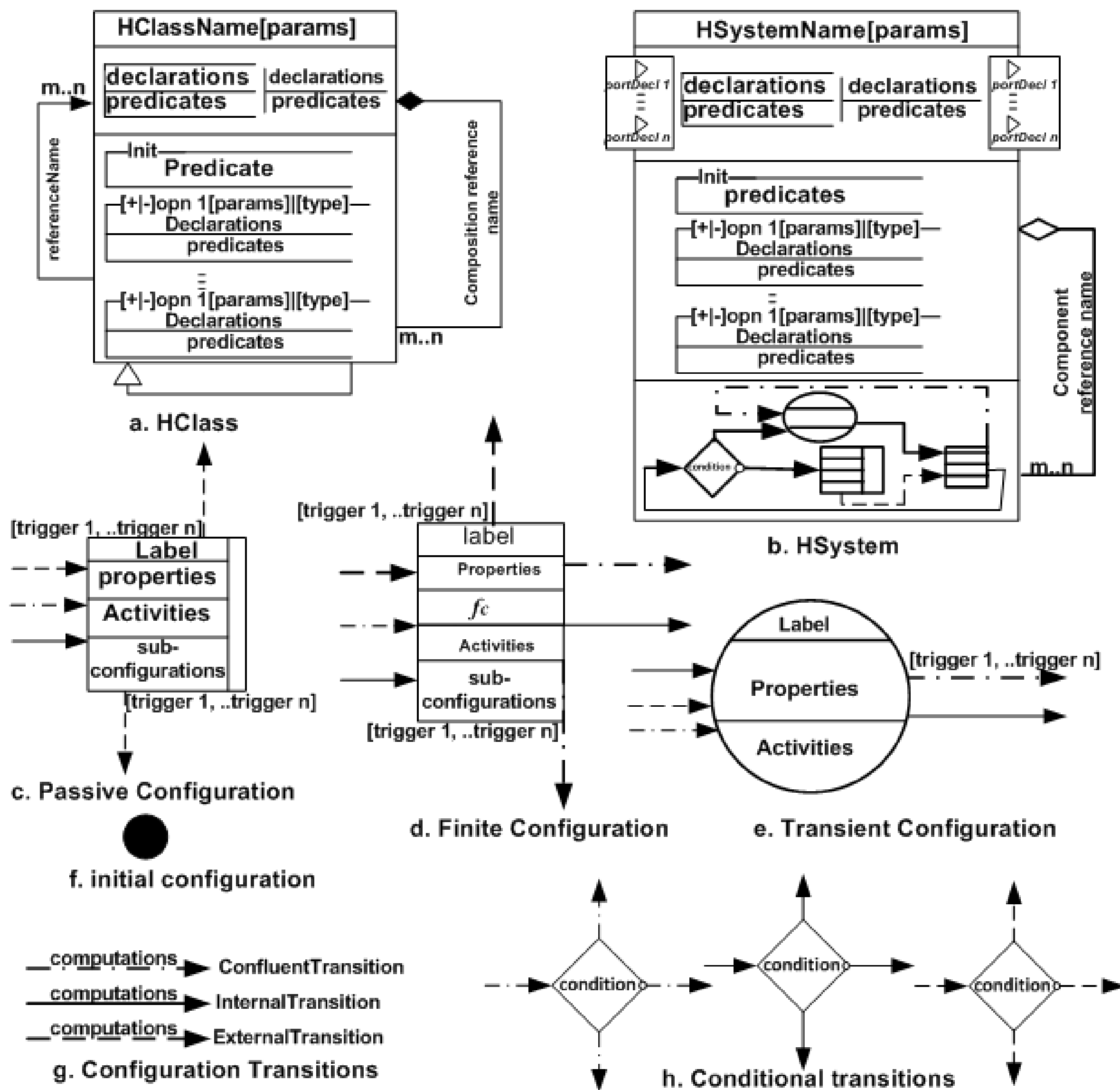

Figure 32. HiLLS Concrete Syntax

III.5 Queuing Network example

We present the modeling of a simple queuing network to illustrate some concepts of HiLLS not shown in the traffic light example. We choose Queuing Network example because it is widely known and intuitive.

We consider a model of a queuing network (*QueueinNetwork*) with two unitary components: a queue and a server. Figure 33 is the specification of the composite *HSystem QueueinNetwork*.

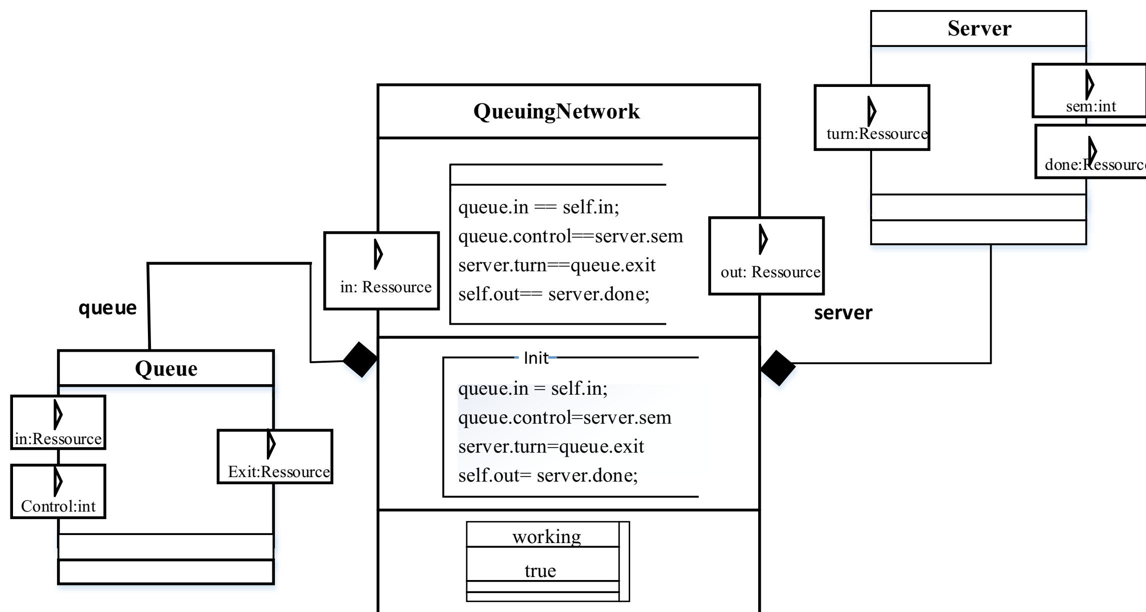

Figure 33. Queueing Network

The *Queue* has a *buffer* to hold resources that are waiting to be served by the *Server*. The *Queue* has ports *Enter* (for receiving resources fed via the *in* port of *QueueinNetwork*), *Control* (for receiving permission from the Server's *Sem* port to send a resource), and *Exit* (for sending a resource to the *Server* when it is its turn, this resource is received by the *Turn* port of the *server*). A served resource is sent out of the *server* through the *Done* port and it leaves the system through the *out* port. The couplings include the input coupling from the input port *in* of *QueueinNetwork* to the input port *in* of the Queue, the internal coupling from the output port *sem* of *server* to the input port *control* of *queue*, the internal coupling from the output port *exit* of *queue* to the input port *turn* of *server*, and the output coupling from the output port *done* of the server to the output port out of in of *QueueinNetwork.*

Figure 34 is the Queue Atomic Model. We consider that the queue component has a *buffer* (realized by aggregation) to hold Resources to be served by the server. It also keeps the *bufferSize*, *max* (the maximum number of elements that can be stored in the buffer) as state variables, and a value for *Check* (integer value used to check if the server is busy when a new Resource appears on an empty queue). The functions *enqueue()*, *dequeue()*, and *setCheck()* are called from the state machine to perform some operations. The *queue* can be in any of three configurations: *Empty*, *Buffering*, and *Full*; depending on the configuration of the bufferSize. It has two input ports: *Control* (to receive signals (1 or 0) from the server when it is/not ready to serve resources) and *in* (used to add new Resources to the queue, as long as it is not *Full*; and an output port, *Exit* (to send resources to the server). When in any configuration, and it receives an input 1 from the *Control* port (external transition event; represented as *Control?1*), the server indicates that it is free to serve Resources. If in *Full*, a resource is sent through an intermediate state and dequeued. (*Exit!dequeue();* representing the output through *Exit* port) and transitions to

Buffering (internal transition). If in the Empty configuration, since no Resource is in the buffer, it *setCheck(1)* (assignment operation). If in the *Buffering* configuration, it checks the buffer size (conditional transition, represented by the diamond shape); if it is equal to 1, it transitions to the Empty configuration after sending a resource through the *exit* port and dequeuing.

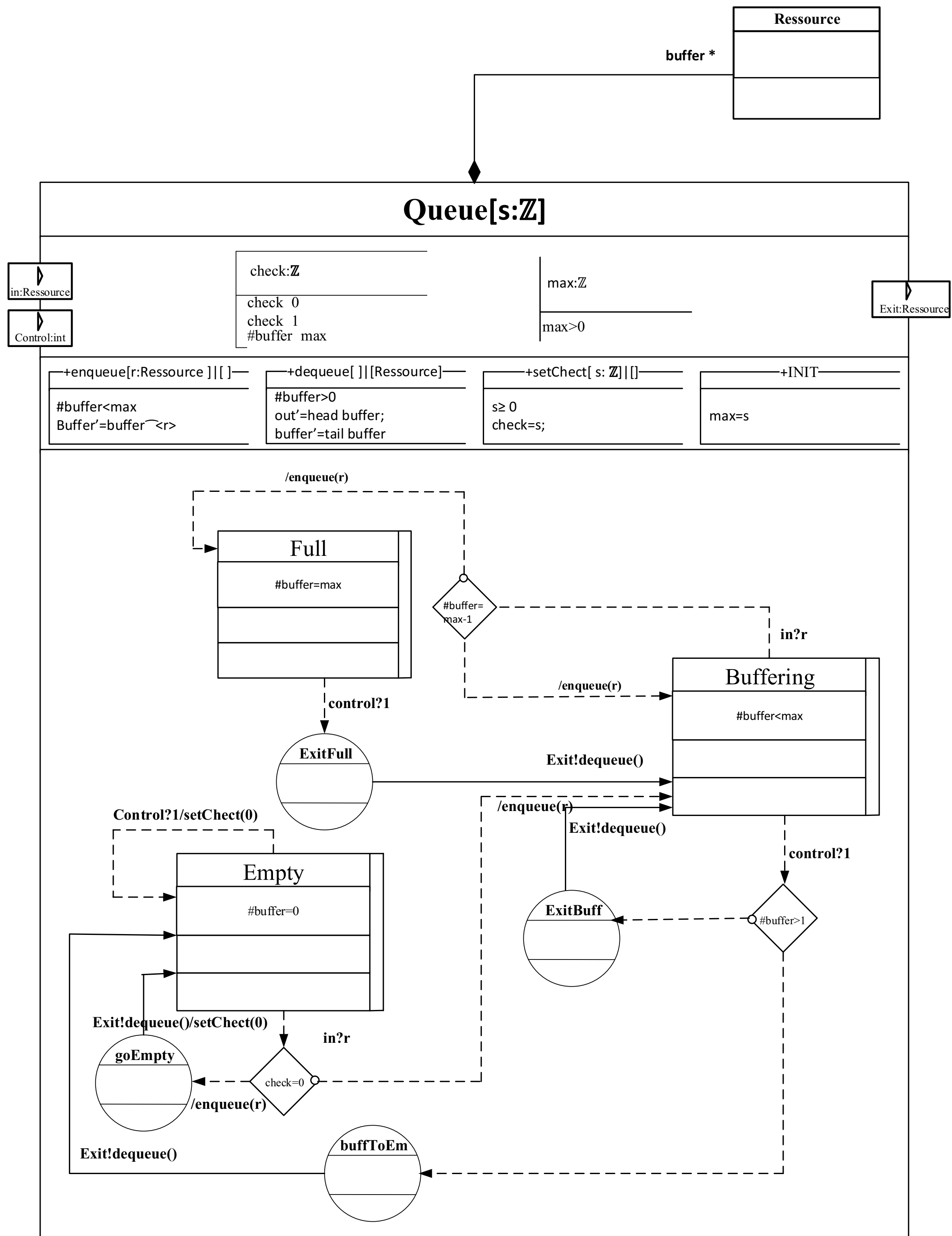

Figure 34. HSystem of the Queue

III.6 Set-theoretic semantics of HiLLS

A good understanding of the semantics of a language enhances the investigation of the quality (e.g., completeness, correctness and consistency) of models [Szlenk 2006]. Though the concepts and their relationships are captured in the abstract syntax, the semantics definition is required to explain and complement it and serve as a guide for its implementation [Kleppe 2008]. Any model specification language must come compete with rigid rules that clarify the legal syntactic representations with clear descriptions of their meanings [Harel & Rumpe 2004]. HiLLS models are meant to serve multiple use cases in different semantic domains. At present, our focus is on three broad areas: simulation, formal analysis and enactment. The mathematical semantics provides a basis for common understanding among all semantic domains by clarifying potential cases of ambiguities in the abstract syntax that could lead to different interpretations. It also clarifies the rules and constraints of the syntax.

The mathematical semantics of HiLLS provides a universal, domain-independent precise description of its abstract syntax. By providing a common point of reference for all concrete semantic domains and avoiding conflicting model interpretations. It is expected to serve a number of purposes:

- To add missing details to the syntax by clearly specifying relationship rules and constraints in unambiguous terms.
- To provide a basis for logical reasoning with models
- To provide a guide for precise understanding of the language's syntax and in the development of its supporting tools.

The motivation for this mathematical foundation of HiLLS lies in the imperativeness of a precise mathematical semantics definition and its associated benefits to a modeling language as have been identified by many research efforts in the literature. Without a precisely defined semantics, a language's syntax, no matter how well defined, could suffer a myriad of conflicting interpretations thereby making it very difficult (if not impossible) to accurately check model qualities. The proper handling of formal mathematics is indispensable for developing automated analysis tools for a modeling language. Transformations of a model into different semantic domains as well as the use of formal specification techniques in industrial applications require a deep understanding of the underlying mathematics [Varro & Pataricza 2003]. The UML, in spite of its proven success in the field of software engineering has been identified by many research works [Diskin 2003; Szlenk 2006; Kleppe 2008] as one important language whose potentials have not been fully exploited due to the non-existence of formal semantics in its specifications. [Diskin 2003] described the various manifestations of the gap created by this deficiency as "lack-of-semantic syndrome". Many tentative to provide formal semantics to UML used standard mathematics to define what is called “semantic model” [Lano and Bicarregui 1999].

As a meta-language for defining the language concepts, we use basic mathematical notations. The advantage of this approach lies in the versatility and universality of mathematical notations. The language of set theory underpins several formalisms [Fishwick 2007]. Thus, we use handy and intuitive mathematical notations based on elementary set theory to explain the HiLLS metamodel. A good knowledge of elementary set theory, relations and simple logics would help the reader understand the expressions used. We consider this semantics as an independent axiomatic semantics of HILLS.

III.6.1 Formalization

We propose here a mathematical basis for HILLS to present the key concepts of the language. A HILLS model can be composed of other models (0 or many). We distinguish two types of models: the unitary model which has no sub-components and composites models which are aggregations of unitary or composite models. The semantic model describes structure and behavior. Structural aspects include attributes and relations between them; the structural part defines the data space of a model. For the definition of the data space, primitive data type (integer, Boolean, strings, reals) and user defined data types (HiLLS classes) are used. The behavior is defined by the transition relation. The behavioral part defines the temporal ordering of the steps of the system depending on system's configurations and received events.

We define the following sets and concepts:

$TypeName$ is the set of type names.

$Universe$ is the universe of values.

We assume also that some specific values are element of $Universe$. $-\infty \in Universe$, $+\infty \in Universe$, $true \in Universe$, $false \in Universe$.

Each type has an associated domain of values. Domains of types are defined by the following function

$dom: TypeName \rightarrow 2^{Universe}$. This function satisfy the following property
$\forall\ t \in TypeName, dom(t) \neq \emptyset$.

We assume that $Integer, Real, Real^{+}, String, Boolean \in TypeName$

$dom(Integer) = \mathbb{Z}$, $dom(Real) = \mathbb{R}$, $dom(Real^{+}) = \mathbb{R} \cup \{-\infty, +\infty\}$, $dom(String) = char$, $dom(Boolean) = \{true, false\}$

Each of these types is associated with its basic operations. For example, $Integer$ is associated with the basic arithmetic operators (+, -,×,÷) and $Boolean$ is associated with the logical connectives. These basic are rather standard, we assume their properties in standard mathematics. The formal semantics of predicates, expressions and related logic concepts are defined for Object-Z; we will not focus on this here.

$VarID$ is the set of variable identifiers.

$IReference$ is the set of input ports references

$OReference$ is the set of output ports references.

$ModelReference$ is the set of model references

$ConfID$ is the set of configurations identifiers

$ClassName$ is the set of all possible class names

$OpName$ is the set of all possible operation names

$Object$ is the universe of objects

$ObjectId$ is the set of object identifiers

III.5.1.1 Variables

A variable is of the form $var =< identifier, domain, initValue, value >$ where:

- $identifier \in VarID$, identify uniquely the variable in the scope
- $domain$ is the set of possible values of the variable. $\exists\ t \in TypeName$ such that $domain \subseteq dom(t)$. The following function associates a type to each variable identifier $type: VarID \rightarrow TypeName$
- $initValue \in domain$ is the initial value provided at initialization (if not, the initial value will be the default one of the domain)
- $value \in domain$ is the current value of the variable

III.5.1.2Variable assignment

A variable assignment is a function that assigns to each variable a value in the domain of that variable $v: VarID \mapsto dom(Type(v))$ ($or\ v: VarID \mapsto v.domain$)

A variable domain can be defined by primitive data types such as: Boolean, integer, double, Strings etc. and user defined HILLS classes.

III.5.1.3 Operations

An operation is of the form $op =< input, output, preCondition, postCondition\ >$

$input$ is the list of input parameters (parameters are also variables),

$output$ is the list of output parameters,

$preCondition$ is the set of predicates that must be satisfied to make the operation possible

$postCondition$ is the set of predicates that must be satisfied after the call of the operation.

III.5.1.4 HiLLS classes

A HILLS class is defined by $C =< Name, VAR_C, OPS_C, Init_C >$ where:

- $Name \in ClassName$ is the name of the class
- VAR_C is the set of attributes (attributes are variables) of the class C,
- OPS_C is the set of methods (or operations) of the class C
- $Init_C$ is the set of possible initial states for instance objects.

III.5.1.5 Classes and objects

A class is a collection of objects. This means that we can create objects sharing the same features (attributes and operations) from a class. Objects are different by their identifiers. Each object has a state which is defined by actual values of is attributes.

An object is of the form $obj =< id, class, as >$ where

- $id \in ObjectId$ is the identifier of the object
- $class$ is the class of the object
- as is the set of assignment to variables of $class$

$ClassToObjID: ClassName \rightarrow 2^{ObjectId}$, is a function that assigns a set of object identifiers to a class name.

$ClassToObj: ClassName \rightarrow 2^{Object}$, is the function that assigns a set of object to a class name

$ObjToClass: Object \rightarrow ClassName$ assigns an object to a unique class

$ObjIdToClass: ObjectId \rightarrow ClassName$ assigns an object identifiers to a class

$IdToObj: ObjectId \rightarrow 2^{Object}$, is the function that assigns a set of object to an object identifier.

An object may have temporary variables i.e. parameters of operations and variables declared only in operations body. The evaluation of an object state depends on the evaluations of its instance variables and temporary variables.

III.5.1.6 Inheritance

A class may inherit for another class. A class may have different subclasses.

$parent: Class \rightarrow IP\ Class$

Let $Cl \in Class$, $\forall c \in parent(Cl)$ we have $VAR_c \subseteq VAR_{Cl}$ and $OPS_c \subseteq OPS_{Cl}$. A class inherits the set of attributes and operations of its parent.

III.5.1.7 Ports

Ports are like external variables, we make here a difference between them because of their particular role.

A port is of the form $port =< reference,\ domain,\ modelReference >$ where $reference \in IReference \cup OReference$ is the reference (or unique identifier) of the port. If $reference \in IReference$, the port is called input port. If $\in OReference$, the port is called output port. The $reference$ attribute defines the role played by the port: receiving or sending events.

Input ports in a model must have different references

Output ports in a model must have different references

From now, we will refer to elements of a tuple by using the dot notation (commonly used in other languages). For example if p is a port, $p.reference$ gives access to the reference of the port.

III.5.1.8 Events

An Event is of the form $ev = < portReference, domain, value >$ where $portReference \in IReference \cup OReference$ is the reference of the port, $domain$ is the type of the event and $value$ is the value received on the associated port. If $portReference \in IReference$, the event is called input event (or external event). If $portReference \in OReference$ it is called output event (or internal event).The $portReference$ defines the source or target of an event.

Many events can have the same port reference at the same time. This means that a port can receive zero or finitely many events at the same time. Events are instantaneous, their occurrence are known in the operational level.

IB^b et OB^b are sets of bags of input and output events.

The notation for an external event on input port p is of the form $p.x | x \in dom(p)$.

Similarly, notation for an output event on output port q is of the form $q^\wedge.y | y \in dom(q)$

III.5.1.9 Configurations

A configuration is of the form $c = < Name, PredPart, Act >$ where

$Name$ is the name of the configuration,

$PredPart$ is the predicate part of the configuration, and

Act is the sequence of activities to be executed in the configuration.

We pose $SC = UC \cup CC$ where UC is the set of unitary configurations and CC is the set of composite configurations (these sets are disjoints $UC \cap CC = \emptyset$).

The set of configuration is like a control graph of the system proposed by the modeler.

We note $RootConf$ as the set of root configurations. Root configurations are top level configurations in the graphical representation of the model. A root configuration can be unitary or composite.

We introduce the relation $\subset$ between configurations such that $s \subset s'$ if s is sub-configuration of s'. The relation verifies:

$S \in RootConf \Longrightarrow \forall s \in SC, (S \not\subset s)$. A root configuration don't have parent

$\forall s \in SC, (s \not\subset s)$. The relation is not reflexive

$\forall s, s' \in SC, (s \subset s' \Longrightarrow s' \not\subset s)$ the relation is not symmetric

$\forall s \in SC, s \notin RootConf \Longrightarrow \exists! s' \in SC \mid s \subset s'$. Every non root configuration has a parent.

$\forall s \in SC, s \notin RootConf \Longrightarrow \exists s' \in RootConf \mid s \subset^* s'$. Reachability of non-root configuration from a root configuration,

$\forall s \in SC, s \not\subset^+ s$. Every chain is acyclic

$\subset^*$ and $\not\subset^+$are respectiveley the reflexive and anti-reflexsive transitive closure of $\subset$.

If two configurations s and s' are such that $s \subset s'$,s' is called the parent of s and s a child of s'.

We define the following functions:

$parent: SC \rightarrow SC$ s the function which associate to each configuration its parent such that

$parent(s) = s' \Leftrightarrow s \subset s'$.The domain of $parent$ is $dom(parent) = SC - RootConf$.

$parent^{*}: SC \rightarrow IPSC$ is the function that associate to each configuration the set of its ascendants parents (i.e. its parent and the parent of its parent and so on).

$parent^{*}(s) = \{s' \in SC | s \subset^{*} s'\}. \forall s \in SC, parent(s) \subseteq parent^{*}(s)$.

$subConf: SC \rightarrow IPSC$ is the function that associate to each configuration its direct sub-configurations defined by:

$\forall s \in SC\ subConf(s) = \{s' \in SC | s' \subset s\}$.

If s is an unitary configuration then $subConf(s) = \emptyset$, else $subConf(s) \neq \emptyset$.

$subConf^{*}: SC \rightarrow IPSC$, is the function which associates to each configuration its related sub-configurations (its direct sub-configuration, the sub-configurations of its sub-configurations, and so on).

$subConf^{*}(s) = \{s' \in SC | s' \subset^{*} s\}. \forall s, subConf(s) \subseteq subConf^{*}(s)$.

III.5.1.10 Predicates

Terms: $t = x | c | f(t_1, \dots, t_n)$

x is a variable, c is a constant or proposition and f is a function symbol with arguments which are terms.

$f = p | {\sim}p | p \wedge q | p \vee q | \forall x\ p | \exists x\ p$.

p and q are predicate terms. The objective here is not to be complete about the notion of predicates. The abstract syntax gives the complete structure of predicates usable in the language.

$C(V)$ is the set of constraints (predicates) on state variables.

$\psi: S \rightarrow \mathbb{P}C(V) \cup C(coupling)$ is a function which associate to each configuration a set of constraints (to be taken in conjunctive form $\bigwedge_{c \in \psi(s)} c$) on state variables or couplings. The set of all constraints of a configuration is defined by the function ψ^{*} as follows:

$S \in SC, \psi^{*}(S) = \psi(S) \cap (\bigcup_{s \in subConf(S)} \psi(s)) \cap (\bigcap_{s \in parent^{*}(S)} \psi(s))$ (to be taken in the form $(\bigwedge_{c \in \psi(s)} c) \wedge (\bigvee_{s \in subConf(S)} \bigwedge_{c \in \psi(s)} c) \wedge (\bigwedge_{s \in parent^{*}(S)} \bigwedge_{c' \in \psi(s)} c')$) i.e. the set all constraints of S is equal to the intersection of the set of constraints directly associated to S, the union of the sets of constraints directly associated to its sub-configuration and the set of constraints directly associated to its parent.

In particular if S is a unitary configuration $\psi^{*}(S) = \psi(S) \cap (\bigcap_{s \in parent^{*}(S)} \psi(s))$

if S is a root configuration $\psi^*(S) = \psi(S) \cap (\bigcup_{s \in subConf(S)} \psi(s))$

let $s \in SC$, a configuration, we note $\bar{s} = \{a \in SM | \vDash of\ \psi^*(s)\}$, the set of states satisfying the constraints of s.

The function ψ^* verifies the following conditions: $\bigcup_{s \in RooConf} \bar{s} = SM \ \wedge \ \forall s, s' \in RootConf, \bar{s} \cap \overline{s'} = \emptyset$ for $s \neq s'$

$\forall S \in CC, \bigcup_{s \in subConf(S)} \bar{s} = \bar{S} \ \wedge \ \forall s, s' \in subConf(S), \bar{s} \cap \overline{s'} = \emptyset$ for $s \neq s'$.

The root configurations constitute a partition of the state space. For each composite configuration, the sub-configurations constitute a partition of the subspace associated to it. These constraints make it possible to avoid ambiguities.

III.5.1.11 HiLLS model classes

$MC_{HILLS} =< DataStruct, StateMachine, Com >$. Com is the possible set of components.

If Com is empty the model class represents a family of unitary models. If Com is non empty the model class represents a family of composite models i.e. composed of sub-components.

$DataStruct =< reference,\ IB, OB, V, OP, P >$

$reference \in ModelReference$ is the unique identifier of the model

IB is the input interface (the possibly empty and finite set of input ports)

$X = \bigcup_{p \in IB} \{(p, v) | v \in dom(p)\}$ is the input space with respect to IB

OB is the output interface (the finite possibly empty set of output ports)

$Y = \bigcup_{q \in OB} \{(q, v) | v \in dom(q)\}$ is the output space with respect to OB

V is the set of state variables;

P is the set of parameters

OP is the set of operations. An operation is a function that modifies the values of one or more variables. Each operation op possesses a pre-condition ($pre\ op$) and a post-condition ($pos\ op$). The pre-condition $pre\ op$ represents the necessary conditions for the operation to be executed and the post-condition represents the effects of the operations on state variables.

$StateMachine =< ConfID, SC, T_{int}, T_{ext}, T_{conf}, \psi, \varphi, Act, T >$:

$\psi: S \rightarrow \mathbb{P}C(V) \cup C(coupling)$ is the function that associate to each configuration a set of constraints.

Act is the set of activities (an activity in HILLS doesn't modify the state variables);

$\varphi: SC \ \rightarrow \ Act$ is the function that associate to each configuration an activity. When an activity is associated to a composite configuration, it is the same activity which is carried out in its sub-

configurations i.e. $\forall S \in CC\ (\varphi(S) = \alpha \Longrightarrow \forall s \in subConf^*(S), ops\ \varphi(s) \supset ops\ \alpha)$. The activity of a composite configuration is shared by all its sub-configurations (Activities of parent configurations must be executed in sub-configurations).

$T: SC \to (IP\ SM \to \mathbb{R}^+ \cup \{+\infty\})$ tel que $\forall s \in SC, T(s): \bar{s} \to \mathbb{R}^+ \cup \{+\infty\}$

$a \in \bar{s}$, $T(s)(a)$ will be noted $T_s(a)$.

$\forall S \in CC$, $\forall s \in subConf^*(S), \forall a \in \overline{s}, T_s(a) = T_S(a))$ if T_S is defined.

$Effect: OP \times Eval(VarID) \to Eval(VarID)$ is the function who indicates for each operation how the evaluations of the variables change with the application of this operation. We assume here that the effect of an operation is deterministic

The effects of a sequence of operation ops on state is the succesive application of all the operation of ops. The evaluation is done atomically. Formally, let $Ev_0 = a$ the current variables evaluation on which the operations of ops will be applied.

$\forall 0 < i < \#ops - 1, Effet(ops(i), Ev_{i-1}) = Ev_i$

$Effet(ops, a) = Ev_{\#ops-1}$

We note $SM = \prod_{v \in VarID} dom(v) = \prod_{v \in VarID} S_v$ the set of all atomic states (the state space).

$\psi: S \to \mathbb{P}C(V)$ associate to each configuration a set of predicates on state variables (only state variables). Coupling constraints are not used in HILLS unitary models specifications.

$T_{int} \subseteq SC \times C(V) \times OB^b \times \mathbb{P}\ OP \times SC$ is the set of internal transitions

The internal transition $(s, c, y, ops, s') \in T_{int}$ will be noted $s \xrightarrow{<c,y,ops>}_i s'$,

In particular $(s, <>, l, ops, s') \in T_{int}$ will be noted $s \xrightarrow{<l,ops>}_i s'$ (where <> represents the absence of constraint).

$C(e)$ is the set of constraints α on the elapsed time e of the form:

$\alpha \coloneqq e{\sim}t | e - t{\sim}t' | e + t{\sim}t' | e * t{\sim}t' | \alpha_1 \wedge \alpha_2 | \alpha_1 \vee \alpha_2$ où $\sim\ \in \{=, \leq, <, \geq, >\}$, e, t, t' are positive real numbers, -, +, * are subtraction, addition and multiplication operators in $\mathbb{R}$, α_1 and α_2 are conditions on e.

$T_{ext} \subseteq SC \times C(V) \times IB^b \times \mathrm{C}(e) \times \mathbb{P}\ OP \times SC$ is the set of external transitions

The external transition$(s, m, x, c(e), ops, s') \in T_{ext}$ will be noted $s \xrightarrow{<m,x,c(e),ops>}_e s'$.

In particular $(s, <>, x, c(e), ops, s') \in T_{ext}$ will be noted $s \xrightarrow{<x,c(e),ops>}_e s'$.

$T_{conf} \subseteq SC \times C(N) \times IB^b \times \mathrm{OB^b} \times \mathbb{P}\ OP \times SC$ is the set of confluent transitions.

Unicity of transitions and coherences of outputs in internal and confluent transitions (avoiding non determinism).

Conditions to avoid external non-determinism:

Let $(s, c, x, c(e), ops, s') \in T_{ext}$ and $(s, c', x', c'(e), ops', s'') \in T_{ext}$ (same source configuration) :

$$(c \Leftrightarrow c') \wedge (x = x') \wedge \big(c(e) \Leftrightarrow c'(e)\big) \Longrightarrow s' = s'' \wedge Effet(ops, a) = Effet(ops', a), \forall a \in \bar{s} \cap \{b \in SM | b \vDash c\}$$

$$s' \neq s'' \Longrightarrow (c \oplus c') \vee (x \neq x') \vee \big(c(e) \oplus c'(e)\big)$$

Conditions to avoid internal non-determinism

Let $(s, c, y, ops, s') \in T_{int}$ and $(s, c', y', ops', s'') \in T_{int}$ (same source configuration):

$$(c \Leftrightarrow c') \Longrightarrow s' = s'' \wedge (y = y') \wedge Effet(ops, a) = Effet(ops', a), \forall a \in \bar{s} \cap \{b \in SM | b \vDash c\}$$

$$s' \neq s'' \Longrightarrow (c \oplus c')$$

Conditions to avoid confluent transition non-determinism.

Let $(s, c, x, y, ops, s') \in T_{conf}$ and $(s, c', x', y', ops', s'') \in T_{conf}$ (same source configuration):

$$(c \Leftrightarrow c') \wedge (x = x') \Longrightarrow s' = s'' \wedge (y = y') \wedge Effet(ops, a) = Effet(ops', a), \forall a \in \bar{s} \cap \{b \in SM | b \vDash c\}$$

$$s' \neq s'' \Longrightarrow (c \oplus c') \vee (x \neq x')$$

Output coherence in internal and confluent transitions

$(s, c, y, ops, s') \in T_{int}$ and $(s, c', x', y', ops', s'') \in T_{conf}$ (we have the same source configuration):

$$(c \Leftrightarrow c') \Longrightarrow (y = y')$$

III.5.1.12 Traces

A trace is a finite or infinite alternate sequence of state and events.

$t = s_0 \alpha_1 s_1 \alpha_2 \ldots \alpha_n s_n \ldots$ with $s_i \xrightarrow{\alpha_{i+1}}_e s_{i+1}$or $s_i \xrightarrow{\alpha_{i+1}}_i s_{i+1}$ or $s_i \xrightarrow{\alpha_{i+1}}_c s_{i+1}$ $\forall i \geq 0$ (α_i is a bag of events). A state s if reachable if there exists a finite trace $s_0 \alpha_1 s_1 \alpha_2 \ldots \alpha_n s_n$ such that $s_n = s$.

Com $=< \{M_d\}_{d \in MReference}, EIC, IC, EOC >$

The M_d are HILLS unitary or composite models

$EIC \subseteq \big\{\big(\big((M, p), (k, d)\big)\big)/p \in IReference, k \in IReference_d, d \in MReference\big\}$where, $IReference_d$ is the $IReference$ of the sub-model of reference d.

$EOC \subseteq \big\{\big((l, d), (q, M)\big) | l \in ORefrence_d, d \in MReference, q \in OReference\big\}$

$IC \subseteq \{((l,d),(k,d'),), l \in OReference_d, d, d' \in MReference, k \in IReference_d\}$

A composite model defines a containment hierarchy which roots is the composite model itself.

We define the relation $\Subset$ between models.

$m \Subset m'$ if m is a sub-model of m' i.e. $m \in \{M_d\}_{d \in MReference_{m'}}$

The relation verifies:

$M \in CompositeModel \Longrightarrow \forall m \in \{M_d\}_{d \in MReference}$ $, m \Subset M$. A root configuration don't have parent

$\forall m \in Model, (m \Subset m)$. The relation is not reflexive

$\forall m, m' \in Model, (m \Subset m' \Longrightarrow \neg(m' \Subset m))$. The relation is not symmetric

$\forall m \in Model, m \notin RootModel \Longrightarrow \exists! \, m' \in CompositeModel \mid m \Subset m'$. Every non root model has a container.

$\forall m \in Model, m \notin RootModel \Longrightarrow \exists m' \in RootModel \mid m \Subset^* m'$. Reachability of non root model from a root model.

$\forall m \in Model, \neg(m \Subset^+ m)$. Every chain is acyclic

$\Subset^*$ and $\Subset^+$are respectiveley the reflexive and anti-reflexsive transitive closure of $\Subset$.

If two models m and m' are such that $m \Subset m'$,m' is called the container of m and m a sub-model of m'.

$subModels: Model \rightarrow \mathbb{P}\ Model$ is a function which associate to each model its sub-models.

$\forall m \in Model\ subModels(m) = \{m' \in Model | m' \Subset m\}$

By definition we have: $m \in UnitaryModel \Longrightarrow subModels(m) = \emptyset$.

For a composite model CM_{HILLS}, $subModels(CM_{HILLS}) = \{M_d\}_{d \in MReference}$

Definition: $|subModels(m)|$ is called the high of the model m.

$subModels^*: Model \rightarrow \mathbb{P}\ Model$ determine the of all models in the containment hierarchy of a model.

$\forall m \in Model, subModels^*(m) = \{m' \in Model | m' \Subset^* s\}$.
$\forall m, subModels(m) \subseteq subModels^*(m)$.

$m \in UnitaryModel \Longrightarrow subModels^*(m) = \emptyset$

$container: Model \rightarrow Model$ s the function which associate to each model its container such that

$container(m) = m' \Leftrightarrow m \Subset m'$.

$container^{*}: Model \rightarrow IPModel$ is the function that associate to each model the set it's related container (i.e. its container and the container of his container and so on).

$container^{*}(m) = \{m' \in Model | m \Subset^{*} m'\}. \forall m \in Model, container(m) \subseteq container^{*}(m).$

III.5.1.13 Parameterized model

A HiLLS model has may have parameters. Non-parameterized models (non-initialized models) can be created form parameterized models by assigning specific values to parameters.

A model is a model with specific values for parameters:

$NIM_{HILLS} = PM_{HILLS} \cup < PInit >$

PM_{HILLS} is a HILLS model (unitary or composite)

$PInit$ is the set of assignments to parameters.

$J: PM_{HILLS} \times \prod_{p \in P} dom(p) \rightarrow NIM_{HILLS}$ is the function that create a non-initialized model from a parameterized model. J is defined as follows:

$J(M, \alpha_1, \dots, \alpha_n) = M \cup < \{p_1 = \alpha_1, \dots, p_n = \alpha_n\} >, \forall M \in PM_{HILLS}, \alpha_i \in dom(p_i)$

$(PInit = \{p_1 = \alpha_1, \dots, p_n = \alpha_n\})$

III.5.1.14 Initialization of a model

The init schema of a HiLLS gives the set of possible initial states. An initialized model can be defined by choosing an initial state in the set of states defined by the init schema.

An imodel is a model with a defined initial state:

$IM_{HILLS} = NIM_{HILLS} \cup < S_0, Init >$

NIM_{HILLS} is a HILLS non-initialized model

$S_0 \in SC$ is the initial configuration of the model.

$Init$ is the set of assignments defining the initial state in the initial configuration S_0

IM_{HILLS} is a HILLS non-initialized model

$I: NIM_{HILLS} \times \prod_{v \in V} dom(v) \rightarrow IM_{HILLS}$ is the function that create an initialized model from a non-initialized model. I is defined as follows:

$I(M, \alpha_1, \dots, \alpha_n) = M \cup < S_0, \{v_1 = \alpha_1, \dots, v_n = \alpha_n\} >$

$(Init = \{v_1 = \alpha_1, \dots, v_n = \alpha_n\} \in \overline{S_0})$

III.5.1.15 Domain model

A domain model (or metamodel of the problem domain) in HiLLS is a structure of the form $CG = (ClassNode, ModelNode, PortNode, PortModel, Assoc, Inherit, CE, MC)$ where

- $ClassNode$ is the set of class nodes ,
- $ModelNode \subseteq ClassNode$ is the set of model class nodes,
- $PortNode$ is the set of port nodes,
- $PortModel = \{(P, M)_{p \in PortNode, M \in ModelNodel}\}$ is the set of containment relations between ports and models,
- $Assoc = \{(C, C') \in ClassNode^2 | C \in Assoc(C')$ is the set of association edges
- $Inherit = \{(C, C') \in ClassNode^2 | C = parent(C')\}$ is the set of inheritance relations.
- $CE = (CE(P, P'), p)_{P,P' \in PortNode, p \in Predicate}$ is the set of communication relations between model nodes (instances of communication edges are coupling edges), and
- $MC = \{(M, M') | M' \in subComp(M))\}_{M,M' \in ModelNode}$ the set of model containment relations. $MC \subseteq Assoc$.

Classes and associations can be instantiated by objects and links respectively.

III.6.2 Closure under coupling of HILLS

The HiLLS is closed under coupling as the DEVS formalism is. We give here the justification is of this important property in system theory which guarantees hierarchical construction of models.

Let $CM =< DataStruct, StateMachine, Com >$ a HiLLS model with $Com \neq <>$. The closure under coupling property states that there exists an unitary HiLLS model $UM =< DataStruct', StateMachine', Com' = \emptyset >$which behavior is equivalent to that of CM.

$DataStruct =< reference, IB, OB, V, OP, P >$

$StateMachine =< ConfID, SC, T_{int}, T_{ext}, T_{conf}, \psi, \varphi, Act, T >$

$Com =< MReference, \{M_d\}_{d \in MReference}, EIC, IC, EOC >$

We obtain UM's constituents as follows:

$DataStruct' =< reference', IB', OB', V', OP', P' >$

Where,

$reference' = reference$

$IB' = IB$

$OB' = OB$

$V' = V \cup (\bigcup_{d \in D} V_d \cup \{e_d\})$ where e_d the implicit elapsed time variable of subcomponents M_d.

$d \in MReference,$

$OP' = OP \cup \left(\bigcup_{d \in MReference} OP_d\right)$ operations

$P' = P \cup \left(\bigcup_{d \in MReference} P_d\right)$ parameters

$StateMachine' =< ConfID', SC', T_{int}', T_{ext}', T_{conf}', \psi', \varphi', Act', T' >$

$ConfID' = ConfID \times \prod_{d \in MReference} ConfID_d$ (naming convention for the element of SC')

$SC' = \prod_{d \in MReference} SC_d$

$C \in SC', \psi'(C) = \psi(\pi_{CM}(C)) \cup \left(\bigcup_{d \in MReference} \psi_d(\pi_d(C))\right)$ (π_d is the projection that returns the configuration of the components of name d and ψ_d is the function that determines the constraints associated to a configuration of the component of name d).

$C \in SC', \varphi'(C) = \varphi(\pi_{CM}(C)) \cup \left(\bigcup_{d \in MReference} \varphi_d(\pi_d(C))\right)$

$Act' = Act \cup \left(\bigcup_{d \in MReference} Act_d\right)$

III.5.2.1 Time advance function

$\forall s \in SM, T_{conf(s)}(s) = \min_{d \in MReference}\left\{T_{conf(s_a)}(s_d) - e_d\right\}$

We define the following sets:

- $IMM(s) = \left\{d \in Mreference | \sigma_d = T_{conf(s)}(s)\right\}$: The set of references of components those configuration duration expired; It is the set of imminent components
- $INF(s) = \left\{d \in D \,/\, \exists i \in I(d), i \in IMM(s) \wedge x_d^b \neq \emptyset\right\}$ where

$x_d^b = \{\lambda_i(s_i) / i \in IMM(s) \cap I(d)\}$

The set of references of influenced components

- $CONF(s) = IMM(s) \cap INF(s)$ the references of confluent components
- $INT(s) = IMM(s) - INF(s)$ determines components that will execute an internal transition
- $EXT(s) = INF(s) - IMM(s)$ this set determines the no-imminent and influenced components;
- $UN(s) = D - IMM(s) - INF(s)$. These components will not execute any transition.

III.5.2.2 The internal transition relation

Let $C = (c, (\dots, c_d, \dots)) \in ConfID\ D = MReference$

$(c, \dots, c_d, \dots) \xrightarrow{p_c \wedge \left(\wedge_{d \in D} p_{c_d}\right), y_c \cup \left(\cup_{d \in D} y_{c_d}\right), ops_c \cup \left(\cup_{d \in D} ops_{c_d}\right)}_i (c', \dots, c'_d, \dots) \in T'_{int}$

where

$c \xrightarrow{p_c, y_c, ops_c} c' \in T_{int}$

$c_d \xrightarrow{p_{c_d}, x_{c_d}, y_{c_d}, ops_{c_d}} c'_d \in T_{conf d} \wedge (e'_d = 0)\ if\ d \in CONF(s)$

$$c_d \xrightarrow{p_{c_d}, x_{c_d}, ce(c_d), ops_{c_d}} c'_d \in T_{extd} \wedge \left(e'_d = e_d + \boldsymbol{T_{c_d}(s)}\right) \; if\ d \in EXT(s)$$

$$c_d \xrightarrow{p_c, y_c, ops_c} c'_d \in T_{intd} \wedge (e'_d = 0) \; if\ d \in INT(s)$$

$$c'_d = c_d \wedge \left(e'_d = e_d + \boldsymbol{T_{c_d}(s)}\right) \; if\ d \in UN(s)$$

III.5.2.3 The external transition relation

Let $C = (c, (\dots, c_d, \dots)) \in ConfID$

$$(c, \dots, c_d, \dots) \xrightarrow{p_c \wedge \left(\wedge_{d \in D} p_{c_d}\right), x_c \cup \left(\cup_{d \in D} x_{c_d}\right), ce_c \wedge \left(\wedge_{d \in D} ce_{c_d}\right), ops_c \cup \left(\cup_{d \in D} ops_{c_d}\right)}_i (c', \dots, c'_d, \dots) \in T'_{ext}$$

where

$$c \xrightarrow{p_c, x_c, ce_c, ops_c} c' \in T_{ext}$$

$$c_d \xrightarrow{p_{c_d}, x_{c_d}, ce(c_d), ops_{c_d}} c'_d \in T_{extd} \wedge \left(e'_d = e_d + \boldsymbol{T_{c_d}(s)}\right) \; if\ UM \in I(d) \wedge x_{c_d} \neq \emptyset$$

$c'_d = c_d \wedge \left(e'_d = e_d + \boldsymbol{T_{c_d}(s)}\right)$ otherwise

III.5.2.4 The confluent transition relation

- $INF'(s) = \{d \in D \,/\, \exists i \in I(d), (i \in IMM(s)\ ou\ N \in I(d)) et\ x_d^b \neq \emptyset\}$
  $x_d^b = \{\lambda_i(s_i) / i \in IMM(s) \cap I(d)\} \cup \{x_d / x_d \in x^b\ et\ M \in I(d)\}$
- $CONF'(s) = IMM(s) \cap INF'(s)$
- $INT'(s) = IMM(s) - INF(s);$
- $EXT'(s) = INF(s) - IMM(s)$

$$(c, \dots, c_d, \dots) \xrightarrow{p_c \wedge \left(\wedge_{d \in D} p_{c_d}\right), x_c \cup \left(\cup_{d \in D} x_{c_d}\right), y_c \cup \left(\cup_{d \in D} y_{c_d}\right), ops_c \cup \left(\cup_{d \in D} ops_{c_d}\right)}_i (c', \dots, c'_d, \dots) \in T'_{conf}$$

$$c \xrightarrow{p_c, x_c, y_c, ops_c} c' \in T_{conf}$$

$$c_d \xrightarrow{p_c, y_c, ops_c} c'_d \in T_{intd} \wedge (e'_d = 0) \; if\ d \in INT'(s)$$

$$c_d \xrightarrow{p_{c_d}, x_{c_d}, y_{c_d}, ops_{c_d}} c'_d \in T_{confd} \wedge (e'_d = 0) \; if\ d \in CONF'(s)$$

$$c_d \xrightarrow{p_{c_d}, x_{c_d}, ce(c_d)(e_d + \boldsymbol{T_{c_d}(s)}), ops_{c_d}} c'_d \in T_{extd} \wedge (e'_d = 0) \; if\ d \in EXT'(s)$$

$c'_d = c_d \wedge \left(e'_d = e_d + \boldsymbol{T_{c_d}(s)}\right)$ otherwise

III.7 HiLLS Usability Assessment

We present here a survey on the usability of HiLLS as a systems modeling langauge. We followed the methodological principles of [Ringert et al. 2013].

Our study involved 48 students in Computer Science:

- 12 from the MSc in Computer Science at the African University of Science and Technology (AUST, Abuja, Nigeria); this degree is carried out in 18 months and admitted students are holding BSc in Electrical and Computer Science; therefore, they have a good knowledge of UML, Object-Orientation and discrete mathematics, but they don't have a significant background in simulation or system theory; and
- 36 from the MSc in Web and Mobile Engineering at the Blaise Pascal University (UBP, Clermont Ferrand, France); this degree is carried out in 24 months and admitted students are holding BSc in mathematics or Computer science with sound knowledge of UML, Object-Orientation and discrete mathematics, but without any background in simulation or system theory.

We have built 12 modeling teams of 4 members each. Each team worked during one month (October 2015) on the modeling of three problems: the Traffic Light (TL), the ABP and the ATM. We distributed a survey to be completed by each team, and our analysis is based on the results of these questionnaires (all filled by all teams). The filling principle adopted by each team is to first fill the survey by each member, and then to summarize results in one unique survey after discussions on individual results and mutual agreement on the final results.

The survey consists of four groups of questions, which are then broken down by technology (UML, DEVS, HiLLS) and/or model built (TL, ABP, ATM), and for which the only possible answers are High, Medium and Low:

1. Time to learn the given technology
2. Cost in time of the building of the given model with the given technology
3. Degree of collaboration made possible within the team by the given technology in building the given model
4. Degree of confidence of the team in the model built with the given technology
5. Availability of software support for the given technology

Globally, students evaluated the time to learn HiLLS to be medium while the time to learn DEVS is high. UML is evaluated to be easy to learn (Figure 35). We think that the use of concrete syntax elements similar to that of UML reduced the complexity of learning HiLLS. Globally, the time spent to model TL, ABP and ATM is medium while it is globally low for UML and high for DEVS. The degree of involvement and discussion in modeling with HiLLS is high for UML and HiLLS while it is globally medium for DEVS. Globally, the confidence in created models is high for all the technologies. Tool support is high for UML while it is low for DEVS and HiLLS.

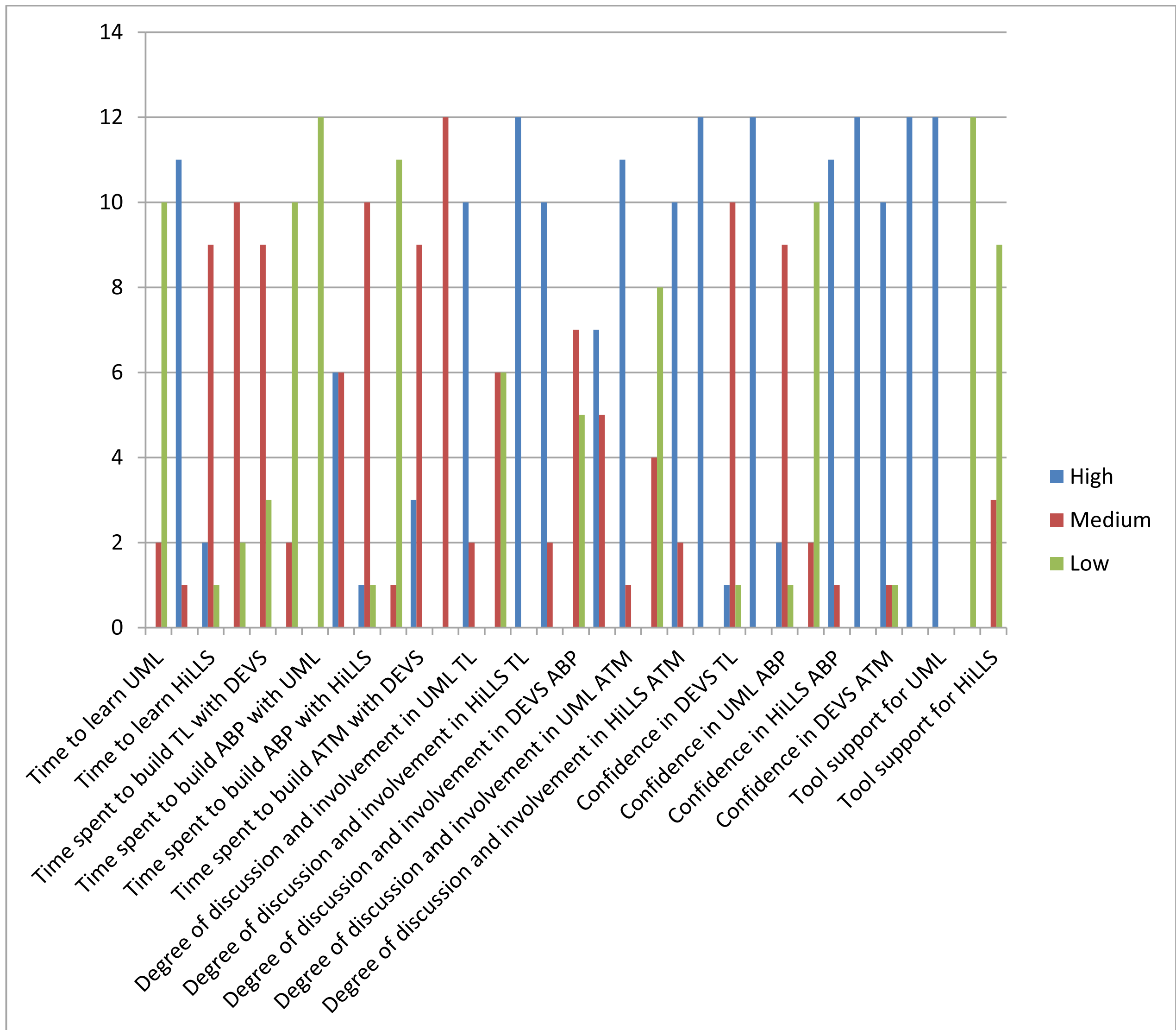

Figure 35. Survey results

The results of the survey show that HiLLS is accessible to students for modeling systems and promotes collaboration between team members [Aliyu et al. 2015 a].

III.8 Conclusion

We have introduced the HiLLS, a graphical modeling language that combines universal modeling paradigms from system theory and software engineering. It is meant to facilitate the specification of complex DESs models for the exhaustive investigation of static and dynamic properties using multiple established computational analysis techniques such as simulation, formal methods and enactment.

We have presented the abstract syntax, concrete syntax and set-theoretic semantics of HiLLS. We have shown the rationale behind the concepts expressed in its syntax, the corresponding

formalisms from which they have been adopted as well as the integration of the concepts to achieve the objectives of HiLLS. We used metamodel integration techniques to define the abstract syntax of HiLLS. Our proposal deals with the problems arising from the manual specifications of separate models of the same system (with disparate formalisms) for investigating different static and dynamic properties of system, i.e., (1) the herculean task of creating and maintaining consistencies between the different models, (2) limited model reuse, (3) constrained communication between stakeholders and (4) collaboration of tools.

From the perspective of language design, this work demonstrates that disparate languages can actually be combined in a symbiotic way to define a manageable integrated language that consistently subsumes all participants. And from system analysis perspective, it bridges the gap between domain experts and formal system analysis methodologies.

We are confident that with appropriate tooling and integration with legacy tools for the underlying formalisms, HiLLS would make the benefits of many analysis methodologies available to a wider audience especially among domain experts through high levels of abstractions, in addition to facilitating the process of formal investigation of system properties through reduced modeling tasks and improved model reusability. The next sections discuss the basis of the analysis aspects of HiLLS models using techniques like simulation, formal methods and enactment.

IV. Translational Semantics of HiLLS

IV.1 Introduction

We have defined HiLLS with multiple semantic domains. The semantics of HiLLS models in these domains are defined by translation. We call them the translational semantics of HiLLS. The translational semantics include the operational semantics for simulation, the operational semantics for enactment, the logical semantics to Z and the process-based semantics to CSP. While the operational semantics for simulation uses virtual/logical time to reveal and forecast the system's behaviors (dynamic properties) in specified experimental frames, the operational semantics for enactment uses clock-time execution to prototype the real-time behavior of the system. The logical semantics based on Z and CSP on the other hand offer the mathematical techniques to investigate statically (no time basis) the conformity of a model to requirements and rigorously prove or disprove logical assertions about the system's properties. Through increased understanding of systems and designs, these methods instinctively complement the conventional testing for software and hardware systems considering the fact that testing can only reveal the presence of errors in a system, but cannot guarantee their absence. The fact that the real system is not required for the investigations also adds to the economics of time and resources guaranteed by these techniques in the process of system development as design errors and faulty assumptions are detected and resolved in the early stages of the development process.

We presents in this chapter, the details of the translational semantics of HiLLS. The operational semantics for simulation is defined by semantic mappings to DEVS and DSDEVS presented in sections V.2.1 and V.2.2 respectively. At the other hand, the logical semantics of HILLS is expressed at three levels of abstraction by using different formal methods to capture the corresponding properties that can be specified. This is done so that we can derive different insights about the model. We can also take advantage of existing tools for formal analysis to derive properties of the model. Figure 36 shows the different mappings from HILLS to formal methods.

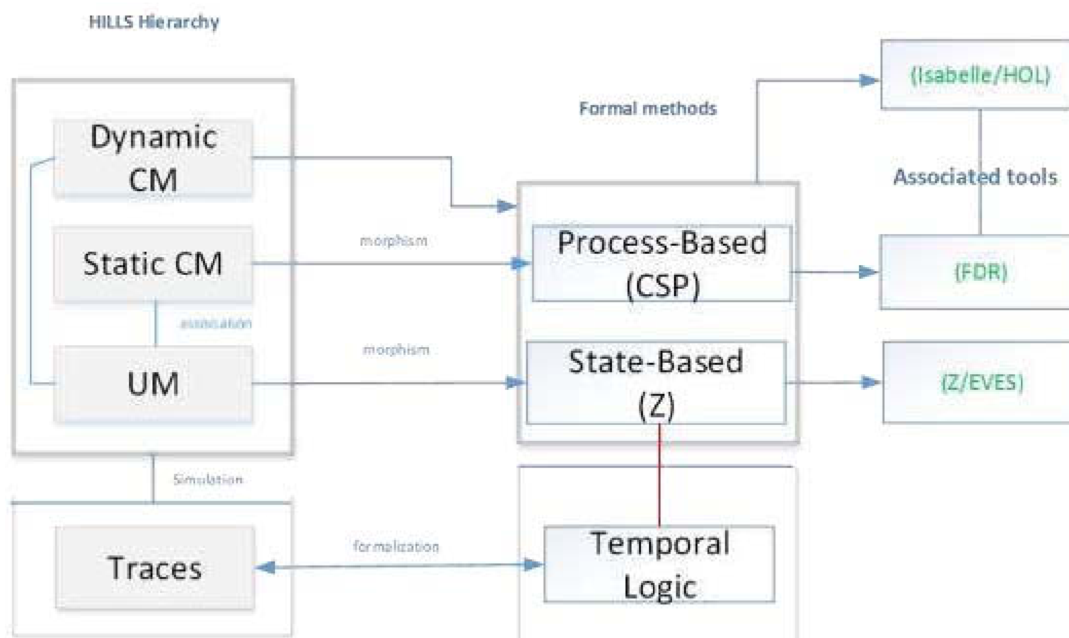

Figure 36. From HILLS to Formal Methods

The HILLS paradigm focuses on three levels of abstraction, which correspond respectively to CN (Coupled level), IOS (Input Output System) and IORO (Input Output Relation Observation) levels of Zeigler's hierarchy of system specification [Zeigler et al. 2000]:

- Composite model level (concerned with structural properties and functional couplings).
- Unitary model level (concerned with system dynamics characterized by states and state transitions.
- Traces level (concerned with traces and trajectories of the system).

System theory uses the concepts of decomposition and composition in systems engineering. Decomposition of a system consists in breaking a large system into smaller pieces that are easy to model and check. Each sub-system realizes a set of functionalities of the original system. Unitary testing is generally used to check and validate each component with some "level of confidence". The composition concept consists of building the model of the original system from smaller components that realize its complete behavior in a hierarchical manner. An error is to consider that, the system composed by individually correct sub-components is also correct. It is known that integration testing is also needed at system level in software engineering. Concurrency and interaction between components in the system can rise to problems (Synchronization, response time, reachability, fairness between components, starvation, livelock, global deadlock etc.) that are not possible to reveals at unitary component level. For example, using traffic lights at crossroads to control traffic flow needs also to take into account some synchronization and scheduling problems between the lights. The validity of individual light model is not sufficient to validate a model composed of concurrent lights at a crossroads. Composite system level verification and validation needs special formalisms that concentrate on concurrent and distributed systems verification. The specification of a composite model defines how the components are inter-connected and how they influence each other. This level does not require knowledge from what occurs inside the models which compose it. Properties of this level are those which target process-based formal methods like process algebras. At this level, a semantic mapping function maps a HILLS composite model onto concurrent processes defined in CSP (communicating sequential processes). We can use FDR refinement checker or Isabelle/HOL to check our models in this domain.

At the unitary level, HILLS highlights the concept of state and transitions. Stated-based formal methods like Z capture the properties of this level perfectly. At this level, a semantic function maps the state transition diagram onto Z notation. The Z specification language is known to be able to capture safety and liveness properties of state transition systems. We can take advantage of Z/EVES [Saaltink 2003] to check our models at this level.

At the traces level, the properties of system traces (which are expressed as footprints of the state transition system) are expressed in CTL (computational tree logic). A state transition system can have safety, invariants, liveness, and fairness properties. It is known that one temporal logic is not able to express all these properties. It is sometimes necessary to use another temporal logic to express adequately some family of properties.

IV.2 Operational Semantics for Simulation

We recall that the operational semantics maps a language into structures of an abstract execution environment. It describes how every valid expression of the language is interpreted in terms of

successive steps giving its value. It is composed of rules which describe the effects of the constructs of the language. An operational semantics allows the description of the evolution of a model during its execution. Transition systems are mathematical foundation of operational semantics of languages. Structural Operational Semantics (SOS) [Plotkin 1981] is an operational semantics definition techniques based on a transition system where states are valid predicates of configurations of the systems. The source of a transition is called premise and the target is called consequence. SOS is generally used in grammar-based language engineering.

We define the operational semantics of HILLS by a semantic mapping to DEVS (Figure 37).

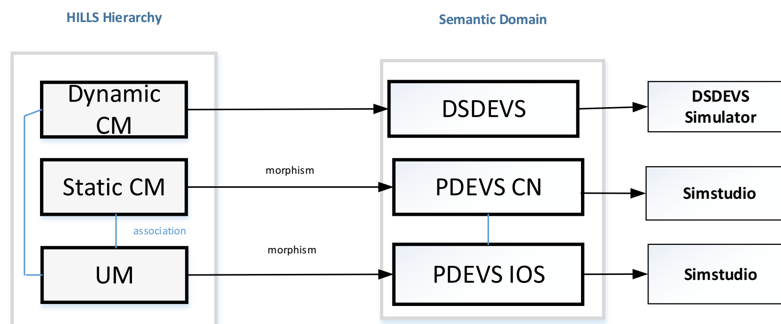

Figure 37. HiLLS operational Semantics mappings

IV.2.1 Semantic mapping to DEVS

The set of models specifiable by HiLLS can be split into two subsets: the subset of static structure models and the subset of dynamic structure models. The original DEVS formalism is only capable of simulating static structure models while DSDEVS is its extension that is capable of simulating dynamic structure models.

We present in this section the semantic mapping of HiLLS static structure models to DEVS. We show here how to build an equivalent DEVS model of a given HILLS model with static structure.

DEVS, being a mathematical formalism solely for system specification has no specific constructs for representing objects. Since the formalism also does not prescribe any concrete syntax, the user may take advantage of the freedom to represent and object as mathematical structure with its essential attributes and operations constituting the elements of the structure. HiLLS operations are also specified as mathematical functions that may be called from the DEVS-specific functions such as the transition and output functions.

IV.2.1.1 HiLLS unitary model to DEVS atomic model

HSystem translates to Atomic or Coupled DEVS models for simulation. A HSystem with empty hComponents translates to a atomic model; otherwise, it translates to coupled model and HSystem.hComponnents.target refers to the set of individual HSystems that are instantiated to form the components of the HSystem in context in this case. HiLLS ports are used to build the set of input and output of the equivalent DEVS model.

A configuration translates to a subset of S, the state space. The sojourn times function of configurations in HiLLS translate to the time advance functions ta, of states in DEVS. Internal, External and Confluent transitions in HiLLS are extracted to build DEVS' δ_{int} δ_{ext} and δ_{conf} functions respectively. The output parts of Internal and Confluent transitions in HiLLS are used to build the λ function of DEVS.

DEVS, like many other conventional simulation formalisms do not take the state activities into account since the advancement of simulation time is virtual while real physical time is required to execute activities. So the mapping of HiLLS to DEVS do not consider the translation of activities associated to configurations.

Let $UM_{HILLS} =< DataStruct, StateMachine, Com >$ with $Com = \emptyset$ be an HILLS unitary model.
We recall that $DataStruct =< reference,\ IB, OB, V, OP, P >$ and $StateMachine =< ConfID, SC, T_{int}, T_{ext}, T_{conf}, \psi, \varphi, Act, T >$.

UM_{HiLLS} has an equivalent DEVS atomic model $A =< X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta >$ where:
$X = \{(p, v) | p \in IReference \wedge \exists d \in DomID, v \in dom(d) \wedge (\ p, d) \in IB\}$.
$Y = \{(q, v) | q \in OReference \wedge \exists d \in DomID, v \in dom(q) \wedge (\ q, d) \in OB\}$.
$S = SM = \prod_{v \in VarID} dom(v)$;
The internal transition function $\delta_{int}: S \to S$ is defined by:
$\forall s \xrightarrow{c,y,ops} s' \in T_{int}$ where c is condition, y is a bag of output events, ops is the set of operations applied then we have:
$\forall a \in \bar{s} \cap \{b \in S | b \vDash c \wedge \bigwedge_{op \in Ops} Pre\ op\}, \delta_{int}(a) = Effect(ops, a)$ such that $Effet(ops, a) \vDash \bigwedge_{op \in Ops} Pos\ op$ ($b \vDash c$ means that the state b verifies the conditions of the sequence c).
The external transition function $\delta_{ext}: Q \times X^b \to S$ is defined by:
$\forall s \xrightarrow{x,c,c(e),ops} s' \in T_{ext}$ where c is a condition, x is the bag of input events, $c(e)$ is the condition on the elapsed time e and ops is the set of operations, we have : $\forall a \in \bar{s} \cap \{b \in S | b \vDash c \wedge \bigwedge_{op \in Ops} Pre\ op\}, \delta_{ext}((a, e), x) = Effet(ops, a)$ such that $Effet(ops, a) \vDash \bigwedge_{op \in Ops} Pos\ op$

The confluent transition function $\delta_{conf}: S \times X^b \to S$ is defined by:
$\forall s \xrightarrow{x,c,y,ops} s' \in T_{conf}$ where c is a condition, x is the bag of input events, y is the bag of output events, and ops is the set of operations, we have: $\forall a \in \bar{s} \cap \{b \in S | b \vDash c \wedge \bigwedge_{op \in Ops} Pre\ op\}, \delta_{conf}(a, x) = Effet(ops, a)$ such that $Effet(ops, a) \vDash \bigwedge_{op \in Ops} Pos\ op$

The output function $\lambda: S \to Y^b$ is defined by:
$\forall s \xrightarrow{c,y,ops} s' \in T_{int}$: where c is a condition, y is the bag of output events, and ops is the set of operations,$\forall a \in \bar{s} \cap \{b \in S | b \vDash c \wedge \bigwedge_{op \in Ops} Pre\ op\}, \lambda(a) = y$
$\forall s \xrightarrow{c,x,y,ops} s' \in T_{conf}$ where c is a condition, x is the bag of input events, y is the bag of output events, and ops is the set of operations, we have $\forall a \in \bar{s} \cap \{b \in S | b \vDash c \wedge \bigwedge_{op \in Ops} Pre\ op\}, \lambda(a) = y$.

The time advance function $ta: S \rightarrow \mathbb{R}^{+} \cup \{+\infty\}$ is defined by:
$s \in SC, \forall a \in \bar{s}, ta(a) = T_s(a)$.

We proved by construction here that every HILLS unitary model UM_{HILLS}has an equivalent DEVS atomic model A.

IV.2.1.2 HiLLS composite model to DEVS coupled model

We now show how a HiLLS composition static structure composite model is mapped to an equivalent DEVS coupled. The equivalent DEVS model is defined by construction from the HiLLS model.

Let $MC_{HILLS} =< DataStruct, StateMachine, Com >$ with $Com \neq \emptyset$ be HiLLS static structure composite model. We recall that Com $=< \{M_d\}_{d \in MReference}, EIC, IC, EOC >$.

MC_{HILLS} has an equivalent DEVS coupled model $CM =< X, Y, D, \{M_d\}_{d \in D}, EIC, EOC, IC >$ where
$X = \{(p, v) | p \in IReference \wedge \exists d \in DomID, v \in dom(d) \wedge (p, d) \in IB\}$.
$Y = \{(q, v) | q \in OReference \wedge \exists d \in DomID, v \in dom(q) \wedge (q, d) \in OB\}$.

$D = \{d | Hcomponents.names\}$
$CM.\{M_d\}_{d \in D} = com.\{M_d\}_{d \in MReference}$
$CM.EIC = com.EIC$
$CM.IC = com.IC$
$CM.EOC = com.EOC$
This proves by construction that every HILLS composite static structure model CM_{HILLS}has an equivalent DEVS coupled model CM.

IV.2.1.3 Traffic Light example

We the traffic light presented in Figure 38. Its behavior extends the behavior of the main traffic light presented in Figure 22. It has the possibility of receiving inputs from a controller through *control* port that switch it on or off. The *signal* port constitutes the single output port, which makes it possible to send the color corresponding to current configuration of the traffic light. The various configurations of the traffic light model are: *off*, *move*, *brake*, *stop*, *shutting-down*, *booting*. The configurations *shutting-down* and *booting* are transitory configurations, i.e. their lifespans are null. For any configuration different from *off*, when it receives the external event of value 0, it makes an external transition to pass to the *shutting-down* configuration in which it sends *Black* as output before making an internal transition immediately to pass to the *off* configuration. While being in the *off* configuration, if it receives an external event of value 1, it makes an external transition towards the *booting* configuration before making an internal transition to the *move* configuration. Without external event, the traffic light behaves like the MainTL presented in Figure 22. The confluent transitions similar to the external transitions.

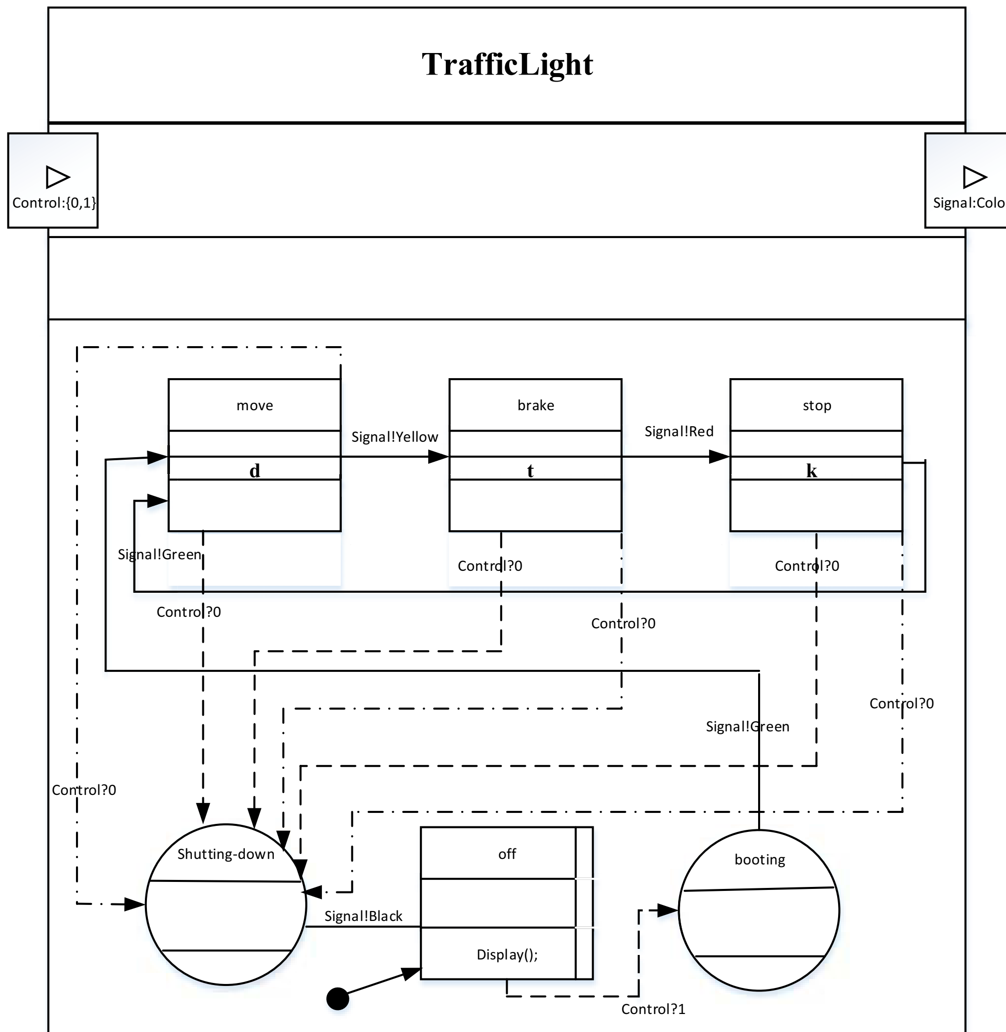

Figure 38. controlled Trafficlight

We give here, the equivalent atomic DEVS model $TrafficLight_{DEVS}$ of the traffic light *HSystem* presented in Figure 38.

$TrafficLight_{DEVS} =< X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta >$ where:
$X = \{(signal, \{0,1\})\}$,
$Y = \{(color, \{Green, Orange, Red, Black\})\}$
$S = \{move, braking, stoping, off, shutting_down, booting\}$.

$\delta_{int}: S \rightarrow S$

$\delta_{int}(move) = brake$
$\delta_{int}(brake) = stop$
$\delta_{int}(stop) = move$
$\delta_{int}(shutting_down) = off$
$\delta_{int}(booting) = move$

Internal transition function is not defined for the "*off*" state because its time advance is infinite as will be shown later. Hence, an external input event is required to change the state via an external state transition function.

$\delta_{ext}: Q \times X^b \rightarrow S$

$\delta_{ext}(s, e, 0) = shutting_down \ \forall s \in S \backslash \{off\}, \forall e \in \mathbb{R}^+$

$\delta_{ext}(off, e, 0) = off \ \forall e \in \mathbb{R}^+$

$\delta_{ext}(off, e, 1) = booting \ \forall e \in \mathbb{R}^+$

$\delta_{ext}(s, e, 1) = s \ \forall s \in S \backslash \{off\}, \forall e \in \mathbb{R}^+$

In any state other than the *off* state, the system transits to the *shutting_down* state upon receiving a zero (0) input and returns to the same upon receiving a 1 input.

$\delta_{conf}: S \times X^b \rightarrow S$

$\delta_{conf}(s, x) = \delta_{ext}(s, ta(s), x) \ \forall s \in S$

$ta: S \rightarrow \mathbb{R}^+ \cup \{+\infty\}$

$ta(moving) = 10$

$ta(braking) = 3$

$ta(stoping) = 5$

$ta(shutting_down) = ta(booting) = 0$

$ta(off) = +\infty$

$\lambda: S \rightarrow Y^b$

$\lambda(moving) = Orange$

$\lambda(braking) = Red$

$\lambda(stoping) = Green$

$\lambda(shutting_down) = Black$

$\lambda(booting) = Green$

We will illustrate the mapping of composite systems to DEVS in chapter V.

IV.2.2 Semantic mapping to DSDEVS

HiLLS models can represent static structure systems or dynamic structure systems. We have presented the mapping of HiLLS static structure models to DEVS in the previous section. We present here the mapping HiLLS dynamic structure models in DSDEVS [Barros 1997]. HiLLS Dynamic structure models can then be simulated by DSDEVS algorithms.

Sine each DSDEVS atomic is exactly the same as DEVS atomic model, the mapping of HiLLS composite dynamic structure will reuse the HiLLS to DEVS semantic mapping rules at the atomic level.

Let $MC_{HILLS} =< DataStruct, StateMachine, Com >$ with $Com \neq \emptyset$ be HiLLS dynamic structure composite model. We recall that $DataStruct =< reference,\ IB, OB, V, OP, P >$, $StateMachine =< ConfID, SC, T_{int}, T_{ext}, T_{conf}, \psi, \varphi, Act, T >$ and Com $=< \{M_d\}_{d \in MReference}, EIC, IC, EOC >$.

MC_{HILLS} has an equivalent DSDEVS model
$DSDN = (\chi, M_\chi =< X_\chi, S_\chi, s_{0,\chi}, Y_\chi, \gamma, \Sigma^*, \delta_\chi, \lambda_\chi, \tau_\chi >)$ where:

$\chi = MC_{HILLS}.name$

$X_\chi = \{(p,v) | p \in IReference \wedge \exists d \in DomID, v \in dom(d) \wedge (p,d) \in IB\}$.
$Y_\chi = \{(q,v) | q \in OReference \wedge \exists d \in DomID, v \in dom(q) \wedge (q,d) \in OB\}$.
$S_\chi = SM = \prod_{v \in VarID} dom(v) \times CONF$ (where $CONF$ is the set of configuration names);
$s_{0,\chi} \triangleq MC_{HILLS}.INIT$ (the initial state is defined by the INIT of MC_{HILLS})

The transition function $\delta_\chi: Q_\chi \times (X \cup \{\emptyset\}) \rightarrow S_\chi$is defined by:
$\forall s \xrightarrow{c,y,ops} s' \in T_{int}$ where c is condition, y is a bag of output events, ops is the set of operations applied then we have: $\forall a \in \bar{s} \cap \{b \in S | b \vDash c \wedge \wedge_{op \in Ops} Pre\ op\}, \delta_\chi((a, \tau_\chi(a)), \phi) = Effet(ops, a)$ such that $Effet(ops, a) \vDash \wedge_{op \in Ops} Pos\ op$.
$\forall s \xrightarrow{x,c,c(e),ops} s' \in T_{ext}$ where c is a condition, x is the bag of input events, $c(e)$ is the condition on the elapsed time e and ops is the set of operations, we have : $\forall a \in \bar{s} \cap \{b \in S | b \vDash c \wedge \wedge_{op \in Ops} Pre\ op\}, \delta_\chi((a, e), x) = Effet(ops, a)$ such that $Effet(ops, a) \vDash \wedge_{op \in Ops} Pos\ op$.
$\forall s \xrightarrow{x,c,y,ops} s' \in T_{conf}$ where c is a condition, x is the bag of input events, y is the bag of output events, and ops is the set of operations, we have: $\forall a \in \bar{s} \cap \{b \in S | b \vDash c \wedge \wedge_{op \in Ops} Pre\ op\}, \delta_\chi((a, x), \phi) = Effet(ops, a)$such that $Effet(ops, a) \vDash \wedge_{op \in Ops} Pos\ op$.

$\Sigma^* = \{CM_{conf,s}\}_{conf \in CONF \wedge s \in S_\chi}$ where $CM_{conf,s} =< D_s, (\{M_i\}_{i \in D})_s, (\{I_i\}_{i \in D})_s, (\{Z_i\}_{i \in D})_s >$

Given a transition between s and s': $s \xrightarrow{x,c,y,ops} s' \in T_{conf}$ or $s \xrightarrow{x,c,ops} s' \in T_{ext}$ or $s \xrightarrow{c,y,ops} s' \in T_{conf}$

$D_{s'} = D_s \cup \{d.name\}_{d \in addedComponent(s)} - \{d.name\}_{d \in droppedComponent(s)}$

$(\{M_i\}_{i \in D})_{s'} = (\{M_i\}_{i \in D})_s \cup addedComponents(s) - droppedComponents(s)$

$(\{I_i\}_{i \in D})_{s'} =$
$(\{I_i\}_{i \in D})_s \cup \{d \in addedComponents(s) | \exists C \in EIC' \cup IC' \cup EOC' \wedge C.target = i\} - \{j \in I_i | j \in droppedComponents(s)\}$

$\gamma: S_\chi \rightarrow \Sigma^*$ is defined by $\forall s_\chi \in S_\chi, \gamma(s_\chi) = M_{conf,s_\chi}$ where $conf$ is the actual configuration.
The output function $\lambda_\chi: S_\chi \rightarrow Y$ is defined by:
$\forall s \xrightarrow{c,y,ops} s' \in T_{int}$: where c is a condition, y is the bag of output events, and ops is the set of operations,$\forall a \in \bar{s} \cap \{b \in S | b \vDash c \wedge \wedge_{op \in Ops} Pre\ op\}, \lambda_\chi(a) = y$
$\forall s \xrightarrow{c,x,y,ops} s' \in T_{conf}$ where c is a condition, x is the bag of input events, y is the bag of output events, and ops is the set of operations, we have $\forall a \in \bar{s} \cap \{b \in S | b \vDash c \wedge \wedge_{op \in Ops} Pre\ op\}, \lambda_\chi(a) = y$.

The time advance function $\tau_\chi: S \rightarrow \mathbb{R}^+ \cup \{+\infty\}$ is defined by:
$s \in SC, \forall a \in \bar{s}, ta(a) = T_s(a)$.

An Object Oriented implementation of the DSDEVS formalism is the DELTA Environment. DELTA has been implemented in Smalltalk [Barros 1995].

IV.3 Operational Semantics for Enactment

IV.3.1 Enactment

By enactment, we mean the synthesis of a software prototype of the system that can be executed to generate the footage of its real-time behavior in a given experimental frame. A typical analysis using enactment pays attention not only to the sequence of successive states but also to the activities associated with such states and their periods of execution. An activity refers to a set of time-consuming and interruptible tasks performed by the system while in a particular state which do not involve the reception of inputs, sending of outputs or change of state variables. For example, the display of advertisements on the screen of an Automated Teller Machine (ATM) when it is in idle state and the playing of the ringing tone of a telephone during an incoming call are activities associated to the respective states. Hence enactment methodology should provide a means to verify a system's real-time behavior by managing the scheduling and processing of its activities and events (internal and external) using the real world physical time. In the case of an embedded system, we may refer to enactment as the synthesis of codes for relevant hardware simulation and prototyping environments.

In this thesis, we present an Object-Oriented approach to system enactment that relies on the synthesis (from system specifications) of executable codes based on Object-Oriented General-Purpose languages. In this section, we will introduce an Object-Oriented description of discrete event systems. Based on the template provided by this description, we will show in subsequent sections how executable program codes can be synthesized from system specification with the aid of Model-Driven Engineering (MDE).

IV.3.2 Enactment Methodology

The methodology we propose is to express DEVS-based concepts using the dialect of the observer design pattern for the purpose of enactment. We do this by registering system ports as observers of other ports that may influence them. However, we acknowledge the fact that the notification process in the observer pattern poses some undesired effects during the exchange of messages between ports; the processes of the system sending the message will be blocked until the receiving system finishes treating the message received. The effect is even more complicated when there is a cascade of notifications. This is due to the synchronous calls to the update methods of the observers. We have tried to address this problem by using the command pattern to decouple the subject from its observer during notifications.

Figure 39 shows our attempt to introduce asynchronous message passing between the subject and its observers to make it more suitable for enacting systems' behaviors in real time. Compared to the command pattern presented in Figure 17, *Subject*, *Observer, Notifier, Notification* and *ConcrerteNotification* are equivalent to *Client, Receiver, Invoker, Command* and *ConcreteCommand* respectively. Therefore, subject will delegate the notifications of observes to

Notifier and continue its activities. Since the subject does not expect any return value from these method calls, it is easy to just use the "fire and forget" approach. *Notifier* has a pool of threads to which the requests are assigned on arrival, hence it does not have to always create threads thereby minimizing the overhead that may be incurred due to thread creation.

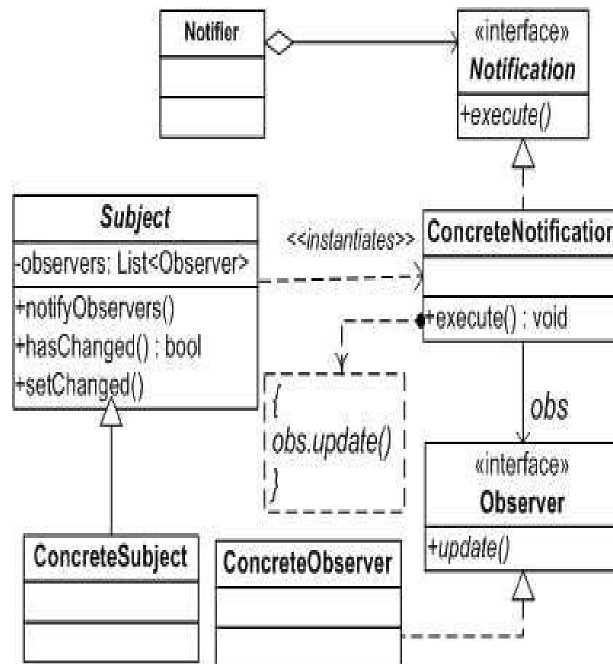

Figure 39. Observer pattern with asynchronous notification

IV.3.3 Metamodel of the framework

We present the metamodel of the enactment framework in the segment of Figure 40 that is enclosed within a dashed box. Using the observer pattern with asynchronous notification, a DES is described as *AbstractSystem* which implements the *Observer* interface while its generic input and output ports can observe and can be observed by other objects. A system has a clock that monitors the time advance of the current state; the clock inherits the Subject class so that it can notify the system at appropriate instants.

All methods in the *AtomicSystem* and *CoupledSystem* classes are abstract; therefore the concrete atomic and coupled system classes using the framework must implement them to provide the specific elements of the system being modeled. The *update* method of the *AbstractSystem* class has the implementation of the enactment protocol (to be provided in the next sub-section) which calls the user-defined functions when they are needed. The *doInternalTransition, doExternalTransition* and *doConfluentTransition* allow the user to describe the internal, external and confluent transition functions respectively. *setCurrentStatus* method is used to define the system's states based on the instantaneous values of the state variables to be defined by the user in the concrete class. Similarly, *mapTimeAdvance* and *mapOutputEvents* methods must be implemented to provide the time advance and output functions respectively. Method *mapActivities* can be used to define the activities to be enacted for each state during execution. An activity is a set of operations that do not lead to change in state variables, reception of inputs

and output events. *Coupling* between any two ports in the *CoupledSystem* is realized by adding the target port to the list of observers of the source port.

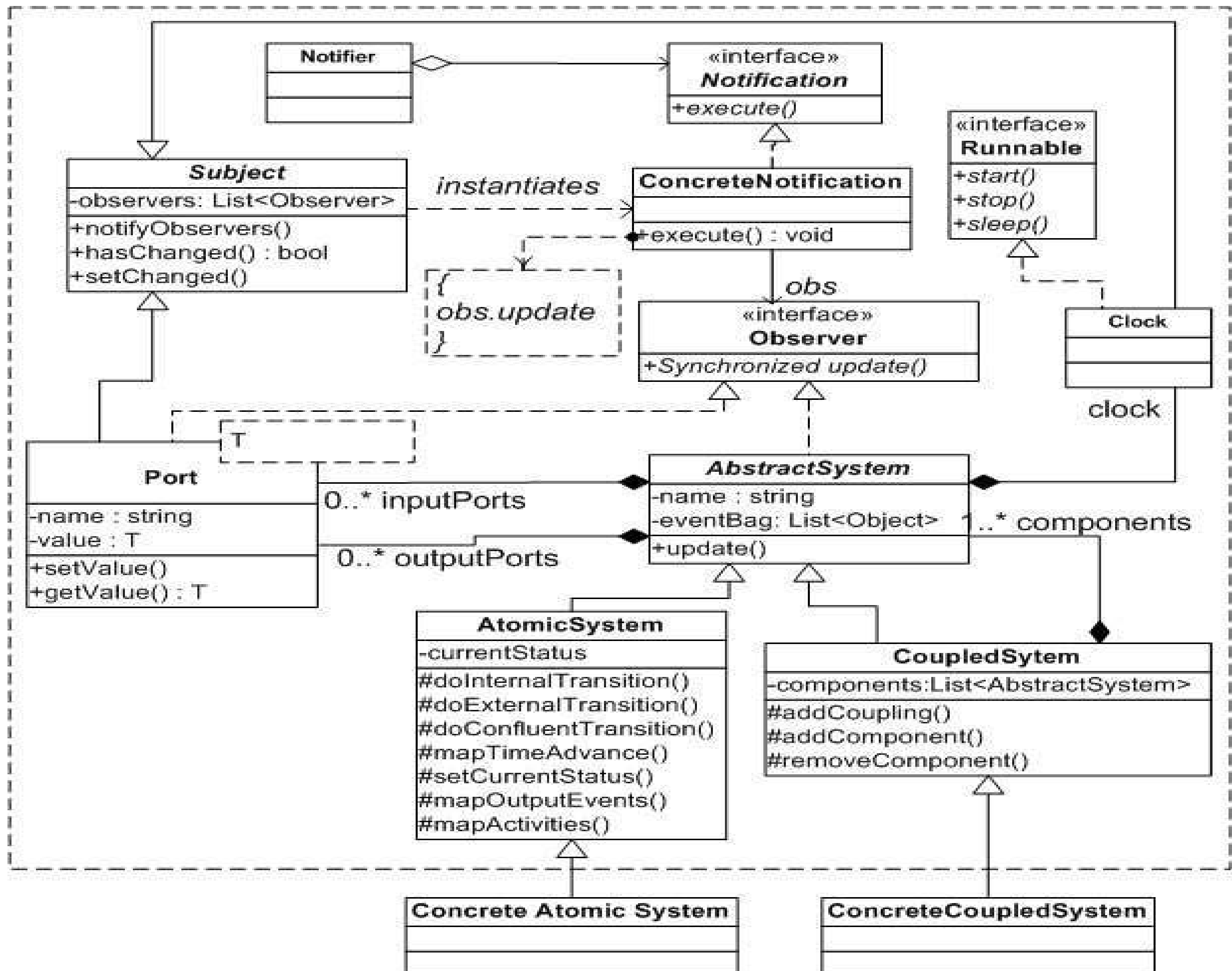

Figure 40. Metamodel of enactment framework

IV.3.4 Enactment protocol

A transition in the state of an *AtomicSystem* can be triggered by a timed event (an automatic notification from the clock when the time advance of the current state has elapsed), an input event (a notification from an input port upon receipt of a new value) or both. By default, an *AtomicSystem* is a registered observer of its *clock* and all its input ports, so this allows for automatic notifications from both sides. In any case, an event is an object that encapsulates a message (value) and information about the nature of its source, whether a port or clock. When the system receives notifications, all events received are stored in the event bag (*eventBag*) of the system. Then the system's reaction will depend on the content of the bag. If the event bag contains one time event, then it sends outputs (if any) to the appropriate output port(s) and then check if there are also input events in the bag. If a port event is found, then the *doConfluentTransition* method is invoked, otherwise *doInternalTransition* method is invoked.
If the event bag contains only input event(s), then *doExternalTransition* method is invoked.

IV.3.5 Implementation

We have implemented the framework's metamodel and enactment protocol in Java. To use the framework, we can simply create classes inheriting from the AtomicSystem and CoupledSystem classes of the framework to get the skeletons of the appropriate system unit. The user only needs to specify the properties that are peculiar to the system under study while the enactment mechanism is driven by the framework.

IV.3.5 Traffic Light Example

We present as example, the enactment of a traffic light control system to illustrate the extension of the enactment framework for real time execution of DES. The system consists of two components, control and display. The four configurations of the control, their durations and the corresponding light color to be on the display unit are summarized in Figure 41. The control unit has only one output port which is connected to the only input port of the display unit. Whenever, there is a change in the state (internal transition) of the control unit, it sends an output which is received by the display unit to show the appropriate light color to the road user.

Control states	**Duration of control state (units)**	**Display color**
ready	3	yellow
moving	10	Green
braking	3	Yellow
stopping	5	Red

Figure 41. Specification of traffic light system

The specification of the system is presented in Figures 42-45. The control unit is shown in Figure 42. It is an atomic unit, so it has to *extend* the framework's *AtomicSystem* class which provides the required system-specific methods to be completed as indicated by the methods with *@override* annotation in Figure 35. The single output port is created using the *addOutputPort* method provided by the framework. Since it has no input port, it cannot receive any *input* event, hence we did not provide implementations for external and confluent transition methods as they will never occur.

For experimental purpose, we have chosen 1000 milliseconds as the unit of the time advance durations specified in Figure 41. Before effecting a transition to a new state, in the internal transition method, the system invokes the *output* function to place a value on the output port depending on the active state. The activity function is also not implemented here as the component is not observed for any action in this context.

```java
package example;
import enactment.AtomicSystem;
import enactment.Port;
import enactment.Utilities;
import enactment.designExceptions.*;
import java.util.ArrayList;

public class TrafficLight extends AtomicSystem {
    private enum Status{MOVING, BRAKING, STOPING, READY};
    private Status state;
public TrafficLight(String name) {
    super(name);
    registerPorts();
    state=Status.READY;}

private void registerPorts(){
    try {//addInputPort(name, type); addOutputPort(name, type);
        addOutputPort("statusOut", new String());
    } catch (DuplicateIdException e){e.printStackTrace();}
}
@Override
protected long computeLifeSpanFunction() {
    long ltime;//ltime is in milliseconds
    if (state==Status.BRAKING) ltime = 3000;
    else if(state==Status.MOVING) ltime = 10000;
    else if(state==Status.STOPING) ltime = 5000;
    else if (state==Status.READY) ltime = 3000;
    else ltime=2000; return ltime;}
@Override
protected void doInternalTranssition() {//internal transition fun
    mapOutputEvents(state); //send output on port"statusOut"
    System.out.println(Utilities.getCurrentTime()+":"+" "+
    getName().toUpperCase()+": Internal transition from "+state);
    if (state==Status.MOVING) state = Status.BRAKING; //set targe
    else if(state==Status.BRAKING) state =Status.STOPING;
    else if(state==Status.STOPING) state = Status.READY;
    else if (state==Status.READY) state =Status.MOVING;
    else state =Status.MOVING;
    System.out.println(Utilities.getCurrentTime()+":"
    +" "+this.getName().toUpperCase()+": New state: "+state);}
@Override
protected void doExternalTransition(ArrayList<Port> eventBag){
    // No external transition specified}
@Override
protected void doConfluentTransition(ArrayList<Port> eventBag){
    // No confluent transition specified}
@Override
protected void mapOutputEvents(Object event) {
    Status st = (Status)event;
    if (st==Status.READY) sendMessage("statusOut","GREEN");
    else if(st==Status.MOVING) sendMessage("statusOut","YELLOW");
    else if(st==Status.BRAKING) sendMessage("statusOut","RED");
    else sendMessage("statusOut","YELLOW");
}
@Override
protected void mapActivities() {
    // no activities specified}
}
```

Figure 42 Control unit of the traffic light system

The display unit is presented in Figure 43. It is also an atomic unit and maintains only one state with approximately infinite time advance as indicates by the *Long.MAX_VALUE* in the *computeLifeSpanFunction*. So, it will never receive a *time event* since the time advance will never expire. Therefore, only external transition is possible. Whenever, it receives an input event (which is an instruction from the control unit), it changes the color of light displayed to the new color received.

```java
package example;
import java.util.ArrayList;
import enactment.AtomicSystem;
import enactment.Port;
import enactment.Utilities;
import enactment.designExceptions.*;
public class Display extends AtomicSystem {
    private String color;
    public Display(String name) {
        super(name); registerPorts();
        color = "RED";  }
    private void registerPorts() {
        try { addInputPort("input", new String());
        }catch(DuplicateIdException e){e.printStackTrace();}}
@Override
protected void doInternalTranssition(){}
@Override
protected void doExternalTransition(ArrayList<Port> eventBag) {
    color = (String) eventBag.remove(0).getValue();
    System.out.println(Utilities.getCurrentTime()+": "
    +this.getName().toUpperCase()+": received: "+ color );
    mapActivities();}
@Override
protected void doConfluentTransition(ArrayList<Port> eventBag){}
@Override
protected long computeLifeSpanFunction() {
    long lspan = Long.MAX_VALUE; return lspan;}
@Override
protected void mapOutputEvents(Object event) { }
@Override
protected void mapActivities() {
    System.out.println(Utilities.getCurrentTime()+": "+
    getName().toUpperCase()+":Displaying color:"+color+"\n");}
}
```

Figure 43 Display unit of the traffic light system

Figure 44 shows the composite system that has the control and display units as components. Being a composite system, it *extends* the *CompositeSystem* class of the framework which provides the required methods to be completed. It basically creates and registers its components and establishes any coupling(s) between them. In this case, there is only one coupling between the components as shown.

```java
package example;
import java.util.ArrayList;
import enactment.CompositeSystem;
import enactment.Port;
import enactment.designExceptions.*;

public class TrafficTest extends CompositeSystem {
    private TrafficLight lightControl;
    private Display displayUnit;

    public TrafficTest(String name) {
        super(name);
        lightControl = new TrafficLight("control");
        displayUnit = new Display("Display");
        registerComponents();
        doCouplings();
}
private void registerComponents() {
    try {
        addComponent(lightControl);
        addComponent(displayUnit);
    } catch (DuplicateIdException e){e.printStackTrace();}
}
@Override
protected void doCouplings() {
    try {//connectIC(source sys, source port, target Sys, target port)
        connectIC(lightControl, "statusOut", displayUnit, "input");
    } catch (InvalidCouplingException e) {e.printStackTrace();
    } catch (NoSuchPortExistsException e) {e.printStackTrace();}
}
}
```

Figure 44 Coupled traffic light system

Figure 45 shows an excerpt from the result of the enactment of the specification. The first column of the result shows the wall clock time, the second column shows the identity o fhte component in context and the third column shows the event being reported. Note that each component has its *clock* that monitors its activities based on the ticks of the wall clock. With a starting state of "*READY*", the "*CONTROL*" received a time event at "15:31:37", sent an output as specified in the model and did an internal transition to assume the "*MOVING*" state. The output sent by "*CONTROL*" was received in by the "*DISPLAY*" at the same time which

concurrently changed its display color (activity) accordingly. Other lines of the result segment can be read similarly.

```
Console
<terminated> TrafficEnact [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (13 Aug 2015 15:31:34)
15:31:37: CONTROL: TIME EVENT RECEIVED
15:31:37: CONTROL: Internal transition from READY
15:31:37: CONTROL: New state: MOVING

15:31:37: DISPLAY: INPUT EVENT RECEIVED
15:31:37: DISPLAY: received: GREEN
15:31:37: DISPLAY:Displaying color:GREEN

15:31:47: CONTROL: TIME EVENT RECEIVED
15:31:47: CONTROL: Internal transition from MOVING
15:31:47: CONTROL: New state: BRAKING

15:31:47: DISPLAY: INPUT EVENT RECEIVED
15:31:47: DISPLAY: received: YELLOW
15:31:47: DISPLAY:Displaying color:YELLOW

15:31:50: CONTROL: TIME EVENT RECEIVED
15:31:50: CONTROL: Internal transition from BRAKING
15:31:50: CONTROL: New state: STOPING

15:31:50: DISPLAY: INPUT EVENT RECEIVED
15:31:50: DISPLAY: received: RED
15:31:50: DISPLAY:Displaying color:RED
15:31:55: CONTROL: TIME EVENT RECEIVED
15:31:55: CONTROL: Internal transition from STOPING
15:31:55: CONTROL: New state: READY

15:31:55: DISPLAY: INPUT EVENT RECEIVED
15:31:55: DISPLAY: received: YELLOW
15:31:55: DISPLAY:Displaying color:YELLOW

15:31:58: CONTROL: TIME EVENT RECEIVED
15:31:58: CONTROL: Internal transition from READY
15:31:58: CONTROL: New state: MOVING

15:31:58: DISPLAY: INPUT EVENT RECEIVED
15:31:58: DISPLAY: received: GREEN
15:31:58: DISPLAY:Displaying color:GREEN
```

Figure 45 Results of the enactment of the traffic light system

IV.4 Semantic mapping to Z

This section presents the mapping of HILLS models to Z notation.

IV.4.1 Transformation approach to Z

Some compromises are necessary between HILLS and Z concepts due to fundamental differences between them. HILLS has notions of internal and external events where external events are initiated by some outside components. Z makes no formal notion of events or distinctions between internal and external operations. In Z, if the precondition of an operation is satisfied it is free to occur and no particular component is assumed to initiate it. In translating the HILLS concepts into Z, we made no difference between internal, external and confluent transitions, they are all considered as operation schemas. Only the differences between their input and output variables and a naming convention make the difference. Since Z is used for specifying systems mainly for time-independent static analysis, we do not consider it necessary to represent the sojourn time in Z.

Each unitary model has in addition to its operations specifying the methods of the class, the operations schemas representing the different transitions in the behavioral part of the HILLS model representing it.

In HiLLS specifications we have simple classes and model classes. Each simple class in HiLLS is mapped to one state schema and associated operations. The mapping is straightforward because HiLLS uses also Z predicates and expressions to specify data and data transformations.

Let $C =< Name, Params, State, OPS_C, Init_C >$ be a HiLLS class (not a model class). C is translated to a Z state schema $CStateSchema$ and associated operations $(COPS_i)_{i\in[1,\#OPS_C]}$.

$CZState$ is defined as follows:

- $CZState.Name = C.Name$
- $CZState.Params = C.Params$
- $CZState.DeclPart = C.State.DeclPart$
- $CZState.PredPart = C.State.PredPart$
- $CZState.INIT = C.INIT$

An abstract type, *Status* is defined with a finite alternative values comprising of configuration names in the HiLLS specification. A secondary variable, *configuration* of type *Status,* is declared in the state schema and the properties of the configuration in HiLLS translate to the constraints to derive corresponding values of *configuration* in the predicate part of the state schema.

For each operation

$op =< Name, input, output, DeclPart, preCondition, postCondition > \in OPS_C$

op is translated to an Z operation schema opZ as follows:

- $opZ.name = op.name + C.Name$ (to precise that the operation is for C because different classes can have operations with the same name)
- $opZ.input = op.input$ ($opZ.input \subseteq opZ.DeclPart$)
- $opZ.output = op.output$ ($opZ.output \subseteq opZ.DeclPart$)

- $opZ.DeclPart = op.DeclPart \cup C.State.DeclPart \cup C.State.DeclPart'$ (or $opZ.DeclPart = op.DeclPart \cup \Delta C.Name$ where $C.Name$ is the reference of $CZState$ resulting from the translation of the state schema to Z)
- $pre\ opZ = C.State.PredPart \wedge pre\ op$
- $post\ opZ = post\ op \wedge \left(\bigwedge_{x \notin op.Deltalist} x' = x\right)$

pre and $post$ are respectively precondition and postcondition operators.

HiLLS model class has activities associated to configurations. An activity is also a Z operation with its own declarations, preconditions and postcondition. They can access to model attributes but don't modify them. Activities are translated as operations. The only difference is the absence of *deltalist* for activities because they don't modify model state.

A HiLLS model class is a class with additional state machine and ports. The state machine is composed of configurations and transitions between configurations. A configuration is just a set of predicate.

Every configuration transition translates to an operation in Z with the name of source configuration constituting part of its pre-conditions, the computations accompanying the transition make the body of the operation and the name of the target configuration becomes the primed value of the derived variable *configuration*. For external and confluent transitions, the triggers translate to input variables of the operation (the type of each input variable is the type of the port from which the trigger is received). Similarly, for every internal and confluent transition where an output(s) is/are produced, an output variable is declared in the operation with same type as the corresponding output port. Finally, three operation expressions, *InternalTransitions*, *ExternalTransitions* and *ConfluentTransition*s that combines all operations derived from internal, external and confluent transitions respectively is defined.

The translation of the different kind of transitions is as follows.

IV.4.1.1 Internal Transitions

Output ports referenced in the transition are declared as output parameters with their corresponding type. The assignments of values to these ports are added as post condition of the corresponding operation.

We recall that $T_{int} \subseteq SC \times C(V) \times OB^b \times \mathbb{P}\ OP \times SC$ is the set of internal transitions.

Let $t = (s, c, y, ops, s') \in T_{int}$ be an internal transition between configuration s and configuration s' where c is a predicate, y is the bag of output events (each event in y is of the form $q^\wedge.v | v \in dom(q)$) and ops is the sequence of reconfiguration operations.

The internal transition t is translated to a Z operation schema tZ where:

- $tZ.name = int + s.name + s'.name + C.name$ (this naming convention is to precise that this operation represents an internal transition between s and s' in the state machine of class C.
- $tZ.input = \bigcup_{op \in t.ops} op.input$
- $tZ.output = (\bigcup_{e \in y} e.port.name\colon dom(e.port)) \cup \left(\bigcup_{op \in t.ops} op.output\right)$
  $tZ.DeclPart = (\bigwedge_{op \in t.ops} op.DeclPar) \cup tZ.output \cup C.State.DeclPart \cup C.State.DeclPart'$

- $pre\ tZ = (C.State.PredPart) \wedge (s.PredPart) \wedge (t.c) \wedge (\bigwedge_{op \in t.ops} pre\ op)$
- $post\ tZ = \left(\bigwedge_{op \in t.ops} post\ op\right) \wedge \left(\bigwedge_{op \in t.ops}(\bigwedge_{x \notin op.Deltalist} x' = x)\right)$
  $\wedge \left(\bigwedge_{e \in y} e.port.name' = e.value\right)$
- A theorem to prove $post\ tZ \vDash s'.PredPart$ i.e. the post condition of the operation results in a state that satisfies the constraints of s'.

The translation is schematized in Figure 46.

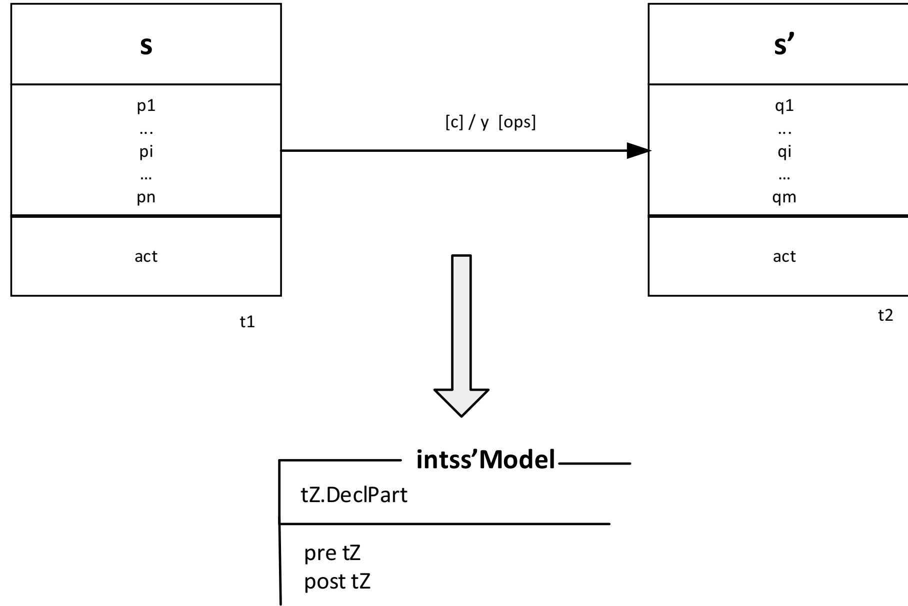

Figure 46. Translation of an internal transition to Z

IV.4.1.2 Conditional internal transitions

In the case of conditional transitions between configurations the translation rules to Z operation is the same for each alternative transition if the target configurations are different (Figure 47). If more than one alternative have the same target configuration, then the name of each operation for each these alternatives is of the form:

$$tZ.name = int + s.name + s'.name + C.name + indice(c)$$

The special integer $indice(c)$ depending on the predicate c is added to the name to distinguish between the different alternative transitions between configuration s and configuration s').

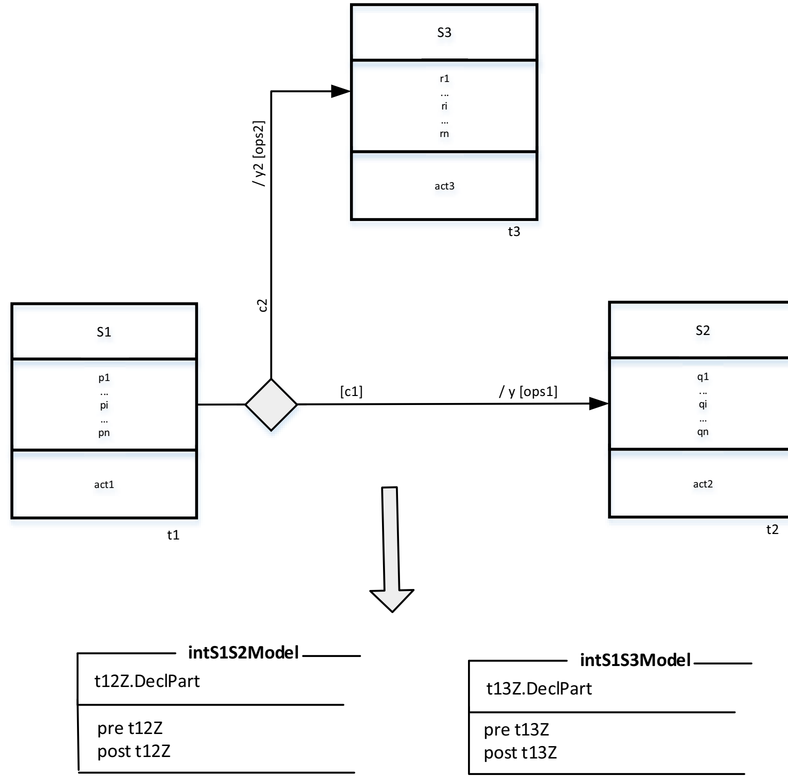

Figure 47. Conditional internal transition to Z

IV.4.1.3 External Transitions

Input ports referenced in the transition are declared as input parameters with their corresponding type. The reception of values on these ports are added as pre conditions of the corresponding operation.

We recall that $T_{ext} \subseteq SC \times C(V) \times IB^b \times \mathrm{C}(e) \times \mathbb{P}\, OP \times SC$ is the external transition relation.

Let $t = (s, c, x, p_e, ops, s') \in T_{ext}$ be an external transition between configuration s and configuration s' where c is a predicate, x is the bag of intput events (each event in x is of the form $q.v|v \in dom(q)$), p_e is a predicate on the elapsed time since the last event, and ops is the sequence of reconfiguration operations.

The external transition t is translated to a Z operation schema tZ where:

- $tZ.name = ext + s.name + s'.name + C.name$ (the prefix ext is used to precise the type of transition)
- $tZ.input = \bigcup_{e \in x} e.port.name: dom(e.port)) \bigcup_{op \in t.ops} op.input$
- $Z.output = \bigcup_{op \in t.ops} op.output$
- $tZ.DeclPart = (\bigwedge_{op \in t.ops} op.DeclPar) \cup (tZ.input) \cup (C.State.DeclPart) \cup (C.State.DeclPart')$
- $pre\ tZ = (C.State.PredPart) \wedge (s.PredPart) \wedge (t.c) \wedge \left(\bigwedge_{op \in t.ops} pre\ op\right) \wedge (t.p_e) \wedge (\bigwedge_{e \in x} e.port.name = e.value)$
- $post\ tZ = (\bigwedge_{op \in t.ops} post\ op) \wedge \left(\bigwedge_{op \in t.ops} (\bigwedge_{x \notin op.Deltalist} x' = x)\right)$

- A theorem to prove $post\ tZ \vDash s'.PredPart$ i.e. the post condition of the operation results in a state that satisfies the constraints of s'.

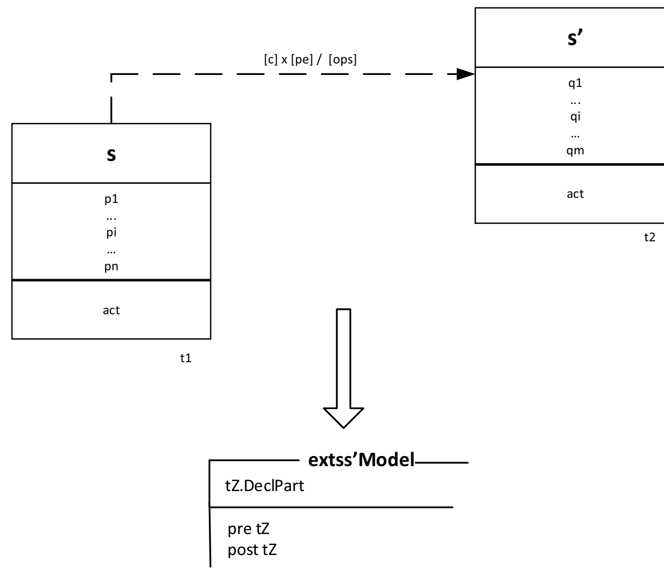

Figure 48. External Transition translation to Z

IV.4.14 Conditional external transitions

As in the case of conditional internal transition, a conditional external transition will be translated to different operations (one for each alternative) and the same naming convention applies (Figure 49).

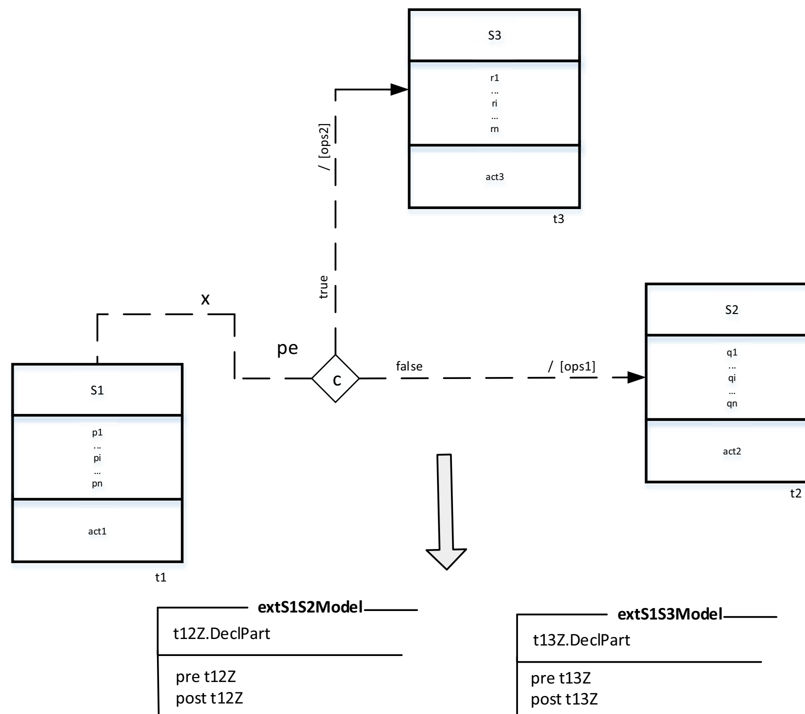

Figure 49. Conditional external transition to Z

IV.4.1.5 Confluent Transitions

Input and output ports referenced in the confluent transition are declared respectively as input and output parameters with their corresponding type. The reception and sending of values on these ports are respectively added as pre conditions and postconditions of the corresponding operation.

We recall that $T_{conf} \subseteq SC \times C(V) \times IB^{b} \times \mathrm{OB}^{\mathrm{b}} \times \mathbb{P}\, OP \times SC$ is the confluent transition relation.

Let $t = (s, c, x, y, ops, s') \in T_{conf}$ be an external transition between configuration s and configuration s' where c is a predicate, x is the bag of intput events, y is the bag of output events, and ops is the sequence of reconfiguration operations.

The confluent transition t is translated to a Z operation schema tZ where:

- $tZ.name = conf + s.name + s'.name + C.name$ (the prefix $conf$ is used to precise the type of transition)
- $tZ.input = \bigcup_{e \in x} e.port.name: dom(e.port)) \cup (\bigcup_{op \in t.ops} op.input)$
- $Z.output = (\bigcup_{op \in t.ops} op.output) \cup \left(\bigcup_{e \in y} e.port.name: dom(e.port))\right)$
- $tZ.DeclPart = \left(\bigwedge_{op \in t.ops} op.DeclPart\right) \cup (tZ.output) \cup (tZ.input) \cup (C.State.DeclPart) \cup (C.State.DeclPart')$
- $pre\ tZ = (C.State.PredPart) \wedge (s.PredPart) \wedge (t.c) \wedge \left(\bigwedge_{op \in t.ops} pre\ op\right) \wedge (\bigwedge_{e \in x} e.port.name = e.value)$
- $post\ tZ = (\bigwedge_{op \in t.ops} post\ op) \wedge \left(\bigwedge_{op \in t.ops} (\bigwedge_{x \notin op.Deltalist} x' = x)\right) \wedge \wedge \left(\bigwedge_{e \in y} e.port.name' = e.value\right)$
- A theorem to prove $post\ tZ \vDash s'.PredPart$ i.e. the post condition of the operation results in a state that satisfies the constraints of s'.

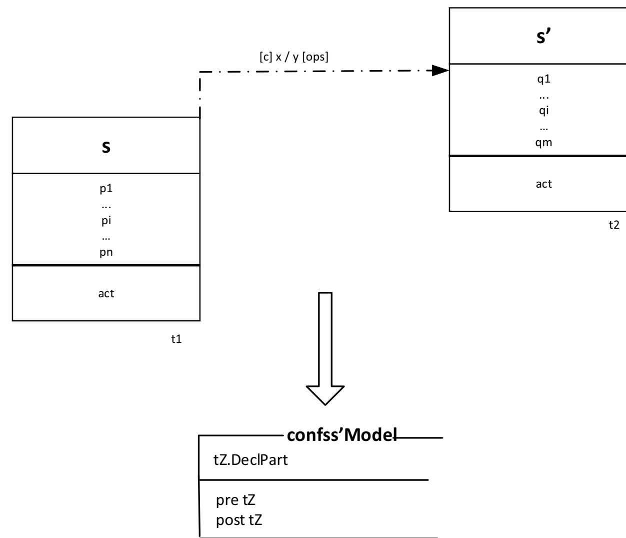

Figure 50. Confluent transition to Z

IV.4.1.6 Conditional confluent transitions

The same translation principle used for internal and external conditional transition applies to conditional confluent transitions.

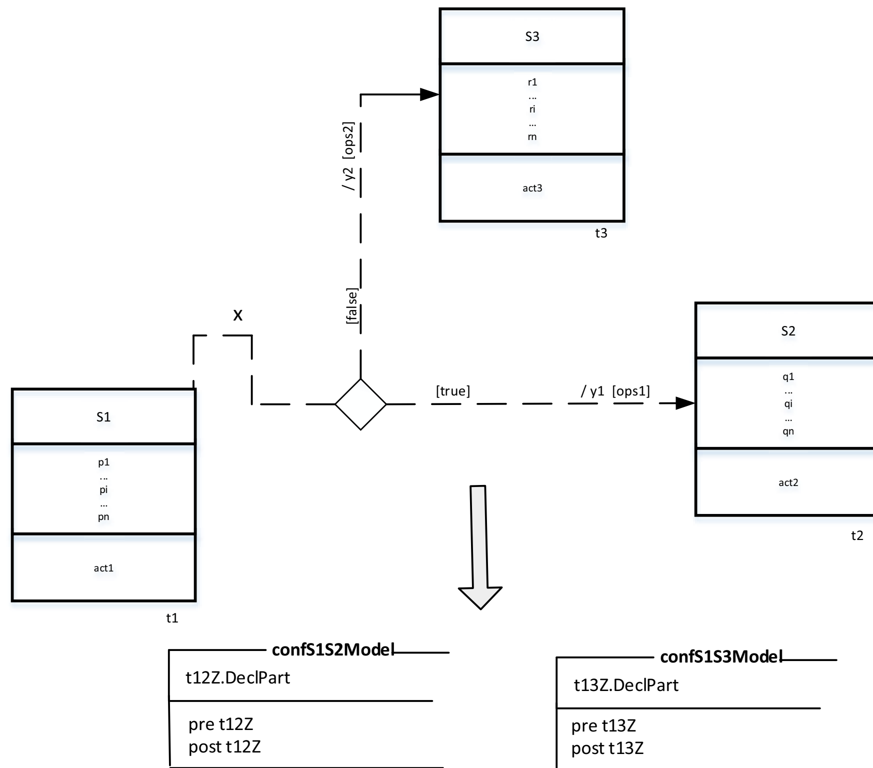

Figure 51. Confluent conditional transition to Z

We will illustrate the mapping of HiLLS to Z in chapter V.

IV.5 Semantic mapping to CSP

The HILLS Composite level is the architectural level defining the overall structure of a system with components and functional couplings between components. At the composite level, the components operate concurrently and their interactions are instantaneous and parallel. At this level of abstraction, HILLS can be translated to several formal methods, notably process calculi. Process calculi are a diverse family of related approaches to formal modeling of concurrent systems. Process calculi [Baeten 2004] provide a technique for high-level description of interactions, communications, and synchronizations between collections of independent processes. They also provide algebraic laws that allow process descriptions to be manipulated and analyzed, and permit formal reasoning about equivalence between processes. Leading examples of process calculi are CSP (Communicating Sequential Processes) [Hoare 1985], CCS (Calculus of Communicating Systems) [Milner 1980], ACP (Algebra of Communicating Processes) [Bergstra and Klop 1987], pi-calculus [Milner et al. 1992a],[Milner et al. 1992b], and LOTOS (Language of Temporal Ordering Specification) [Bolognesi and Brinksma 1989]. Each of these provides access to different kinds of techniques, automated tools and analysis. However, CSP has emerged as our preferred formalism for three main reasons. First, CSP is well established for modeling concurrent systems [Abdallah et al. 2004] making it a target for industrial and academic research and development. Second, CSP gives access to several analytical tools like the industrial FDR2 refinement checker [Broadfoot and Roscoe 2000], ProBE animator [Formal Systems 2011a],

CSP TypeChecker [Formal Systems 2011b], CSP-Prover [Isobe et al. 2007], Isabelle/HOL [Camilleri 1990] … These tools allow process expressions to be automatically checked for properties, such as deadlock freedom and compliance with system specifications also expressed in CSP. Third, CSP supports hierarchical development of process models. Furthermore, CSP models can be transformed into other process algebra to enable analysis in their domains. For a very comprehensive review of CSP, the reader should consult Hoare's book [Hoare 1985] and Roscoe's Understanding Concurrent Systems [Roscoe 2010].

The basic building blocks of a HILLS coupled network are models, ports, and couplings. In general, every model stands for an instantiation of a specific CSP process. The HILLS framework maps the behavior of unitary models to Z. Here, we are interested in events that are visible to the environment. In HILLS, these are realized with Ports. Ports are mapped to CSP channels (compound events) with names and types. These channels belong to the alphabet of some process. Processes communicate through compatible channels. Such communications are analogous to couplings in HILLS. The couplings are modeled as parallel composition of processes with appropriate synchronization sets demonstrating the flow of objects between processes.

Generally, an arbitrary port of a HILLS model defines a logical point of interaction between the model and its environment. A port typically represents the set of interaction events with the environment through its interface. Hence, ports in HILLS can be mapped onto CSP compound events. These events are realized through CSP channels. We define a translation function ψ that maps a HILLS port to CSP channel local to corresponding process of its associated model:

$$\psi : Port \rightarrow Channel$$

Hence, if c is the name of a port and T is the type of object communicated down it, we would have that

$$\psi(p) = c.T = \{c.x \mid x \in T\} \subseteq \Sigma$$

HiLLS couplings are also mapped to CSP channels that are visible outside of individual subcomponents: $\Psi: Coupling \rightarrow Channel$ because they only relate to compatibles, which exactly correspond to role of channels in CSP.

A model in HILLS is defined by a set of ports (input and output ports) and a component specification that represent the abstract behavior of the model. Since models can have many ports, ports allows a model to define multiple interface to its environment. A HILLS model can be mapped to a CSP process. The alphabet contains only external events (or channels). We define a function $\mathcal{M}$ that maps a HILLS model to CSP process

$$\mathcal{M} : Model \rightarrow Process$$

Hence, if A_m is a HILLS model and $S_{internal}$ represents the set of all internal events,

$$\mathcal{M}(A_m) = P_A \setminus S_{internal} = A_p$$

Where P_A is a process and the alphabet of A_p is given as

$$\alpha A_p = \bigcup\{\psi(p) \mid p \in ModelPorts\}$$

HiLLS unitary models are represented by their equivalent abstract processes where data transformation is not taken into account. This CSP abstract process takes into account only events. Each event in the equivalent CSP process is an abstract representation of a HiLLS operation which defines the real state changes and communication effects caused by its occurrence. A unitary model is translated into a process which is written as an external choice

of all its possible events, where after an event occurrence, the process recursively offers all events again.

Let $M =< State, Init, Ops >$ be a unitary model represented in its Z-based form as a state schema $State$, init schema $Init$ and operations Ops where
$Ops = \{tZ|\ t \in T_{int}\} \cup \{tZ|\ t \in T_{ext}\} \cup \{\{tZ|\ t \in T_{conf}\}\}$ where tZ is the equivalent Z operation schema of the transition t. It comes naturally that Ops is a finite set because the number of transitions in HiLLS model is finite.

The equivalent CSP process of M is as follows:

$P(M) = \boxdot_{com_op \in Ops}\ pre\ com_op \wedge op \rightarrow P(com_op(State_A))$

Composite models are top level processes composed of other processes with defined synchronization between them.

We refer to the equivalent CSP process of a model by its name.
A composite model in HILLS is composed of several interacting models. These models are coupled or connected via port attachments or couplings. Communications between the models take place between ports that are compatible. Such communication is synchronous and can be generalized as follows.
Composite models belong to the domain of the function $\mathcal{M}$. Hence, if *C* is a composite model there exists an equivalent CSP process.

$$\mathcal{M}(C) = C_p$$

Let *Sub* be the set of all sub-models of the coupled model *C*. Then

$$V = \|_{P:Sub}\ (P[\![R_C]\!], X_p)$$

V is the parallel composition of all the sub-models.

$$C_p = V\ [\![R_{EC}]\!])\ \setminus\ H_{IC}$$

This is the *n*-way parallel composition of all the sub-models in *Sub*. Where

$P[\![R_C]\!]$ is a renaming operation that changes port names to coupling names:

X_p represents the alphabet of *P*. This is the set of external channels (after renaming).

$V\ [\![R_{EC}]\!]$ is the renaming operation used to rename coupling names from a sub-model to the name of an external port of the composite model.

H_{IC} denotes all the internal couplings that are hidden.

We illustrate the HiLLS to CSP transformation by using the example shown in Figure 52.

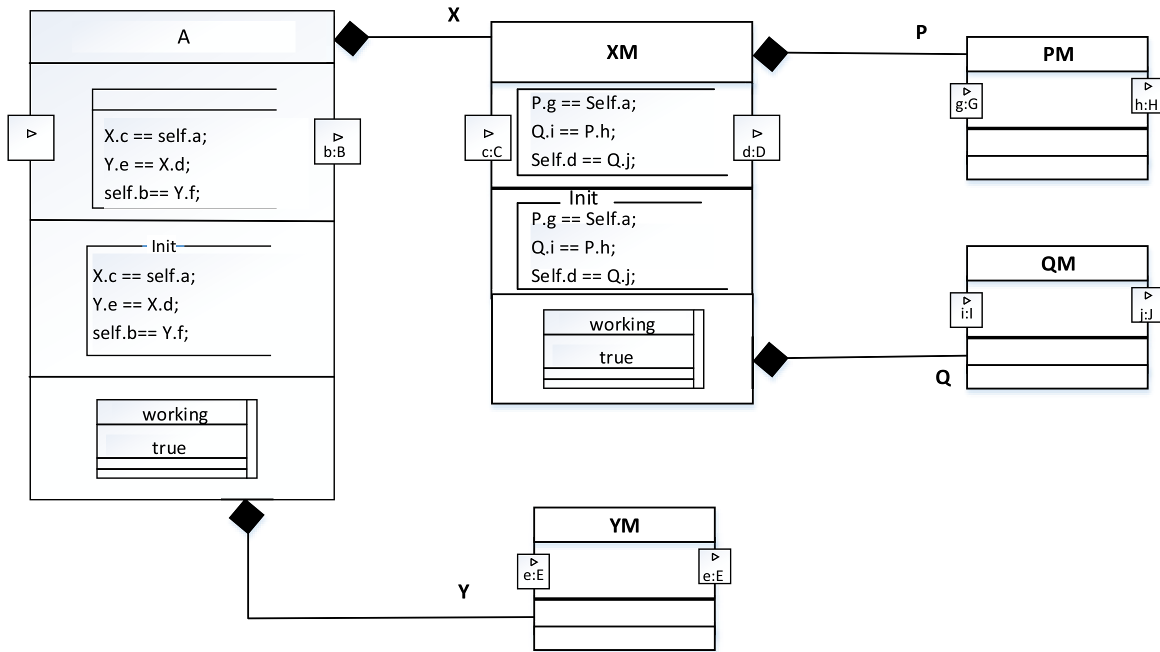

Figure 52. Illustration of HILLS to CSP translation

A is a composite model with input port a and output port b. It is composed of two sub-models (X and Y). X is a composite model with sub-models P and Q and ports c and d. For the composite model A, we note m as the name of the $InputCoupling$ between X and A, u is the of the $InternalCoupling$ between X and Y, and r is the name of the $OutputCoupling$ between Y and A. For X, r is an $InputCoupling$ between X and P, s is an $InternalCoupling$ between P and Q, and t is an $OutputCoupling$ between between Q and X. We start with the composite model X with unitary models P and Q. For these two we assume to have some definitions at hand (e.g. as given by their IOS definitions in Z) and just use their names: P, Q.

To determine the process term of the coupled model X, first we have to compose processes P and Q in parallel. Note that for communication to take place between two processes, they must synchronize on a common channel. So if we use the names of the ports h and i attached to the c, communication cannot take place. Instead, we use the name of the coupling in the synchronization set and carry out an appropriate renaming of the port names to the coupling names on the coupled model by $P\left[\!\left[{}^{s,r}/_{h,g}\right]\!\right]$ which renames h and g to s and r respectively and $Q\left[\!\left[{}^{s,t}/_{i,j}\right]\!\right]$ which renames i and j to s and t respectively. Thus the communication between P and Q is modeled by the CSP paradigm of synchronous communication:

$$P\left[\!\left[{}^{s,r}/_{h,g}\right]\!\right]\ {}_{\{s,r\}}\|_{\{s,t\}}\ Q\left[\!\left[{}^{s,t}/_{i,j}\right]\!\right]$$

To complete the process description of the coupled model X, the channel r has to be renamed into c (the input port of X) and t has to be renamed into d (the output port of X). Finally, the internal channel ($InternalCoupling$) s between P and Q has to be hidden. Hence

$$X = ((P\left[\!\left[{}^{s,r}/_{h,g}\right]\!\right]\ {}_{\{s,r\}}\|_{\{s,t\}}\ Q\left[\!\left[{}^{s,t}/_{i,j}\right]\!\right])\ \setminus \{s\})\left[\!\left[{}^{c,d}/_{r,t}\right]\!\right]$$

We hide the channel s because when X is treated like a component within another composite model, channel s is seen as an internal event. The only visible channels become r and t which are accessible from the outside by ports c and d. We can apply the same principles to derive

the CSP process of the complete network A. The communication between processes X and Y is also modeled by the CSP synchronous communication model:

$$X\left[\!\left[{}^{m,u}/_{c,d}\right]\!\right]{}_{\{m,u\}}\|_{\{u,v\}}\, Y\left[\!\left[{}^{u,v}/_{e,f}\right]\!\right]$$

Hence, the process description of A is:

$$A = \left(X\left[\!\left[{}^{m,u}/_{c,d}\right]\!\right]{}_{\{m,u\}}\|_{\{u,v\}}\, Y\left[\!\left[{}^{u,v}/_{e,f}\right]\!\right]\right)\left[\!\left[{}^{a,b}/_{m,v}\right]\!\right] \setminus \{u\}$$

For the example we have considered, the coupled models have at most two sub-models. In more practical examples, this is not usually the case. We shall show how to derive the CSP process description of a coupled model with more than one sub-model. This would also be realized by a parallel composition of the child models. Generally, from CSP theory, if we have three concurrent processes (P, Q and R) with X, Y and Z being their respective alphabets. The parallel composition can be given as:

$$(P\ {}_{X}\|_{Y}\ Q)\ {}_{X\cup Y}\|_{Z}\ R = P\ {}_{X}\|_{Y\cup Z}\ (Q\ {}_{Y}\|_{Z} R)$$

In composing a large network, we use the indexed notation for n-way parallel composition:

$$||_{i=1}^{n}(P_i, X_i) = P_1\ {}_{X1}\|_{X2\cup\ldots\cup Xn}\ (\ldots(P_{n-1}\ {}_{Xn\text{-}1}\|_{Xn}\ P_n)\ \ldots)$$

Consider the example the coupled model below with three atomic models P, Q, R (Figure 53).

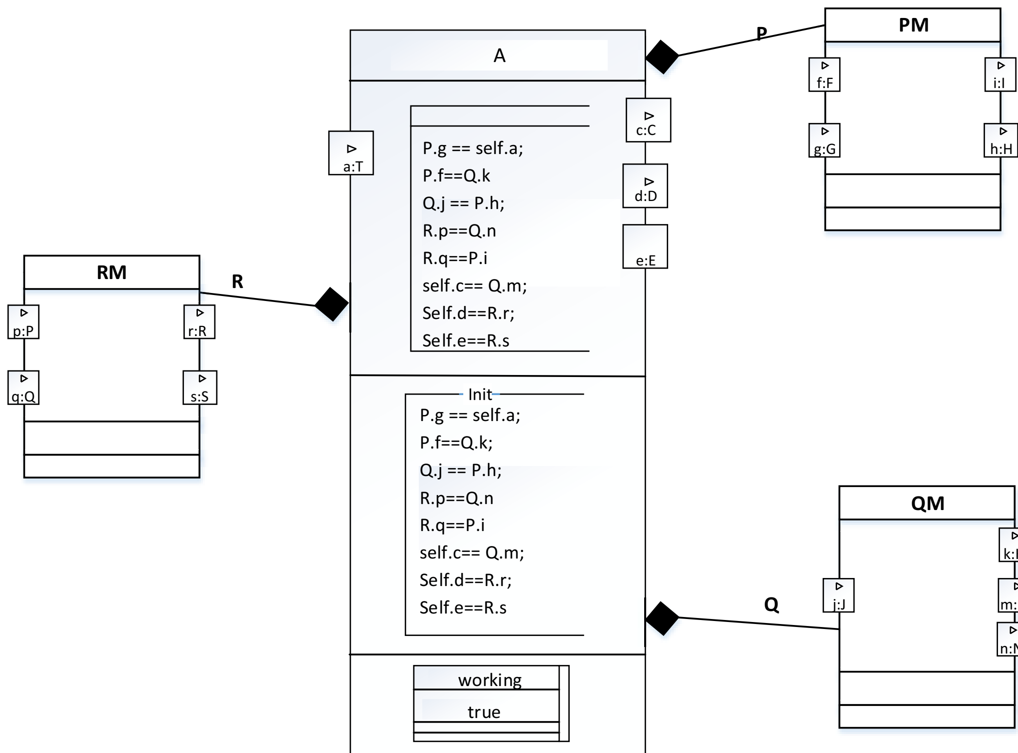

Figure 53. Translation of HILLS composite model to CSP

After carrying out the renaming operations of the ports to the coupling names, we have

$$P\left[\!\left[{}^{x,t,y,o}/_{f,g,h,i}\right]\!\right], Q\left[\!\left[{}^{y,x,w,z}/_{j,k,m,n}\right]\!\right] \text{ and } R\left[\!\left[{}^{o,z,u,v}/_{q,p,s,r}\right]\!\right].$$

The parallel composition of processes P, Q and R becomes:

If $X = \{x, t, y, o\}$ and $Y = \{y, x, w, z\}$, and $Z = \{o, z, u, v\}$, then

$$V = (P \left[\!\left[{}^{x,t,y,o}/_{f,g,h,i} \right]\!\right] {}_{X}||_{Y}\, Q \left[\!\left[{}^{y,x,w,z}/_{j,k,m,n} \right]\!\right]) {}_{X\cup Y}||_{Z}\, R \left[\!\left[{}^{o,z,u,v}/_{q,p,s,r} \right]\!\right]$$

x, t, y, o, w, z, u are the names of the couplings between RM, P, Q and R
V is the parallel composition of all the sub-models. Thus, the process description of the composite model A is defined as follows. Let S be the set of internal coupling channels that would be hidden to form the composite model A,

$$S = \{x, y, o, z\}$$

$$A = (V \setminus S) \left[\!\left[{}^{a,c,d,e}/_{t,w,v,u} \right]\!\right]$$

IV.6 Tooling Framework

The HILLS-to-DEVS mapping allows the automation of simulation code synthesis since already implemented DEVS simulators are available. Formal analysis of HILLS models are also done by taking advantage of already existing tools using model-driven techniques. A critical aspect of success of Model Driven Software Development (MDSD) is using a common metamodel. In our case, the EMF's Ecore is the meta-modeling language because HILLS graphical editor is developed using Eclipse Framework. The federated tooling framework, as shown in Figure 54 , permits to transform concepts of the architectural model of HILLS (HILLS.ecore) into the semantically equivalent concepts of the architectural models of the formal methods chosen, for example CSP.ecore, Z.ecore, and CTL.e (corresponding to the three levels of abstraction in HILLS). All these architectural models conform to Ecore. Thereafter, a semantic mapping (or meta-transformation) that relates the architectural concepts of the HILLS meta-model to the semantic equivalent elements in the meta-model of the input languages of the tools is defined. This mapping is done at two levels: Model-to Model Mapping (meta-model mapping of HILLS to the input languages; done with QVT/ATL) and Model to Text mapping (mapping the meta-models of the input languages of the tools to textual templates for code generation; done with JET/XPand/Acceleo). The tool integration methodology is based on a model-driven plugin architecture consisting of the HILLS graphical editor (for creating the models) and Transformers.

The transformers are at the core of the development of the federated tooling framework. Their main function is to transform the model files created by the graphical editor to the input files that would be used to feed the tool for formal analysis.

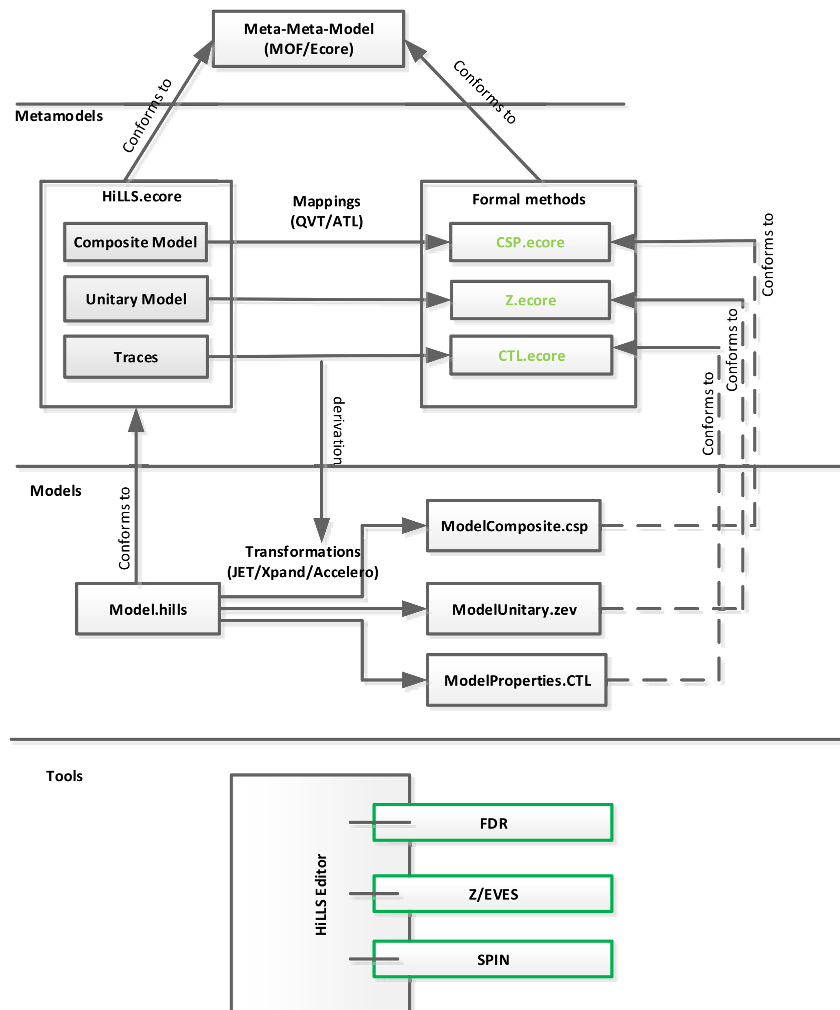

Figure 54. Overview of HiLLS tooling Framework

VI.7 Conclusion

We have given, in this chapter, the translational semantics of HiLLS, opening the way to simulation, formal analysis and enactment of HiLLS-specified models:

- For simulation, we have mapped HiLLS onto DEVS (for general systems) and onto DSDEVS (for variable structure systems).
- For enactment, we have presented an Object-Oriented framework that provides a template to guide the synthesis/writing of program codes from HiLLS models and the protocol for real time enactment of system's behavior. The main idea is to be able to generate or specify an operational model in form of software systems to verify and validate the real-clock time behavior of system models with respect to requirements.
- For formal analysis, we have presented semantic mappings between HiLLS at one hand, and Z and CSP at the other hand. These mappings allow the use of Z-supported tool (like Z/EVES) and CSP-supported tools (like FDR). The semantic relationship between HiLLS and CTL is still in investigation to provide a formal way of expressing trace-based properties. This is part of our future work.

V. Application

V.1 Introduction

HILLS describes the possible system behavior in a mathematically (using state transitions and Z-based specifications of data and operations) precise and unambiguous manner. The domain modeling capability of HiLLS enables domain experts to participate in the system development process by capturing their domain knowledge in precise and comprehensible models on which other technical experts can collaborate. The HILLS specification of models provides the basis for many verification techniques ranging from model checking, theorem proving, simulation, or model-based testing. Once the HILLS model is built and properties are formalized using the appropriate property specification language, possible analyses may consist in:

- Studying system behaviour and producing traces of it by using HILLS simulators or DEVS-based simulators. At the first time some simulations can be performed before any other form of verification to eliminate simple modelling errors. Even though a specification has been successfully and formally verified by model checking or theorem proving techniques i.e. no errors have been detected. This does not guaranty total absence of design flaws because only correctly formulated properties or theorems have been verified. For example, some required behavior may have been omitted in the modeling phase unintentionally, or some important requirements may not be discovered and formalized or may be stated incorrectly. There is no guarantee that formal analysis will detect these kinds of errors. Simulation is necessary in these cases to explore the behavior of the system and probably discovering faults and completing the requirements.
- The precise modelling of discrete event dynamic systems often leads to the discovery of incompleteness, ambiguities, conflicts and inconsistencies in informal specifications of the system under study. It is necessary to check these conflicts in specifications earlier because such problems are usually only discovered at a much later stage of the design. A first step in identifying ambiguities, conflicts and inconsistencies in the specification is to run partial checks while doing the specification of the model. This is very helpful to drive a complete and concise specification in an incremental specification process. More can be done after the specification is achieved (e.g., verifying that there is an initial state), since such a specification is opened to all the verification and theorem proving means that are available in the tooling framework associated to selected formal methods.
- Demonstrating properties. Given a formal definition of requirements through experimental frames, and a formal specification of the model, theorem proving tools or model checking tools can be used to assist in proving that the specification meets the requirements, i.e., that the experimental frame can be applied to the model and the expected results correctly obtained. For each property of the system, the model checker analyzes system states to check whether they satisfy the desired property. If model checker reaches a state that violates the desired property, the model checker provides a counterexample that indicates how the model could reach the undesired state. The counterexample describes an execution path that leads from the initial state to a state that violates the property. HILLS simulators can be used to replay the violating scenario, to obtain useful debugging information, and adapt the model (in case of modeling error) or the property (in case of property specification error) to resolve the problem. If the property verification by model checking doesn't provide a result, theorem proving can be used or try to reduce the model and repeat the process.

This can open the way to the management of models libraries in one hand, and experimental frames libraries in the other hand, and automatic selection, retrieval and matching processes.

- Internal consistency checks: checking that each class operation preserves the model invariants.
- Semantic consistency checks: semantic consistency checking can consist of checking that model invariants are preserved, conflicts do not exist between invariants defined in the domain model and constraints defined in the discrete event behaviour of the model, no inconsistencies exist between operation specifications of a HiLLS class and state machines configurations and transitions.

We present in this chapter some applications of HiLLS in modelling and analysis of complex systems.

V.2 Alternating Bit Protocol

The alternating bit protocol [Bochmann and Gecsei 1977], is a communication protocol being used for the transmission of messages. This protocol is conceived to make reliable transmissions even in the situations of errors. An error can occur when the communication channel is disturbed. These disturbances can cause a loss, deterioration or duplication of transmitted messages. In this case, only the transmissions made successfully are taken into account, when an acknowledgement of delivery is sent to the source of the message. The protocol ensures also the fact that a message is not lost definitively.

The communication protocol has the following components: a transmitter (Sender), a receiver (Receiver), two transmission channels: a channel for the transmission of the messages and another for the emission of acknowledgement of delivery.

Each communication channel is assumed to be a FIFO (First In First Out) channel such that messages are treated in the order in which they enter the channel. The instantaneous reliability of a communication channel may be affected by factors such as congestion levels and the presence of disturbances from the environment. Depending on the channel's condition, a message may be delivered immediately or after some delay to the other end of the channel. It may also be duplicated or lost within the channel.

The basic idea in the alternating bit protocol is that each message accepted for first transmission is tagged alternatively with 0 or 1 called control bit. This control bit used to distinguish retransmissions of previous message from transmissions of new messages. The tagged message is then periodically retransmitted until the tag is returned by the receiver as an acknowledgement of message receipt.

The Sender accepts a message from its environment and sends it to the Receiver component via the communication channel dedicated to transmitting messages. When the receiver receives the message with the good control bit, it delivers it to its environment and sends (via the transmission channel dedicated) an acknowledgement consisting of the control bit of the message just received to the transmitter (Sender) which authorizes it to accept a new message and to transmit it with a control bit equal to the opposite of the control bit of the previous message. To ensure reliability of communications between $Sender$ and $Receiver$ in the presence of possible message losses, $Sender$ resorts to retransmissions if the acknowledgement is not received after a defined waiting period by resending the message (with the same control bit) on the assumption that the message is lost in the channel. This

process is repeated until the expected acknowledgement is received when the next message in the buffer (if any) is sent with a control bit which is the binary complement of that of the last message sent.

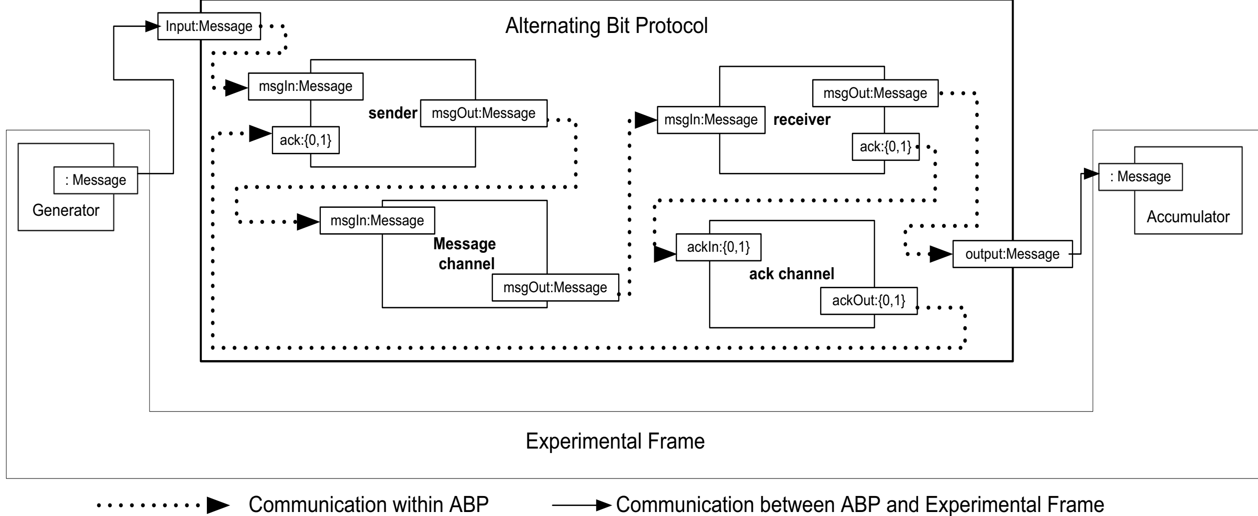

Figure 55. Alternating Bit Protocol and its Experimental Frame

Figure 55 shows a block diagram of the protocol coupled with an experimental frame. The experimental frame is a representation of the environment within which the ABP specification is observed. The Generator and Accumulator represent the source and destination respectively of the message being transmitted using the protocol.

V.2.1 HiLLS specification of ABP

Figure 56 shows a black box view of the ABP's specification in HiLLS showing the various components of the system with their input and output interfaces as well as the hierarchical composition of the entire system. The ABProtocol has four components: sender (an instance of HSystem Sender), receiver (an instance of HSystem Receive), msgChannel (an instance of HSystem CommLine[T] with T as Message) and ackChannel (an instance of HSystem CommLine[T] with T as Integer). Each of sender and receiver has a complex attribute *buffer* which is a list of instances of HClass Message. CommLine[T] is a generic(template) HSystem for communication lines, hence msgChannel and ackChannel are communication lines for transmitting messages (Message) and acknowledgement bits (Integer) respectively. Message is modelled as an HClass and not HSystem for the same reason that it is considered to be a non-autonomous object. It can be seen from Figure 56 that the specification consists of a composed HSystem *ABProtocol* with hcomponent references to three atomic HSystem specifications: *Sender, Receiver* and *CommLine*[T]. We will zoom into one of the components, and the composed system to show the details of their internal definitions. We expect that this will be sufficient to show the reader the differences between the internal details of an atomic and a composed system specification in HiLLS. The main difference between the two is that a composed system defines couplings between its components while an atomic system does not.

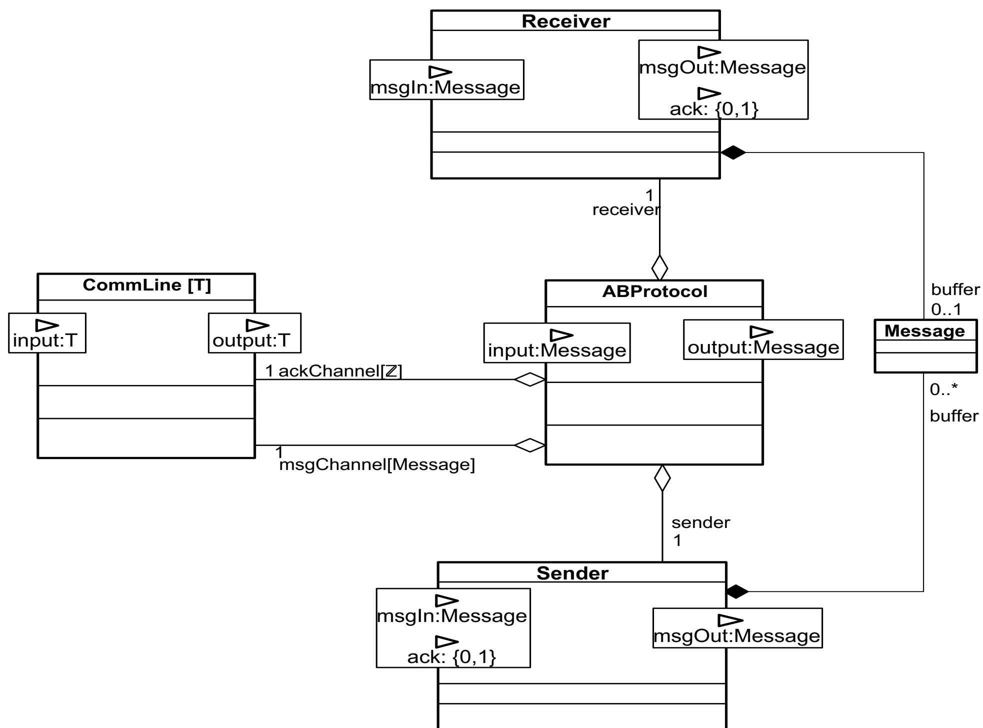

Figure 56. HiLLS metamodel of the alternating bit protocol

V.2.1.1 HiLLS' specification of Message

Message HClass is shown in Figure 57. The state schema declares two state variables *header* with a constraint that restricts the possible legal values of *header* to 0 or 1 representing the header (control) bit of the message. The *Init* specifies the default values of the state variables. In addition to the *Init*, four operations are defined. It is important to state here that every operation with a defined type has an implicit output variable, *out!*, having same type as the operation. This *out!* variable used to store the output produced by the operation. The decorations '!' and apostrophe (') is a notation we have adopted from Object-Z to denote output variables and the final value of a state variable after an operation is executed respectively.

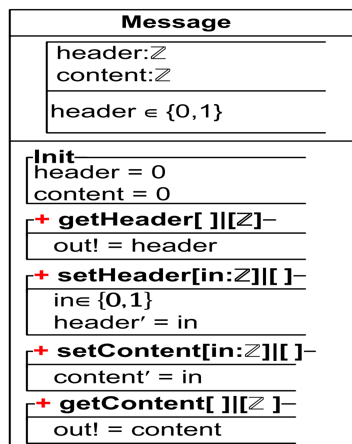

Figure 57. Message HClass

V.2.1.2 Generator

The Generator (Figure 58) represents the component of the experimental frame that is responsible of generating messages to send for the sender. The Generator has an attribute $messages$ which is a sequence of Message and a real parameter $hatchingPeriod$ which is the time period that it spends before generating a new message ($hatch()$) for transmission through the unique output port $outPut$. It has only one configuration which duration is defined by the parameter ($duration = hatchingPeriod$).

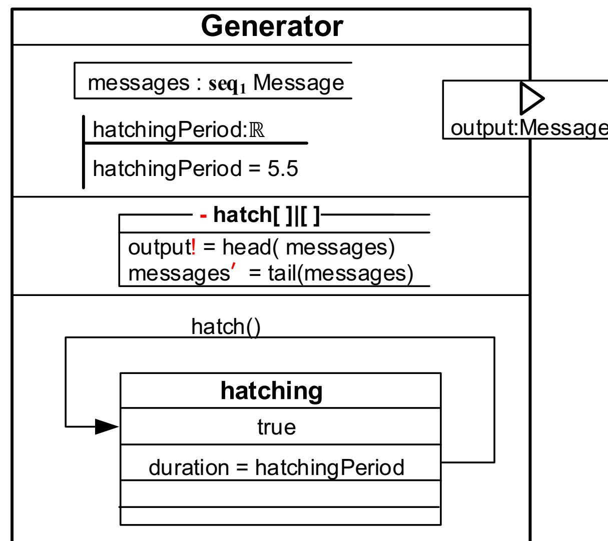

Figure 58. HiLLS model of the Generator

V.2.1.3 Sender

The Sender model is described by the HSystem in Figure 59. Sender has an attribute $buffer$ (Figure 56) which is a sequence of messages for storing messages for transmission. Sender also has an attribute $flag$ of type integer that represents the control bit of the last transmitted message and a parameter $waitinPeriod$ that define the waiting duration of the Sender after the sending of a message. It is duration after which the Sender timeout and resend the message remaining in $buffer$.

The sender has two ports: a port $msgIn$ for the acceptance of the messages from the environment and a port ack for the acknowledgements of delivery. It has an output port $msgOut$ for the transmission of received messages with their control bit. Initially the buffer is empty ($buffer = <>$) and the flag is equal to zero ($flag = 0$); this situation corresponds to the $idle$ configuration. When a new message is received in the $idle$ configuration, the sender adds it to the buffer ($buffer' = buffer \cap < m >$), flip the flag ($FlipFlag()$) and set the control bit to the actual value of the flag ($head\ (buffer).setFlag(flag)$) for transmission in the transient $sending$ configuration.

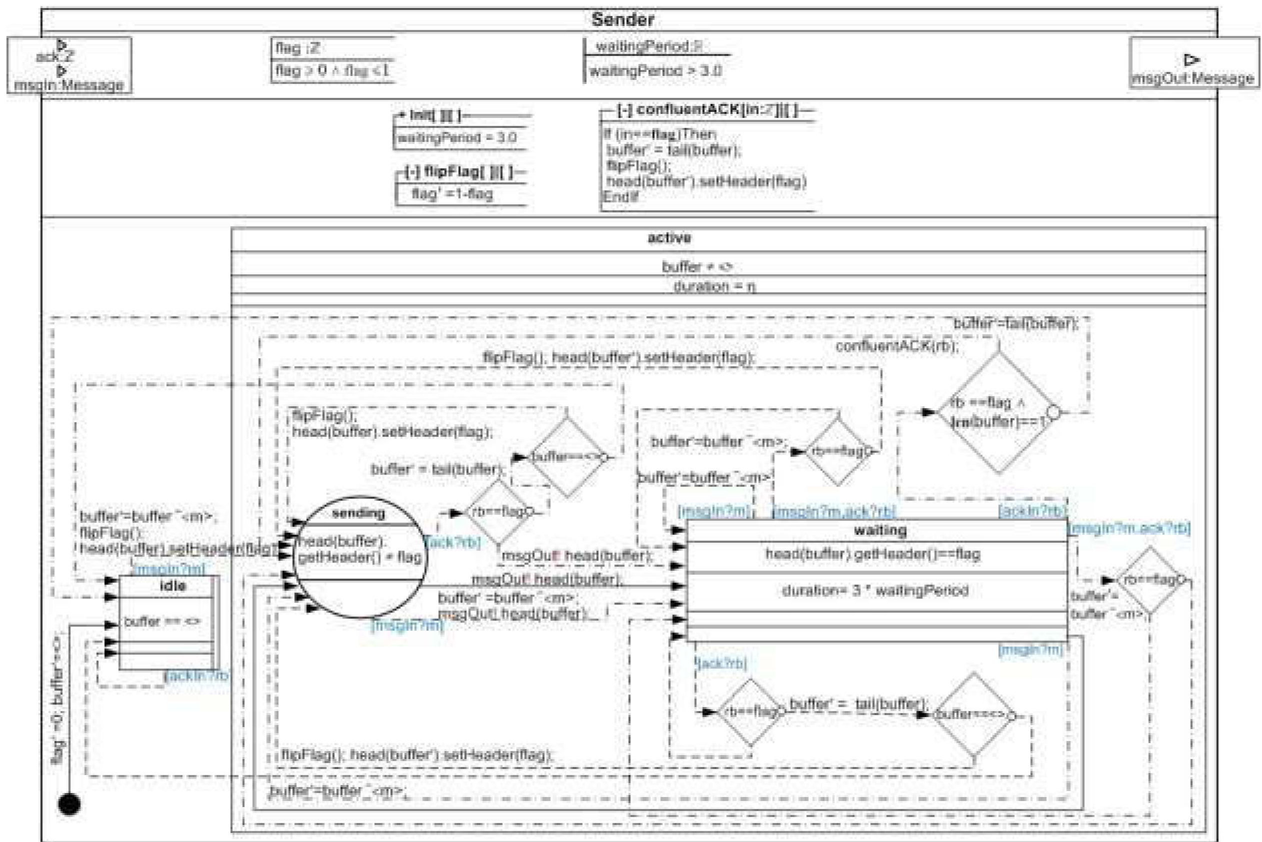

Figure 59. HiLLS model of the Sender

V.2.1.4 Receiver

Receiver is described by the HSystem in Figure 60. Recall from Figure 56 that Receiver has an attribute *buffer*, which is a collection of messages as depicted by the containment reference from Receiver to Message. Figure 56 shows an additional state attribute, *flag* of type integer whose legal values are restricted to the elements of the set {0,1} by the state constraints. When a new message is received, its header bit is extracted and stored in *flag* before it is acknowledged and delivered; the value of *flag* is compared to the header bits of subsequent messages that arrive, same value implies duplicate messages while a different value implies a new message. This behavior is described in detail by the fourth compartment. The receiver has a global variable, *indicator* of type Colors as defined in the second compartment. This variable is used to model the light indicator as an *activity* that manifests the instantaneous states of the system to an observer in real time. It has an input port, *msgIn* of type Message and two output ports, *ack* and *msgOut* of types Integer and Message respectively. Messages are received through the *msgIn* and acknowledged through *ack* while messages are delivered (without duplicates) to the target device through *msgOut.*

The third compartment contains the *Init* and two operations, *setFlag* and *checkHeader*. The fourth compartment contains the description of the system's behavior with the configuration transition diagram. Two configurations, *waiting* and *receiving* are defined with *predicate* properties *buffer* = <> and *buffer* ≠<> respectively. i.e., each configuration is assumed whenever its specified predicate (on the state variables) is satisfied. Hence, from the specification of the *Init,* the initial configuration of any object of *Receiver* is *waiting.* At the assumption of each configuration, the activity function invokes the *display* operation which displays the specified color of light indicator for a period equal to the sojourn time of the

configuration. For example, the yellow indicator is displayed during the *waiting* configuration. Since *waiting* is a passive configuration as depicted by its concrete representation, the system remains in this state until a message *m* is received at the input port *msgIn* which triggers an *external transition* to the *receiving* configuration.

The computation accompanying the transition adds the received message to buffer which leads to the satisfaction of the property of the target configuration. The prime decoration on *buffer'* in the computation indicates its final state after the transition. The *receiving* configuration is transient, hence an internal transition occurs automatically; the first computation invokes the *checkHeader* operation to determine whether the received flag has the same flag as the current value of *flag*. Note that the *flag* stores the header bit of the last message received, so if *checkheader* returns *true*, it means the received message is a duplicate of the previous one; hence only acknowledgement is sent to sender by taking the path described by the upper arrow to *waiting* configuration. If *checkHeader* returns false, the message received is genuinely a new message; hence it is acknowledged and delivered as described by the operations accompanying the lower arrow that leads to the *waiting* configuration.

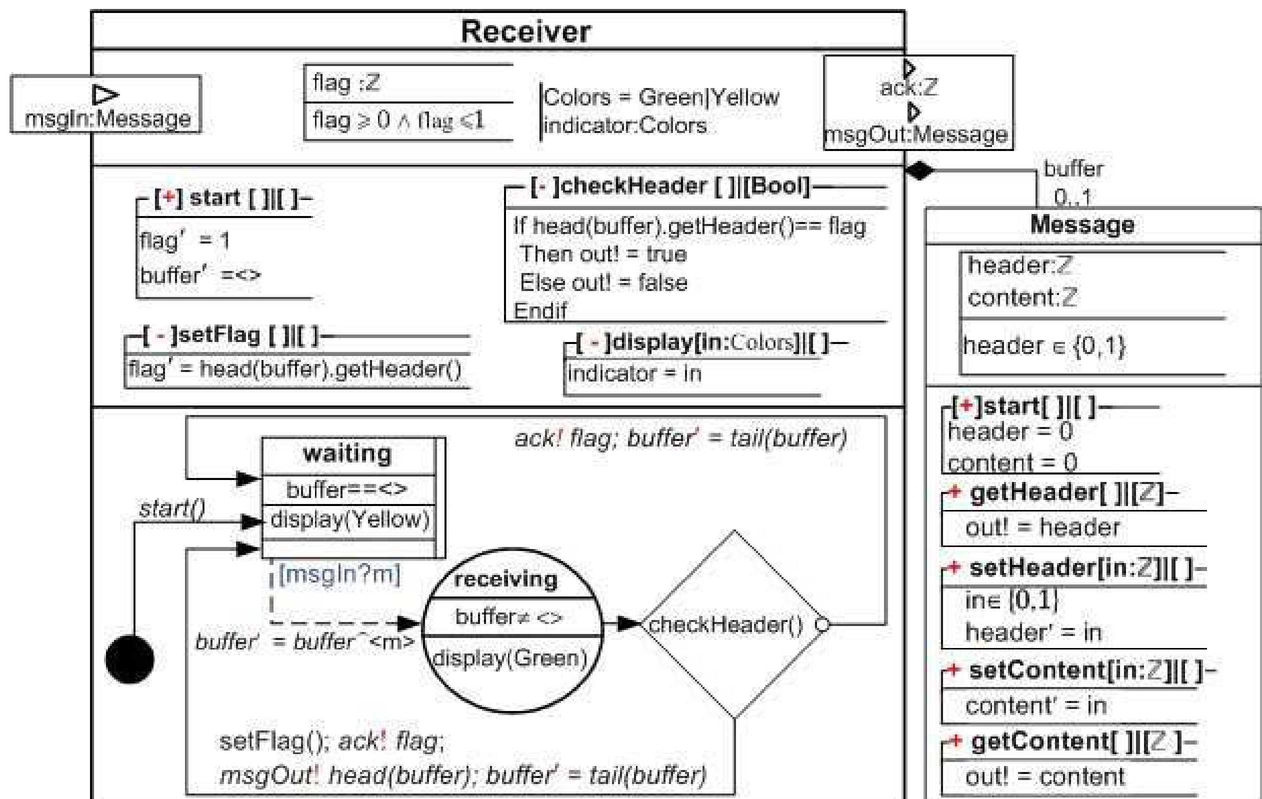

Figure 60. Receiver HSystem

V.2.1.5 Communication line

The Communication line $ComLine[T]$ presented in Figure 61 is a unitary HSystem with generic parameter T that is the type of data that can be transmitted through the line. $ComLine[T]$ has one input port $input$ and one output port $output$ both of type T. It has two configurations $wainting$ ($buffer =<>$) and $sending$ ($buffer \neq <>$). Initially, the communication line waits for a data of type T (message or acknowledgment) in the $waiting$ configuration. At the reception of a message msg on its input port ($input?msg$), it adds it in

the $buffer$ and go to the $sending$ configuration for the transmission of the message ($pushForward()$) after some period ($duration = 2 * delay()$). It transmits, or delays or lost the received message ($pushForward()$)) depending on the level of reliability c_level of the line and passes to the $waiting$ configuration if $=<>$, and becomes ready to accept a new message. If $buffer \neq <>$ it go back again to the $sending$ configuration. While in the $sending$ configuration the line can receive a new message which will be added to the $buffer$.

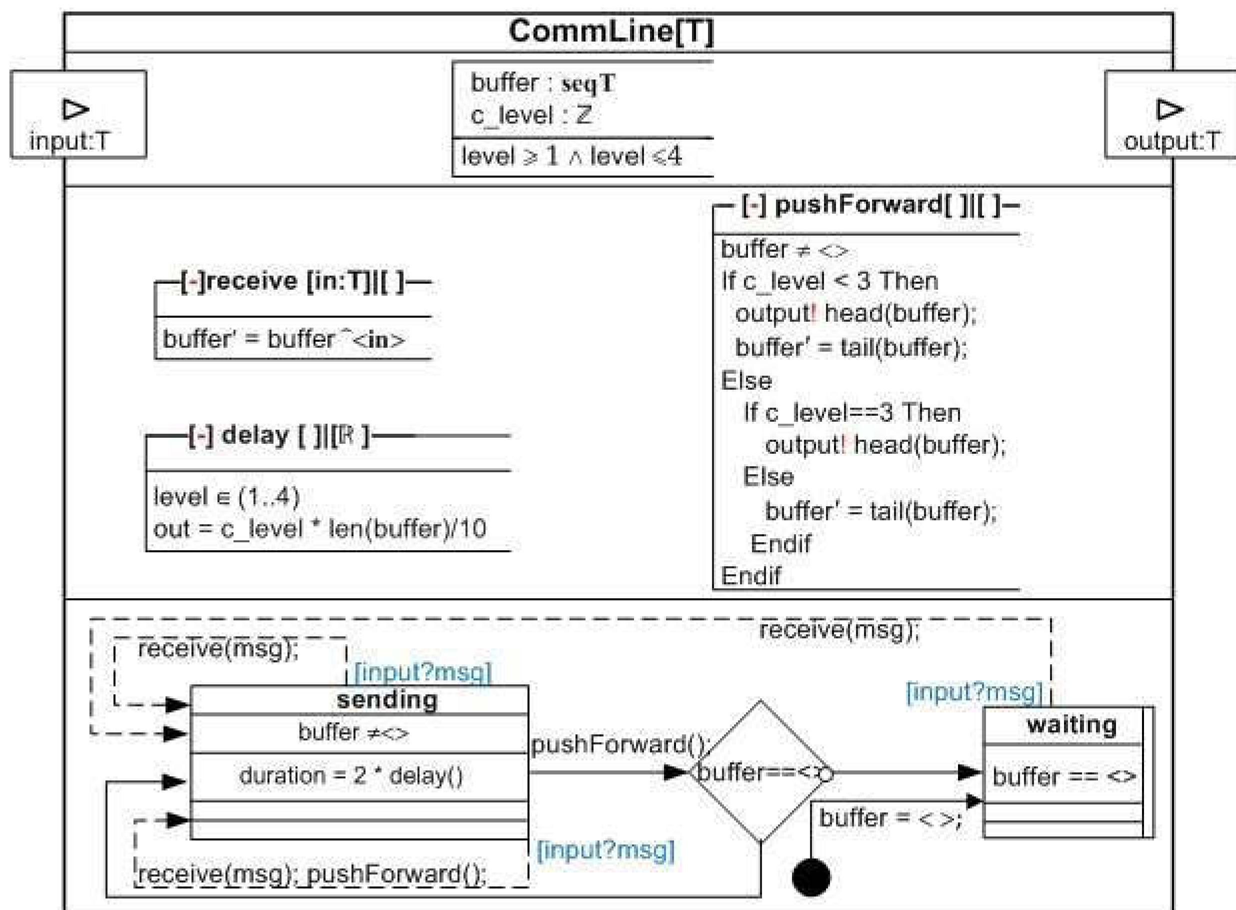

Figure 61. The HiLLS model of the Medium

V.2.1.6 Accumulator

The Accumulator (Figure 62) is the component of the experimental frame that plays the role destination of the messages transmitted. It has one input port $input$ of type $Message$ and one attribute $messages$ which is a sequence of type $Message$. The Accumulator only receives messages ($input?msg$) and store them in $messages$ ($messages' = messages \cap < msg >$). It has no internal and confluent transition because it has no output port.

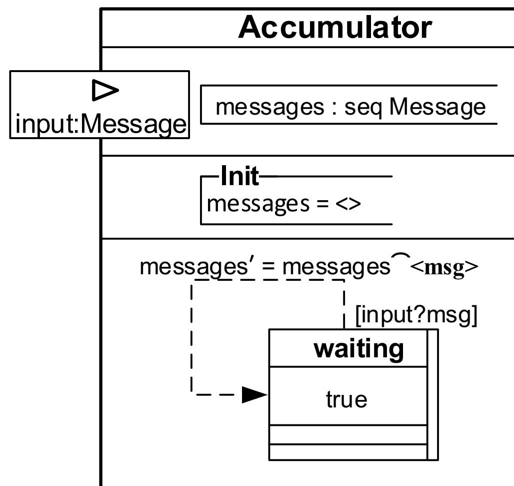

Figure 62. The Model of accumulator

V.2.1.7 Protocol

The definition of the ABP is described by Figure 63. An input port and an output port are defined each having type *Message.* The state variables are described by the hcomponent references *sender, receiver, ackChannel* and *msgChannel* in Figure 56. The predicate part of the state schema defines the constraints that specify the ports that must be connected (coupled) permanently. The *connect()* operation simply ensures the couplings are established between appropriate ports at the creation of an object of ABProtocol. The first coupling specification, *sender.msgIn=self.input,* is an *EICg* that connects the an input port, *input,* of *ABProtocol* to an input port, *msgIn* of *sender*, which is one of its components.

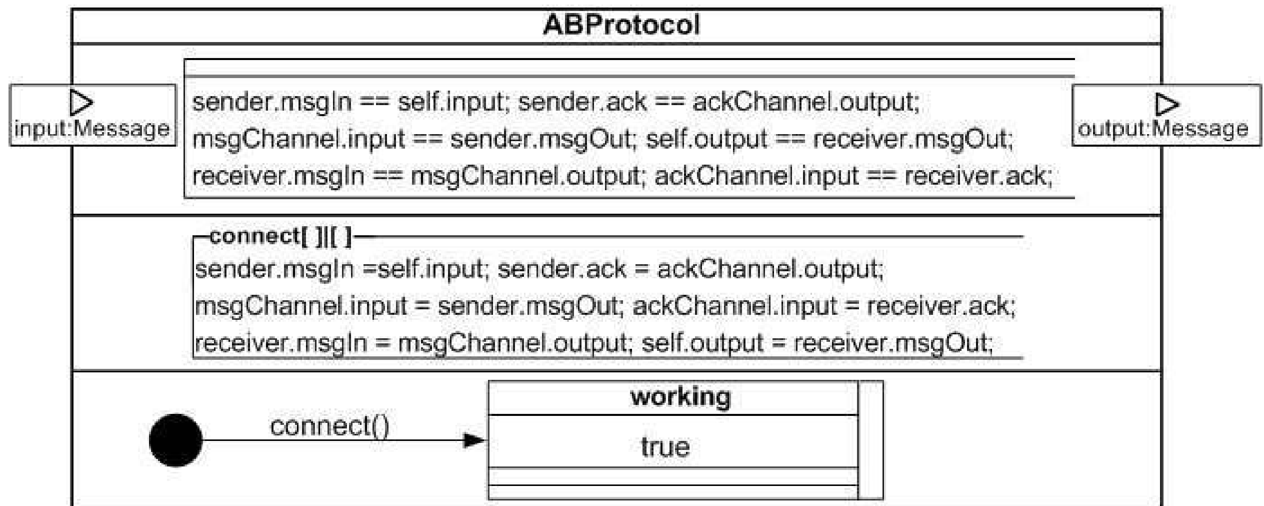

Figure 63. ABProtocol HSystem

The couplings *sender.ack = ackChannel.output, msgChannel.input = sender.msgOut, receiver.msgIn = msgChannel.output* and *ackChannel.input = receiver.ack* are all *Internal Couplings* between peer elements of *ABProtocol.* In each case, the right hand side describes the sending port and the receiving port is described by the left hand side. For example, the first one states that the output port *output* of *ackChannel* is coupled with the input port *ack* of *sender.* In the fourth compartment, only one passive configuration *working* is specified which does not change. This implies that the composed model does not add any extra behavior to that which is defined by the interactions between its components.

V.2.2 Equivalent models of the ABP in DEVS

To illustrate how HiLLS models are generated, we will show equivalent DEVS models only for the Receiver and the Protocol itself.

To add clarity to the example presented here, we describe the message object based on the rules presented in Section 5. Therefore, we represent *Message* as a mathematical object in the form of a structure as:

$$Message = \langle V, F \rangle$$

Where V and F are sets of variables and functions respectively. Therefore, from the HiLLS specification of Message HClass in Figure 11, we derive V and F as follows:

$$V = \{(header, \mathbb{Z}), (content, \mathbb{Z}) |\ header \in \{0,1\}\}$$

$$F = \{getHeader, setHeader, getContent, setContent\}$$

$getHeader: Message \rightarrow \mathbb{Z}$ is a function that returns the *header* bit of a message
$setHeader: \mathbb{Z} \rightarrow \mathbb{Z}$ is a function that sets the value of the *header* bit of a message
$getContent: Message \rightarrow \mathbb{Z}$ is a function that returns the *content* of a message
$setContent: \mathbb{Z} \rightarrow \mathbb{Z}$ is a function that sets the *content* of a message.

V.2.2.1 Receiver in DEVS

The *Receiver* in Figure 60 has no component, hence it is translated to an Atomic DEVS.

$Receiver_{DEVS} = \langle X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta \rangle$
$X = \{(msgIn, Message)\}$
$Y = \{(ack, \{0,1\}), (msgOut, Message)\}$

$S = \{((buffer, seq\ Message), (flag, \{0,1\}),$
$(phase, \{waiting, sending\})) |\ phase = waiting \Leftrightarrow buffer = \langle\ \rangle, phase = sending \Leftrightarrow$
$buffer \neq \langle\ \rangle\}$

The state variables *flag* and *buffer* constitute a subset of the *S* in DEVS and the complement is provided by the variable *phase* whose domain is the set of configuration names specified in the fourth compartment of the HSystem in Figure 60. The collection of the *properties* of each configuration constitute the predicate part of set *S*.

V.2.2.1.1 Internal transition function, $\boldsymbol{\delta_{int}: S \rightarrow S}$

$\delta_{int}(sending, buffer, flag) = (waiting, tail\ (buffer), flag),$
$if\ head(buffer).getHeader(\ ) = flag$

$\delta_{int}(sending, buffer, flag) = \big(waiting, tail\ (buffer), head(buffer).getHeader(\ )\big),$
$if\ head(buffer).getHeader(\ ) \neq flag$

The derived internal state transition function is described by these two equations. From a state in which *phase = sending* and *if* condition is satisfied, then an internal transition event will result into a state in which *phase=waiting, buffer* is its initial value before transition without the first element and *flag*'s value retains its value before transition. This equation is derived from the upper path from *sending* to *waiting* configuration in Figure 60 by extracting all computations along the path except those meant for sending output events to some ports. Any

state variable that is not modified in the computations is assumed to be unchanged; that is the case with *flag* as it is not modified in the computations and it is reported in the derived DEVS model as unchanged.

Similarly following the lower path from *sending* to *waiting* in Figure 60, from a state in which *phase = sending* and *if* condition is satisfied, then an internal transition event will result into a state in which *phase=waiting, buffer* is its initial value before transition without the first element and *flag* has a value equal to the output of the function invoked in the third element of the triple (target state).

V.2.2.1.2 External transition function, $\boldsymbol{\delta_{ext}}: \boldsymbol{S} \times \mathbb{R}^{+} \times \boldsymbol{X^b} \rightarrow \boldsymbol{S}$

$\delta_{ext}((waiting, buffer, flag), e, m \in Message) = (sending, buffer ^\frown \langle m \rangle, flag)$

From a state in which *phase=waiting* and the corresponding predicate stated is satisfied, the receipt of an event $m \in Message$ will trigger an external state transition into a state in which *phase=sending*, *buffer* is equal to its value before transition with the received event *m* appended to the sequence and *flag*'s value is equal to its value before transition. This equation is derived from Figure 60 by following the external transition from configuration *waiting* to *sending;* the trigger [*msgIn?m*] translates into the input event $m \in Message$ where *Message* is the type of input port *msgIn* in Figure 60.

V.2.2.1.3 Confluent transition function, $\boldsymbol{\delta_{ext}}: \boldsymbol{S} \times \boldsymbol{X^b} \rightarrow \boldsymbol{S}$

No confluent configuration is specified in Figure 60, hence $\delta_{ext}(s) = \emptyset. \forall s \in S$

V.2.2.1.4 Output function, $\boldsymbol{\lambda}: \boldsymbol{S} \rightarrow \boldsymbol{Y^b}$

$\lambda(sending, buffer, flag) = \{(ack, flag), (msgOut, head\ (buffer))\},$
$if\ head(buffer).getHeader(\,) = flag$

$\lambda(sending, buffer, flag) = \{(ack, flag)\},\ if\ head(buffer).getHeader(\,) \neq flag$

The derived output functions are described by these two equations. They are obtained by traversing the (non-external) configuration transition paths in a HiLLS specification and extracting the output expressions if any. The first equation is derived by traversing the lower path from *sending* to *waiting* in Figure 60; this path contains two output instructions *ack!=flag* and *msgOut!=head(buffer)* which translate into (*ack, flag*) and (*msgOut, head(buffer)*) respectively. Similarly, the second equation is derived by traversing the upper internal transition path from *sending* to *waiting* in Figure 60.

V.2.2.1.5 Time advance function, $\boldsymbol{ta}: \boldsymbol{S} \rightarrow \mathbb{R}^{+} \cup \{+\infty\}$

$ta(waiting, buffer, flag) = +\infty, \forall\ buffer \in seq\ Message, flag \in \{0,1\}$

$ta(sending, buffer, flag) = 0, \forall\ buffer \in seq\ Message, flag \in \{0,1\}$

The time advance function derived from the HiLLS specification in Figure 60 is described by these two equations. They are simply derived from each configuration by extracting the specified *sojourn time*. In this example, the two configurations specified have pre-defined sojourn times inherent in their concrete representation as explained previously in Section 4; the passive configuration *waiting* has a pre-defined sojourn time of positive infinity while the

transient configuration *sending* has a pre-defined sojourn time of zero. Supposing a finite configuration is specified, the sojourn time defined by the modeler will be extracted to build the time advance function of the DEVS equivalent model.

V.2.2.2 ABProtocol in DEVS

ABProtocol (Figure 63) translates to a Coupled DEVS model because its set of *hcomponents* is not empty.

$ABProtocol_{DEVS} = \langle X, Y, D, \{M_d\}_{d\in D}, EIC, IC, EOC\rangle$
$X = \{(input, Message)\}$
$Y = \{(output, Message)\}$
$D = \{sender, msgChannel, ackChannel, receiver\}$

X and *Y* are derived from the input and output interfaces respectively of Figure 63. The set D is built from Figure 56 by extracting the names of all hComponent references having ABPRotocol as source. The set $\{M_d\}_{d\in D}$ refer to the complete DEVS equivalent of each of the target HSystem of the hComponent references from which set *D* is built. We have provided the DEVS equivalent of the HSystem referenced by $receiver \in D$. i.e., $Receiver_{DEVS}$. Therefore, $M_{receiver} = Receiver_{DEVS}$. Similarly, if the detailed specifications of other HSystems in Figure 56 were given, then we would have $M_{sender} = Sender_{DEVS}$, $M_{ackChannel} = CommLine[\mathbb{Z}]_{DEVS}$ and $M_{msgChannel} = CommLine[Message]_{DEVS}$.

From Figure 56, the DEVS coupling relations can be derived from the coupling predicates specified in the Init schema as follows:

The RHS of a coupling predicate translate to the influencer system and port of the DEVS coupling predicate while the LHS translate to the influenced system and port. We provide a comprehensive guide in Figure 64 to remind the reader of how to identify to which of the three coupling relations it belongs. For example, considering the first coupling predicate.

LHS (Left Hand Side)		**RHS (Right Hand Side)**		**Coupling relation**
System	**port**	**System**	**port**	
$d \in D$	*input*	*self*	*input*	*EIC*
$d \in D$	*input*	$d \in D$	*output*	*IC*
Self	*output*	$d \in D$	*output*	*EOC*

Figure 64. couplings mapping

sender.msgIn = self.input; LHS = sender.msgIn and RHS = self.input (self = coupled model in context), the system reference of the LHS belongs to the group $d \in D$ (i.e., a component) and its port reference refers to an input port. The system reference of the RHS is *self* and its port reference is also an input port; therefore, this coupling predicate translates to an element of EIC as shown below. The remaining five coupling predicates can be translated in the same manner to derive the elements of EOC and IC below.

$EIC = \{ ((self, input), (sender, msgIn))\}$
$EOC = \{((receiver, msgOut), (self, output))\}$

$IC = \{((sender, msgIn), (msgChannel, input)),$
$((msgChannel, output), (receiver, msgIn)),$
$((receiver, ack), (ackChannel, input)),$
$((ackChannel, output), (sender, ack))\}$

Suppose we have an experimental frame comprising a message generator and a message acceptor representing the source and destination of messages respectively, we can couple the two to the input and output respectively of the ABProtocol to simulate the behavior of the protocol. We may couple these components with the protocol to obtain a closed system described as:

$ABP_{expFrame} =< X, Y, D, \{M_d\}_{d \in D}, EIC, IC, EOC >$

$X = Y = \{\}$ because it is a closed system.
$D = \{generator, abp, accumulator\}$
$\{M_d\}_{d \in D}$: $M_{generator} = Generator, M_{abp} = ABProtocol\ and\ M_{accumulator} = Accumulator$

$EIC = EOC = \{\ \}$ because it is a closed system.
$IC = \{((generator, output), (abp, input)),$
$((abp, output), (accumulator, input))\}.$

V.2.3 Z specification and Analysis of the ABP

Each unitary model is translated in Z notation in Z/EVES theorem prover. Operations pre and post conditions are verified. The composite model of the protocol is a CSP process.

The Message class is translated to a state schema *Message* and operation schemas (Figures 65-67): *Init*, *getHeader*, *setHeader*, *getContent* and *setContent*;

Figure 65. Message and Init

Figure 66.getHeader and setHeader

Figure 67. getContent and setContent

The receiver specification in Z consist of the state schema *Receiver, ReceiverStatus* enumeration (Figure 68) and operation schemas: *InitReceiver* (Figure 72), *waiting2sentingTransitionExt* (Figure 72), *senting2waitingTransitionINT1* (Figure 74), and *sending2waitingTransitionINT2* (Figure 75).

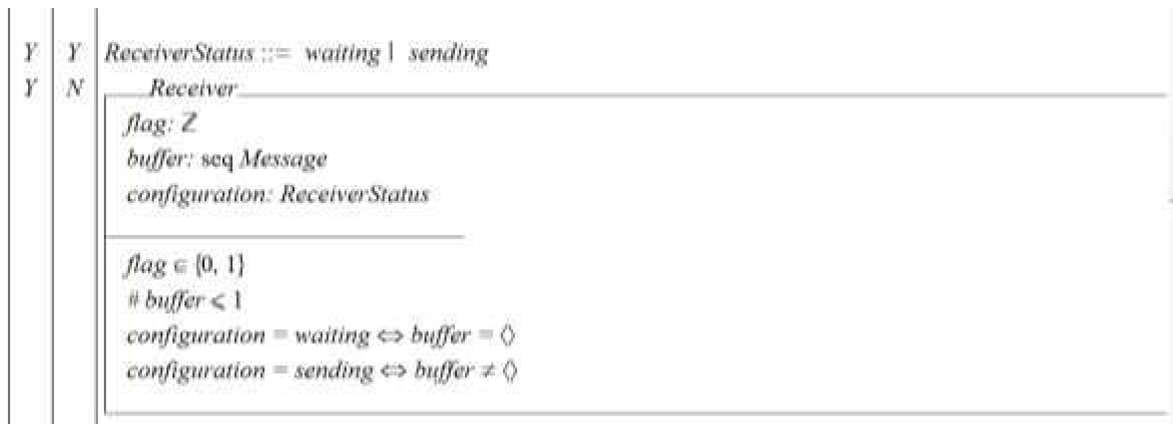

Figure 68. Receiver in Z

Z/EVES tool offer an automatic support for checking for errors (syntax and type checking, domain checking, consistency checking) and exploring a specification (schema expansion, preconditions, invariants, refinement, test cases, test theorems). The specification can be checked incrementally and user can solve errors accordingly.

Domain checking is a strongly recommended verification to do with Z specifications. When we checked the Receiver schemas, its status in the left of the tool shows that it is syntactically and type correct, but that it has an unproved goal. Selecting “show proof” shows the following predicate (Figure 69):

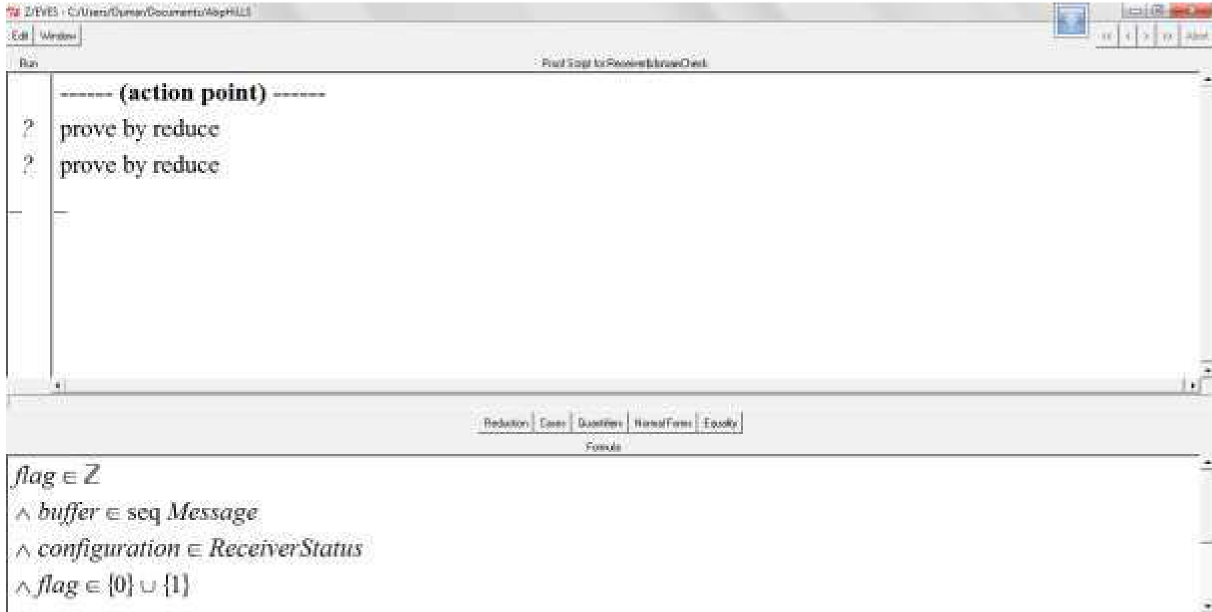

Figure 69. Show proof of Receiver State schema

There is one conjunct to prove in the conclusion. To prove it, some simplifications are necessary. We can do this by using a rewrite or reduce command. The rewrite command produces the following result (the predicate is proven to be true) (Figure 70):

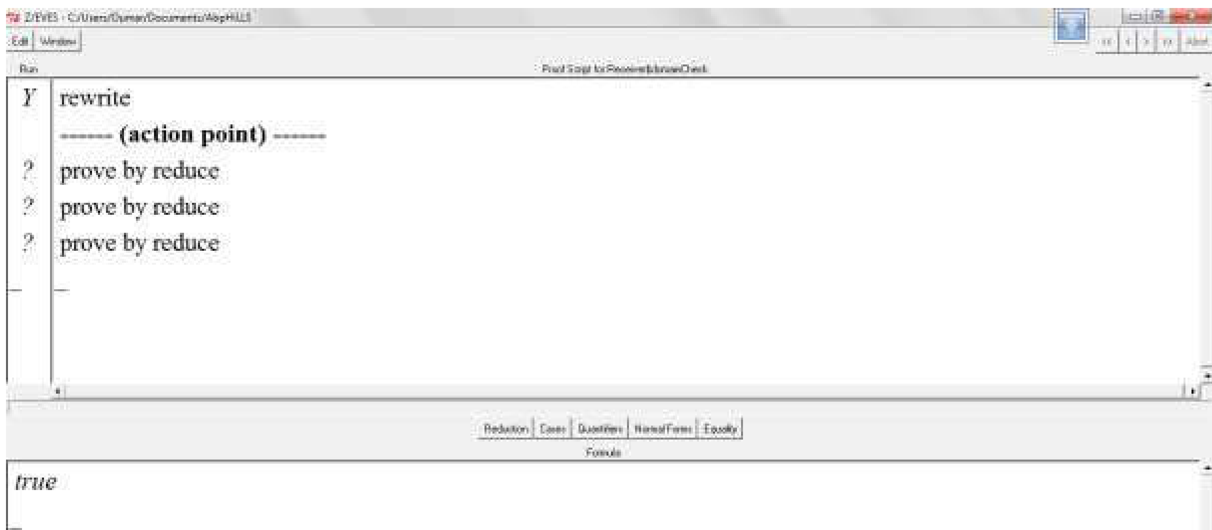

Figure 70. Proof of Reciver state schema

The proof by reduce command produce the same result (Figure 71).

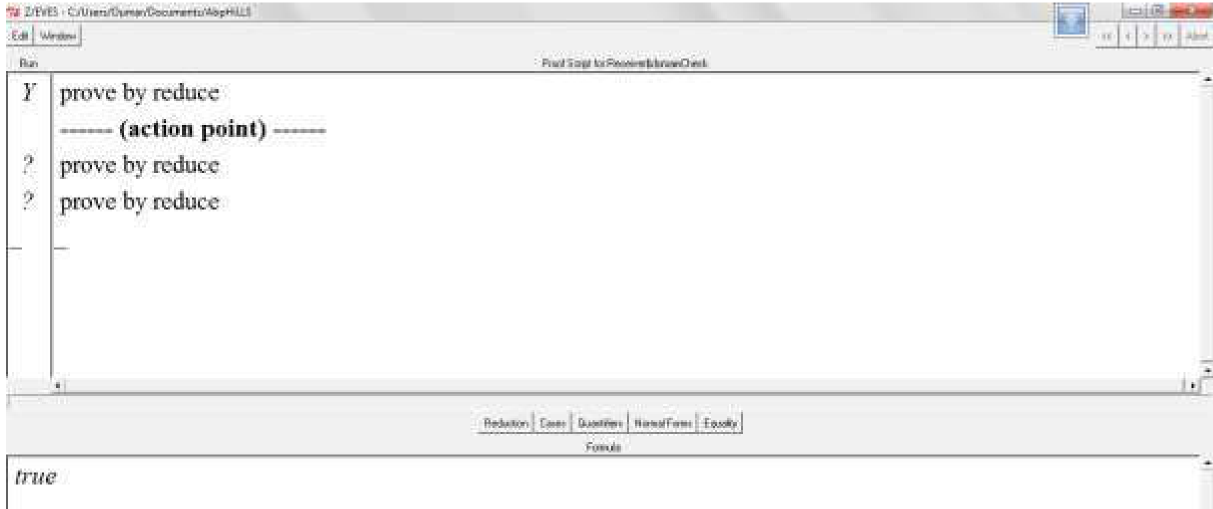

Figure 71. proof by reduce of Receiver state schema

The *waiting2sendingTransitionEXT* operation schema corresponds to the mapping to Z of the external Transition going from waiting configuration to sending configuration in HiLLS specification of the Receiver.

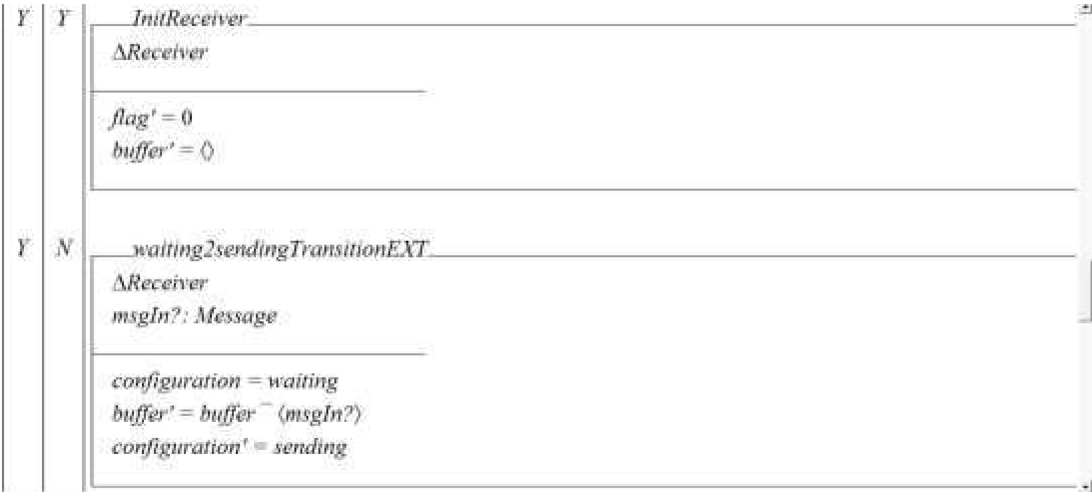

Figure 72. Init and waiting2sendingTransition

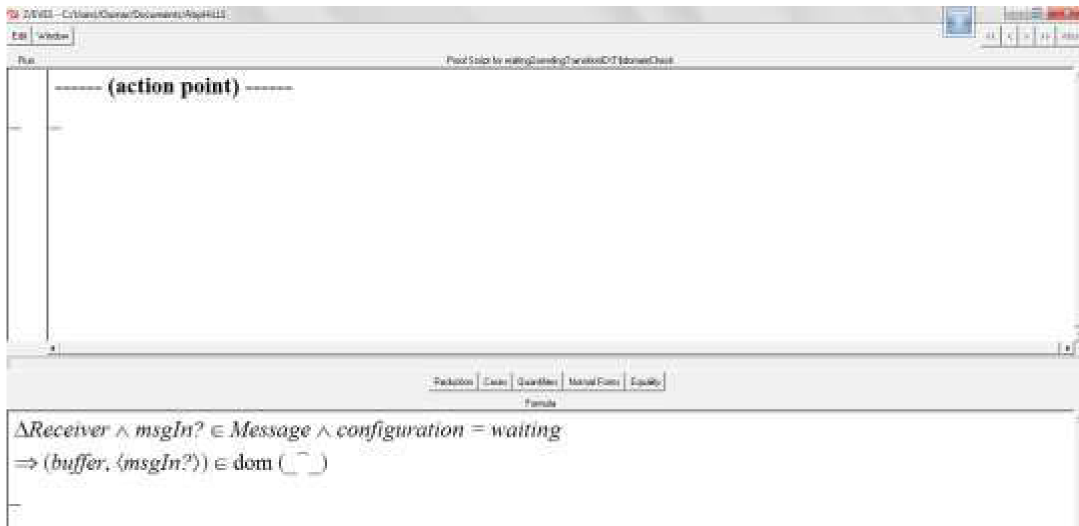

Figure 73. show Proof on waiting2sendingTransitionExt

The *senting2waitingTransitionINT1* and *senting2waitingTransitionINT2* operations schemas are the mapping to Z of the conditional internal transition from *sending* configuration to *waiting* configuration.

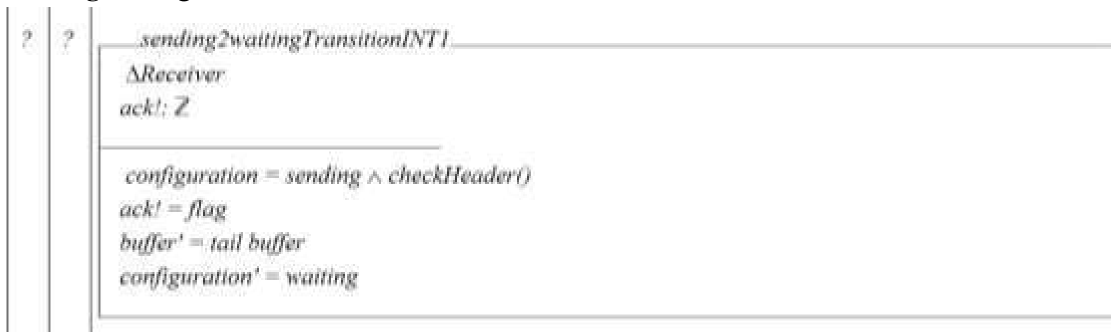

Figure 74. sendint2waintingTransitionINT1

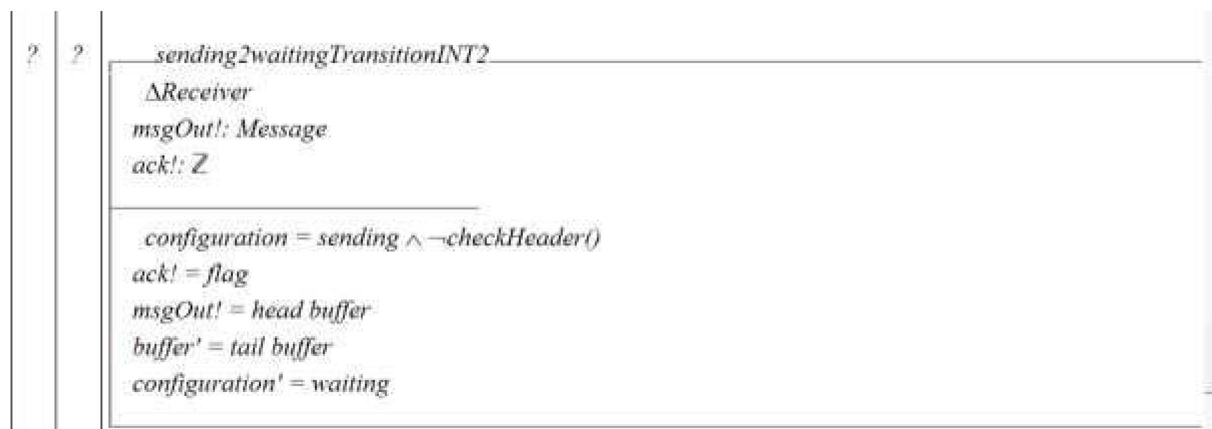

Figure 75. sending2waitingTransitionINT2

By checking the specification of *senting2waitingTransitionINT1* and *senting2waitingTransitionINT2,* Z/EVES shows that the syntaxes are not correct (Figure 76) and the schemas are not proved to be consistent.

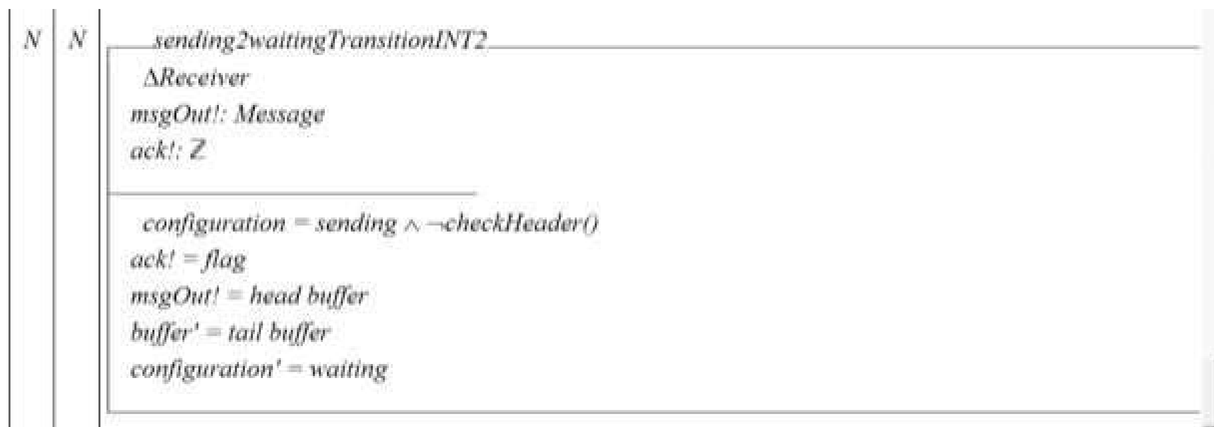

Figure 76. syntax error in sending2waitingTransitionINT2

In Z/EVES, one can also check if initialization of the system is possible, i.e. initial state exist with respect to the constraints of the initialization schema. For the Receiver we have to prove the following theorem (Figure 77) in order to show that initial state exists and the state schemas Receiver is consistent:

Figure 77. Existence of initial state

The Rewrite command has failed to prove the theorem (Figure 78).

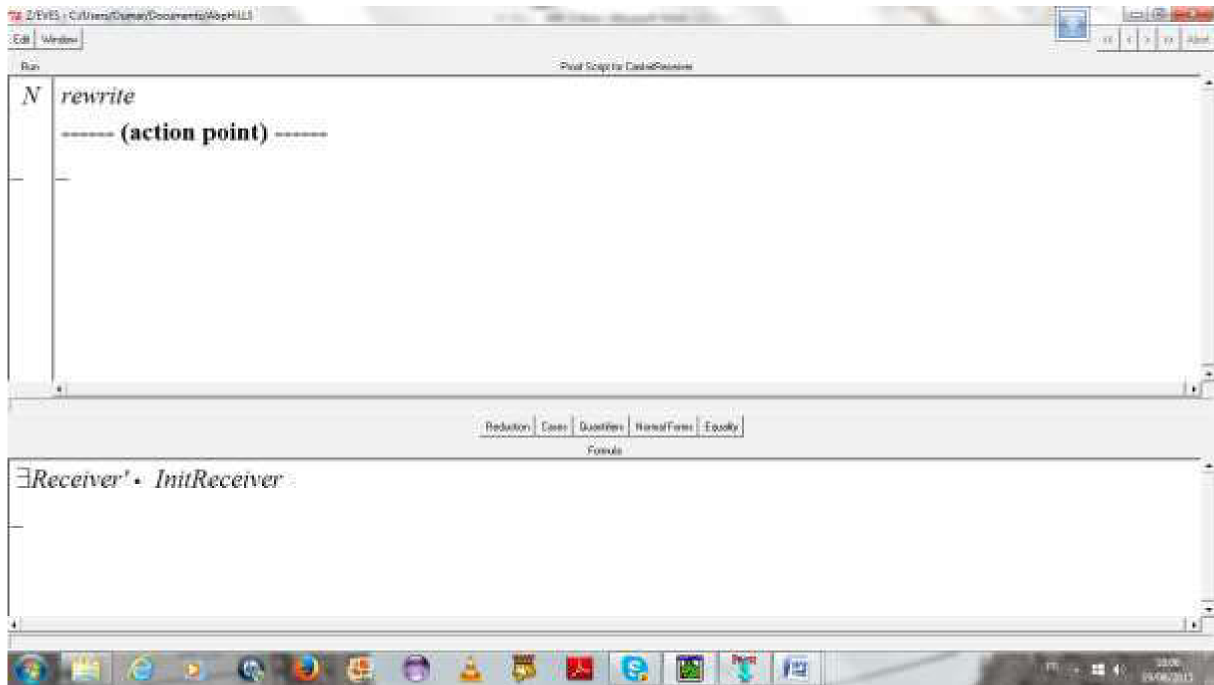

Figure 78. unsuccessful proof of the theorem

The prove by reduce command has succeeded to prove the theorem (Figure 79).

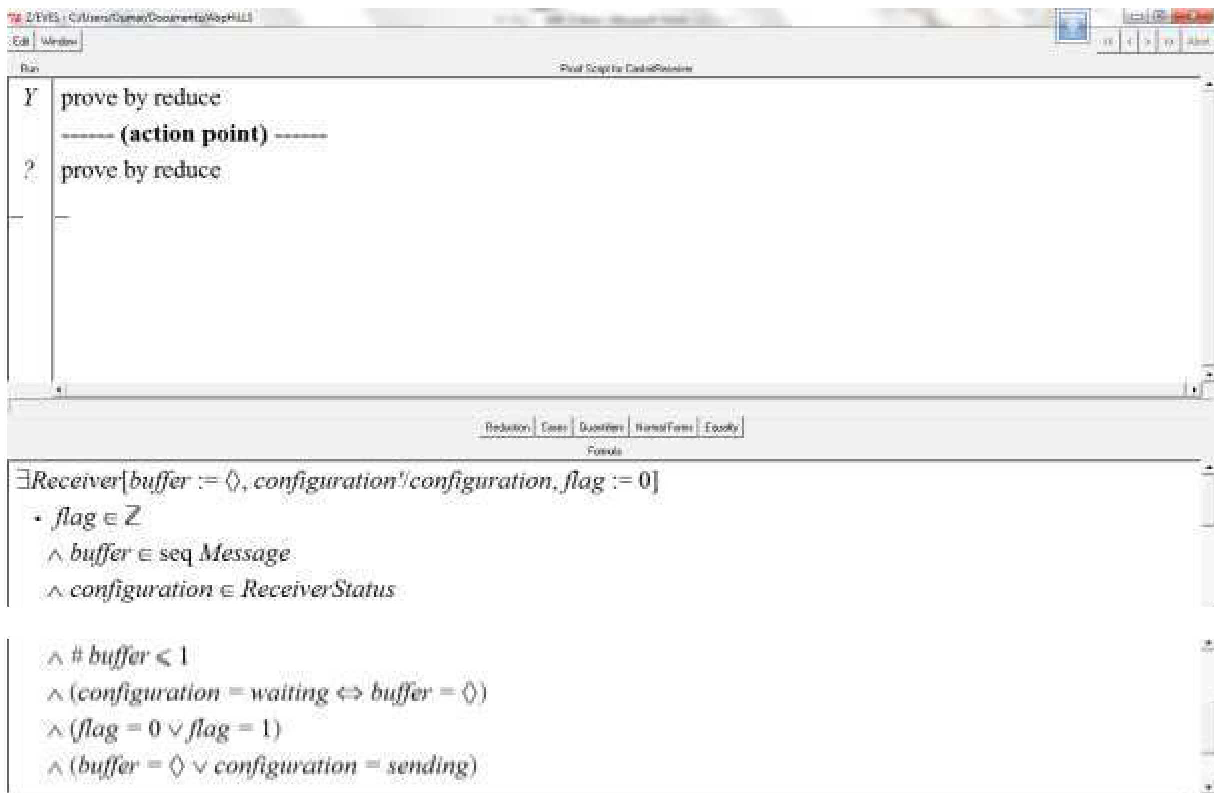

Figure 79. proof of the theorem

It is also possible to check for local inconsistency by using similar theorems.

More verification can be done by using the history of the model (example for Sender) (Figure 80).

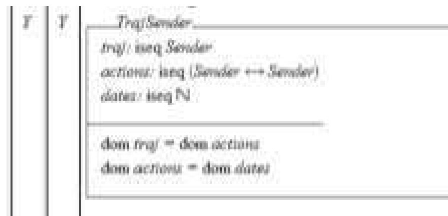

Figure 80. Trajectories in Z

The schemas trajSender allow one to express complex properties on trace of the model such as safety and liveness properties and fairness constraints. These properties can be written in temporal logic formulas and translated into Z theorem because Z/EVES does not support directly temporal logic formulas.

V.3 Automated Teller Machine (ATM)

We present a model of the mechanical processes of the Cash Deposit Module (CDM) of an Automated Teller Machine (ATM) to illustrate dynamic structure system modelling and analysis with HiLLS.

The Cash Deposit Module (Figure 81) allows a customer to deposit a bundle of currency notes into an account. It composes six components that collaborate to process the bills. It first checks the genuineness of the bills presented by checking for some security properties, any unrecognized bill is returned to the customer. The accepted bills are temporarily held in the machine to request for a confirmation of the transaction from the customer. If the transaction is confirmed, the bills are permanently stored in the machine while the transaction runs to completion. Otherwise, the bills are returned to the customer while the transaction is being cancelled. The following are the components and their respective roles in processing the bills.

1. Bundle Accepter (BA): It receives a bundle of currency notes (maximum of 50 per transaction) from the input slot and sends one bill at a time for processing. After sending the last bill, it notifies the controller. It also receives returned bills from the machine and present them in a bundle to the customer in the event of unrecognized currencies or cancellation of transaction. It is guarded by a shutter that opens only at the beginning of a transaction and when returning notes (in the case of rejection or cancellation of transaction)
2. Bill checker (BC): It receives a bill at a time from the BA and investigates its genuineness. Accepted and rejected notes are passed on to the to the escrow and reject box respectively.
3. Escrow (ES): A temporary stack area for validated bills until the transaction is confirmed or cancelled. In the event of confirmation, the bills are sent to the cassette for permanent storage. If the transaction is cancelled, the bills are sent to the reject box. The ES has only one output slot that may be linked to either the reject or the cassette depending on the situation. ES has a maximum capacity of 50 bills per transaction
4. Reject box (REJ): stacks rejected notes and returns them in bundle. A temporary stack area for rejected and returned bills. Upon receiving the appropriate control instruction,

the stack is transferred to the BA for onward delivery to the customer. REJ has a maximum capacity of 50 bills per transaction

5. Cassette (CAS): A permanent stack area for deposited bills. It has the capacity to accommodate 3000 bills. The bills are manually evacuated by the custodian of the machine
6. Control Board (CON): A micro-electronic board that coordinates the activities by maintaining the flow of communication signals to activate some components when necessary.

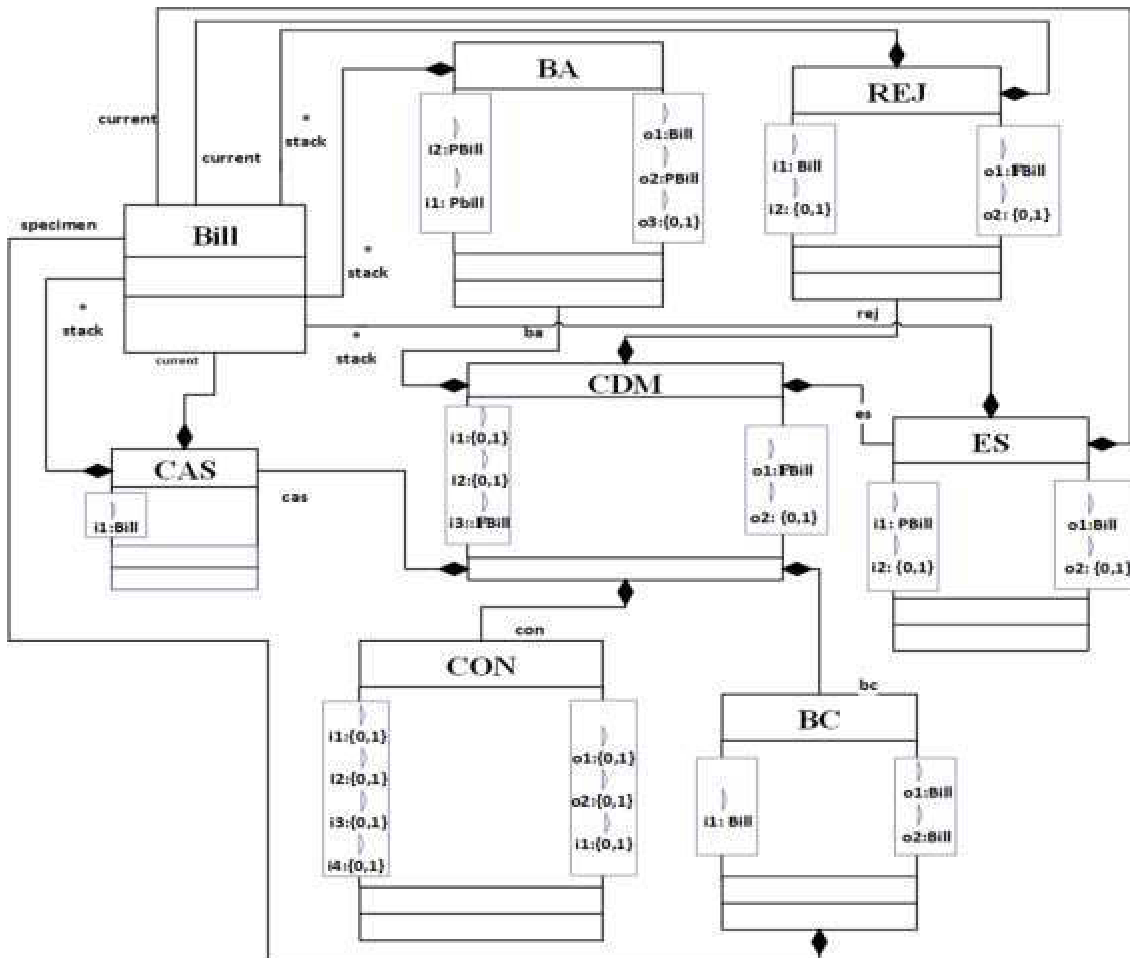

Figure 81. Domain Model of the CDM

V.3.1 HiLLS specification of ATM

We present the HiLLS model of each of the six components followed by the specification of the assembly of those components to make up the CDM. Each component is an autonomous system with its own behaviour; they interact with one another by exchanging bills and low voltage signals.

V.3.1.1 Bill

Recall that the HiLLS' syntax supports the specification of objects and systems through the instantiations of HClass and HSystem respectively (see abstract and concrete sysntaxes in Figure 30 and Figure 32 respectively). Since it has no autonomic behavior, input and/or output ports to influence or be influenced, the bank note (Bill) is modelled as an object (i.e.,

an instance of HClass). The Bill (Figure 83 with the Cassette) class has four parameters *d, t, l* and *w* representing the denomination, thickness, length and width repectively of the bank note. The first line in the second compartment is a list of public operations that can be invcked on an object of the class. The state schema declares the state variables and constraints that define the features of the bill. The Init operation initializes the class' attributes with the values of the parameters. Lastly, the third and last compartment houses the declaration and definition of the class' operations.

V.3.1.2 The Cassette

The Cassette is shown in Figure 82.

It has only one port in its input interface. It has no output interface since it needs not produce any output. It has two parameters; volume and period representing its capacity and time taken to stack a bill respectively. We define three variables in the state schema: (i) *stack*, a list of bills, is an abstraction of the stack of bills in the cassette. The length of stack cannot exceed the volume of the cassette as specified by the constraint, (ii) *current* holds the currency bill received at the input of the cassette, and (iii) *f* is used to store the *duration* of the reigning configuration at any instant.

Three configurations are specified in the last compartment; The *available* and *full* configurations are specified as *passive configurations* with infinite *duration* because external influences are required in both cases to change its status. The third configuration, *acquisition* is a finite configuration with *duration* equal to the time taken to put a received bill in the stack.

We define three operations ($store()$,$resetCurrent()$) in the operation compartment. The operations are called during configuration transitions to effect the reconfiguration of state variables.

It waits in the *available* configuration when the number of bills is less than its capacity. Once the cassette is filled to its capacity, it assumes the *full* configuration until its contents are manually evacuated.

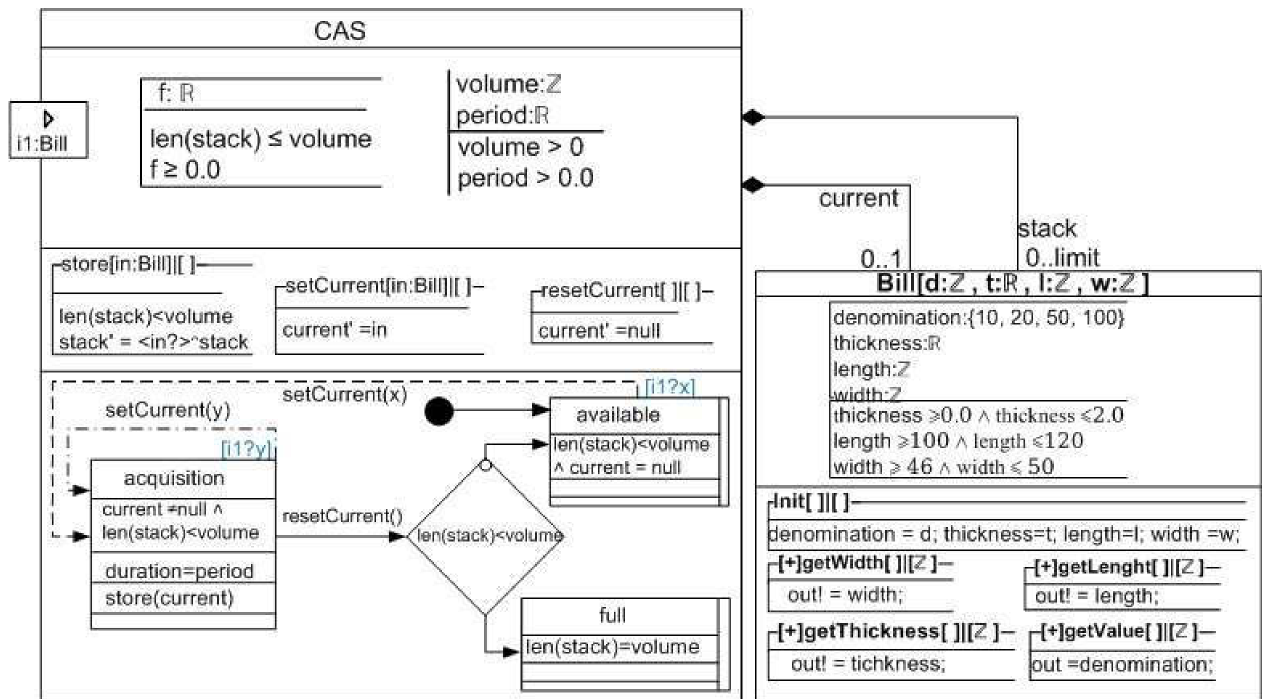

Figure 82. HiLLS specification of the Cassette

V.3.1.3 The Bundle Acceptor

The BundleAcceptor is described by the HSystem in Figure 83.

The BundleAccepter has two input ports ($i_1: PBill$ and $i_2: PBill$) and three output ports ($o_1: Bill$, $o_2: PBill$ and $o_3: \{0,1\}$). It accept a set of Bills through i_1 and send them one by one through o_1 to the BillChecker for processing. After sending the last Bill, it notifys the Controller through o_3. Returned Bills from the machine in case of error or cancellation of the transaction are received through i_2. These returned Bills are presented to the customer through o_2.
It has three state variables ($stack: PBill$, $status: \mathbb{Z}$ and $f: \mathbb{R}^+$) and three parameters ($limit: \mathbb{Z}, pickTime: \mathbb{R}^+$ and $returnTime: \mathbb{R}^+$). If a set of Bills is received its content is stored in the $stack$ variable. $limit$ is the parameter that defines the maximum number of Bills that the BundleAcceptor can store. $pickTime$ and $returnTime$ are timing parameters for picking a Bill from the $stack$ and returning a Bill.

It has three configurations: $idle$, $picking$ and $returning$. The $idle$ configuration correponds to the situation where the $stack$ is empty and the $status$ is equal to zero ($stack = <>$ and $status = 0$). In $picking$ configuration the $stack$ contains at least one Bill and the $status$ is equal to 1 ($stack \neq <>$ and $status = 1$). The predicates $stack = <>$ and $status = 2$ define the $returning$ configuration.

We have three operations:$ReturnBundle()$,$notify()$ and $postBill()$. They specify respectively how a bundle is returned, a notification is sent and a Bill is posted.

Initially the Bundle Accepter is in the $idle$ passive configuration. From $idle$, two external transitions are possible. If Bills are received through i_1 ($i_1?b$) one of the external transition lead to the $picking$ configuration. The other external transition targets the $returning$

configuration when Bills are received through i_2 ($i_2?b$). The received Bills are stored in the $stack$ ($stack = b$). In $returning$ configuration, we have only one internal transition to $idle$ during which it return the bundle ($o_2!returnBundle()$) and notify ($o_3!notify(1)$) the Bontrol Board. From $picking$ we have a conditional internal transition target $idle$ (if the $stack$ is empty) or $picking$ (if the $stack$ is not empty) after posting a Bill ($o_1!postBill()$). The internal transition to $idle$ include sending a notification ($o_3!notify(0)$).

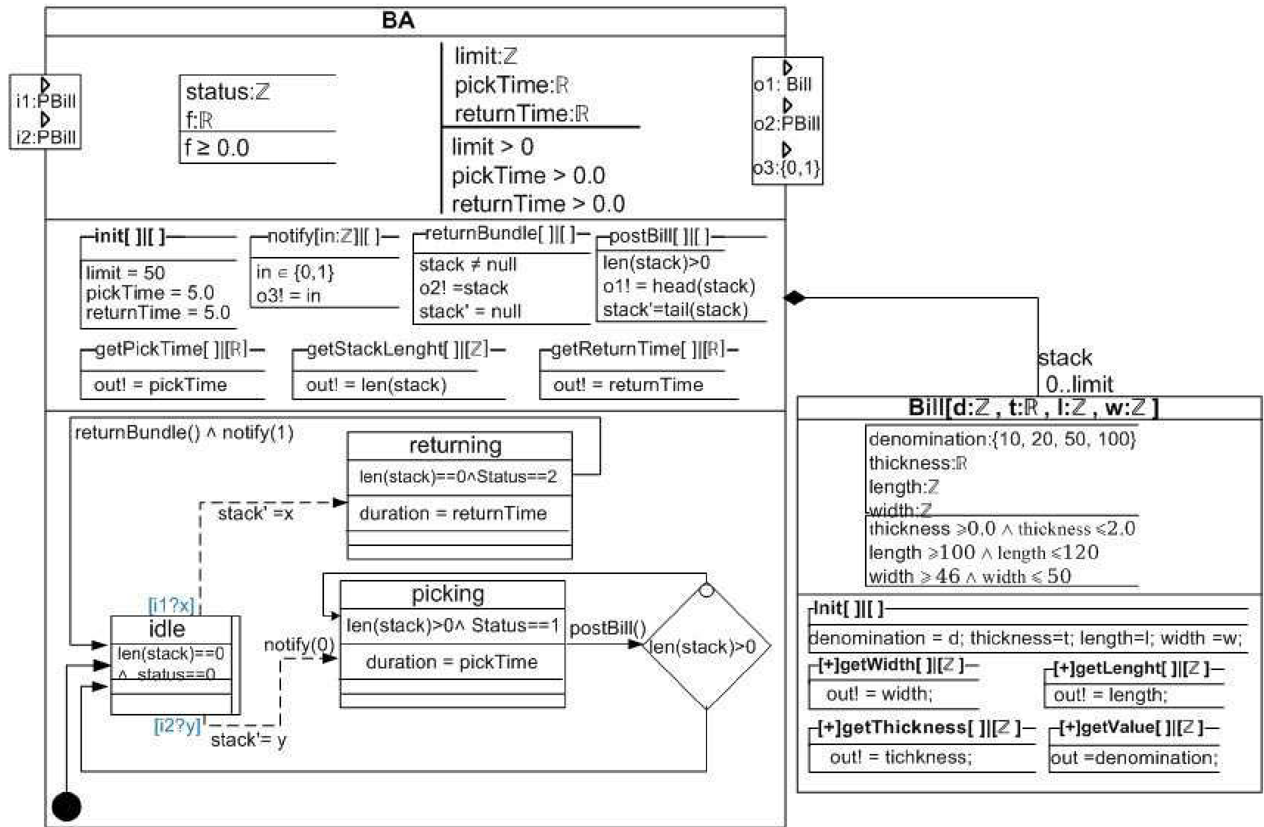

Figure 83. HiLLS specification of the Bundle Acceptor

V.3.1.4 The Bill Checker

The BillChecker is described by the HSystem in Figure 84.

It has one input port ($i_1: Bill$) and two output ports ($o_1: Bill$ and $o_2: Bill$). It receive a Bill for validation through i_1. Validated Bills and non-validated Bills are respectively sent to Escrow (through o_1) and the Reject Box (o_2). The Bill Checker has two state variables: $specimen: Bill$ and $genuine: true|false$. The $specimen$ variable store the actual Bill being checked for validation. $genuine$ is the genuineness of the $specimen$. An additional timing parameter $vTime: \mathbb{R}$ is used as the $duration$ of the validation process.

We defined two configurations for the Bill Checker: $waiting$ and $validating$. In the $waiting$ there no Bill for validation ($specimen = \emptyset$). The $validating$ configuration is defined the condition that there is a Bill for validation ($specimen = \emptyset$).
It has one operation $validate$ () for checking the genuineness of a given Bill.

The behaviour of the Bill Checker is defined by three transitions. In $waiting$, the reception of a Bill $i_1?b$ provoque an external transition to $validating$ for validation of the received Bill which is stored in the $specimen$ variable ($specimen = b$). We have an internal transition from $validating$ to $waiting$ after the validation process. The last transition is a confluent one from $validating$ to $validating$.

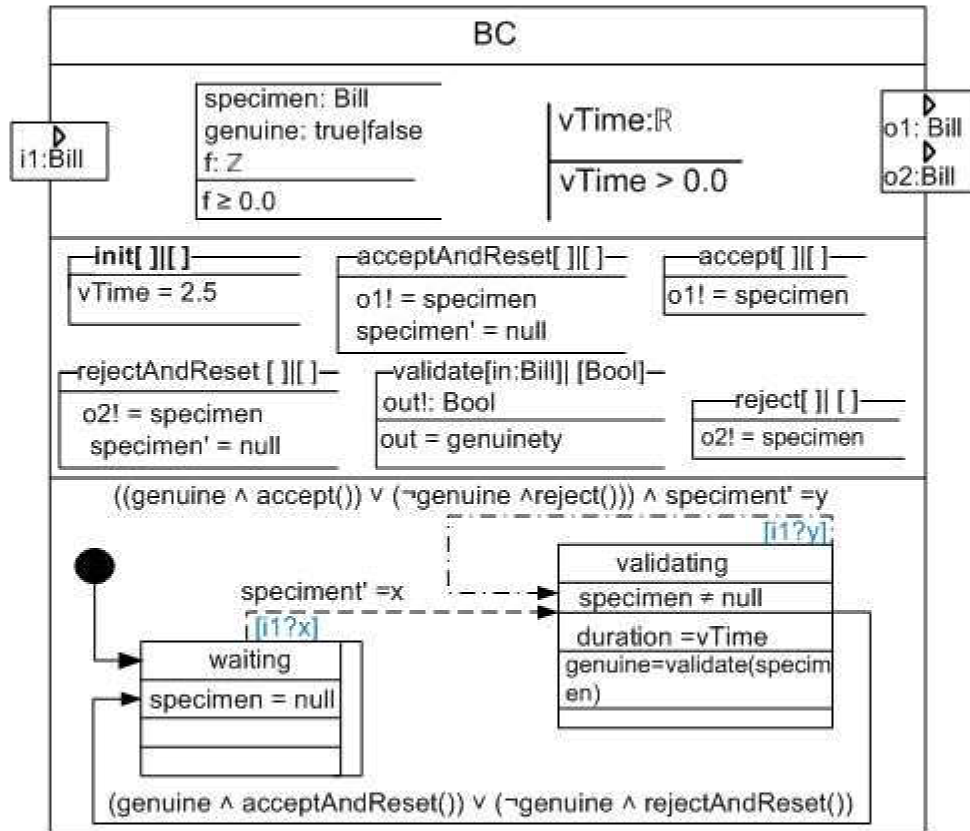

Figure 84. HiLLS specification of the Bill Checker

V.3.1.5 The Escrow

The HSystem in Figure 85 gives the formal description of the Escrow.

The Escrow possesses two input ports ($i_1: PBill$ and $i_2: \{0,1\}$) and two output ports ($o_1: Bill$ and $o_2: \{0,1\}$). It receives through $i_1$the validated Bills during the process or the set of Bills in case of cancelled operation. The stored Bills are sent to the Reject Box (in case of cancellation) or the Cassette (in case of confirmation) through o_1.It receives confirmation information about the transaction through i_2. The reception of 0 is synonym of cancellation and reception of 1 means confirmation.

The Escrow has four state variables ($stack: PBill$, $current: Bill$, $flag: \mathbb{Z}$ and $interrupt: \mathbb{Z}$) and two parameters ($limit: \mathbb{Z}$ and $period: \mathbb{R}$). $stack$ is used to store the received Bills. $current$ represent the current Bill that must be sent. $flag$ and $interrupt$ are control variables. $limit$ is the parameter that represents the maximum number of Bills that can be stored in the $stack$. $period$ is a timing parameter that represents the duration of processing a Bill.

The state space of the Escrow is partitioned to four configurations: $waiting$, $loading$, $offloading$ and $response$. The $waiting$ configuration is a passive configuration defined by the situation where there is no current Bill ($current = \emptyset$) for processing and $interrupt$ is equal to 0 ($interrupt = 0$). In $loading$ configuration there is a current Bill ($current \neq \emptyset$) and $interrupt = 0$. The $offloading$ configuration is characterized by the fact that $interrupt = 1$. The $response$ configuration is transient configuration where $interrupt = 2$.

The Escrow has the following operations: $dispense()$, $load()$, $report()$ and $resetCurrent()$. $dispense()$ return the first element ($head(stack)$) of the $stack$ if there at least one Bill in it ($len(stack) > 0$) and remove it ($stack' = tail(stack)$). The $load()$ operation consist of adding a Bill to the $stack$ ($stack' = stack \cap < in? >$) if the number of Bills in the $stack$ is less than the $limit$ ($len(stack) < limit$). $resetCurrent()$ is used reset the $current$ variable ($cuurent' = \emptyset$).

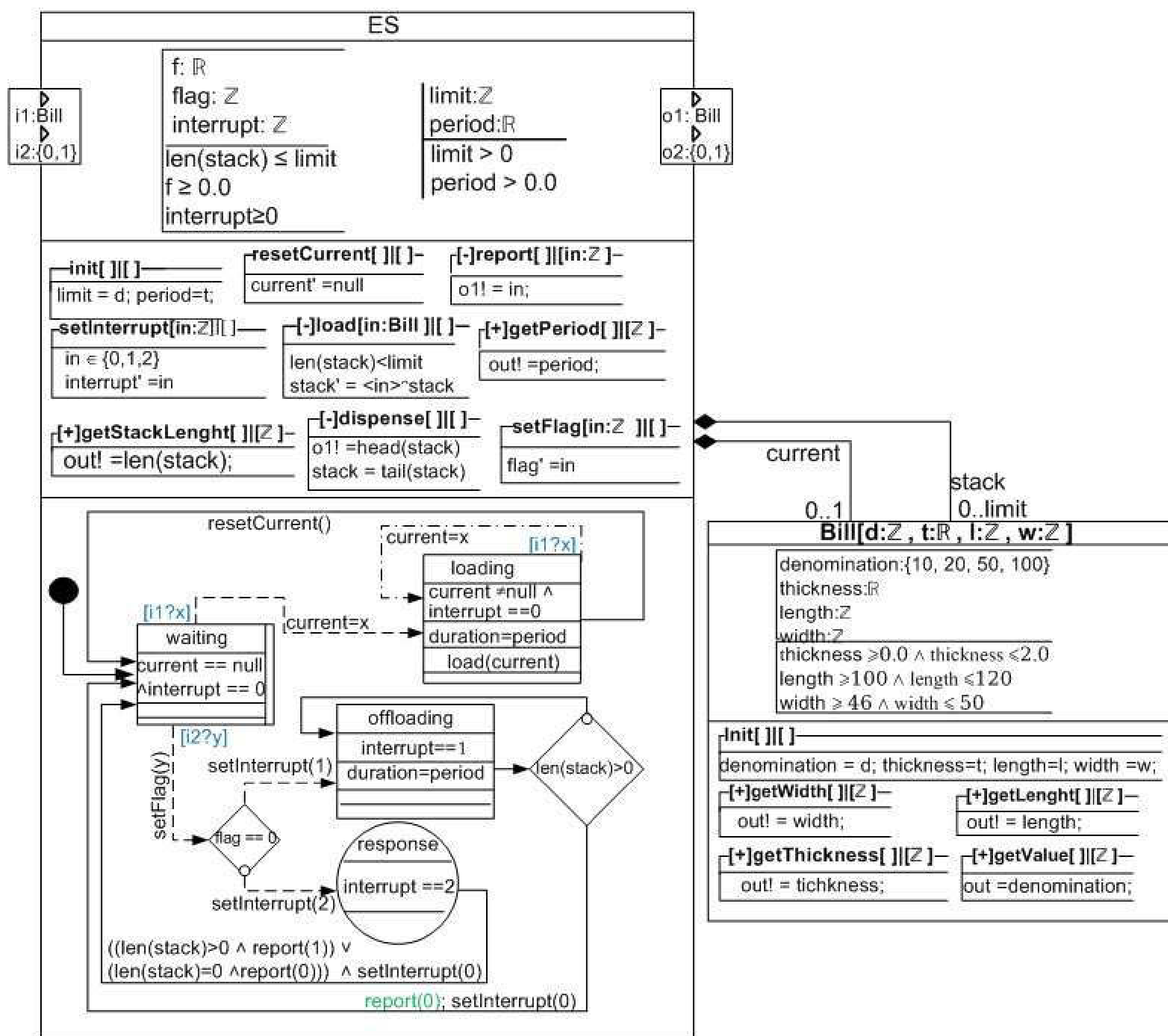

Figure 85. HiLLS specification of the Escrow

Initially the Escrow is $waiting$. The reception of a Bill b on i_1 ($i_1?b$) leads to external transition which target is the $loading$ configuration with an associated computation that set the value of $current$ to be the received Bill ($current' = b$). From $waiting$, the reception of a confirmation leads to a conditional external transition which target is $response$ if $flag =$

0. If $flag \neq 0$ the $offloading$ configuration is assumed. The value of $flag$ is defined by the received confirmation. In $response$ we have an istantaneous internal transition to the $waiting$ configuration with the following associated computation: $o_2!report()$. From $offloading$ we have conditional internal transition that can lead to $waiting$ and sending 0 as output on o_2 ($o_2!0$) if $len(stack) = 0$ or $offloading$ if $len(stack) > 0$. We hase also an internal transition from $loading$ to $waiting$ witch effect is the reset of $current$ ($current' = \emptyset$).

V.3.1.6 The Reject Box

The reject box (Figure 86), being an entity that influences and is influenced through the exchange of bills with other components and which responds to and produses low voltatge signals is modeled as an HSystem. The input interface has two ports *i1* and *i2* for receiving bills and digital signals respectively. Simlarly, the output interface has two ports *o1* and *o2* for sending bundles of blls and digital signals respecively.

The reject box has four state variables three of which are explicit and the remaining one implicit. The explicitly declare state variables as shown in Figure 5 are *interrupt* of type positive integer; *current* of type Bill and stack, a list of Bills. Since every instantiation of HSystem must have at least one specified configuration, it declares an implicit variable *duration* which may be of type positive real number or positive infinity to hold the instantaneous values of the sojourn times of active configurations. Simlar to the Bill class, RejectBox defines a list of public elements comprising all input and output ports, the Init operation and two other operations; for every instantiation of RejectBox, these elements are visible to its environment. The axiomatic schema in the second compartment defines two constants: *limits* and *period* representing the component's capacity and time required to exchange bills respectively as specified in the system's requirement. Note that *limit* is used as the cardinaity of the variable *stack* (see the composition relationship from RejectBox to Bill) indicating the maximum nuber of bills that can be stacked in the reject box at any instant and for any transacton. The *Init* operation defines the initial configuration by setting the state variables to the appropriate values or ranges of values. According to the assigned values, the initial configuration of *Reject Box* is *waiting.*

The third compartment holds the definitions of three operations; *getStackLenght, getPeriod* and *load* meant to read the length of variable *stack*, value of constant *period*, and add *current bill* to the *stack* respectively. The first two operations do not require input parameters, hence the empty brackets following the operation names but each returns an integer value as output as indicate in the *type* bracket. In HiLLS, any operation with type different from void has an intrinsic variable, *out*, of the same type as the operation's type; this variable must be assigned the computed value of the output of the operation. *load* has a type void denoted by the empty type bracket but it declares an input parameter of type Bill.

The fourth and last compartment contains the specification of the behavior of the reject box in the form of configurations and configuration transitions. The state space is partitioned into four configurations *waiting, loading, offloading* and *reporting* with their respective properties indicated in their second compartments. As stated previously, the computed sojourn time for each configuration is assigned to the implicit variable *duration*.

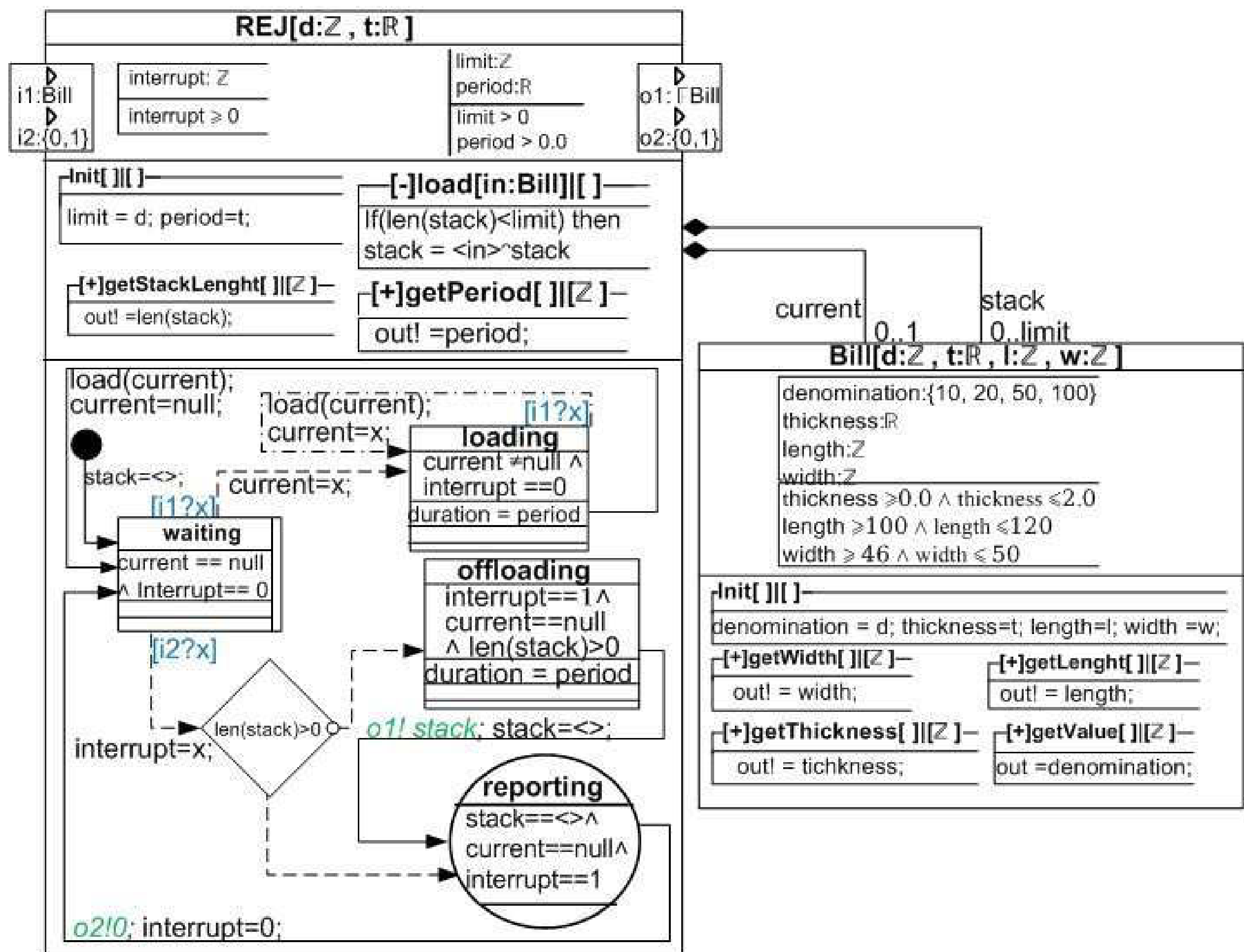

Figure 86. HiLLS' specification of Bill and Reject box

Configurations *loading* and *offloading,* being finite configurations have their sojourn times explicitly defined in the third compartments while *waiting* and *reporting* have implicit sojourn times of positive infinty and zero respectively.

The behaviour of the component is described by the configurations and transitions between them. Like the state space, the set of state transitions is partitioned into seven configuration transitions as depicted by the three different kinds of arrow lines entering the four configurations from their left sides. The sequence of computations accompanying each transition reconfigures the state variables to satisfy the properties of the target configuration. For the sake of clarity, let us discuss the transitions by considering one source configuration at a time.

Being a passive configuration, only external influences at the input ports (i.e., causing *external* transition) can force the system out of the *waiting configuration* once it is assumed. There are three external transitions emanating from *waiting*, each targeting one of the three other configurations. The *waiting2loading* transition is triggered when a bill is received at the input port *i1*. Just before the transition, the bill received at *i1* is assigned to the state variable *current* and the port is reset to null, making the state space configuration satisfy the properties of *loading*. The other two transitions with *waiting* as the source, *waiting2offloading* and *waiting2reporting* are both triggered by the reception of a non-zero value (high voltage

signal) at the port *i2.* In either case, the state variable *interrupt* is assigned the value received at the port before resetting it (the port) to zero. The target configuration *offloading* is assumed if the variable *stack* is not empty; otherwise, configuration *reporting* is assumed as the target.

There are two transitions in this category, i.e., one *internal* and one *confluent* transition. The internal transition *loading2waiting* is triggered when the reign of a*waiting* configuration expires without interruption. The value of variable *current* is reset to null in the computations accompanying the transition. If just upon the expiration of the sojourn time of *waiting*, a bill is received at *i1,* a confluent transition *loading2loading* is triggered during which the bill received at *i1* is assigned to *current* and the configuration *loading* is re-assumed.

An internal transition, *offloading2reporting*, is specified with configuration *offloading* as its source. It is triggered at the expiration of the sojourn time of *offloading.* The sequence of computations accompanying the transition includes an output on the port *o2;* the list of bills stored in the *stack* is assigned to the output port *o2* and *stack* is set to empty.

The last in the list of transitions is the internal transition *reporting2waiting* which is triggered immediately the system assumes the transient configuration *reporting*. It is also accompanied by an output of zero on the output port *o2* followed by the assignment of the value zero to the variable *interrupt,* a computation that leads to the reconfiguration of the state variables to satisfy the properties of *waiting.*

V.3.1.7 The Control board

The HSystem in Figure 87 represents the Control Board. Its role is to coordinate the activities of the different components to realize a transaction.

Ports, variables and parameters
For the coordination purpose it maintains four input (i_1,i_2,i_3 and i_4) and three output ports (o_1, o_2 and o_3) with $\{0,1\}$ as common domain.The input ports are used to receive confirmation information from other components and the output ports are used to send control information to the components. It has four state variables ($sign$: $\{0,1\}$, $stage$: $\mathbb{Z}$,$ecount$: $\mathbb{Z}$ and $confirm$: $\{0,1\}$) and one parameter ($delay$: $\mathbb{Z}$).

We have two configurations: $idle$ and $busy$. $idle$ is a passive configuration that correspond to the beginning of the transaction process ($stage = 0$). $busy$ is a composite configuration that have five other subconfigurations: $waiting$, $active2$, $active3$, $active4$ and $delaying$.

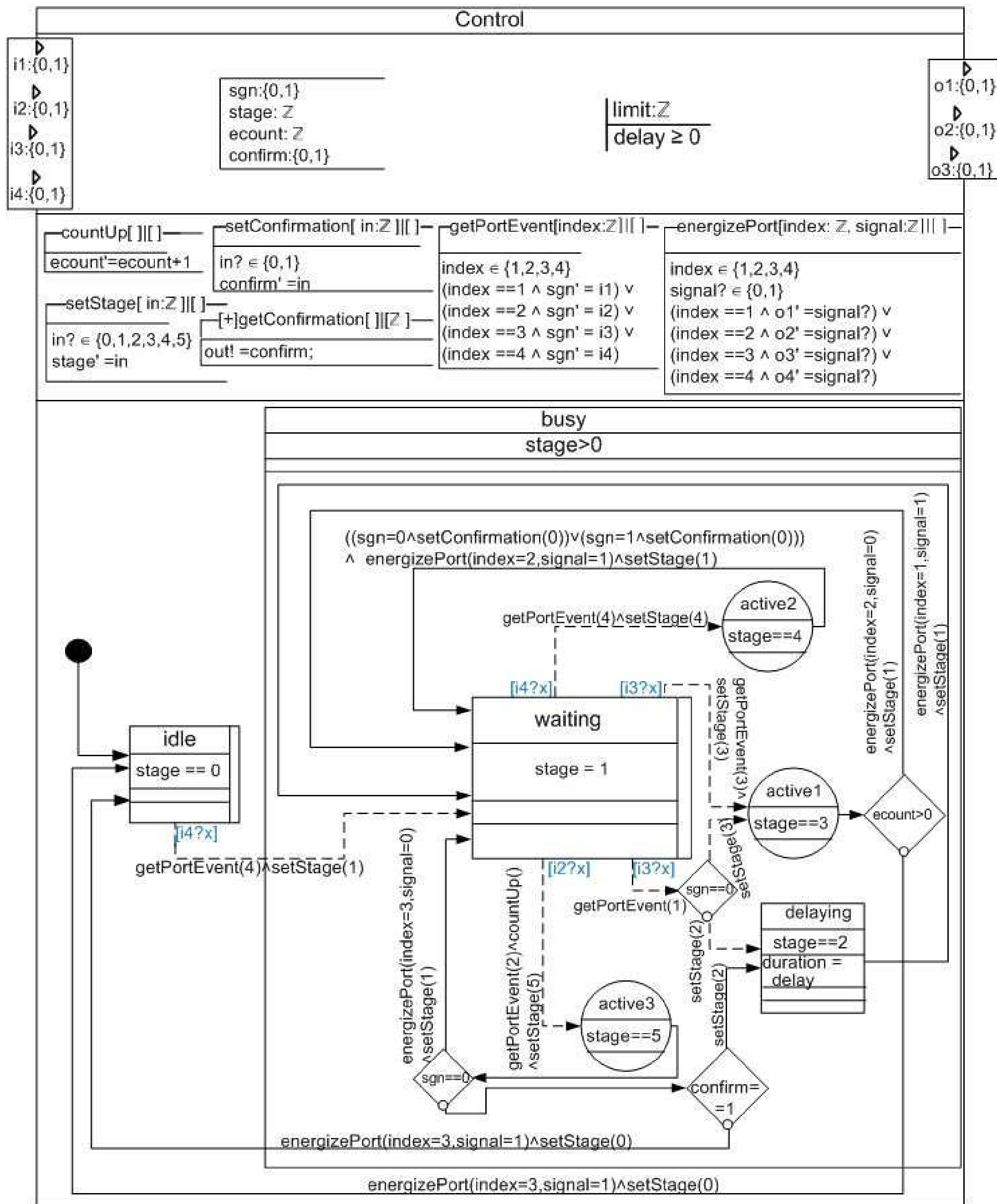

Figure 87. HiLLS specification of the Control board

V.3.1.8 The cash deposit Module

The CDM (Figure 88) is an HSystem comprising six components; *ba, bc, es, rej, cas* and *con* which are instantiations of BundleAcceptor, BillChecker, Escrow, RejectBox, Cassette and ControlBoard respectively. Normally, the full model of every component should appear at the other end of the relationship; however, space would not allow us to show legible models of all components and the relationships on the same page. Therefore, we designate dashed boxes to represent each component in Figure6 to enable us show the relationships between them and the parent system. The full specification of the RejectBox has been presented previously (Figure 5) to give the reader a perception of what the dashed boxes represent. The CDM has three input ports *i1, i2* and *i3.* While each of the first two ports takes inputs of 0 and 1, the last takes stacks of bills as input. It also has two output ports *o1* and *o2*; while the former sends

stacks of bills as outputs, the later sends 0 and 1. In addition to the components, the CDM specification declares two state variables, *transaction* and *shutter* that keep information about the presence of an active transaction and the status of the shutter respectively. The shutter is the opening through which the user presents stacks of bills to the machine or receives rejected and/or returned stacks from the machine. The last line of invariants in the state schema specifies the static couplings between components of the system. The full model of each component specifies input ports $i_1,..,i_n$ and output ports $o_1, ..,o_n$ such that n > 0 and all ports are included in the visible list as shown in RejectBox (Figure 86). Therefore, a coupling specification $A.port_r = B.port_s$ implies that $port_s$ of system B influences $port_r$ of system A. In addition to initializing the state variables to a starting configuration, the *Init* operation of *CDM* also invokes the Init of each of its components to initialize them as well and establishes the static couplings between the ports of components. Note that the CDM does not specify a *visible* list.

The CDM defines three operations as shown in the third compartment of Figure 88. The same explanations for the operations defined in RejectBox can be used to understand them.

The CDM's behavioural specification defines two major configurations, *idle* and *busy* with the state variable *transaction* as the main distinguishing property between the duo. The value of the variable is set to "1" once a user initiates a request to make deposit indicating the *busy* configuration. Once the transaction is completed, it returns to the *idle* state by setting the *transaction* variable to "0". The second distinguishing property is that the *shutter* is always in the closed state when the machine is idle. The busy configuration is further refined into sub-configurations *exchange* and *processing* which are themselves further decomposed as shown in the model. Among the unique features of HiLLS demonstrated in this example is the way structural dynamics is specified graphically. *Coupling*s are used extensively to specify the properties of many of the configurations of the CDM especially the sub-configurations of *busy*. For example, when the *validation* configuration is assumed, the port *o2* of *bc* (i.e., *bc.o2*) is coupled to port *i1* of *rej* (i.e., *rej.i1*) so that the former can influence the later.

In the *rejecting* configuration, *rej.i1* is coupled to another component, *es.o1* to be influenced by it while *es.o1* influences yet a different port, *cas.i1* when the system assumes *completing* configuration.

In addition to the explicit and implicit specifications of sojourn times for configurations demonstrated in the RejectBox specification, the CDM presents two special functions, η and π, to specify the sojourn times in some specific cases. η is used in cases where a configuration has some sub-configurations where its own sojourn time cannot be precisely specified. Hence, the function indicates that the sojourn time at any instant is equal to that of its active sub-configuration at that time. For example, the sojourn time of *processing* at any instant is equal to that of whichever is active among *validation, confirmation* and *escrow_dispense.* π on the other hand is used when all sub-configurations under the same parent have identical sojourn time; the sojourn time may be specified once on the parent configuration while all its children inherit this property from it. For instance, configurations *completing* and *rejecting* have identical sojourn time which specified once on *escrow_dispense* from which all its sub-configurations inherit the property. Configuration transitions of the CDM can be read just the same way those of *RejectBox* were presented previously.

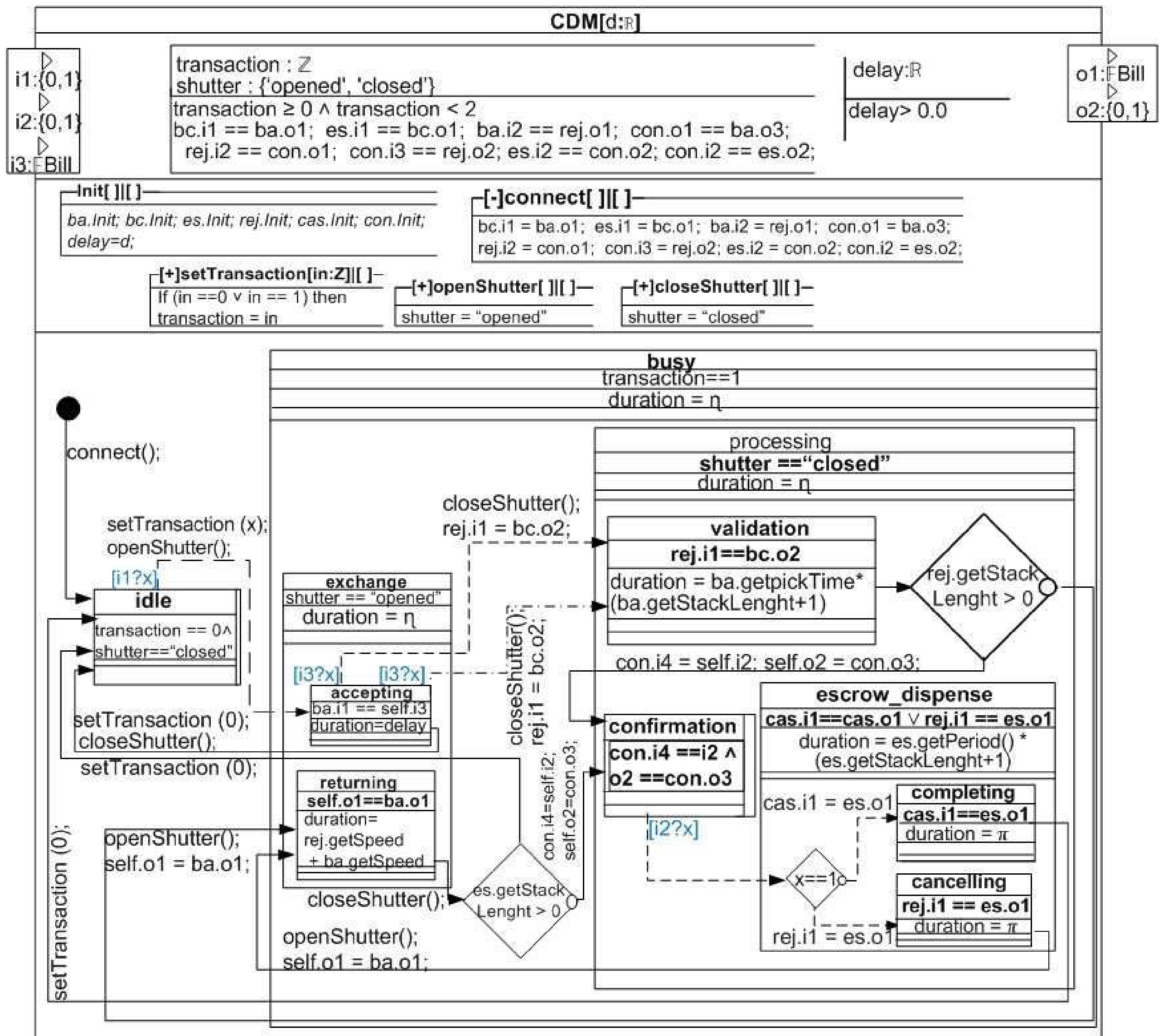

Figure 88. HiLLS specification of the Cash Deposit Machine (CDM)

The block diagram in Figure 89 is to further give the reader a perception of the assembly of the system's components as presented in Figure 81. The eight static couplings, specified as invariants in the state schema of the HiLLS specification are represented as solid arrow lines connecting the source and target ports while the dynamic couplings that are specified in the configurations and configuration transitions are represented as dashed arrow lines in this block diagram.

It is important to note that we only intend to take advantage of the cognitive property of the formalism of this block diagram to give further explanation (at least for the first time) of the CDM model in Figure 88 , it does not serve as a replacement to the HiLLS specification. Though it offers a quick grasp of the physical structure of the assembly specifically the static couplings, it cannot effectively describe the conditions under which the dynamic couplings are established and/or broken. Moreover, representing all components inside the whole is highly cognizable but it is highly deficient in terms of reusability of specification. i.e., the Object-Oriented approach adopted in the HiLLS representation allows one specification of a component to be reused by many assemblies through references.

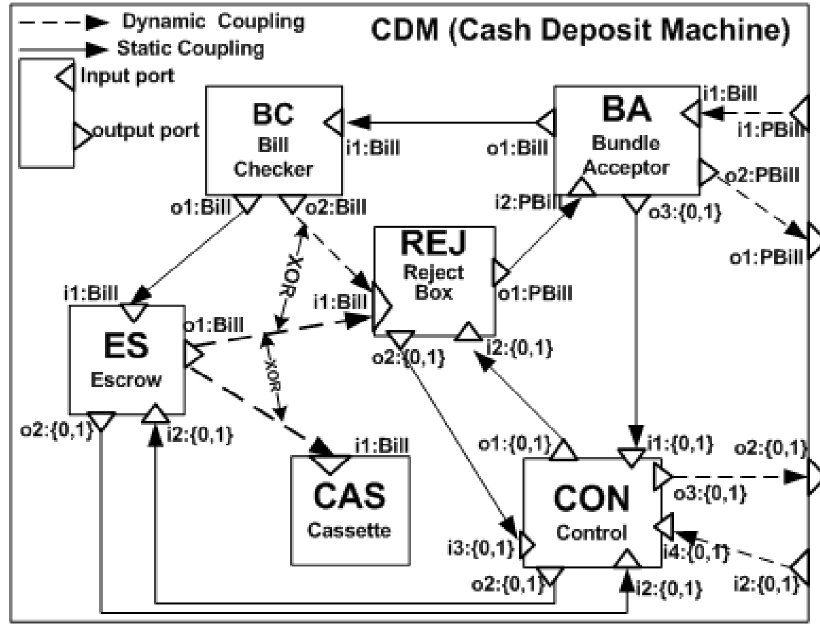

Figure 89. Block diagram illustrating the structure of the CDM

V.3.2 DSDEVS specification of ATM

We present here the equivalent DEVS models of the Reject Box and the CDM.

V.3.2.1 DSDEVS model of the Reject Box

Since the Reject Box is an unitary, the equivalent DSDEVS model of it is also the equivalent DEVS atomic model.

$RejectBox =< X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta >$

$X = \{(i_1, Bill), (i_2, \{0,1\})\}$

$Y = \{(o_1, Bill^b), (o_2, \{0,1\})\}$

$S = \{(stime, \mathbb{R}^+), (interrup, \mathbb{Z}^+), (stack, Bill^b), (current, Bill), (limit, \mathbb{Z}^+), (priod, \mathbb{R}^+),$

$(conf, \{waiting, loading, offloading, reporting\})\}$

External transition function $\delta_{ext}: Q \times X^b \rightarrow S$

$\delta_{ext}(+\infty, 0, <> ,50,1.5, \emptyset, waiting, e, i_1)(period, 0, stack.add(cur),50, 1.5, i_1, loading)$ $\forall e > 0 \wedge i_1 \in Bill$

$\delta_{ext}(+\infty, 0, <>, 50,1.5, \emptyset, waiting, e, i_2)$

$= (period, 1, stack.add(cur),50,1.5, \emptyset, offloading)\ if\ length(stack) = 0$

$= (0,2, <> ,50,1.5, \emptyset, reporting)\ if\ length(stack) = 0$

Internal transition function :$\delta_{int}: S \rightarrow S$

$\delta_{int}(period, 2, <>, 50, 1.5, \emptyset, reporting) = (+\infty, 0, <>, 50, 1.5, \emptyset, waiting)$

$\delta_{int}(period, 0, stack, 50, 1.5, \emptyset, loading) = (+\infty, 0, stack, 50, 1.5, \emptyset, waiting)$

$\delta_{int}(period, 1, stack \neq <>, 50, 1.5, \emptyset, offloading) = (+\infty, 0, stack, 50, 1.5, \emptyset, waiting)$

Confluent transition function δ_{conf}: $S \times X^b \rightarrow S$

$\delta_{conf}(period, 0, stack, 50, 1.5, cur, loading, i_1)$

$= (period, 0, stack.load(cur), 50, 1.5, i_1, loading)$

Output function $\lambda: S \rightarrow Y^b$

$\lambda(stime, inter, stack, limit, period, cur, conf = loading) = \emptyset$

$\lambda(stime, inter, stack, limit, period, cur, conf = offloading) = \{(o_1, stack), (o_2, 0)\}$

$\lambda(stime, inter, stack, limit, period, cur, conf = reporting) = \{(o_2, 0)\}$

Time advance function $ta: S \rightarrow \mathbb{R}^+ \cup \{+\infty\}$

$ta(s) = s.stime \; \forall s \in S$

V.3.2.2 DSDEVS model of the CDM

$DSDN_{CDM} = (\chi, M_\chi =< X_\chi, S_\chi, s_{0,\chi}, Y_\chi, \gamma, \Sigma^*, \delta_\chi, \lambda_\chi, \tau_\chi >)$ where $\chi = CDM$

$X_\chi = \{(i_1, \{O, 1\}), (i_2, \{O, 1\}), (i_3, IPBill)\}$

$Y_\chi = \{(o_1, IPBill), (o_2, \{O, 1\})\}$

$S_\chi = \{(transaction, \mathbb{Z}^+), (shutter, string), (stime, \mathbb{R}^+ \cup \{+\infty\}), (conf, CONF)\}$ where

$CONF = \{idle, ready, cancelling, validating, confirmation, completing, rejecting\}$

$s_{0,\chi} = (transaction = 0, shutter = "closed", stime = +\infty, conf = idle)$

$\Sigma^* = \{M_{conf}\}_{conf \in CONF}$ with $M_{conf} =< D_{conf}, \{M_i\}_{i \in D_{conf}}, \{I_i\}_{i \in D_{conf}}, \{Z_i\}_{i \in D_{conf}} >$

Let $D_0 = \{M.name\}_{M \in Comps}$ where $Comps = \{ba, bc, es, rej, cas, \; con\}$

$\forall M_{conf} \in \Sigma^*, D_{conf} = D_0 \; \wedge \{M_i\}_{i \in D_{conf}} = Comps$

$\forall s_\chi = (transaction, shutter, stime, conf) \in S_\chi, \gamma(s_x) = M_{conf}$

Let $sI_{CDM} = \{ \quad \}$, $sI_{ba} = \{rej\}$, $sI_{bc} = \{ba\}$,$sI_{es} = \{bc, con\}$, $sI_{rej} = \{con\}$,$sI_{cas} = \{ \quad \}$

$sI_{con} = \{ba, rej, es\}$

$M_{idle} =< D_{idle}, Comps_{idle,} I_{idle}, Z_{idle} >$

$I_{idle} = \{sI_{CDM}, sI_{ba}, sI_{bc}, sI_{es}, \; sI_{rej}, sI_{cas} \; sI_{con}\}$

$Z_{idle,ba}: Y_{rej} \rightarrow X_{ba}$

$Z_{idle,bc}: Y_{ba} \rightarrow X_{bc}$

$Z_{idle,es}: Y_{bc} \rightarrow X_{es}$

$Z_{idle,rej}: Y_{con} \rightarrow X_{rej}$

$Z_{idle,con}: Y_{rej} \times Y_{ba} \times Y_{es} \rightarrow X_{con}$

$M_{ready} =< D_{ready}, Comps_{ready,} I_{ready}, Z_{ready} >$

$I_{ready} = \{sI_{CDM}, sI_{ba} \cup \{CDM\}, sI_{bc}, sI_{es}, \ sI_{rej}, sI_{cas}\ sI_{con}\}$

$Z_{ready,ba}: X_{CDM} \times Y_{rej} \rightarrow X_{ba}$

$Z_{ready,bc}: Y_{ba} \rightarrow X_{bc}$

$Z_{ready,es}: Y_{bc} \rightarrow X_{es}$

$Z_{ready,rej}: Y_{con} \rightarrow X_{rej}$

$Z_{ready,con}: Y_{rej} \times Y_{ba} \times Y_{es} \rightarrow X_{con}$

$M_{cancelling} =< D_{cancelling}, Comps_{cancelling,} I_{cancelling}, Z_{cancelling} >$

$I_{cancelling} = \{sI_{CDM} \cup \{ba\}, sI_{ba} \cup \{CDM\}, sI_{bc}, sI_{es}, \ sI_{rej}, sI_{cas}\ sI_{con}\}$

$Z_{cancelling,CDM}: Y_{ba} \rightarrow Y_{CDM}$

$Z_{cancelling,ba}: X_{CDM} \times Y_{rej} \rightarrow X_{ba}$

$Z_{cancelling,bc}: Y_{ba} \rightarrow X_{bc}$

$Z_{cancelling,es}: Y_{bc} \rightarrow X_{es}$

$Z_{cancelling,rej}: Y_{con} \rightarrow X_{rej}$

$Z_{cancelling,con}: Y_{rej} \times Y_{ba} \times Y_{es} \rightarrow X_{con}$

$M_{validating} =< D_{validating}, Comps_{validating,} I_{validating}, Z_{validating} >$

$I_{validating} = \{sI_{CDM} \cup \{ba\}, sI_{ba} \cup \{CDM\}, sI_{bc}, sI_{es}, \ sI_{rej} \cup \{bc\}, sI_{cas}\ sI_{con}\}$

$Z_{validating,CDM}: Y_{ba} \rightarrow Y_{CDM}$

$Z_{validating,ba}: X_{CDM} \times Y_{rej} \rightarrow X_{ba}$

$Z_{validating,bc}: Y_{ba} \rightarrow X_{bc}$

$Z_{validating,es}: Y_{bc} \rightarrow X_{es}$

$Z_{validating,rej}: Y_{bc} \times Y_{con} \rightarrow X_{rej}$

$Z_{validating,con}: Y_{rej} \times Y_{ba} \times Y_{es} \rightarrow X_{con}$

$M_{awaiting} =< D_{awaiting}, Comps_{awaiting,} I_{awaiting}, Z_{awaiting} >$

$I_{awaiting} = \{sI_{CDM} \cup \{ba\}, sI_{ba} \cup \{CDM\}, sI_{bc}, sI_{es},\ sI_{rej} \cup \{bc\}, sI_{cas}\ sI_{con} \cup \{CDM\}\}$

$Z_{awaiting,CDM}: Y_{ba} \rightarrow Y_{CDM}$

$Z_{awaiting,ba}: X_{CDM} \times Y_{rej} \rightarrow X_{ba}$

$Z_{awaiting,bc}: Y_{ba} \rightarrow X_{bc}$

$Z_{awaiting,es}: Y_{bc} \rightarrow X_{es}$

$Z_{awaiting,rej}: Y_{bc} \times Y_{con} \rightarrow X_{rej}$

$Z_{awaiting,con}: X_{CDM} \times Y_{rej} \times Y_{ba} \times Y_{es} \rightarrow X_{con}$

$M_{completing} =< D_{completing}, Comps_{completing,} I_{completing}, Z_{completing} >$

$I_{completing} = \{sI_{CDM} \cup \{ba\}, sI_{ba} \cup \{CDM\}, sI_{bc}, sI_{es},\ sI_{rej} \cup \{bc\}, sI_{cas} \cup \{es\}, sI_{con} \cup \{CDM\}\}$

$Z_{completing,CDM}: Y_{ba} \rightarrow Y_{CDM}$

$Z_{completing,ba}: X_{CDM} \times Y_{rej} \rightarrow X_{ba}$

$Z_{completing,bc}: Y_{ba} \rightarrow X_{bc}$

$Z_{completing,es}: Y_{bc} \rightarrow X_{es}$

$Z_{completing,rej}: Y_{bc} \times Y_{con} \rightarrow X_{rej}$

$Z_{completing,cas}: Y_{es} \rightarrow X_{cas}$

$Z_{completing,con}: X_{CDM} \times Y_{rej} \times Y_{ba} \times Y_{es} \rightarrow X_{con}$

$M_{rejecting} =< D_{rejecting}, Comps_{rejecting,} I_{rejecting}, Z_{rejecting} >$

$I_{completing} = \{sI_{CDM} \cup \{ba\}, sI_{ba} \cup \{CDM\}, sI_{bc}, sI_{es},\ sI_{rej} \cup \{bc, es\}, sI_{cas} \cup \{es\}, sI_{con} \cup \{CDM\}\}$

$Z_{completing,CDM}: Y_{ba} \rightarrow Y_{CDM}$

$Z_{completing,ba}: X_{CDM} \times Y_{rej} \rightarrow X_{ba}$

$Z_{completing,bc}: Y_{ba} \rightarrow X_{bc}$

$Z_{completing,es}: Y_{bc} \rightarrow X_{es}$

$Z_{completing,rej}: Y_{es} \times Y_{bc} \times Y_{con} \rightarrow X_{rej}$

$Z_{completing,cas}: Y_{es} \rightarrow X_{cas}$

$Z_{completing,con}: X_{CDM} \times Y_{rej} \times Y_{ba} \times Y_{es} \rightarrow X_{con}$

$\delta_\chi: Q_\chi \times (X_\chi \cup \{\emptyset\}) \rightarrow S_\chi$

$\delta_\chi\big((0, closed, +\infty, idle), e, i\big) = (1, opened, con.getDelay(), ready), \quad \forall e > 0$. This transition corresponds to the external transition between the $idle$ configuration and the $ready$ configuration when an event is received on the port i_1. This transition creates also a new input coupling between the input port i_3 of the cash deposit machine CDM and the input port i_1 of its bundle acceptor ba. The coupling creation is part of changes specified in the set of different possible structures.

$\delta_\chi\big((1, opened, delay, ready), e, i\big) = (1, closed, t, validation) \forall\, i \in \{0,1\} \wedge \forall e \geq 0$ where $t = ba.getPictTime() * (len(ba.getStackLength() + 1)$, this transition takes into account the two transitions (the external one and the confluent one) from the ready configuration to validation configuration. This transition specifies a new internal coupling between the output port o_2 of the bill checker bc and the input port i_1 of the reject box rej.

$\delta_\chi\big((1, closed, +\infty, confirmation), e, i\big) = (1, closed, t, completing), \quad if\ i = 1$ where $t = es.getPeriod() * (es.getStackLength() + 1)$ Is the external transition between $confirmation$ and $completing$ configurations when the value of the input received from port i_2 is equal to 1. It establish an internal coupling between the cassette cas and the escrow $es$by connecting the input port $cas.i_1$ and the output port $es.o_2$.

$\delta_\chi\big((1, closed, +\infty, confirmation), e, i\big) = (1, closed, t, rejecting), \quad if\ i = 0$ where $t = es.getPeriod() * (es.getStackLength() + 1)$ Is the external transition between $confirmation$ and $rejecting$ configurations when the value of the input received from port i_2 is equal to 0. In this case a connection is established between the reject box and the escrow. This connection is done by a new internal coupling between linking the output porto_2 of es and the input port i_1 of rej.

$\delta_\chi\big((1, closed, t, completing), 0, \emptyset\big) = (0, closed, +\infty, idle)$, where $t = es.getPeriod() * (es.getStackLength() + 1)$. This represents the internal transition from the $completing$ configuration to the $idle$ configuration.

$\delta_\chi\big((1, closed, t, rejecting), 0, \emptyset\big) = (0, closed, +\infty,\ idle)$, where $t = es.getPeriod() * (es.getStackLength() + 1)$.

$\delta_\chi\big((1, closed, t, rejecting), 0, \emptyset\big) = (1, opened, t',\ cancelling) \quad t = es.getPeriod(\ \ ) * (es.getStackLength() + 1)$ and $t' = ba.getPictTime() * (len(ba.getStackLength() + 1)$ This transition represents the rejecting and the canceling phase of an operation. It creates a new output coupling which links the output port o_1 of the bundle acceptor ba and the output port o_1 of the cash deposit machine itself.

$\delta_\chi\big((1, opened, con.getDelay(), ready), 0, \emptyset\big) = (0, closed, +\infty,\ idle)$.

$\delta_\chi\big((1, opened, con.getDelay(), cancelling), 0, \emptyset\big) = (0, closed, +\infty,\ idle)$

$\tau_\chi: S_\chi \rightarrow \mathbb{R}^+ \cup \{+\infty\}$

$\forall s_\chi \in S_\chi, \tau_\chi(s_\chi) = stime$

$\lambda_\chi: S_\chi \rightarrow Y_\chi$

$\forall tr \in \{0, 1\}, shut \in String, st \in \mathbb{R}^+ \cup \{+\infty\}$, we have

$\lambda_\chi(tr, shut, st, cancelling) = \{0\}$

$\lambda_\chi(tr, shut, st, completing) = \{1\}$

$\lambda_\chi(tr, shut, f, rejecting) = Bill$

V.4 Conclusion

We have presented in this chapter the HiLLS modeling of the Alternating Bit Protocol (ABP) and the Cash Deposit Machine (CDM) of an ATM. The ABP is a static structure system which equivalent models in DEVS and Z have been presented. Some basic verification of its properties with Z/EVES has been shown. The CDM is an illustration of the modeling of dynamic structure systems with HiLLS. We have presented its equivalent DSDEVS model that captures its essential dynamic structure properties for simulation. These case studies intuitively show the expressiveness and usability of HiLLS. They can serve as a basis for thorough evaluation of HiLLS by potential users. They show the expressive power of HiLLS to model and analyze complex dynamic systems with discrete event perspective.

VI. General conclusion

This thesis proposed an integrated approach for systems modeling and analysis that uses metamodel integration techniques to define a unified formalism with multiple semantic domains for different system analysis methodologies like simulation, formal methods and enactment. Precisely we integrated DEVS, Object-Z and MOF to consistently define the abstract syntax of HiLLS. We adopted similar representation to UML diagrams to define the concrete of the language in order to facilitate the specification of systems. Illustrative examples have provided to show the expressiveness and usability of HiLLS.

We provided a set-theoretic semantics for HiLLS to make the language amenable to rigorous symbolic reasoning. This semantics based on standard mathematics provides a basis for common understanding among all the semantic domains by providing clarifications to the definition of the concepts presented in the abstract syntax to ensure their consistent interpretations. It also clarifies the rules and constraints of the syntax

We selected DEVS, DSDEVS, UML, Z and CSP as semantic domains for HiLLS to benefit from the capability of these domains for the analysis of different aspects of a system. We have presented the different semantic mappings to these semantics domains:

- Operational semantics for simulation: the mappings of HiLLS to DEVS and DSDEVS represent the operational semantics for simulation of HiLLS models. The consequence of these semantic mappings is that one can use existing DEVS simulator implementations to simulate HiLLS static structure systems and use DSDEVS simulators to simulate HiLLS dynamic structure systems.
- Operational semantics for enactment: is defined by an Object-Oriented Framework using UML design patterns. We used the Object-Oriented Observer design pattern to express HiLLS system constructs by mapping the system's structural and behavioral properties to the structure and semantics respectively of the observer pattern. The subject-observer relations are used to establish couplings between ports of the components of a system while the notification mechanisms are used to trigger state transitions. We provided a Java implementation of the framework and a case study to illustrate its use to specify and enact discrete events systems.
- Logical semantics: is defined by establishing semantics mappings between HiLLS unitary level and Z and HiLLS composite level and CSP to capture respectively state-based properties and process-based properties of a *HSystem*.

We have illustrated these semantic mappings by examples.

HiLLS is suitable for modeling a broad range of systems including those with variable structure [Maïga et al. 2015] for simulation, formal analysis and prototyping. However, the current specification of the language does not provide support for variable interface structure and behavior. The dynamic structure modeling capability of HiLLS is illustrated by the study of the Cash Deposit Machine of an ATM. The modeling of static structure systems is also illustrated by the study of the alternating Bit Protocol.

HiLLS provide the following advantages:

- HiLLS models are communicable because of its intuitive graphical concrete similar to UML diagrams.
- HiLLS is highly expressive because of the expresivness of its base languages: DEVS and Object-Z

- HiLLS models can be analyzed by simulation using DEVS and DSDEVS existing simulators.
- HiLLS models are amenable to formal analysis using Z and CSP associated tools.
- HiLLS allows modular and hierarchical construction of models because it adopts modular and hierarchical structuring concepts from the DEVS formalism and object oriented structuring concepts from Object-Z and MOF (Meta Object Facility).
- Simulation and enactment codes can be generated from HiLLS models. See [Aliyu et al. 2015b] for more information about our enactment template of discrete event systems.
- HiLLS specifications can be used as front-ends from witch other views are generated to maintain consistency between the different views and reduce the task of updating them.

The definition of HiLLS is in the starting phase of a larger project to integrate proven scientific techniques and methodologies employed in the exhaustive investigation of the dependability attributes of systems. Further works are in pipeline towards the integration of the supporting tools in a common framework namely SimStudio. Future works will also consider the link between HiLLS and CTL for the formalization of system traces.

HiLLS is in use as modeling language in some projects in the following domains:

- Business Process Management (PhD thesis of Shaowei Wang)
- Military operations (PhD thesis of Hawa Bado)
- Healthcare systems (PhD thesis of Ignace Djitog)
- Urban Traffic systems (PhD thesis of Youssouf Koné)
- Network Security (PhD thesis of Moussa Koïta)
- Collaborative framework for simulation (PhD thesis of Hamzat Aliyu)

References

[Abdallah et al. 2004] Abdallah, Ali E., Cliff B. Jones, and Jeff W. Sanders, eds. Communicating Sequential Processes. The First 25 Years: Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004. Revised Invited Papers. Vol. 3525. Springer Science & Business Media, 2005.

[Abrial 1996] Abrial, J. R. "The B-book: assigning programs to meanings Cambridge University Press." (1996).

[Alencar and Goguen 1991] Alencar, Antonio J., and Joseph A. Goguen. "OOZE: An object oriented Z environment." ECOOP'91 European Conference on Object-Oriented Programming. Springer Berlin Heidelberg, 1991.

[Aliyu et al. 2015 a] Hamzat, Aliyu O., Maïga O., Abdoul-Wahab Hajara I. and Traoré Mamadou K. Introducing HiLLS: High Level Language for System Specification. In Proceedings of AUSTECH'2015, Abuja, Nigeria, October 2015.

[Aliyu et al. 2015 b] Hamzat, Aliyu O., Maïga O., and Traoré Mamadou K. A Framework for Discrete Event Systems enactment. In proceedings of European Simulation and Modeling Conference (ESM'2015), Leicester, United Kingdom, October 2015

[Alur 1999] Alur, Rajeev. "Timed automata." Computer Aided Verification. Springer Berlin Heidelberg, 1999.

[Aït-Ameur et al. 2004] Aït-Ameur, Yamine, Rémi Delmas, and Viginie Wiels. "A framework for heterogeneous formal modeling and compositional verification of avionics systems." Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on. IEEE, 2004.

[Attiogbé 2008] Attiogbé, J. Christian. "Mastering specification heterogeneity with multifacet analysis." Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on. IEEE, 2008.

[Baâti et al. 2007] Baati, Lassaad, Claudia Frydman, and Norbert Giambiasi. "LSIS_DME M&S environment extended by dynamic hierarchical structure DEVS modeling approach." Proceedings of the 2007 spring simulation multiconference-Volume 2. Society for Computer Simulation International, 2007.

[Baeten 2004] Baeten, Jos CM. "A brief history of process algebra." Theoretical Computer Science 335.2 (2005): 131-146.

[Balci 1997] Balci, Osman. "Verification validation and accreditation of simulation models." Proceedings of the 29th conference on Winter simulation. IEEE Computer Society, 1997.

[Baldassari and Bruno 1991] Baldassari, Marco, and Giorgio Bruno. "PROTOB: An object oriented methodology for developing discrete event dynamic systems." High-Level Petri Nets. Springer Berlin Heidelberg, 1991. 624-648.

[Baldassari et al. 1989] Baldassari, Marco, et al. "PROTOB a hierarchical object-oriented CASE tool for distributed systems." ESEC'89. Springer Berlin Heidelberg, 1989. 424-445.

[Barret et.al. 2011] Bryant, Barrett R., et al. "Challenges and directions in formalizing the semantics of modeling languages." Computer Science and Information Systems 8.2 (2011): 225-253.

[Barros 1995] Barros, Fernando J. "Dynamic structure discrete event system specification: a new formalism for dynamic structure modeling and simulation." Proceedings of the 27th conference on Winter simulation. IEEE Computer Society, 1995.

[Barros et al.1998] Barros, Fernando J., Bernard P. Zeigler, and Paul A. Fishwick. "Multimodels and dynamic structure models: an integration of DSDE/DEVS and OOPM." Proceedings of the 30th conference on Winter simulation. IEEE Computer Society Press, 1998.

[Bergstra and Klop 1987] Bergstra, Jan A., and Jan Willem Klop. "ACPτ a universal axiom system for process specification." Algebraic Methods: Theory, Tools and Applications. Springer Berlin Heidelberg, 1989. 445-463.

[Bhasker 2002] Bhasker, J., and A. SystemC Primer. "Star Galaxy Publishing." Allentown, PA (2002).

[Bochmann and Gecsei 1977] Bochmann, Gregor V., and Jan Gecsei. "A unified method for the specification and verification of protocols." In Proceedings of IFIP Cong., Toronto, Canada, August. 1977, pp. 229-234.

[Bolduc and Vangheluwe 2002] Bolduc, Jean-Sébastien, and Hans Vangheluwe. "A modeling and simulation package for classic hierarchical DEVS." MSDL, School of Computer McGill University, Tech. Rep (2002).

[Bolognesi and Brinksma 1989] Bolognesi, Tommaso, and Ed Brinksma. "Introduction to the ISO specification language LOTOS." Computer Networks and ISDN systems 14.1 (1987): 25-59.

[Broadfoot and Roscoe 2000] Broadfoot, Philippa, and Bill Roscoe. "Tutorial on FDR and its Applications." SPIN Model Checking and Software Verification. Springer Berlin Heidelberg, 2000. 322-322.

[Büchi 1960] Büchi, Julius Richard. On a Decision Method in Restricted Second Order Arithmetic. In Proceedings of International Congress on Logic, Method, and Philosophy of Science, Stanford, Stanford University Press, pp. 1-12,1960.

[Budinsky et al. 2003] Budinsky, Frank. Eclipse modeling framework: a developer's guide. Addison-Wesley Professional, 2004.

[Büssow et al. 1998] Büssow, Robert, Robert Geisler, and Marcus Klar. "Specifying safety-critical embedded systems with statecharts and Z: A case study." Fundamental Approaches to Software Engineering. Springer Berlin Heidelberg, 1998. 71-87.

[Butler 1999] Butler, Michael. "csp2B: A practical approach to combining CSP and B." FM’99 Formal Methods. Springer Berlin Heidelberg, 1999. 490-508.

[Camilleri 1990] Camilleri, Albert John. "Reasoning in CSP via the HOL Theorem Prover." Information Technology, 1990.'Next Decade in Information Technology', Proceedings of the 5th Jerusalem Conference on (Cat. No. 90TH0326-9). IEEE, 1990.

[Chao 1971] Chao, Yen W. "Simulating with SIMSCRIPT." Proceedings of the 5th conference on Winter simulation. ACM, 1971.

[Chen 1976] Chen, Peter Pin-Shan. "The entity-relationship model: toward a unified view of data." ACM Transactions on Database Systems (TODS) 1.1 (1976): 9-36.

[Chen et al. 1999] Chen, Liming, et al. "A Conceptual Framework for Modeling and Simulation of Variable Structure Hybrid Systems." (1999): 35-41.

[Chi 1997] Chi, Sung-Do. "Model-based reasoning methodology using the symbolic DEVS simulation." TRANSACTIONS of the Society for Computer Simulation International 14.3 (1997): 141-151.

[Cho and Kim 2001] Cho, Seong Myun, and Tag Gon Kim. "Real time simulation framework for RT-DEVS models." Transactions of the Society for Modeling and Simulation International 18.4 (2001): 203-215.

[Chow and Zeigler 1994] Chow, Alex Chung Hen, and Bernard P. Zeigler. "Parallel DEVS: a parallel, hierarchical, modular, modeling formalism." Proceedings of the 26th conference on Winter simulation. Society for Computer Simulation International, 1994.

[Ciacciac et al. 1995] Ciaccia, Paolo, Paolo Ciancarini, and Wilma Penzo. "A formal approach to software design: The Clepsydra methodology." ZUM'95: The Z Formal Specification Notation. Springer Berlin Heidelberg, 1995. 5-24.

[Circirelli et al. 2010] Cicirelli, Franco, et al. "Temporal verification of RT-DEVS models with implementation aspects." Proceedings of the 2010 Spring Simulation Multiconference. Society for Computer Simulation International, 2010.

[CLEARSY 2009] CLEARSY. "B4Free: Academic tool enabling the operational use of formal Method B for proven software development." http://www.b4free.com/indexen.
php. (Accessed May 21, 2015).

[Clarke, and Wing 1996] Clarke, Edmund M., and Jeannette M. Wing. "Formal methods: State of the art and future directions." ACM Computing Surveys (CSUR) 28.4 (1996): 626-643.

[Cristia 2007] Cristiá, Maximiliano. "A TLA+ encoding of DEVS models." International M&S Multiconference, Buenos Aires, Argentina, 2007.

[Dahl and Nygaard 2003] Dahl, Ole, and Nygaard Kristen. Simula. Encyclopedia of Computer Science 4th. Pages 1576-1578 John Wiley and Sons Ltd. Chichester, UK,2003.

[Dacharry and Giambiasi 2005] Dacharry, Hernán P., and Norbert Giambiasi. "Formal Verification with Timed Automata and DEVS Models: a case study." Proc. of Argentine Symposium on Software Engineering. 2005.

[Dacharry and Giambiasi 2007] Dacharry, Hernán P., and Norbert Giambiasi. "A formal verification approach for DEVS." Proceedings of the 2007 Summer Computer Simulation Conference. Society for Computer Simulation International, 2007.

[De Lara and Vangheluwe 2002] De Lara, Juan, and Hans Vangheluwe. AToM3: A Tool for multi-formalism and meta-modelling. In European Joint Conference on Theory And Practice of Software (ETAPS), Grenoble, France, Fundamental Approaches to Software Engineering (FASE), Lecture Notes In Computer Science 2306, Springer Verlag, April 2002, pages 174-188.

[Dines et al.1978] Bjøorner, Dines, and Cliff B. Jones, eds. The Vienna Development Method: The Meta-language. Springer, 1978.

[Diskin 2003] Diskin, Zinovy. Mathematics of UML: Making the Odysseys of UML less dramatic. In Haim Kilov and Kenneth Baklawski (Eds)Practical Foundations of Business Systems Specifications. Khiwer Academic Publishers, 2003. 348pp. ISBN 1-4020-1480-5

[EBNF 1996] Standard, EBNF Syntaxt Specification. "Ebnf: Iso/iec 14977: 1996 (e)." URL http://www. cl. cam. ac. uk/mgk25/iso-14977. pdf 70 (1996).

[Emerson 1991] Emerson, E. Allen. "Temporal and modal logic." Handbook of Theoretical Computer Science, vol. B (Jan van Leeuwen (Ed.). MIT Press, Cambridge, MA, USA, 1991.

[Emerson and Sztipanovits 2006] Emerson, Matthew, and Janos Sztipanovits. "Techniques for metamodel composition." OOPSLA–6th Workshop on Domain Specific Modeling. 2006. pp. 123-139.

[Erdweg et al. 2012] Erdweg, Sebastian, Paolo G. Giarrusso, and Tillmann Rendel. "Language composition untangled." Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications. ACM, 2012.

[FDR 2010] FDR2 user manual. http://www.formal.demon.co.uk/fdr2manual/index.htm. Accessed May 2010.

[Feng 2004] Feng, Huining. Dcharts, a formalism for modeling and simulation based design of reactive software systems. Diss. McGill University, Montréal, Canada, 2004.

[Feng et al.2007] Feng, Thomas Huining, Miriam Zia, and Hans Vangheluwe. "Multi-formalism modelling and model transformation for the design of reactive systems." Proceedings of the 2007 Summer Computer Simulation Conference. Society for Computer Simulation International, 2007.

[Fischer 2000] Fischer, Clemens. Combination and Implementation of Processes and Data: from CSP-OZ to Java. PhD thesis, University of Oldenburg, 2000.

[Filippi and Bisgambiglia 2004] Filippi, Jean-Baptiste, and Paul Bisgambiglia. "JDEVS: an implementation of a DEVS based formal framework for environmental modelling." Environmental Modelling & Software 19.3 (2004): 261-274.

[Fishwick 1995] Fishwick, Paul A. Simulation model design and execution: building digital worlds. Prentice Hall PTR, 1995.

[Fishwick 2007] Fishwick, Paul A. The Languages of Dynamic System Modeling In Fishwick, P.A (ed) Handbook of Dynamic System Modeling .Chapter 1. CHAPMAN & HALL/CRC.

[Fishwick et al. 2003] Fishwick, Paul. A., Lee, J., Park, M., and Shim, H. 2003. RUBE: A customized 2D and 3D modeling framework for simulation. In Proceedings of the 2003Winter Simulation Conference: 755–762.

[Formal Systems 2011a] ProBE: Formal Systems, http://www.fsel.com/probe_download.html, accessed August 2011.

[Formal Systems 2011b] CSP Type Checker: Formal Systems, http://www.fsel.com/typechecker_download.html, accessed August 2011.

[ForSyde 2014] http://www.ict.kth.se/forsyde/, accessed October 2014.

[Galloway and W. J. Stoddart 1997] Galloway, A. J., and W. J. Stoddart. "An operational semantics for ZCCS.". In Proceedings of 1st International Conference on Formal Engineering Methods (ICFEM '97), (IEEE Computer Society, Washington, DC, USA, 1997), 272-

[Gamma et al. 1994] Gamma, Erich, et al. Design patterns: elements of reusable object-oriented software. Pearson Education, 1994.

[Geisler et al. 2000] Geisler, R., M. Klar, and S. Mann. "Precise semantics of integrated modeling languages by formal metamodeling." Proc. 5th World Conference on Integrated Design and Process Technology. Society for Design and Process Science. 2000..

[Gholami and Sarjoughian 2012] Gholami, Soroosh, and Hessam S. Sarjoughian. "Observations on real-time simulation design and experimentation." Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative M&S Symposium. Society for Computer Simulation International, 2013.

[Giambiasi et al. 2001] Giambiasi, Norbert, Bruno Escude, and Sumit Ghosh. "Generalized discrete event simulation of dynamic systems." Transactions of the Society for Computer Simulation International 18.4 (2001): 216-229.

[Gogolla et al. 2007] Gogolla, Martin, Fabian Büttner, and Mark Richters. "USE: A UML-based specification environment for validating UML and OCL." Science of Computer Programming 69.1 (2007): 27-34.

[Gordon and Melham 1993] Gordon, Michael JC, and Tom F. Melham. "Introduction to HOL A theorem proving environment for higher order logic." Cambridge University Press, 1993.

[Groβe-Rhode 2004] GroBe-Rhode, Martin. Semantic Integration of Heterogeneous Software Specifications. EATCS Monographs in Theoretical Computer Science. Springer Verlag, Berlin Heidelberg New-York, 2004.

[Hagendorf et al. 2009] Hagendorf, Olaf, Pawletta Thorsten. and Deatcu Christina. "Extended dynamic structure DEVS". Proceedings of the European Moddling and Simulation Symposium (EMSS), 2009.

[Hardebolle and Boulanger 2007] Hardebolle, Cécile, and Frédéric Boulanger. "ModHel'X: A component-oriented approach to multi-formalism modeling." Models in Software Engineering. Springer Berlin Heidelberg, 2008. 247-258.

[Harel and Rumpe 2004] Harel, David, and Bernhard Rumpe. "Meaningful modeling: what's the semantics of" semantics?" IEEE Computer, vol.37, no.10, pp.64,72, 2004.

[Harel et al. 1990] Harel, David, et al. "STATEMATE: A working environment for the development of complex reactive systems." Software Engineering, IEEE Transactions on 16.4 (1990): 403-414.

[Harel 1998] Harel, David, and Michal Politi. Modeling reactive systems with statecharts: the STATEMATE approach. McGraw-Hill, Inc., 1998.

[Henriksen et al. 1996] Henriksen, Jesper G., et al. Mona: Monadic second-order logic in practice. In TACAS '95, LNCS 1019, Springer Berlin Heidelberg, 1996.

[Himmelspach and Uhrmacher 2004] Himmelspach, Jan, and Adelinde M. Uhrmacher. "A component-based simulation layer for JAMES." Proceedings of the eighteenth workshop on Parallel and distributed simulation. ACM, 2004.

[Hoare 1985] Hoare, Charles Antony R. Communicating Sequential Processes. Prentice Hall, 1985.

[Holzmann 2003] Holzmann, Gerard J. The SPIN model checker: Primer and reference manual. Vol. 1003. Reading: Addison-Wesley, 2004.

[Hong et al.1997] Hong, Joon Sung, et al. "A real-time discrete event system specification formalism for seamless real-time software development." Discrete Event Dynamic Systems 7.4 (1997): 355-375.

[Hong and Kim 2004 a] Hong, S., and Tag Gon Kim. "Embedding UML subset into object-oriented DEVS modeling process." SIMULATION SERIES 36.4 (2004): 161.

[Hong and Kim 2004 b] Hong, Ki Jung, and Tag Gon Kim. "Timed i/o test sequences for discrete event model verification." Artificial Intelligence and Simulation. Springer Berlin Heidelberg, 2005. 275-284.

[Hong and Kim 2006] Hong, Ki Jung, and Tag Gon Kim. "DEVSpecL: DEVS specification language for modeling, simulation and analysis of discrete event systems." Information and Software Technology 48.4 (2006): 221-234.

[Hui and Wainer 2006] Shang, Hui, and Gabriel Wainer. "A simulation algorithm for dynamic structure DEVS modeling." Proceedings of the 38th conference on Winter simulation. Winter Simulation Conference, 2006.

[Hu & Zeigler, 2002] Hu, Xiaolin, and Bernard P. Zeigler. An integrated modeling and simulation methodology for intelligent systems design and testing. Arizona University Tucson, 2002.

[Hu & Zeigler 2004] Hu, Xiaolin, and Bernard P. Zeigler. "Model continuity to support software development for distributed robotic systems: A team formation example." Journal of Intelligent and Robotic Systems 39.1 (2004): 71-87.

[Hwang 2012] Hwang, Moon Ho. "Qualitative verification of finite and real-time DEVS networks." Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium. Society for Computer Simulation International, 2012.

[Hwang and Cho 2004] Hwang, Moon Ho, and Su Kyoung Cho. "Timed behavior analysis of schedule preserved DEVS." Simulation series 36.4 (2004): 173.

[Hwang and Zeigler 2009] Hwang, Moon Ho, and Bernard P. Zeigler. "Reachability graph of finite and deterministic devs networks." IEEE Transactions on Automation Science and Engineering 6.3 (2009): 468.

[Ighoroje et al. 2011] Ighoroje, U. B., O. Maïga, and M. K. Traore. "Formal Framework for the DEVS-Driven Modeling Language." In Proceedings of European Modeling and Simulation Symposium (EMSS) 2011, Rome (Italy), 2011.

[Ighoroje et al. 2012] Ighoroje, Ufuoma Bright, Oumar Maïga, and Mamadou Kaba Traoré. "The DEVS-driven modeling language: syntax and semantics definition by meta-modeling and graph transformation." Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium. Society for Computer Simulation International, 2012.

[ISO 2002] Information Technology. Z Formal Specification Notation - Syntax, Type System and Semantics. ISO/IEC 13568. 2002.

[Isobe et al. 2007] Isobe, Yoshinao, and Markus Roggenbach. "Proof principles of CSP–CSP-Prover in practice." Dynamics in Logistics. Springer Berlin Heidelberg, 2008. 425-442.

[Khan and Risoldi 2012] Khan, Yasir Imtiaz, and Matteo Risoldi. "Language enrichment for resilient MDE." Software Engineering for Resilient Systems. Springer Berlin Heidelberg, 2012. 76-90.

[Kim 1994] Kim, Tag Gon. "DEVSIM++ user's manual." Department of Electrical Engineering, KAIST, Korea (1994).

[Kim et al. 2001] Kim, T. G., S. M. Cho, and W. B. Lee. "DEVS framework for systems development: Unified specification for logical analysis, performance evaluation and implementation." Discrete Event Modeling and Simulation Technologies. Springer New York, 2001. 131-166.

[Kleppe 2008] Kleppe, Anneke. Software language engineering: creating domain-specific languages using metamodels. Pearson Education, 2008.

[Krahn et al. 2010] Krahn, Holger, Bernhard Rumpe, and Steven Völkel. "MontiCore: a framework for compositional development of domain specific languages." International journal on software tools for technology transfer 12.5 (2010): 353-372.

[Krasner & Pope 1988] Krasner, Glenn E., and Stephen T. Pope. "A description of the model-view-controller user interface paradigm in the smalltalk-80 system." Journal of object oriented programming 1.3 (1988): 26-49.

[Kreutzer 1986] Kreutzer, W. "System Simulation: Programming Languages and Styles." Addison-Wesley, Reading, MA, 1986.

[Kuhn et al. 2003] Kuhn, D. Richard, Dan Craigen, and Mark Saaltink. "Practical application of formal methods in modeling and simulation." Summer Computer Simulation Conference. Society for Computer Simulation International; 1998, 2003.

[Kwon et al. 1996] Kwon, Yi Wan, et al. "Fuzzy-DEVS formalism: concepts, realization and applications." Proceedings of the 1996 Conference on AI, Simulation and Planning In High Autonomy Systems. 1996.

[Lamsweerde 2000] Lamsweerde, Axel van. "Formal specification: a roadmap." Proceedings of the Conference on the Future of Software Engineering. ACM, 2000.

[Lano and Biccaregui 1999] Lano, Kevin, and Juan Bicarregui. "Formalising the UML in structured temporal theories." Kluwer international series in engineering and computer science (1999): 161-174.

[Lano 2009] Lano, Kevin. UML 2 Semantics and Applications. John Wiley & Sons, 2009.

[Larsen et al. 1997] Larsen, Kim G., Paul Pettersson, and Wang Yi. "UPPAAL in a nutshell." International Journal on Software Tools for Technology Transfer (STTT) 1.1 (1997): 134-152.

[Ledeczi et al. 2001] Ledeczi, Akos, Peter Volgyesi, and Gabor Karsai. "Metamodel composition in the generic modeling environment." Comm. at workshop on Adaptive Object-Models and Metamodeling Techniques, Ecoop. Vol. 1. 2001.

[Leuschel and Butler 2008] Leuschel, Michael, and Michael Butler. "ProB: an automated analysis toolset for the B method." International Journal on Software Tools for Technology Transfer 10.2 (2008): 185-203.

[Linington et al. 2011] Linington, Peter F., et al. Building enterprise systems with ODP: an introduction to open distributed processing. CRC Press, 2011.

[Look et al. 2013] Look, Markus, et al. "Black-box Integration of Heterogeneous Modeling Languages for Cyber-Physical Systems." In GEMOC Workshop 2013 - International Workshop on The Globalization of Modeling Languages, Miami, Florida (USA), Volume 1102 of CEUR Workshop Proceedings, Eds.: B. Combemale, J. De Antoni, R. B. France, CEUR-WS.org, 2013.

[MacDougall 1987] MacDougall, Myron H. Simulating computer systems: techniques and tools. MIT Press, Cambridge, 1987.

[Mahony and Dong 2000] Mahony, Brendan, and Jin Song Dong. "Timed communicating object Z." Software Engineering, IEEE Transactions on 26.2 (2000): 150-177.

[Maïga et al. 2012 a] Maïga, Oumar, Ufuoma Bright Ighoroje, and Mamadou Kaba Traoré. "DDML: A Support for Communication in M&S." Enabling Technologies: Infrastructure for

Collaborative Enterprises (WETICE), 2012 IEEE 21st International Workshop on. IEEE, 2012.

[Maïga et al. 2012 b] Maiga, Oumar, Ufuoma Bright Ighoroje, and Mamadou Kaba Traoré. "Integration des methodes formelles dans la specification, la verification et la validation de modeles de simulation à evenements discrets." 9th International Conference on Modeling, Optimization & SIMulation. 2012.

[Maïga and Traoré 2013] Maiga, Oumar, and Mamadou Kaba Traoré. Approche Formelle de vérification et validation des modèles de simulation. In Journées Scientifiques de l'Ecole Doctorale des Sciences pour l'Ingénieur (EDSPI), Clermont-Ferrand (France), June 2013.

[Maïga and Traoré 2015] Maiga, Oumar, and Mamadou Kaba Traoré. An Integrated approach to the specification, simulation, formal analysis and enactment of discrete event systems. AUSTECH 2015, Abuja, Nigeria, October 2015

[Maïga et al. 2015] Maïga, Oumar, Hamzat Olanrewaju Aliyu, and Mamadou Kaba Traoré. A New Approach to Modeling Dynamic Structure Systems. In European Simulation and Modeling Conference (ESM'2015), Leicester, United Kingdom, October 2015.

[Mathias et al. 2010] Bonaventura, Matías, Gabriel A. Wainer, and Rodrigo Castro. "Advanced IDE for modeling and simulation of discrete event systems." Proceedings of the 2010 Spring Simulation Multiconference. Society for Computer Simulation International, 2010.

[McMillan 1992] McMillan, Kenneth L.. Symbolic Model Checking: An approach to the state explosion problem. Thesis submitted in Carnegie Mellon University 1992.

[MetaGME 2014] MetaGME. Meta-model Composition. http://w3.isis.vanderbilt.edu/projects/gme/composition.html. Accessed 12 August, 2014.

[Micheal 1997] C Michael, Holloway. "Why Engineers Should Consider Formal Methods." Technical Report, NASA Langley Technical Report Server (1997).

[Milner et al. 1992a] Milner, Robin, Joachim Parrow, and David Walker. "A calculus of mobile processes, I." Information and computation 100.1 (1992): 1-40.

[Milner et al. 1992b] Milner, Robin, Joachim Parrow, and David Walker. "A calculus of mobile processes, II." Information and computation 100.1 (1992): 41-77

[Milner 1980] Milner, Robin. "A calculus of communicating systems." Springer Verlag (1980).

[Mittal and Douglass 2012] Mittal, Saurabh, and Scott A. Douglass. "DEVSML 2.0: The language and the stack." Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium. Society for Computer Simulation International, 2012.

[Mittal et al 2007] Mittal, S., J. L. R. Martin, and B. P. Zeigler. "DEVSML: Automating DEVS simulation over SOA using transparent simulators." DEVS Syposium 2 (2007).

[Moellami and Wainer 2011] Moallemi, Mohammad, and Gabriel Wainer. "I-DEVS: imprecise real-time and embedded DEVS modeling." Proceedings of the 2011 Symposium on

Theory of Modeling & Simulation: DEVS Integrative M&S Symposium. Society for Computer Simulation International, 2011.

[Money 2008] Mooney, Joseph. DEVS/UML–A Framework for Simulatable UML Models. Diss. MS Thesis, Computer Science and Engineering Dept., Arizona State University, Tempe, AZ, USA, 2008.

[Mosterman and Vangheluwe 2004] Mosterman, Pieter J., and Hans Vangheluwe. "Computer automated multi-paradigm modeling: An introduction." Simulation 80.9 (2004): 433-450.

[Muzy and Zeigler 2014] Muzy, Alexandre, and Bernard P. Zeigler. "Specification of dynamic structure discrete event systems using single point encapsulated control functions." International Journal of Modeling, Simulation, and Scientific Computing 5.03, 2014, 1450012.

[Nikolaidou et al. 2008] Nikolaidou, Mara, et al. "A sysml profile for classical devs simulators." Software Engineering Advances, 2008. ICSEA'08. The Third International Conference on. IEEE, 2008.

[Nutaro 1999] Nutaro, James. "ADEVS (a discrete EVent system simulator)." Arizona Center for Integrative Modeling & Simulation (ACIMS), University of Arizona, Tucson. Available at http://www. ece. arizona. edu/nutaro/index. php (1999).

[OMG 2010a] Object Management Group (OMG) Unified Modeling Language TM (UML) Version 2.3 May 2010. http://www.omg.org/spec/UML/2.3/

[OMG 2010b] Object Management Group (OMG) System Modeling Language TM (SysML) Version 1.2, June 2010. http://www.omg.org/spec/SysML/1.2/

[OMG 2012] OMG Business Process Modeling Notation (BPMN) 2.0. http://www.omg.org/spec/BPMN/2.0/

[OMG 2013] Object Management Group (OMG) (2013). A Foundational Subset for Executable UML. http://www.omg.org/spec/FUML/1.1. August 2013.

[OMG 2015] Object Management Group (OMG). http://www.omg.org/spec/MOF/2.5

[Ören 1991] Ören, Tuncer I. "Dynamic templates and semantic rules for simulation advisors and certifiers." Knowledge-based simulation. Springer New York, 1991. 53-76.

[Owre et al. 1992] Owre, Sam, John M. Rushby, and Natarajan Shankar. "PVS: A prototype verification system." Automated Deduction—CADE-11. Springer Berlin Heidelberg, 1992. 748-752.

[Pagliero et al. 2003] Pagliero, E., M. Lapadula, and E. Kofman. "Power-DEVS. An Integrated Tool for Discrete Event Simulation." Proceedings of RPIC, San Nicolas, Argentina (2003).

[Patel and ShuKla 2004] Patel, Hiren, and Sandeep Kumar Shukla. SystemC kernel extensions for heterogeneous system modeling: a framework for Multi-MoC modeling & simulation. Springer Science & Business Media, 2004.

[Paulson 1993] Paulson, Lawrence C. The Isabelle reference manual. University of Cambridge, Computer Laboratory, 1993.

[Pawletta et al. 2002] Pawletta, Thorsten, et al. "A DEVS-Based approach for modeling and simulation of hybrid variable structure systems." Modelling, Analysis, and Design of Hybrid Systems. Springer Berlin Heidelberg, 2002. 107-129.

[Plotkin 1981] Plotkin, Gordon D. "A structural approach to operational semantics." DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[Posse 2008] Posse, Ernesto. Modeling and Simulation of Dynamic Structure Discrete-Event Systems, Ph.D Thesis, MCGill University, 2008.

[Präehofer et al. 1999] Praehofer, Herbert, Johannes Sametinger, and Alois Stritzinger. "Discrete event simulation using the JavaBeans component model." SIMULATION SERIES 31 (1999): 107-112.

[Ptolemaeus 2014] Ptolemaeus, Claudius. System Design, Modeling, and Simulation: Using Ptolemy II. Berkeley, CA, USA: Ptolemy. org, 2014. http://ptolemy.org/systems

[Quesnel et al. 2009] Gauthier, Quesnel, Raphaël Duboz and Eric Ramat, "The Virtual Laboratory Environment - An operational framework for multi-modelling, simulation and analysis of complex dynamical systems," Simulation Modelling Practice and Theory, Vol. 17, April 2009, pp. 461-653.

[Raje et al. 1997] Raje, Rajeev R., Joseph I. Williams, and Michael Boyles. "Asynchronous remote method invocation (ARMI) mechanism for Java." Concurrency - Practice and Experience 9.11 (1997): 1207-1211.

[Reed and Roscoe 1986] Reed, George M., and A. William Roscoe. "A timed model for communicating sequential processes." Automata, Languages and Programming. Springer Berlin Heidelberg, 1986. 314-323.

[Ringert et al. 2013 a] Ringert, Jan Oliver, Bernhard Rumpe, and Andreas Wortmann. "MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems." arXiv preprint arXiv:1409.2310 (2014).

[Ringert et al. 2013 b] Ringert, Jan Oliver, Bernhard Rumpe, and Andreas Wortmann. "A Case Study on Model-Based Development of Robotic Systems using MontiArc with Embedded Automata." arXiv preprint arXiv:1408.5692 (2014).

[Risco-Martin et al. 2009] Risco-Martín, José L., et al. "eUDEVS: Executable UML with DEVS theory of modeling and simulation." Simulation 85.11-12 (2009): 750-777.

[Roscoe 2010] Roscoe, A.W. Understanding Concurrent Systems (1st ed.). Springer-Verlag New York, Inc., New York, NY, USA. (2010)

[Rozenblit 1985] Rozenblit, Jerzy W. "Experimental frames for distributed simulation architectures." Proc. of the 1985 Distributed Simulation Conference, San Diego, California. 1985.

[Rozenblit 1991] Rozenblit, Jerzy W. (1991) "Experimental Frame Specification Methodology for Hierarchical Simulation." International Journal of General Systems, Vol. 19, 1991, pp. 317-336.

[Rumbaugh et al. 2004] Rumbaugh, James, Ivar Jacobson, and Grady Booch. Unified Modeling Language Reference Manual, The. Pearson Higher Education, 2004.

[Rusu and Lucanu 2011] Rusu, Vlad, and Dorel Lucanu. "K Semantics for OCL-a Proposal for a Formal Definition for OCL." 2nd International K Workshop. 2011.

[Saadawi and Wainer 2009] Saadawi, Hesham, and Gabriel Wainer. "Verification of real-time DEVS models." Proceedings of the 2009 Spring Simulation Multiconference. Society for Computer Simulation International, 2009.

[Saadawi and Wainer 2010] Saadawi, Hesham, and Gabriel Wainer. "From DEVS to RTA-DEVS." Distributed Simulation and Real Time Applications (DS-RT), 2010 IEEE/ACM 14th International Symposium on. IEEE, 2010.

[Saaltink 1997] Saaltink, Mark. "The z/eves system." ZUM'97: The Z Formal Specification Notation. Springer Berlin Heidelberg, 1997. 72-85.

[Saaltink 1999] Saaltink, Mark. "The Z/EVES 2.0 user's guide." Ora Canada (1999): 31-32.

[Sander and Jantsch 2004] Sander, Ingo, and Axel Jantsch. "System modeling and transformational design refinement in ForSyDe [formal system design]." Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 23.1 (2004): 17-32.

[Santucci and De Gentili 2009] Santucci, Jean-François, and Emmanuelle De Gentili. "Dynamic variable structure modelling and simulation of the Claude Levi-Strauss's mythical thought morphodynamics." Proceedings of the 2009 Spring Simulation Multiconference. Society for Computer Simulation International, 2009.

[Sargent 2000] Sargent, Robert G. "Verification, validation, and accreditation: verification, validation, and accreditation of simulation models." Proceedings of the 32nd conference on Winter simulation. Society for Computer Simulation International, 2000.

[Sargent 2001] Sargent, Robert G. "Verification and validation: some approaches and paradigms for verifying and validating simulation models." Proceedings of the 33nd conference on Winter simulation. IEEE Computer Society, 2001.

[Sarjoughian and Zeigler 1998] Sarjoughian, Hessam S., and B. R. Zeigler. "DEVSJAVA: Basis for a DEVS-based collaborative M&S environment." Simulation Series 30 (1998): 29-36.

[Sarjoughian 2005] Sarjoughian, Hessam, and Dongping Huang. "A multi-formalism modeling composability framework: Agent and discrete-event models." Distributed Simulation and Real-Time Applications, 2005. DS-RT 2005 Proceedings. Ninth IEEE International Symposium on. IEEE, 2005.

[Sarjoughian and Gholami 2013] Sarjoughian, Hessam and Gholami S. "Action-level real-time DEVS modeling and simulation." ACM Transactions on Modeling and Computer Simulation, 2013.

[Schneider and Treharne 2002] Schneider, Steve, and Helen Treharne. "Communicating B machines." ZB 2002: Formal Specification and Development in Z and B. Springer Berlin Heidelberg, 2002.

[Schriber 1986] Schriber, Thomas J. "Introduction to GPSS." Proceedings of the 18th conference on Winter simulation. ACM, 1986.

[Schulz et al. 2000] Schulz, Stephan, T. C. Ewing, and Jerzy W. Rozenblit. "Discrete event system specification (DEVS) and statemate statecharts equivalence for embedded systems modeling." Engineering of Computer Based Systems, 2000.(ECBS 2000) Proceedings. Seventh IEEE International Conference and Workshopon the. IEEE, 2000.

[Smith 2000] Smith, Graeme. "The Object-Z Specification Language." In: Hinchey M. (ed.) Advances in Formal Methods. Kluwer, New York, 2000.

[Smith and Derrick 2001] Smith, Graeme, and John Derrick. "Specification, refinement and verification of concurrent systems—an integration of Object-Z and CSP." Formal Methods in System Design 18.3 (2001): 249-284.

[Spivey 1992] Spivey, J. Michael. Understanding Z: a specification language and its formal semantics. No. 3. Cambridge University Press, 1988.

[Song and Kim 2010] Song, Hae Sang, and Tag Gon Kim. "DEVS diagram revised: a structured approach for DEVS modeling." Proc. European Simulation Conference (Eurosis, Belgium, 2010). 2010.

[Steiniger et al. 2012] Steiniger, Alexander, Frank Kruger, and Adelinde M. Uhrmacher. "Modeling agents and their environment in multi-level-DEVS.", Proceedings of the 2012 Winter Simulation Conference (WSC). IEEE, 2012.

[Sung and Kim 2010] Sung, Changho, and Tag Gon Kim. "Object-Oriented Comodeling Methodology for Development of Domain Specific Models". In G. A. Wainer and P. J. Mosterman (Eds.) Discrete-Event Modeling and Simulation: Theory and Applications, CRC Press, 2010.

[Szlenk 2006] Szlenk, Marcin. "Formal semantics and reasoning about uml class diagram." Dependability of Computer Systems, 2006. DepCos-RELCOMEX'06. IEEE, 2006.

[Thomas et al. 2006] Thomas, C., H. Luckhoff, and T. G. Kim. "OpenDEVS: a proposal for a standardized DEVS model exchange format." Proceedings of AIS. Vol. 96. 1996.

[Traoré 2006] Traoré, Mamadou K. "Making DEVS Models Amenable to Formal Analysis." In Proceedings of Spring Simulation Multiconference (Society for Computer Simulation International, Huntsville, Alabama, USA, 2006: 33-39.

[Traoré and Muzy 2005] Traoré, Mamadou K., and Alexandre Muzy. "Capturing the dual relationship between simulation models and their context." Simulation Modelling Practice and Theory 14.2, Elsevier, (2005): 126-142.

[Traoré 2008] Traoré, Mamadou K. "SimStudio: a next generation modeling and simulation framework." Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[Traoré 2009] Traoré, Mamadou K. "A graphical notation for DEVS." Proceedings of the 2009 Spring Simulation Multiconference. Society for Computer Simulation International, 2009.

[Tripakis et al. 2013] Tripakis, Stavros, et al. "A modular formal semantics for Ptolemy." Mathematical Structures in Computer Science 23.04 (2013): 834-881. Available at: http://chess.eecs.berkeley.edu/pubs/999.html,doi:10.1017/S0960129512000278

[Trojet et al. 2009] Trojet, Mohamed Wassim, Claudia Frydman, and Maâmar El-Amine Hamri. "Practical application of lightweight Z in DEVS framework." Proceedings of the 2009 Spring Simulation Multiconference. Society for Computer Simulation International, 2009.

[Uhrmacher 2001] Uhrmacher, Adelinde M. "Dynamic structures in modeling and simulation: a reflective approach." ACM Transactions on Modeling and Computer Simulation (TOMACS) 11.2 (2001): 206-232.

[Uhrmacher et al. 2006] Uhrmacher, Adelinde M., et al. "Introducing variable ports and multi-couplings for cell biological modeling in DEVS". Proceedings of the Winter Simulation Conference, 2006. WSC 06. IEEE, 2006.

[Vallecillo 2010] Vallecillo, Antonio. "On the combination of domain specific modeling languages." Modelling Foundations and Applications. Springer Berlin Heidelberg, 2010. 305-320.

[Vangheluwe 2000] Vangheluwe, Hans LM. "DEVS as a common denominator for multi-formalism hybrid systems modelling." Computer-Aided Control System Design, 2000. CACSD 2000. IEEE International Symposium on. IEEE, 2000.

[Varro and Pataricza 2003] Varró, Dániel, and András Pataricza. "VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML (The Mathematics of Metamodeling is Metamodeling Mathematics)." Software and Systems Modeling 2.3 (2003): 187-210.

[Wainer 2004] Wainer, Gabriel A. "Modeling and simulation of complex systems with Cell-DEVS." Proceedings of the 36th Winter Simulation Conference, 2004.

[Weisel et al. 2005] Weisel, E. W., M. D. Petty, and R. R. Mielke. "A comparison of DEVS and semantic composability theory." Proceedings of the Spring Simulation Interoperability Workshop. 2005.

[Xtext 2014] http://www.eclipse.org/Xtext/. Acessed on octobre 2014.

[Zeigler 1976] Zeigler, Bernard P. "Theory of Modeling and Simulation". Willey Interscience, 1976.

[Zeigler 1990] Zeigler, Bernard P. "Object-oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems". Academic Press, Boston, 1990.

[Zeigler and Kim 1993] Zeigler, Bernard P., and Jinwoo Kim. "Extending the DEVS-Scheme knowledge-based simulation environment for real-time event-based control." IEEE Transactions on Robotics and Automation 9.3 (1993): 351-356.

[Zeigler et al. 1996] Zeigler, Bernard P., et al. "DEVS-C++: a high performance modelling and simulation environment", Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences, Vol. 1. IEEE, 1996.

[Zeigler et al. 2000] Zeigler, Bernard P., Herbert Praehofer, and Tag Gon Kim. Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems. Academic press, 2000.

[Zinoviev 2005] Zinoviev, Dmitry. "Mapping DEVS models onto UML models." In Proceedings of the 2005 DEVS Integrative M&S Symposium, SpringSim05, pages 101–106, San Diego, CA, April 2005.