UNIVERSITÉ NICE - SOPHIA ANTIPOLIS

ÉCOLE DOCTORALE STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION
ET DE LA COMMUNICATION

T H È S E

pour l'obtention du grade de

Docteur en Sciences

de l'Université Nice - Sophia Antipolis

Mention : Informatique

presentée et soutenue par

Carlos Ernesto Gómez Cárdenas

Une approche multi-vue pour la modélisation système de propriétés fonctionnelles et non-fonctionnelles

Modeling Functional and Non-Functional Properties of Systems Based on A Multi-View Approach

Thèse dirigée par: Frédéric Mallet

et encadrée par: Julien DeAntoni

soutenue le 20 décembre 2013

Jury :

M.	Frédéric Boulanger	Prof.	SUPÉLEC	*Rapporteur*
M.	Abdoulaye Gamatié	C.R.	LIRMM-CNRS	*Rapporteur*
M.	Michel Auguin	D.R.	LEAT-CNRS	*Examinateur*
M.	Jean-Philippe Diguet	D.R.	LabSTICC-CNRS	*Examinateur*
M.	Frédéric Mallet	M.C.	INRIA/I3S-CNRS	*Directeur de Thèse*
M.	Julien DeAntoni	M.C.	INRIA/I3S-CNRS	*Encadrant*

A mis padres, mi hermana y mi Tita...

"Il y a des hommes qui luttent un jour et qui sont bons.
Il y en a d'autres qui luttent un an et qui sont meilleurs.
Il y en a qui luttent pendant des années et qui sont excellents.
Mais il y en a qui luttent toute leur vie et ceux-là sont indispensables."

"There are men who struggle for a day and they are good.
There are men who struggle for a year and they are better.
There are men who struggle many years, and they are better still.
But there are those who struggle all their lives:
These are the indispensable ones."

"Hay hombres que luchan un día y son buenos.
Hay otros que luchan un año y son mejores.
Hay quienes luchan muchos años y son muy buenos.
Pero hay los que luchan toda la vida, esos son los imprescindibles."

Bertolt Brecht

Résumé

Au niveau système un ensemble d'experts spécifient des propriétés fonctionnelles et non fonctionnelles en utilisant chacun leurs propres modèles théoriques, outils et environnements. Chacun essaye d'utiliser les formalismes les plus adéquats en fonction des propriétés à vérifier. Cependant, chacune des vues d'expertise pour un domaine s'appuie sur un socle commun et impacte directement ou indirectement les modèles décrits par les autres experts. Il est donc indispensable de maintenir une cohérence sémantique entre les différents points de vue et de pouvoir réconcilier et agréger chacun des points de vue avant les différentes phases d'analyse.

Cette thèse propose un modèle, dénommé *PRISMSYS*, qui s'appuie sur une approche multi-vue dirigée par les modèles et dans laquelle pour chacun des domaines chaque expert décrit les concepts de son domaine et la relation que ces concepts entretiennent avec le modèle socle. L'approche permet de maintenir la cohérence sémantique entre les différentes vues à travers la manipulation d'événements et d'horloges logiques. *PRISMSYS* est basé sur un profil UML qui s'appuie autant que possible sur les profils SysML, dédié à l'ingénierie système, et MARTE, dédié à la conception de systèmes temps-réel embarqués. Le modèle sémantique qui maintient la cohérence est spécifié avec le langage CCSL qui est un langage formel déclaratif pour la spécification de relations causales et temporelles entre les événements de différentes vues.

L'approche est illustrée en s'appuyant sur une architecture matérielle dans laquelle le domaine d'analyse privilégié est un domaine de consommation de puissance. Le modèle contient différentes vues de cette architecture : modèle fonctionnel, modèle architectural, modèle équationnel de propriétés liées à la température et à la puissance, modèle temporel. L'environnement proposé par *PRISMSYS* permet la co-simulation du modèle et l'analyse. La simulation s'appuie conjointement sur TimeSquare pour les aspects événementiels et liés au contrôle, et sur *SciLab* pour la prise en compte des propriétés non-fonctionnelles (température et puissance). L'analyse est conduite en transformant le modèle multi-vue dans un format adéquat pour *Aceplorer*, un logiciel expert dédié à l'analyse de consommation.

Abstract

At the system-level, experts specify functional and non-functional properties by employing their own theoretical models, tools and environments. Such experts attempt to use the most adequate formalisms to verify the defined system properties in a specific domain. Nevertheless, each one of these experts' views is supported on a common base and impacts directly or indirectly the models described by the other experts. As a consequence, it is essential to keep a semantic coherence among the different points of view and also to be able to reconcile and to include all the points of view before undertaking the different phases of the analysis.

This thesis proposes a specific domain model named *PRISMSYS*. This model is based on a model-driven multi-view approach where the concepts, and the relationships between them, are described for each expert's domain. Moreover, these concepts maintain a relation with a backbone model. *PRISMSYS* allows keeping a semantic coherence among the different views by means of the manipulation of events and logical clocks. *PRISMSYS* is represented in a UML profile, supported as much as possible by SysML, devoted to the systems engineering, and MARTE, dedicated to the design of real-time embedded systems. The semantic model, which preserves the view coherence, is specified by using CCSL, a declarative formal language for the specification of causal and temporal relationships between events of different views.

The approach is illustrated taking as case study an electronic system, where the main domain analysis is power consumption. The system model incorporates various views: a functional model, a power model, a time performance model and a thermal model. In turn, these views are divided in three parts: control, structural, and equational. These parts interact with each other to characterize the temperature and power consumption of the system. The environment proposed by *PRISMSYS* allows the co-simulation of the model and its analysis. The simulation is supported by TimeSquare, for the event aspects and correlated to the control, and by *SciLab*, for taking into account the non-functional properties (temperature and power consumption). The analysis is conduced by transforming the multi-view model in the internal format accepted by *Aceplorer*, an expert tool dedicated to power consumption analysis.

Acknowledgements

This Ph.D. has been a very rewarding experience, both in a personal and professional level for me. It is not only an end, but also the incredible journey it has represented. I hope I will find the right words to properly express my gratitude and recognition to everyone who, directly or indirectly, participated in this work. I will try my best. To begin with, I owe my gratitude to Frédéric Boulanger, Abdoulaye Gamatié, Michel Auguin and Jean-Philippe Diguet, for accepting and evaluating my thesis. I appreciate their comments and the improvement suggestions for this research. To my advisors, Frédéric Mallet and Julien DeAntoni I am deeply in debt. Throughout these years, Frédéric has shared his researching experience with me, he has given me an external optic of my work, and showed me how I could improve it; but above all he has taught me the importance of presenting my ideas written in a clear and proper way. Julien has been beside me during all the development process of my thesis, he has been a continuous guide, who helped me to address and improve my research. I appreciate all the time he has granted me, his permanent availability and also his willingness to redress my initial writing and presentation skills. I am very grateful for their patience and their personal and professional support when I have needed it throughout these last three years.

I want to thank the STIC Doctoral School and the University of Nice-Sophia Antipolis for having rewarded me with the scholarship to develop this thesis. I cherished the opportunity I had of teaching at the Computer Science Department of the University during the last two years. I am deeply grateful to INRIA and I3S/CNRS laboratories who provided the resources and all the administrative support for this research project. I would also express my gratitude and recognition to the French Ministry of Higher Education and Research who provided these scholarships to promote the development of our research. In particular, I would like to express my great appreciation to Patricia Lachaume, our assistant, who has been very attentive, kind and extremely helpful. For our good fortune, she is strongly influenced with the Colombian warmth.

It has been a pleasure for me, to work in the AOSTE team during these years. I would like to address my special thanks to Charles André, who read and commented my first manuscript version, I miss his questions on Spanish grammar, always difficult to answer. My gratitude to Robert de Simone for welcoming me in the team, and also integrating me to the ANR-HELP project, which was in part an inspiration for this thesis; to Marie-Agnès Peraldi and Arda Goknil for their interest and feedback, to Matias Vara I will miss our discussions about our research, to Kelly Garcés and Ameni Khecharem for their help during the implementation of my work. I also want to thank Jean-Vivien, Sid, Jeff,

Calin, Regis, Luc, Nicolas, Benoit, Amin, Emilien, Yuliia, Ling and Zhichao for sharing these years in the group. Thanks to each and every one of you, I was able to find what I was looking for in a research team, and I am extremely glad of sharing it with you.

To the people who motivated me to follow this graduate formation, Philippe Esteban, Jean-Claude Pascal, Mario Paludetto, Fernando Jimenez and Nicanor Quijano. Thank you very much for showing me that it was really possible to begin this journey full of promises and dreams. Now I know it has been worth it.

I am in gratitude with the Docea Power technical support for its help in the use of *Aceplorer*. I also want to thank to the Arcsis-CIM PACA program to provide the access to the *Aceplorer* tool.

Finally, some words in Spanish...

Quiero dar infinitas gracias a mi papá y a mi mamá que me han acompañado y apoyado durante todos estos años, por darme la fuerza y el valor de seguir adelante todos los días y así haber hecho de mi la persona que soy hoy, de la que espero se sientan orgullosos. A mi hermana Carola con quien siempre he contado y con quien nos apoyamos y ayudamos a salir adelante, sobre todo cuando las cosas no han estado tan bien y nos reírnos cuando si lo están, aunque no parezcan. A mis tíos, quienes han sido un gran soporte para mí en Colombia, a mis abuelos por quererme tanto y a mis primos. Me siento muy feliz de tener la familia que tengo y los quiero mucho.

Hoy al cierre de mi tesis tengo mucho que agradecerles, a Kelly, Michael, Camilo, Clara, Oscar, Ruby y Rafael, por acompañarme y apoyarme duranten estos tres años. A Carlos Quintero por ser un gran amigo y hermano desde que llegamos a Francia. A a mis hermanas la Beba y Eugenia y mi segunda mamá Rosario por su soporte y ánimo. A Rebeca por hacernos más agradable nuestra estancia en Europa. A Isabel y Françcois por su apoyo incondicional. Finalmente, a mi esposa Margarita, que siempre será mi novia, por haberse aventurado conmigo en este sueño, por su paciencia, su apoyo, sus correcciones, por leer toda mi tesis, muchas veces, y por ser tú. Tita, gracias por mostrarme que los sueños se pueden alcanzar. Te amo.

Contents

Résumé III

Abstract IV

Acknowledgements V

List of Figures X

List of Tables XII

Abbreviations XIII

1. Introduction (Version en Français) 1

1. Introduction 5

2. Background 9
2.1. Introduction . . . 10
2.2. Structural Concerns . . . 10
2.2.1. Multi-View Modeling . . . 11
2.2.2. Multi-View Approaches and Model Composition . . . 15
2.2.3. Discussion . . . 23
2.3. Behavioral Concerns . . . 24
2.3.1. Models of Computation . . . 25
2.3.2. Heterogeneous Models . . . 27
2.3.3. Discussion . . . 29
2.4. Conclusion . . . 30

3. PRISMSYS: A Multi-View Modeling Language for Specifying Systems 31
3.1. Introduction . . . 32
3.2. PRISMSYS Framework . . . 33
3.2.1. Structural SubView . . . 40
3.2.2. SubView Element . . . 41
3.2.3. Equational SubView . . . 43
3.2.4. Control SubView . . . 46
3.3. UML Profile for PRISMSYS . . . 49

3.3.1. UML Concepts for PRISMSYS . . . 50
3.3.2. MARTE Concepts for PRISMSYS . . . 54
3.3.3. SysML Concepts for PRISMSYS . . . 55
3.4. Semantics of Execution . . . 56
3.4.1. Finite State Machine Semantic Specification . . . 58
3.4.1.1. Finite State Machine Clocks . . . 58
3.4.1.2. Finite State Machine Clocks Relationship . . . 59
3.4.2. Equational View Semantic Specification . . . 64
3.5. Conclusion . . . 69

4. Power Consumption Modeling **71**
4.1. Introduction . . . 72
4.2. Dynamic Power Consumption . . . 73
4.3. Static Power Consumption . . . 74
4.4. Characterization for Power Consumption . . . 75
4.5. Power Management Techniques . . . 77
4.5.1. Clock-Gating . . . 78
4.5.2. Power-Gating . . . 78
4.5.3. Dynamic Voltage-Frequency Scale . . . 80
4.6. Power Design Specification . . . 81
4.6.1. UPF, CPF and IEEE 1801 . . . 81
4.6.2. SystemC . . . 84
4.6.3. UML . . . 84
4.7. Discussion . . . 85
4.8. Conclusion . . . 86

5. PRISMSYS Framework for Power-Aware Modeling **87**
5.1. Introduction . . . 88
5.2. Views . . . 89
5.2.1. Hardware View . . . 89
5.2.2. Application View . . . 92
5.2.3. Power View . . . 93
5.2.4. Clock View . . . 99
5.2.5. Thermal View . . . 102
5.3. Correspondences . . . 105
5.3.1. Allocation . . . 106
5.4. Sub-Correspondences . . . 107
5.5. Conclusion . . . 108

6. PRISMSYS Power-Aware Model Analysis **111**
6.1. Introduction . . . 112
6.2. PRISMSYS Power-Aware Model Simulation . . . 112
6.2.1. Scilab Solver . . . 113
6.2.2. The PRISMSYS Power-Aware Model Scenario . . . 115
6.2.2.1. Application View . . . 115
6.2.2.2. Hardware View . . . 121
6.2.2.3. Clock View . . . 125

6.2.2.4. Power View 129
6.2.2.5. Thermal View 133
6.3. PRISMSYS Power-Aware Model Analysis in *Aceplorer* 135
6.3.1. Transformation Overview 136
6.3.2. *Aceplorer* Domain Model 137
6.3.3. PRISMSYS to *Aceplorer* Transformation 139
6.3.4. *Aceplorer* Code Generation 140
6.3.5. Test Scenario Generation 140
6.4. Conclusion 144

7. Conclusion (Version en Français) 145
7.1. Perspectives 147

7. Conclusion 149
7.1. Future works 151

List of Figures

2.1. Conceptual model for the system architecture context from [2]. 11
2.2. Multi-view modeling according to IEEE-42010. 12
2.3. Architecture Framework concept model [2] 13
2.4. Abstraction levels in MDE. 14
2.5. Abstraction levels of IEEE-42010 concepts [17]. 15
2.6. Relationship between modeling approaches and specific domains. 18
2.7. Petri Net meta-model and a Petri Net model example. 26
2.8. Composition between Synchronous Data Flow and Finite State Machine in Ptolemy II. 28

3.1. *PRISMSYS Framework* meta-model. 34
3.2. Relationship between *Abstraction* correspondence and *viewElement*. . . 36
3.3. Component meta-model and its relationship with *View*, *SubView*, *SubViewElement* and *ConnectorCorrespondence*. 37
3.4. Correspondences and Sub-Correspondences in *PRISMSYS* Framework. . 39
3.5. Example of *structuralSubViews* including the abstraction correspondence. 41
3.6. *SubViewElement* meta-model. 42
3.7. *EquationalSubView* meta-model. 44
3.8. *EquationalSubView* Example 45
3.9. Example of the characterization and equivalence correspondences use. . 46
3.10. Controller meta-model 47
3.11. Example of the use of *ControlSubView* to control the water level of a tank. 48
3.12. Simplified meta-model of *EncapsulatedClassifier*. 52
3.13. State stereotype. 53
3.14. Abstraction of CPU in a layout component view. 54
3.15. Simplified Constraint Block meta-model from the SysML specification. . 56
3.16. Representation of an active state by clocks 60
3.17. Representation of the clock ticks leading to a change between two states caused by a *guardEvent*. 61
3.18. Representation of the clock ticks leading to a change between two states caused by a *triggerEvent*. 63
3.19. PRISMSYS model where the temperature of a CPU is characterized in the *equationalSubView*. 66
3.20. Temperature evolution through time according a predefined execution scenario. 69

4.1. CMOS inverter circuit. 72
4.2. Leakage currents of a NMOS transistor. 74

4.3. Example of a clock gating implementation. 78
4.4. Example of a power gating implementation. 79
4.5. Example of a retention register. 79
4.6. Example of Power Domain association. 83

5.1. Hardware View meta-model. 90
5.2. Hardware View of the *PRISMSYS* power-aware model. 91
5.3. Application View Meta-model. 92
5.4. Application View of the *PRISMSYS* power-aware model. 93
5.5. Power View Meta-model. 94
5.6. Power View of the *PRISMSYS* power-aware model without including its *equationalSubView*. 96
5.7. *EquationalSubView* of *PowerView*. 98
5.8. Clock View Meta-model. 100
5.9. Clock View of the *PRISMSYS* power-aware model without including its *equationalSubView*. 101
5.10. Equational Sub-view of Clock View. 102
5.11. Thermal view Meta-Model. 103
5.12. Thermal view of the *PRISMSYS* power-aware model. 104
5.13. Equational Sub-View of Thermal View. 105
5.14. Example of the *Abstraction* and *ControlConnector* correspondences between *PowerView* and *HardwareView*. 106
5.15. Example of *Allocation* correspondence between *ApplicationView* and *HardwareView*. 107
5.16. Example of *Characterization* sub-correspondence in *PowerView*. 108
5.17. *PRISMSYS* Power-Aware Model Overview. 110

6.1. Overview of the *PRISMSYS* framework co-simulation implementation. . 113
6.2. Sequence diagram of the PRISMSYS model Simulation. 114
6.3. Execution of *ApplicationView* and its interaction with *HardwareView*. . 116
6.4. *ApplicationView* simulation in TimeSquare. 120
6.5. Execution of the *HardwareView controlSubView* and its interaction with *ApplicationView*, *PowerView* and *ClockView*. 122
6.6. *HardwareView* simulation in TimeSquare. 124
6.7. Execution of the *ClockView controlSubView* and its interaction with its internal *subViewElements* and with *HardwareView*. 126
6.8. *ClockView* simulation in TimeSquare. 128
6.9. Execution of the *HardwareView controlSubView* and its interaction with *ApplicationView*, *PowerView* and *ClockView*. 130
6.10. *Power View* simulation in TimeSquare. 132
6.11. *Thermal View* simulation in TimeSquare. 134
6.12. Transformation Overview. 137
6.13. Simplified *Aceplorer* meta-model. 138
6.14. Control View Scenario generated by TimeSquare (above) and the power consumption response in *Aceplorer* (below). 143

List of Tables

3.1. PRISMSYS - UML Mapping. 51
3.2. PRISMSYS - MARTE Mapping. 54
3.3. PRISMSYS - SysML Mapping. 56
3.4. Clocks representing the relevant actions in a Finite State Machine for both *SubViewElement* and *Controller*. 59

6.1. Action execution in *cpu* clock cycles and time. 126
6.2. Multi-View - *Aceplorer* Mapping. 139

Abbreviations

ATL	**A**TLAS **T**ransformation **L**anguage
CPF	**C**ommon **P**ower **F**ormat
CTM	**C**ompact **T**hermal **M**odel
DVFS	**D**ynamic **V**oltage-**F**requency **S**cale
DSML	**D**omain **S**pecific-**M**odeling **L**anguages
ESL	**E**lectronic **S**ystem-**L**evel
FSM	**F**inite **S**tate **M**achine
HDL	**H**ardware **D**escription-**L**anguage
MARTE	**M**odeling and **A**nalisis of **R**eal **T**ime and **E**mbedded Systems
MDA	**M**odel-**D**riven **A**rchitecture
MDE	**M**odel-**D**riven **E**ngineering
MOF	**M**eta **O**bject **F**acility
MoC	**M**odel **o**f **C**omputation
NFP	**N**on-**F**unctional **P**roperty
EMF	**E**clipse **M**odeling **F**ramework
QVT	**Q**uery **V**iew **T**ransformation
RTL	**R**egister-**T**ransfer **L**evel
SysML	**Sys**tems **M**odeling **L**anguage
TLM	**T**ransaction-**L**evel **M**odeling
UML	**U**nified **M**odeling **L**anguage
UPF	**U**nified **P**ower **F**ormat
VCD	**V**alue **C**hange **D**ump
VHDL	**V**HSIC **H**ardware **D**escription **L**anguage
VSL	**V**alue **S**pecification **L**anguage

Chapitre 1

Introduction (Version en Français)

La notion de système englobe des environnements plus ou moins complexes. Les téléphones filaires autrefois limités à l'aspect communication ont été remplacés par les téléphones GSM qui combinent l'envoi de texto, le guidage GPS des utilisateurs, la lecture d'un journal et/ou d'un livre ou encore la navigation sur Internet. Les systèmes ont aussi été mis-à-jour avec une technologie plus sophistiquée, où l'optimisation de certaines propriétés est une priorité aujourd'hui. Les systèmes électroniques sont maintenant intégrés dans les voitures, les avions, les bateaux et les trains. Ces systèmes numériques se veulent plus efficaces et plus flexibles que les systèmes purement mécaniques en aidant à réduire la consommation de carburant, les coûts de maintenance et en améliorant la qualité fonctionnelle.

Dans le but de gérer la complexité des systèmes modernes, les architectes des systèmes divisent les aspects en plusieurs domaines. Chaque domaine est conçu, étudié et analysé par des experts spécifiques qui s'y intéressent spécifiquement. Ces préoccupations sont quantifiées par les propriétés établies dans le cahier des charges du système. Ces propriétés peuvent être soit fonctionnelles (arrêter une voiture quand la pédale du frein est appuyée), ou non fonctionnelles (déterminer un budget sur la consommation de puissance et de carburant, les temps de réponse, la taille et les coûts). Habituellement, les experts ont leurs propres langages et outils pour modéliser et analyser un domaine

spécifique. Cependant, ces domaines sont liés et interagissent afin de respecter les exigences du système. Par exemple, dans les voitures électriques ou hybrides, l'action de freinage pourrait générer de l'énergie qui peut être stockée dans les batteries pour être réutilisée lorsque la voiture a besoin d'accélérer. Ce cycle peut réduire la consommation de puissance ou de carburant de la voiture en améliorant certaines propriétés non fonctionnelles.

Nous proposons d'exprimer comme des *vues*, chacun des domaines du système. IEEE-1471 [1] et IEEE-42010 [2] sont des standards qui proposent une structure générique afin de spécifier un système avec de multiples vues. Cette manière de décrire un système est appelée *modélisation multi-vue*. Cependant, ces standards sont extrêmement généraux, ils peuvent donc être appliqués de différentes façons. En plus, en utilisant ces standards, c'est difficile de décrire les concepts réutilisables définis dans une architecture pour les appliquer ailleurs.

Dans cette thèse, nous proposons *PRISMSYS*, un langage de modélisation muti-vue qui permet de spécifier les domaines des experts dans une variété de vues. *PRISMSYS* est inspiré par les concepts définis dans IEEE-42010 [2]. Néanmoins, nous proposons des éléments spécifiques inclus dans les vues, ses comportements, ses associations et ses interactions. En utilisant l'Ingénierie Dirigée par les Modèles, nous donnons une syntaxe à *PRISMSYS*, *i.e.,* la structure de l'architecture du système. La structure de *PRISMSYS* est spécifiée par un *méta-modèle*.

PRISMSYS inclut deux types de comportements : un comportement à événements discrets, représenté par des machines à états et l'interaction parmi des vues définie par des événements. Il prévoit aussi un comportement EN temps continu, exprimé par des 2quations. Nous définissons la sémantique d'exécution de ces comportements en utilisant CCSL [3], un langage déclaratif qui décrit les relations causales et temporelles entre événements. En employant CCSL, nous spécifions la coordination du comportement des différents domaines d'exécution. Nous orchestrons aussi les différent modèles (a priori hétérogènes) du comportement dans les vues définies, comme la synchronisation entre l'activation des états d'une machine à états finis (un comportement à événements discrets) et l'évaluation des équations (un comportement en temps continu).

Nous représentons *PRISMSYS* comme un profil UML. Le profil de *PRISMSYS* utilise autant que possible les concepts définis dans les profils UML SysML [4] et MARTE [5].

Une fois que la sémantique d'exécution de *PRISMSYS* est définie, nous utilisons TimeSquare [6] afin de simuler la partie discrète du modèle. Pour évaluer la partie continue, nous choisissons *Scilab* [7], une outil de calcul numérique qui offre les fonctions pour résoudre les équations. Nous avons développé un *connecteur* entre TimeSquare et *Scilab* pour orchestrer la simulation discrète avec la partie continue.

Pour illustrer le potentiel de *PRISMSYS*, nous avons développé un modèle d'un système dont la principale préoccupation est la consommation de puissance. Dans ce modèle, nous définissons les vues et les éléments qui décrivent et impactent la consommation de puissance d'un système. Ce modèle est simulé et les comportements discrets et continus sont présentés (*e.g.,* le comportement de la machine d'états finis, et aussi l'évolution de la consommation de puissance et la température). Finalement, nous proposons une autre manière d'utiliser le modèle *PRISMSYS*. Nous spécifions une transformation du modèle *PRISMSYS* vers un autre modèle d'un outil de domaine spécifique. En prenant comme cas d'étude le modèle *PRISMSYS* dédié à la consommation de puissance, nous le transformons dans le format interne d'*Aceplorer* afin de simuler et analyser la consommation de puissance. *Aceplorer* [8] est un outil commercial qui modélise et simule le comportement de la consommation de puissance d'un système. *Aceplorer* a été utilisée dans le cadre du projet ANR-HeLP (référence ANR-09-SEGI-006).

Le contenu de cette thèse est organisé en deux parties principales : La définition de la structure de *PRISMSYS*, et le développement du cas d'étude de *PRISMSYS*, un modèle du système dédié à la consommation de puissance.

La première partie introduit les concepts principaux de la modélisation multi-vue et de l'hétérogénéité du comportement spécifié dans le modèle d'un système. En conséquence, cette partie est consacrée à la spécification de la structure de *PRISMSYS*. Cette partie est composée des chapitres 2 et 3. Le premier chapitre introduit l'état de l'art des préoccupations structurelles et comportementales afin de modéliser les systèmes. Nous introduisons les concepts de modélisation multi-vue identifiés par la spécification IEEE-42010. Finalement, Nous identifions une relation entre la modélisation multi-vue et la composition des modèles. Sur les préoccupations comportementales, nous introduisons la notion de Modèle de Calcul (MoC), les outils qui les implémentent, comme Ptolemy II [9] et ModHel'X [10], et nous discutons également le problème d'hétérogénéité parmi différents MoCs. Le chapitre 3 définit la structure de *PRISMSYS*, sa syntaxe et sa

sémantique pour spécifier un modèle multi-vue d'un système. La syntaxe de *PRISMSYS* est spécifié par un *meta-modèle*. *PRISMSYS* suit une approche par composants, où les concepts multi-vue sont specifiés en accord avec cette approche. Une *vue* est exprimée par trois *sous-vues* principales : *controlSubView*, *StructuralSubView* et *EquationalSubView*. Chaque sous-vue joue un rôle spécifique dans la construction d'une vue.

La deuxième partie de cette thèse est dédiée à la modélisation d'un système dont la préoccupation principale est la consommation de puissance. Ce modèle est défini en utilisant la structure de *PRISMSYS*. Cette partie de la thèse est composée des chapitres 4, 5 and 6. Le chapitre 4 introduit les concepts, les techniques, et les outils employés pour modéliser la consommation de puissance d'un système. Nous spécifions les vues et ses éléments afin d'évaluer et d'analyser le modèle *PRISMSYS* dédié à la consommation de puissance dans le chapitre 5. Nous simulons, évaluons et analysons le modèle *PRISMSYS* dédié à la consommation dans le chapitre 6 en utilisant TimeSquare, *Scilab* et le connecteur *Scilab Solver* construit pour l'occasion. Dans ce chapitre, nous spécifions également la transformation de *PRISMSYS* vers *Aceplorer*.

Finalement, nous concluons ce travail, en soulignant les contributions principales et nous donnons quelques perspectives futures dans le chapitre 7.

Chapter 1

Introduction

Nowadays, the complexity of systems is increasing. It began with simple devices that performed a specific functionality, such as a telephone that makes calls through a cable, and now, these devices are much more complex including new functionalities and new technologies. For instance, the telephone is being replaced by mobile phones, which are wireless and have multiple functionalities such as sending messages, orienting people to arrive to a destination or allowing to read news and books or to surf on the Internet. Systems have also been upgraded with a more sophisticated technology, where the optimization of certain properties is a priority today. Electronic systems are now integrated in cars, airplanes, boats and trains. These systems are more precise than the mechanical ones helping to reduce gas consumption, maintenance costs and improving the functional quality.

To deal with the complexity of modern systems, system architects split them in various domains. Each domain is designed, studied and analyzed by experts that specify determined stakeholder's concerns. These concerns are quantified by properties stated in system requirements. Such properties can be either functional, such as stopping a car when the brake pedal is pressed, or non-functional, like power and gas consumption, time performance, size and costs. Usually, the experts have their own languages and tools to model and analyze a specific domain. However, these domains are connected and they interact to fulfill the system requirements. For instance, in electric or hybrid cars, the braking action could generate some energy that can be stored in batteries to

be re-used once the car needs to accelerate. This cycle can reduce the power or gas consumption of the car, improving certain non-functional properties.

The multiple domains that could be defined in a system are tackled by expressing them in *views*. IEEE-1471 [1] and IEEE-42010 [2] are standards that propose a generic framework to specify a system in multiple views. This way to describe a system is named *multi-view modeling*. Nevertheless, these standards are extremely general, therefore they can be applied in different ways. Moreover, by using these standards, it is difficult to describe re-usable concepts defined in an architecture in order to apply them in a different one.

In this thesis, we propose *PRISMSYS*, a multi-view modeling language that allows specifying expert's domains in various views. *PRISMSYS* is inspired by the concepts defined in IEEE-42010 [2]. However, we propose specific elements included in the views, their behavior, associations and interactions. By using Model Driven Engineering, we give a syntax to *PRISMSYS*, *i.e.,* the system architecture structure. The *PRISMSYS* structure is specified by *meta-models*. Model Driven Engineering defines a clear separation of abstraction levels where *meta-model* is one of them. Thanks to these abstraction levels, we can split those specified in IEEE-42010.

PRISMSYS includes two kinds of behaviors: a discrete event behavior, represented by state machines and the event interaction between views, as well as a continuous time behavior, expressed by equations whose values are evaluated through time. We define the execution semantics of this behavior in CCSL [3], a declarative language that describes causal and temporal relationships between events. By employing CCSL, we specify the coordination of the behavior from different execution domains. We also orchestrate the heterogeneity in the behavior modeling in the defined views, such as the synchronization between a finite state machine (a discrete event behavior) and the evaluation of an equation (a continuous time behavior).

We represent *PRISMSYS* in UML by specifying a profile. The *PRISMSYS* profile uses as much as possible the concepts defined in other UML profiles, such as SysML [4] and MARTE [5]. The concepts that are not included in UML or in the other two profiles, are defined as stereotypes in the *PRISMSYS* profile, extending the UML concepts whose meaning is compatible with the *PRISMSYS* concept semantics.

Once the semantics of the *PRISMSYS* execution is defined, we use TimeSquare [6] to simulate the discrete part of the model. To evaluate the continuous part, we chose *Scilab* [7], a numerical computing tool that provides the functions to solve equations. We have developed a *connector* between TimeSquare and *Scilab* to orchestrate the discrete simulation with the continuous one.

To prove the potential of *PRISMSYS*, we have developed a model of a power-aware system. First, we introduce a background in power consumption characterization and power management. We continue defining the views and the elements that describe and impact the power consumption of a system. This model is simulated and the discrete and continuous behaviors are depicted (*e.g.,* finite state machine behavior, and also power and temperature evolution). Finally, we propose another way to use the *PRISMSYS* model. We specify a transformation of the *PRISMSYS* model to a model of a specific domain tool. Taking as use case the *PRISMSYS* power-aware system model, we transform it to an *Aceplorer* model in order to simulate and analyze the power consumption. *Aceplorer* [8] is a commercial tool that models and simulates the power behavior of a system. *Aceplorer* was used in the context of the ANR Project HeLP (reference ANR-09-SEGI-006).

The content of this thesis is organized in two main parts: The definition of the *PRISMSYS* framework, and the development of the *PRISMSYS* use case, a power-aware system model.

The first part introduces the main concepts of multi-view modeling and highlights the behavior heterogeneity specified in a system model. Therefore, this first part is the stronghold in the specification of the *PRISMSYS* framework. This part is composed of chapters 2 and 3. The former introduces the background about structural and behavioral concerns to model systems. We present that the complexity of a system architecture could be managed following the multi-view approach. We introduce the multi-view concepts specified in IEEE-42010. We also split the abstraction level defined in IEEE-42010 by using the Model-Driven Engineering abstraction levels. Finally, we identify a relationship between the multi-view modeling and the model composition. In the behavioral concerns, we introduce the notion of Model of Computation (MoC), the tools that implement them, such as Ptolemy II [9] and ModHel'X [10], and we also discuss the

heterogeneity problem between various MoCs. Chapter 3 defines the *PRISMSYS* framework, its syntax and semantics to define a multi-view system model. The *PRISMSYS* syntax is specified by *meta-models*. *PRISMSYS* follows a component approach, where the multi-view concepts are specified accordingly. A *view* is expressed by three main *sub-views*: *controlSubView*, *StructuralSubView* and *EquationalSubView*. Each sub-view plays a specific role in the construction of a view.

The second part of this thesis is dedicated to the modeling of a power-aware system by using *PRISMSYS*. This part consists of chapters 4, 5 and 6. Chapter 4 introduces the concepts, techniques, and tools employed to model the power consumption of a system. We specify the views and their elements to describe various domains that are involved in the system power consumption in Chapter 5. We simulate, evaluate and analyze the *PRISMSYS* power-aware model in Chapter 6 by using TimeSquare, *Scilab* and their connector *Scilab Solver*. In this chapter, we also specify the transformation of *PRISMSYS* to *Aceplorer*.

Finally, we provide the conclusion of this work, highlighting its main contributions and we give some future perspectives in Chapter 7.

Chapter 2

Background

Contents

2.1. Introduction 10

2.2. Structural Concerns 10

2.2.1. Multi-View Modeling 11

2.2.2. Multi-View Approaches and Model Composition 15

2.2.3. Discussion 23

2.3. Behavioral Concerns 24

2.3.1. Models of Computation 25

2.3.2. Heterogeneous Models 27

2.3.3. Discussion 29

2.4. Conclusion 30

2.1. Introduction

Systems have a strong foothold in our daily life. In the customer electronics market, mobile phones, tablets, video and music players, and TVs are some examples of these systems. They provide a quick and direct access to the information (email, news, articles, books, etc) and they are marking a milestone in communications, giving a great mobility to consumers. These systems are also installed in cars, airplanes, boats and submarines to upgrade certain mechanical controllers or optimize energy consumption, time performance and costs. Medicine is also an important domain where systems play an important role, *e.g.,* measuring blood pressure, dosing medicament or pacing the heart.

Experts from different domains work together in the design of systems. These experts fulfill the strict system requirements, generally specified by non-functional properties such as time performance, security, power consumption, temperature and cost. Each expert has his/her own language to describe the model of the system from his/her point of view. Therefore, a system model is represented by multiple languages where each language satisfies certain system requirements.

Whatever its complexity, a language is always defined by a syntax and a semantics. In this thesis, we use the term "syntax" to refer to the structural definition of the language. In contrast, the term "semantics" describes the behavior of the language.

In this chapter, we present the concepts and the approach that we use in this thesis to define the structure and the behavior of the languages that model systems.

2.2. Structural Concerns

According to IEEE-1471 [1], a system is *"a collection of components organized to accomplish a specific function or set of functions"*. This standard also defines *architecture* as *"the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution"*. Taking into account these two definitions, an architecture specifies the structure of a system, based on a component approach.

To define a system architecture, it is important to identify the elements involved in the design of a system. IEEE-15288 standard [11] defines a *system* as *"man-made, created and utilized to provide products and/or services in defined environments for the benefit of users and other stakeholders"*. Following this definition, we identify that a *system* is associated with two main entities: *environment* and *stakeholder*. Figure 2.1 presents a conceptual model of the identified elements that are associated with a system. In this figure, a *system* responds to the *stakeholder* needs and it is placed in an *environment*. An *environment* may contain other systems or subsystems that interact with each other. A *system* exposes one and only one *architecture*.

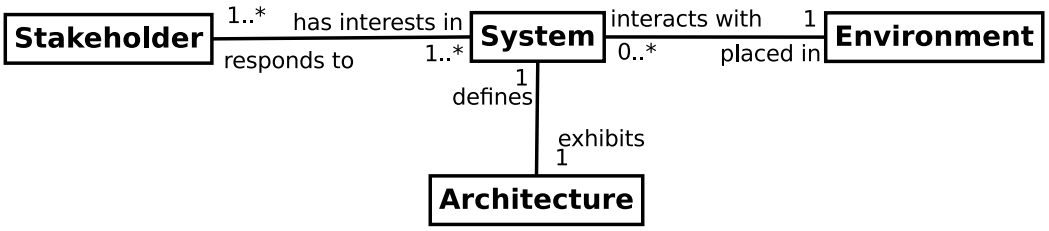


FIGURE 2.1: Conceptual model for the system architecture context from [2].

The stakeholder needs are represented by *concerns* in IEEE-1471 [1]. These concerns are defined in various specific domains that are studied by different experts. These experts build system models that include functional and non-functional properties to tackle the concerns related to their domain. The modeling activity where concerns are divided into various domains is called *multi-view modeling*.

In Section 2.2.1, we present the main concepts of multi-view modeling using the IEEE-42010 standard [2]. This standard is a reference in this kind of modeling.

2.2.1. Multi-View Modeling

Multi-view modeling was proposed as a solution to manage the complexity of the system design. This technique defines a system architecture in different views where each view addresses a set of stakeholder's concerns [1]. Views are defined by domain experts that have their own concepts and languages to express the domain elements and their relationships. An example of this modeling technique is applied to construction. To construct a building, architects design floor plans, electrical engineers draw electrical blueprints and hydraulic engineers create pipe networks. The electrical blueprints and

the pipe networks are defined based on the floor plans, therefore, in this particular case, there is a reference model to build the other domain models. Similar to the construction domain, systems can be specified with diverse views; for instance, power consumption view, financial view, structural view and time performance view.

In this thesis, we use the vocabulary specified in the IEEE-42010 standard [2] to describe the multi-view concepts. This standard is an updated version of IEEE-1471 [1] and it is inspired by various multi-view approaches such as DoDAF [12], MODAF [13], TOGAF [14], the "4+1" view model [15] and Zachman's framework [16].

According to the IEEE-42010 standard, a system architecture is represented by an *architecture description*. The standard emphasizes that an *architecture* is *"abstract, consisting of concepts and properties"*, whereas *architecture description* is a work-product used to define an architecture. Figure 2.2 presents the conceptual model defined in IEEE-42010. In the figure, an *architecture description* owns *views* and *correspondences*. A view contains *models* that are the modeling artifacts describing the view. *Correspondence* builds associations among architecture elements that define the considered system, *i.e.,* the relationship between models, views, the architecture description, stakeholders, and concerns. The main purpose of *Correspondence* is to identify the view elements that have some kind of association in a system architecture in order to maintain the consistency of the architecture description.

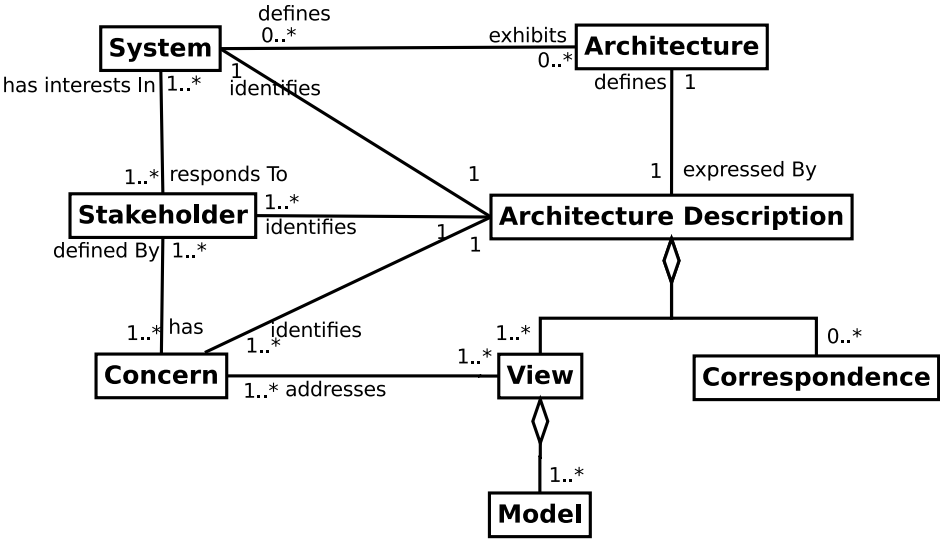


FIGURE 2.2: Multi-view modeling according to IEEE-42010.

This standard also specifies a mechanism to build architecture descriptions which could be reused in various projects that share the same architecture concepts. For this objective, IEEE-42010 introduces the *Architecture Framework* concept. *Architecture description* is the reification of *architecture framework*, *i.e.,* the architecture framework concepts are used to build the architecture description of a system architecture. Figure 2.3 presents the conceptual model of *architecture framework*. An *architecture framework* owns *viewpoints*, and *correspondence rules*. *Views* and *correspondences* conform to *viewpoints* and *correspondence rules*, respectively. A *viewpoint* contains *model kinds* where *models* conform to them.

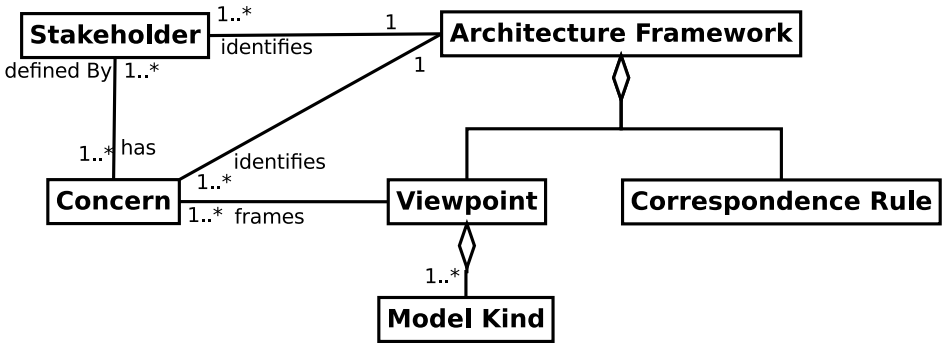


FIGURE 2.3: Architecture Framework concept model [2]

IEEE-42010 defines a conceptual model where *architecture framework* concepts and *architecture description* concepts are mixed, *i.e., models*, *model kinds*, *views*, *viewpoints*, *correspondences* and *correspondence rules* are contained in an *architecture description*. Demirli et al. [17] consider that *architecture framework* concepts and *architecture description* concepts are different abstraction levels. Demirli proposes to use the Model-Driven Engineering approach to model the abstraction levels of the architecture defined in IEEE-42010.

Model-Driven Engineering (MDE) is a software design technique where the main artifact is *model*. The Object Management Group (OMG) defines that *"a model is a representation of a part of the function, structure and/or behavior of a system. The model specification is based on a language that has a well-defined form (syntax), meaning (semantics) and possible rules of analysis, inferences or proof for its constructs."* [18]. According to this definition, a model is built based on a language that gives the necessary expressivity to represent the elements of a specific domain. This language is described through a *meta-model*. A *meta-model* expresses the concepts and relationships to build

a model. A meta-model is a model by itself, so that it has another language that contains the required concepts and relationships to define one or more meta-models. Such a language is called *meta-meta-model*. Examples of meta-meta-models are MOF [19] and Ecore [20]. MDE does not propose another language to build meta-meta-models. A meta-meta-model is rather considered as a self-defined model, *i.e.,* its concepts and relationships are represented by them-selves. This self-definition avoids the multiplication of abstraction levels. In Figure 2.4, we present the abstraction levels in MDE. In the figure, we identify an association of conformity between the concepts of each level, *i.e.,* each level relies on the concepts defined in the upper abstraction level. The *M0* level denotes the real world. In this level, the concrete objects are represented by the elements of a model.

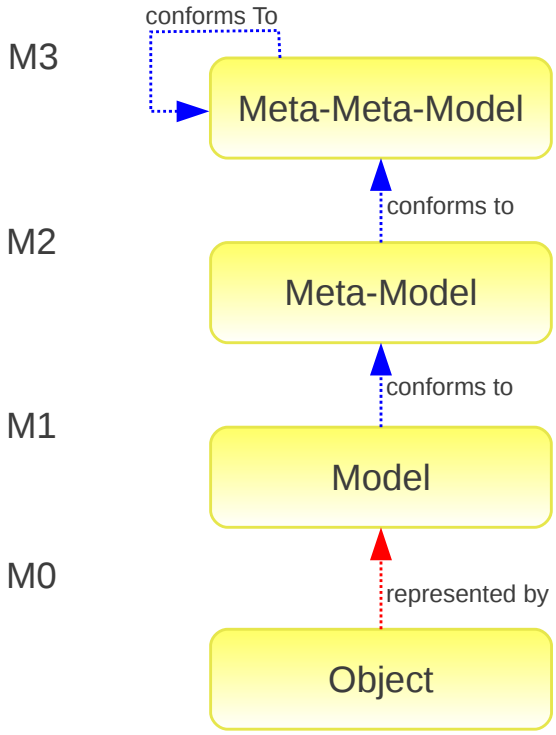


FIGURE 2.4: Abstraction levels in MDE.

Following the MDE abstraction levels, Demirli identifies that the *architecture framework* conceptual model is the *meta-model* of *architecture description* conceptual model. Figure 2.5 depicts the abstraction level representation of IEEE-42010 concepts according to Demirli's work [17].

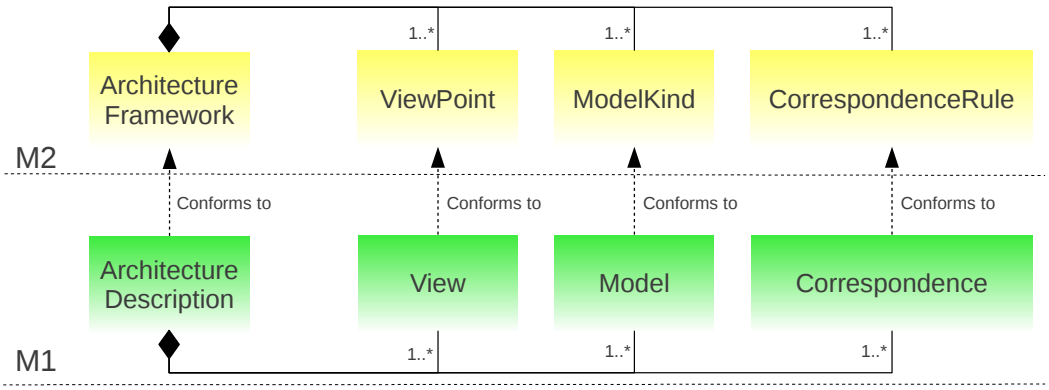


FIGURE 2.5: Abstraction levels of IEEE-42010 concepts [17].

MDE offers two alternative solutions for the definition of models: either through a General-Purpose Modeling Language (GPML) or through a Domain-Specific Modeling Language (DSML). GPML proposes to use a unique meta-model that has enough expressivity to define any domain. UML [21] and XML are examples of GPMLs. DSML proposes to define one dedicated meta-model for each specific domain. SysML [4], MARTE [5], AADL [22] and ATL [23] are examples of DSMLs. Hence, we consider *architecture framework* as a set of DSMLs with a set of correspondence rules between the DSML elements.

An example of the IEEE-42010 implementation is MEGAF [24]. MEGAF is a tool to build architecture frameworks according to the IEEE-42010 standard. This infrastructure allows creating viewpoints, stakeholders and concerns to describe a specific system. MEGAF also defines associations between the specified architecture elements to enable consistency checks based on the defined correspondences.

In the following subsection, we present approaches based on the multi-view modeling requirements defined in IEEE-42010. We also explore an alternative solution through the so-called /model composition/ and we compare the two solutions.

2.2.2. Multi-View Approaches and Model Composition

There are two approaches that use the multi-view concept specified in IEEE-42010: *synthetic* and *projective* [2]. A synthetic approach defines one *viewPoint* for each specific domain, independently. It integrates these *viewPoints* in an architecture framework by

using correspondence rules. In contrast, a projective approach specifies a reference meta-model, where the *viewPoints* are built by hiding irrelevant elements from its meta-model. In this approach, correspondence rules are already defined in the reference meta-model.

Model composition is another modeling approach used in software engineering to combine models with a specific purpose. These models can conform to a common meta-model, or to different ones. Clavreul [25] defines that *Model Composition* is an activity that "*enables to build a system from the union of independent or dependent software artifacts*".

Similarly to the multi-view approaches, model composition specifies correspondences between the elements of the models (or meta-models) to be combined. Clavreul defines four main types of correspondences to classify the model element relationships. These correspondences are: *operator-based*, *rule-based*, *model-based* and *delta representation-based*. *Operator-based* is a set of functions whose actions define the correspondences among model elements. *Rule-based* finds the similarity between model elements, such as term-matching on names or satisfies certain constraints to associate model elements, such as pre- or post-conditions. *Model-based* is a correspondence type that is formally defined as part of the modeling language specification, *e.g.,* DSMLs. Finally, *delta representation-based* is a correspondence that identifies by analysis the differences between two or more versions of the same model.

Clavreul also identifies various interpretations to these correspondences. He defines two interpretation categories in modeling structural associations: *overlapping* and *cross-cutting*. *Overlapping* is to merge one or more models gathering the model elements that have equal or similar interpretation. *Cross-cutting* is to weave new model elements (aspects) to a base model, modifying the structure and/or behavior. Clavreul also defines two additional categories: *add* and *delete*. These categories insert/delete model elements in a model. Clavreul considers that the designer must know the internal model structure in order to use the latter two categories. In contrast, using the previous three interpretation categories does not require a knowledge of the internal model structure to define correspondences.

Multi-view approaches and model composition have in common the notion of correspondence. Clavreul defines correspondence as *"any kind of implicit or explicit relationships*

between sets of models or sets of model elements". This definition is shared with IEEE-42010. However, IEEE-42010 specifies *correspondence* through *correspondence rules*, *i.e.,* a *correspondence* is the use of a *correspondence rule* definition in a model.

The correspondence and interpretation given by Clavreul could be applied to the definition of *correspondence rules*. Nevertheless, the application of *correspondence rules* in model composition and the multi-view approaches are different. While the synthetic approach only uses *correspondence rules* to associate concepts of various DSMLs without generating a new DSML, model (or meta-model) composition has as goal to get a resulting model (or meta-model) that is built by combining one or more models of the same language or from different languages using *correspondence rules*. In the case of the projective approach, *correspondence rules* are defined in the reference meta-model from where the *viewPoints* are derived.

Figure 2.6 depicts the relationship between languages, defined by meta-models, and the modeling approaches. In this figure, *MM1* and *MM2* are independent meta-models (or languages). The elements of both meta-models are associated by *correspondence rules*. The correspondence rules can be in both senses, *i.e.,* they associate elements from *MM1* to *MM2* or vice versa. The two languages (*MM1* and *MM2*) and their correspondence rules define a multi-view synthetic approach. The idea of this approach is to define the correspondence rules between *viewPoint* elements, in order to maintain the coherence between *viewPoints*. Using the synthetic approach, we can generate a composed language (*MM3*) that is the result of the interpretation of correspondence rules between *MM1* and *MM2*. The projective approach is the decomposition of a language in other languages, *i.e., MM3* can be decomposed in *MM1* and *MM2*. The correspondence rules in *MM3* are internal relationships between its elements, *i.e.,* it is part of the domain definition. Therefore, the composition of *MM1* and *MM2* keeps the correspondence rules defined between *MM1* and *MM2*. Once the projective approach is applied, the correspondence rules between *MM1* and *MM2* are identified in *MM3* in order to extract such correspondences and to define associations between *MM1* and *MM2*.

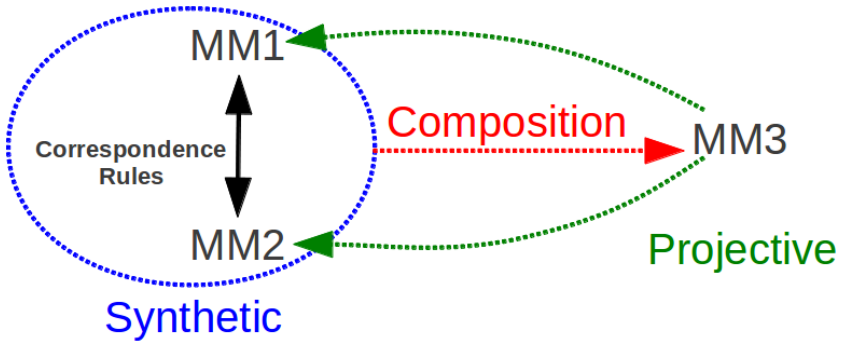


FIGURE 2.6: Relationship between modeling approaches and specific domains.

It is important to note that the multi-view approaches have as objective to maintain the independence between specific domains. Correspondence rules are the connections that these domains have. In contrast, the aim of the composition modeling approach is to generate a model (or meta-model) that contains the elements of the source models according to the correspondence rules. We could apply the composition approach in a multi-view model to generate analysis models from a selected number of views (projective or synthetic) to a specific purpose. These analysis models could study the impact of the modeled concerns from different views of a system. For instance, the impact of increasing the clock frequency in power consumption and time performance.

In the following items, we analyze some examples that are somehow associated with synthetic, projective and composition approaches:

- ***Aspect-Oriented Programming:*** In an object-oriented program, the non-functional and the cross-cutting concerns are interwoven in the code. Kiczales et al. [26] propose to extract these non-functional and cross-cutting concerns from the main concern of the program. These extracted concerns are known as *aspects*. The composition of aspects in the main code is called *weaving*. An aspect is composed by an *advice* and a *pointcut*. The former is the code of the concern that is *woven* in a specific place of the main code (*joint point*). The latter identifies the joint point where the aspect is added in the main code. An example of language that implements this kind of programming is AspectJ [27].

  This programming approach follows the model composition approach. The aim is to weave aspects into a base model to build a composed model. A set of aspects is

not a view of the model and does not specify specific domains such as the multi-view approach. All the models (aspects and base model) are specified using the same language, *i.e.,* the elements of a model (aspects), conform to a meta-model, are injected (woven) to another model that conforms to the same meta-model. The *joint points* are correspondences between the aspects and the target model.

- ***Kompose:*** Kompose [28] is a generic model composition tool that merges models conforming to the same meta-model. The merging process is defined by two main steps: *matching* and *merging*. *Matching* identifies the elements that have the same concepts in the models that are to be composed. *Merging* generates a model that is the result of merging the matched elements. The elements that are not matched, are defined in the resulting model without any changes.

  Kompose follows the model composition approach. *Matching* process identifies the correspondences between the elements of the models to be composed. According to Clavreul, the Kompose correspondences are rule-based and their interpretation is overlapping, *i.e.,* the elements that fulfill the defined composition rules are merged adding the non-common attributes and relations of each element. These composition rules are defined by a pattern between the elements of the models to compose. This pattern is generally found in the equivalence of the semantics and the structure of the elements to merge.

- ***VUML:*** View-based UML (VUML) [29] is a UML profile that uses the multi-view modeling to provide limited access to the system actors[1] through views. The VUML author points out that the given IEEE-1471 [1] recommendations to build system architectures are specified in a general way, and it does not propose the use of a language to be implemented. VUML is a language inspired by the IEEE-1471 concepts to model system architectures. VUML employs a base class diagram of the system to extract the actors' views according to the actor's access rights. The view defines the system elements (classes, attributes and methods) that the user can access in the system.

  VUML defines a common stereotype called *DefaultView*. This class owns the elements that are shared between the system actors. Other views are specified according to the actor's access rights. Theses classes are stereotyped by *View* and they

[1] VUML considers an actor as a logical or physical entity that interacts with the system at run-time.

contain the elements only related to the actor's profile. *Views* and *DefaultView* are associated by UML dependency associations stereotyped by *view-extension*. This association allows accessing to the information shared among actors. VUML also defines relationships among *Views* to guarantee the correct updating of information among the views that share system elements. This relationship is represented by a dependency association stereotyped by *view-dependency*. The attributes dependency between views is constrained by OCL[2] expressions.

VUML follows the projective approach. From a base meta-model, the *viewPoints* are extracted according to the user's profile. We identify that *view-extension* and *view-dependency* are *correspondence rules* between *viewPoints*. According to the Clavreul's correspondence types, both VUML correspondence rules are model-based, they are defined in the language specification. We also identify that the correspondence interpretation is overlapping: each view contains part of the features of the reference model and these features can be shared among views, *i.e.,* a feature of the reference model can be included in one or more views.

- ***SysML:*** System Modeling Language (SysML) [4] is an OMG[3] specification that specifies a UML profile for systems engineering domain. Some of the elements of this standard represents the main IEEE-1471 standard concepts to define a multi-view approach. SysML uses packages to represent *views*, classes to describe *viewpoints*, and *conform* associations to specify relationships between *views* and *viewpoints*. This conform relationship is represented by a UML *dependency* association.

  The SysML viewpoint contains two properties: *stakeholders* and *concerns*. These properties are defined by strings. Therefore, the stakeholders and concerns shared among viewpoints must be rewritten in each viewpoint without guaranteeing the conformance among viewpoints.

  The SysML *View* limits the package elements to comments, constraint elements, package import and element import; therefore, the view elements must be defined in a common model to be imported and constrained according to the view. SysML also specifies that a view must follow the methods and languages defined in the associated viewpoints. However, SysML does not define a verification policy for the

[2]The Object Constraint Language (OCL) is a language defined by the OMG to constrain UML models.

[3]Object Management Group

concerned viewpoint properties. Moreover, *methods* and *languages* are represented as strings in *Viewpoint*, making the verification task more difficult.

SysML implements a projective approach where each view is built by the element models imported from the main model. However, there are not explicit correspondences between views. Moreover, a *viewpoint* does not have the same meaning as in IEEE-42010 or IEEE-1471, but rather it is interpreted as the viewpoint features that a view must answer. SysML viewpoint does not define the language used to express views. The *conform* association is not a correspondence according to the way we interpret the IEEE-42010. This association represents that the view elements conform to the concerns defined by stakeholders from their point of view and it is not a relationship between model elements from different views.

- ***Obeo Designer:*** Obeo Designer is a system design tool developed by Obeo[4]. This tool not only allows system modeling through graphical modeling standard languages such as UML and SysML, but it also provides a graphical environment to build DSMLs in Ecore. Obeo Designer includes *viewpoints* that are a specific representation of the concepts from one or more meta-models. These representations can be predefined (tables, trees, diagrams) or they can be customized by the system designer[5].

  We consider that Obeo's *Viewpoint* concept does not follow any of the multi-view approaches. An Obeo's *viewpoint* is a representation of a model, but it does not define a portion of the model (projective approach) or an independent model (synthetic approach).

- ***Hybrid multi-view modeling:*** Cicchetti et al. [30] present a multi-view modeling approach that is both projective and synthetic. They define a base meta-model to represent every possible concept of a specific system following the projective approach. However, the architect can build *viewPoints* in various meta-models following the synthetic approach. The connection between both approaches is in the base meta-model used to create the *viewPoints*. *ViewPoints* are defined according to the base meta-model, therefore the concepts and associations specified in the *viewPoint* must also be specified in the base meta-model.

[4] http://www.obeo.fr/pages/obeo-designer
[5] http://www.obeo.fr/resources/WhitePaper_ObeoDesigner.pdf

A base model and *view* models are built and they conform to their corresponding meta-models (base meta-model and *viewPoints*). The base model is the synchronization reference to the other *view* models, *i.e.,* if a *view* model is changed, the modifications are propagated initially to the base model and then to the other *view* models. This synchronization mechanism is implemented according to the difference between the base meta-model and the *viewPoints*.

This hybrid multi-view modeling approach solves the consistency problem present in the synthetic approach by having a common reference between the defined views. However, we consider that the duplication of information between the view models and the base model is a drawback since it requires some effort to maintain consistency.

In this modeling approach, the correspondences are explicitly defined in the base meta-model. According to Clavreul's classification, the correspondences specified in Cicchetti's approach are model-based, *i.e.,* every relationship between *viewPoints* is defined in the base meta-model. Nevertheless, we find that there is also a *delta representation-based* correspondence in the synchronization between views and the base model when there is a change of information in a view model.

- ***Heterogeneous points of view with ModHel'X:*** Boulanger et al. [31] present a synthetic approach, defining independent views of a system model in ModHel'X blocks. Each block represents an observable behavior of a system. In the context of multi-view modeling, a block specifies the behavior of a system from a specific point of view. For instance, a system could have a functional behavior, a power consumption behavior or a temperature behavior. In this work, the correspondences are represented by the behavioral relationships among views, *i.e.,* using the ModHel'X relations, we define the view connections and the way that the view behaviors are synchronized.

  This approach proposes to use a single language (defined in ModHel'X) to express the multi-view representation of a system (*viewPoints* and *correspondence rules*). However, there is neither a notion of view nor correspondence in this language. Views and correspondences are interpretations of a ModHel'X concept using blocks (views) and relations (correspondences).

The type of correspondences are model-based, they are defined in the ModHel'X meta-model. We consider that their interpretation is associated with the behavior of the model. In Section 2.3.1, we present it in details.

2.2.3. Discussion

All multi-view approaches have advantages and disadvantages. The projective approach allows observing a system model from different perspectives or *viewPoints* focused on the elements and properties that are important for the stakeholders. However, maintaining and extending a unique meta-model to describe every possible view in a system is a difficult task. For instance, in VUML, when a new *viewPoint* is added to the system meta-model, it can affect the previously defined *viewPoints* and also their associated information. One possible solution is to define consistency mechanisms to preserve the system model information once a new *viewPoint* is added. This kind of mechanism is developed in the Cichetti's work.

The synthetic approach has the advantage of defining independent *viewPoints* of a system splitting the system concerns. This *viewPoint* independence allows the definition of new *viewPoints* without altering the previous ones. However, the main challenge is the definition of *correspondence rules* between *viewPoints*. Unlike the projective approach, where the correspondence rules are explicitly defined in the reference meta-model, in the synthetic approach such correspondence rules are not explicit and they must be established once a *viewPoint* is specified. The domain experts define the relationships between the concepts of the *viewPoint* concepts.

Model composition could be seen as a way to unify projective and synthetic approaches. For instance, when having a multi-view model that follows a synthetic approach, the correspondences among views could be used to generate composed models that have as main goal the analysis of certain properties of the modeled system and the quantification of the impact of the properties from different points of view. In contrast, a composed model (or meta-model) could represent a reference model (or meta-model) in the multi-view projective approach. Using decomposition rules, *viewpoints* could be extracted or projected from the reference meta-model and correspondence rules could be identified in the reference meta-model to be explicitly defined in the decomposition process.

The correspondences and interpretations defined by Clavreul cannot be applied only to model composition. We identify that the Clavreul's correspondences meaning could also be applied to the correspondence rule definition in the multi-view approach. We note that correspondence rules among structural elements of different *viewPoints* are used to maintain the consistency between *viewPoints*, *i.e.,* these structural elements could represent a single element, but from a different point of view. We call these kinds of correspondence rules *syntactic correspondences*. In the multi-view modeling examples, we have identified some syntactic correspondences, such as VUML, SysML, Obeo Designer and Ciccheti's work. However, another kind of correspondences could be applied, *i.e.,* behavioral correspondence rules among *viewPoints*. This sort of correspondence rules was identified in Boulanger's work and is further discussed in Section 2.3.1.

Most of the works that apply the multi-view approaches are oriented to the design of software systems. Nevertheless, we consider that such approaches can be also applied to the system design. In this thesis, we propose a multi-view model for system design. The definition of this multi-view model gathers the advantages of both multi-view approaches: the definition of explicit correspondence rules to maintain the model consistency and the definition of independent *viewPoints* for each expert domain. We also use the Clavreul's terms to identify the correspondence rules among *viewPoints*.

Another important feature to analyze in this chapter is the behavior in a multi-view modeling approach. Identifying the behavioral relationships between *viewPoints* and placing them in a modeling behavior context. Section 2.3 presents the description of the behavioral concerns in the design of systems.

2.3. Behavioral Concerns

In multi-view modeling, each *viewPoint* is described by a language with a specific semantics of execution. In a DSML, while the syntactic domain is represented by a metamodel, the semantic domain is defined though different approaches. In the language theory, we can find three types of semantic definitions. The first type is *Operational Semantics* [32]. It uses functions (endogenous transformations) to manipulate data that represent the execution state of the model. Each execution of these functions represents a step in the model evolution. The second type is *Axiomatic semantics* [33]. It

characterizes the execution state by properties that enable reasoning about the models and their correct evolution. The last type is *Transformational semantics* [34]. It is an exogenous transformation from the syntactic domain to an existing language with well defined semantics.

The concurrent theory has also proposed other ways to describe the behavior of a model. This behavior is characterized by the so-called Models of Computation (MoCs).

2.3.1. Models of Computation

A model of computation (MoC) is *"a formal abstraction of execution in a computer"* [35]. In other words, it defines the behavioral semantics of a model. MoCs are used in different specific domains to express and to evaluate the behavior of a system. For instance, the control experts uses ordinary differential equation (ODE) solvers to analyze the behavior of the system to be controlled in continuous time. However, these solvers discretize the continuous time in order to be computed. The specification that defines the execution rules of these continuous systems in the computing world is a type of MoC. Modelica [36] and Simulink [37] are tools that implement MoCs that allow to model continuous systems and they are often used by control and mechanic experts to represent and to analyze their specific domains.

Ptolemy II [9] and ModHel'X [10] are tools that implement a variety of MoCs. Using these tools, sequential processes, discrete event and continuous time systems can be modeled. These tools share the way they define their modeling syntax, based on the component approach. While Ptolemy II uses *actors*, ModHel'X uses *blocks* to describe the structure of the system behavior. However, this generic use of the component-based modeling restricts the application of the DSML approach. Moreover, if we consider that a *viewPoint* is a DSML in a multi-view approach, the behavioral semantics of the *viewPoint* could be hardly specified using these tools because of the incompatibility of the structure definition.

On the other hand, we note that MoCs in these tools are independent from the structure definition. Ptolemy II represents the MoCs implementation by *directors* and ModHel'X calls them with the same name, MoCs. They associate a specific MoC to a determined structure and this MoC manages the execution of the structure elements. The separation

between semantics and syntax helps to use the MoC definition to specify the DSML semantics. For instance, Petri net is a modeling language that represents the control execution of a system. A Petri net syntax could be defined by a meta-model. Figure 2.7 presents the Petri net meta-model (left-side) and a Petri net instance (right-side) that follows the concepts and relationships defined in the meta-model. To define the execution of this meta-model, we can use a formal language in order to specify the rules that the behavior of the Petri net model must follow. Nevertheless, the mentioned tools implement these rules in programming language such as Java, creating a gap between the formal definitions and their implementation. In this thesis, we propose to use CCSL [3] as a formal language to specify the rules that the DSML must fulfill during its execution. Using CCSL, the mentioned gap could be reduced, thanks to the proximity of the formal semantics and its implementation.

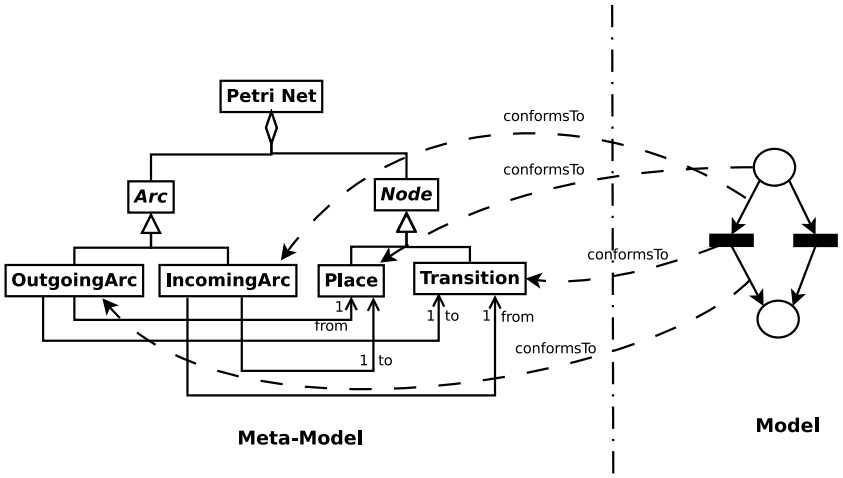


FIGURE 2.7: Petri Net meta-model and a Petri Net model example.

We have explained in Section 2.2 that a system can be represented by various *viewPoints*. These *viewPoints* are associated with each other in their structural definition by *syntactic correspondences*. However, these *viewPoints* also have a semantic definition, whose actions can affect the behavior of other *viewPoints*. For this reason, there are also correspondence rules in the semantic definition of the views.

Clavreul [25] has already identified a correspondence interpretation to describe the execution relationship between models. This interpretation is called *interaction*. It consists in describing the execution ordering of the model elements according to their associations and to control elements, *e.g.*, sequence and parallel execution. Clavreul also defines

two design activities that are associated with the interactions between models, in order to define a composed model behavior. The first activity is *Orchestration* that synchronizes the service execution of two or more models to create a fully running process. The second activity is *Integration* that produces a composed system from the interaction of several independent and running systems. We consider that these activities are strongly associated with the correspondence rules between the behavioral semantics among DSMLs, *i.e.,* we could identify a behavioral impact among DSMLs by using *behavioral correspondences*.

In the multi-view approach, the behavioral correspondences among *viewPoints* are the combination of homogeneous or heterogeneous behavioral semantics. This combination is known in the MoC community as *heterogeneous models*.

2.3.2. Heterogeneous Models

There are different approaches that propose a way to combine heterogeneous MoCs. Ptolemy II and ModHel'X specify the combination of MoCs by using a hierarchical execution. Figure 2.8 depicts a model example where the semantics of execution is a hierarchical MoC combination in Ptolemy II. In this figure, there are two MoCs: Synchronous Data Flow (SDF) and Finite State Machine (FSM). The structure of the model contains four actors: a main composite actor that owns two atomic actors[6] (*A1* and *A2*) and a composite actor (*C1*). The composite actor *C1* contains a FSM that has two atomic actors (S1 and S2). The main composite actor specifies its behavioral semantics by a SDF director. In contrast, *C1* has a FSM director. The domain execution ordering is controlled by the director at the highest level in the model hierarchy, *i.e.,* SDF director. During the execution sequence in the SDF graph, the SDF director executes C1 and then the FSM director is activated to execute the FSM. Once the execution of the FSM finishes, SDF director resumes its execution.

[6] An atomic actors is an actor that does not contain other actors.

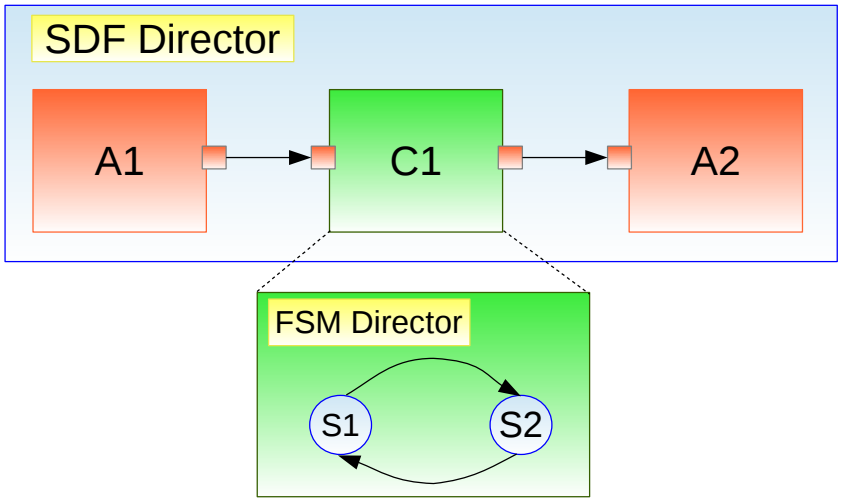


FIGURE 2.8: Composition between Synchronous Data Flow and Finite State Machine in Ptolemy II.

In Figure 2.8, there is a behavioral correspondence between SDF and FSM directors. Once the SDF director executes C1, the FSM director takes the external information to execute the FSM. According to Clavreul, we could consider that this correspondence is an *Orchestration* between two MoC directors. The orchestration between MoCs is implemented in a different way in Ptolemy II and ModHel'X. On one hand, Ptolemy II offers a fixed and encoded interaction semantics between MoCs that the modeler must use. On the other hand, ModHel'X proposes the use of *adapters* to define the semantics between the internal and external execution of a hierarchical model. However, *adapters* are operators that implement the MoC interaction according to the modeler needs. Therefore, there is not guarantee that properties defined in each MoC, such as deadlock or safety properties, are kept after the orchestration of MoCs.

Another approach to combine heterogeneous MoCs is by synchronizing the actions between MoCs. BIP [38] is a component-based language that defines the behavior of each component and their interactions by a specific algebra. The BIP semantics is described by extending the automaton definition. In the BIP approach, the use of the automaton model to define the component interaction allows to study properties, such as deadlock and safety issues. However, the dependency to the automaton model does not allow to describe MoCs that follow other kinds of behavior such as flow-oriented behavior. This behavior is commonly used to define and analyze image processing algorithms.

2.3.3. Discussion

MoCs are a way to define the behavioral semantics of a DSML. A DSML could contain other DSMLs that have their own behavioral semantics, or a DSML could specify their semantics by using various behavioral semantics. For instance, Figure 2.8 could be represented by two DSMLs: $DSML_1$ that defines the first hierarchy level (A1, C1 and A2) and $DSML_2$ that specifies the internal behavior of C1. Both DSMLs have a *syntactic correspondence* that associates the $DSML_1$ element C1 with $DSML_2$. This correspondence represents that the internal behavior of C1 is expressed by $DSML_2$. $DSML_1$ and $DSML_2$ have also a *behavioral correspondence* where the synchronization between SDF and FSM execution is defined. Following the Ptolemy II and ModHel'X approach, we can represent the example of Figure 2.8 by using a single DSML definition (actor-based or block-based representation). In these tools, the behavioral correspondence is defined to a specific element of the DSML, *i.e.,* the DSML can have a different meaning according to the MoC assigned to the model element. We consider that it is more clear to have a DSML with a single meaning, *e.g.,* a Petri Net structure whose behavior follows the Petri Net rules.

In the multi-view approach, each *viewPoint* is a DSML, and each DSML has its own behavior definition specified by a MoC. As *syntactic correspondence*, we identify that there are also other kinds of correspondences between views that we call *semantic correspondences*. These correspondences define the interactions between the elements of different views, *i.e.,* the result of the interaction specification between MoCs. The interactions between views highlight the impact of the view execution on a system design that would be difficult to grasp using only syntactic correspondences.

In this thesis, we use syntactic and semantic correspondences to define the multi-view modeling of systems. We give specific examples where both correspondences are used to maintain the structure consistency among views, the synchronization of the view execution and the impact of the view execution.

2.4. Conclusion

In this chapter, we have presented a background of the pivotal concepts used in the following chapters. We have introduced the architecture concept visualized in the system domain. Afterwards, we have presented the multi-view modeling vocabulary specified in the IEEE-42010 standard and its relationship with MDE. We have noted that a *viewPoint* is a DSML in the MDE context. We have identified the connection between the multi-view approaches and model composition. We have determined that the model composition work could be used in the multi-view approach to characterize the correspondence rules and their interpretations. We have presented some works that implement these approaches (multi-view and model composition) and we have identified the correspondences and their interpretations according to Clavreul's work.

We have continued with the behavioral definition in the multi-view approach. The importance to separate semantics and syntax in the definition of a *viewPoint* has been highlighted. MoCs are adopted as the modeling approach to specify the semantic domain in a *viewPoint*. We stressed the importance of behavioral correspondences in addition to purely structural correspondences in the multi-view modeling. Such behavioral correspondences are bound to the heterogeneous behavior associated with MoC interactions. We have presented two approaches (hierarchy and automaton based) frequently used to specify the interactions between MoCs.

In the next chapter, we use the concepts from this chapter to define a multi-view framework to model systems.

Chapter 3

PRISMSYS: A Multi-View Modeling Language for Specifying Systems

Contents

3.1. Introduction **32**

3.2. PRISMSYS Framework **33**

3.2.1. Structural SubView 40

3.2.2. SubView Element 41

3.2.3. Equational SubView 43

3.2.4. Control SubView 46

3.3. UML Profile for PRISMSYS **49**

3.3.1. UML Concepts for PRISMSYS 50

3.3.2. MARTE Concepts for PRISMSYS 54

3.3.3. SysML Concepts for PRISMSYS 55

3.4. Semantics of Execution **56**

3.4.1. Finite State Machine Semantic Specification 58

3.4.2. Equational View Semantic Specification 64

3.5. Conclusion **69**

3.1. Introduction

This chapter presents the definition of our language named *PRISMSYS*[1]. *PRISMSYS* is a domain specific modeling language (DSML) dedicated to the specification and analysis of functional and non-functional properties at the system level through multiple *views*. Each view describes a part of the system, by using the language commonly employed by domain experts focusing on a specific concern. For instance, a safety expert uses a domain language whose concepts describe a safety infrastructure, at the same time as it presents the safety properties of the system. The system views are independently specified, but the existing relationships inside each view are extremely important to maintain the consistency of the system. In a multi-view model, these relationships are correspondences among views. They should bring semantic consistency between the different parts of the system specified in the views.

The multi-view concepts of *PRISMSYS* are inspired by the notions defined in IEEE-42010. However, the standard is a general framework, therefore we have had to specialize in *PRISMSYS* the concepts defined in IEEE-42010. Our specialization aims at identifying concepts needed to have a semantic consistency between the different views. For instance, the abstract concept of *View* from the IEEE specification is refined into three well-identified *subViews* in *PRISMSYS*, each of them representing sub-concerns of a domain-specific language. This specialization helps us to provide a semantics to the correspondences depending on the kind of elements they refer to.

MDE is largely used to define the *PRISMSYS* domain language. The abstract syntax of *PRISMSYS* is specified as meta-models in Ecore [20], while the behavioral event-based semantics is defined in CCSL [3]. CCSL is a formal declarative language used to define causal and temporal constraints between events. An event represents a specific evolution of a system, such as the sampling of a robot position or a state change in a finite state machine. Events are spread along all the views to bring consistency through the model. Similarly to tagged signals [39] they serve as anchor points to specify the model of computation (MoC) [40] of the system model. We introduce in *PRISMSYS* specific correspondences as a predefined way to coordinate the execution of two MoCs.

[1]PRISMSYS is a composed name where *PRISM* refers to prism, which is a transparent optical element that refracts any composite light producing a variety of colors. We identify the prism behavior as an analogy to define our multi-view approach. *SYS* denotes *system*.

We begin this chapter by defining the *PRISMSYS* framework. This framework specifies the basic elements needed to represent views that capture the different concerns of a system. We continue the chapter by describing the correspondences that can be applied between the views to tight them together; we detail each *PRISMSYS subView* definition, we present its uses and we give some examples to illustrate the use of the *subViews* and the identified correspondences. Taking as reference the *PRISMSYS* domain model, *i.e.,* the meta-model of the *PRISMSYS* framework and the detailed description of each one of its views, we have built a UML profile as a light-weight mechanism to implement the PRISMSYS concepts. The *PRISMSYS* profile applies, as much as possible, the elements defined in SysML and MARTE, including UML elements as well. Finally, we define the semantics of the *PRISMSYS* framework execution by using CCSL to express the actions presented in the behavior evolution of a *PRISMSYS* model.

3.2. PRISMSYS Framework

The *PRISMSYS* framework provides predefined rules and elements that can describe and coordinate different views in the specification of a multi-view system. More precisely, based on a system backbone representation, it allows defining specific views that are focused on the management of its non-functional properties. By applying this framework, experts from various domains (time performance, power, finance, etc.) can build a system from their own point of view while specifying explicitly the relationships with the other points of view. For instance, a time performance expert can specify temporal constraints by using the concepts frequently used in his/her domain (deadline, worst case execution time, etc.). However, domain experts do not specify again the elements already defined in other domains on which they state their constraints (like the hardware or software elements). They just import them and provide an abstraction of existing elements from their point of view.

We use MDE to define the syntax of the *PRISMSYS* framework. Figure 3.1 depicts the *PRISMSYS* framework meta-model. The root element is *ArchitectureDescription*. IEEE-42010 defines *architecture description* as the base concept to specify the architecture of a system through views. To re-use an architecture description in various system designs, IEEE-42010 defines the *architecture framework* concept that governs

the construction of architecture descriptions. IEEE-42010 has needed the definition of these two separated concepts in order to describe the abstraction levels in its multi-view system framework. However, these two concepts are not needed if we use MDE. MDE establishes the needed abstraction levels to specify the vocabulary to express a specific domain (*i.e.,* a meta-model), and the way to use it (*i.e.,* a model conforming to its meta-model). As a consequence, if we define *ArchitectureDescription* as a meta-class in the *PRISMSYS* framework meta-model, it represents the *architecture framework* concept defined in IEEE-42010. Similar reasoning can be made with *view-viewpoints*, *correspondence-correspondence rules* and *model-model kind*. We decide to employ the IEEE-42010 terms that define an architecture description to specify the concepts of the *PRISMSYS* framework meta-model, *i.e., view*, *model* and *correspondence*.

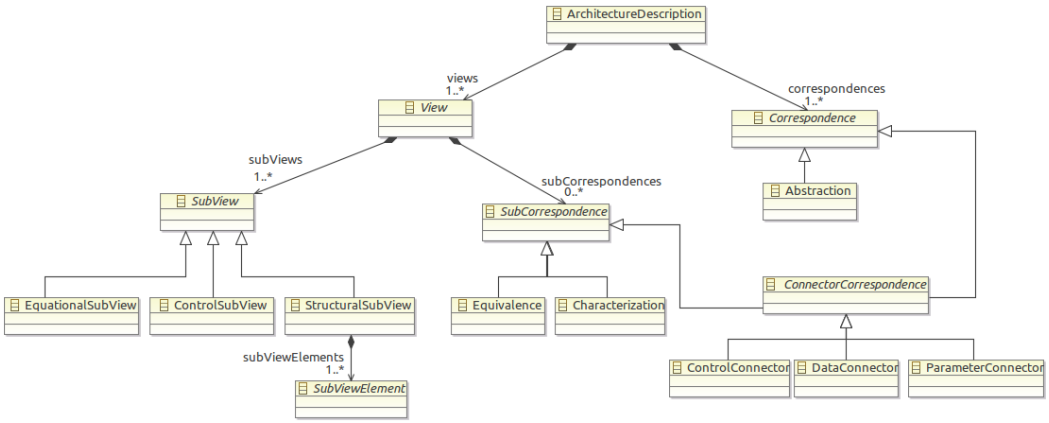


FIGURE 3.1: *PRISMSYS Framework* meta-model.

In *PRISMSYS* meta-model, an *ArchitectureDescription* is a set of *views* and *correspondences*. A *view* defines the needed elements to describe a specific domain. According to IEEE-42010, a *view* is composed of one or more *models*. The standard defines a *model* as "*modeling conventions appropriate to the concerns to be addressed*" [2]. With this very abstract vision of what is comprised in a view, it is not straightforward to guarantee the semantic consistency of a multi-view system model. To ease the automated management of a multi-view system model, the *PRISMSYS* framework proposes to specify systematically three *models* used for the description of each view.

In this context, a domain specific language for a multi-view system model (*i.e.,* a view) is specified by *models* of different nature. Such *models* have their own features that describe view parts. Indeed, these parts are sub-domains needed to specify a complete

view. We name them *subViews*. We have identified three main *subViews* that provide the required elements to define a *view*: a *structuralSubView*, an *equationalSubView* and a *controlSubView*. *StructuralSubView* states the concepts and relations of a specific domain with a component-based approach. A *StructuralSubView* is composed of *subViewElements*. Such elements are the internal concepts that express the structure of a specific domain. A *ControlSubView* controls/schedules the execution of the *subViewElements*. Finally, *EquationalSubView* characterizes the evolution of non-functional properties of a *StructuralSubView*, such as frequency, voltage and temperature, by using mathematical equations.

For each system, there is always a *reference* or *backbone* view. Relying on the backbone view, the other views can "import" existing elements to define the (non-functional) properties of the specific domain. For instance, considering a thermal domain example, the thermal view definition depends on the elements included in the hardware architecture view, *i.e.,* thermal experts reference elements from another view to build their own view. The "importing" action is identified as a *correspondence* between views.

In the *PRISMSYS* framework meta-model, *Correspondence* is an abstract concept specialized into a type of relationship named *Abstraction*. An *abstraction* specifies that the source *subViewElement* is a representation of the target *subViewElement* between two *structuralViews* of different *views*, *i.e.,* a structural element defined in a view is used in another view to specify features that belong to this particular view. This correspondence plays the role of "importing" a *subViewElement* from a view to another. For instance, a memory component defined in a *structuralSubView* of a hardware architecture could be abstracted in a *structuralSubView* of a time performance view. This abstraction allows the definition of temporal features, such as maximum time of writing and reading data. Figure 3.2 depicts the relationship between the *Abstraction* correspondence and *ViewElement*. To express this relationship, we define two abstract concepts: *AssociationElement* and *AssociationEnd*. Such abstract concepts are associated by an oriented relationship (*source* and *target*). As *Abstraction* inherits from *AssociationElement* and *ViewElement* from *AssociationEnd*, therefore *Abstraction* links two *viewElements* in an oriented way.

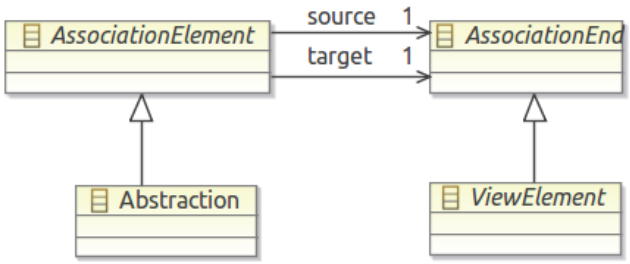


FIGURE 3.2: Relationship between *Abstraction* correspondence and *viewElement*.

Just as *subViews* are sub-elements of *View*, *subCorrespondences* are relationships that maintain the consistency between *subViews*. Moreover, *SubViews* must be linked together in order to fully describe a *view*. For instance, the relationship between a structural element and an equational description is different to the relationship between a hardware component and the hardware component representation in a time performance view. While the first relationship is a *subCorrespondence* that associates a structural sub-view element with an equational sub-view element, the second relationship is a *correspondence* between two different expert domains, a hardware architectural view and its representation in a time performance view.

We have determined two main types of *subCorrespondences* in a *view*: *Equivalence* and *Characterization*. *Equivalence* is the equality of the value between a property defined in a *subViewElement* and a parameter in an equation specified in a *equationalSubView*. For instance, if the *level* property is defined in a *subViewElement* to quantify the water level of a tank; *level* could also be specified as parameter of an equation in an *equationalSubView* to calculate the output flow of the tank. *Level* is expressed in two different *subViews* and the consistency between these *subViews* is defined by the *Equivalence subCorrespondence*. *Characterization* is the association between the behavior of a *subViewElement* and an equation defined in the *EquationalView*. A change in the *subViewElement* behavior causes the change of the active equation designated by the *Characterization* relationship. For instance, the *subViewElement* behavior is described by a finite state machine (FSM). Each state is associated by a *Characterization subCorrespondence* with a specific equation in the *EquationalSubView*. Thus, when a state is active, the associated equation is activated. These two *subCorrespondence* are explained in details in Subsection 3.2.3.

View, *SubView* and *SubViewElement* follow the component approach. Such an approach is used by several domains in the design of systems. MARTE [5], a domain language for the design and analysis of real-time systems, defines the hardware structure following the component approach. Other examples are SysML [4], AADL [22], EAST-ADL [41] and Rosetta [42]. Moreover, The IEEE-1471 and IEEE-42010 standards, which are the inspiration source of *PRISMSYS*, have also based the architecture definition of a system on components. *View*, *SubView* and *SubViewElement* share different kinds of information that can be exposed through *ports* and transmitted through *connectors*. Figure 3.3 depicts a generic component meta-model and its relationship with the *PRISMSYS* framework concepts. *View*, *SubView* and *SubViewElement* inherit from *Component*, *i.e.,* they contain *ports*, *connectors* and owned components. The owned components of a *View* are *subViews*, and the internal components of *subViews* are *subViewElements*. *SubViewElements* can contain other *subViewElements*.

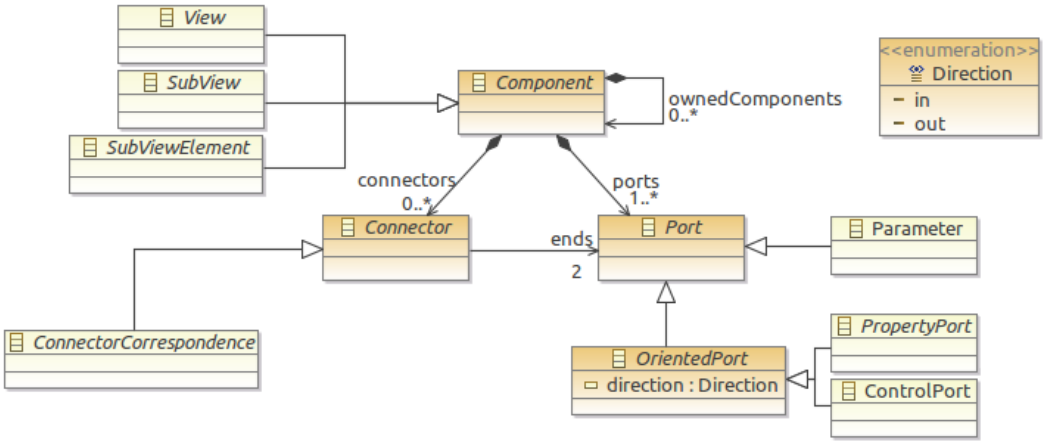


FIGURE 3.3: Component meta-model and its relationship with *View*, *SubView*, *SubViewElement* and *ConnectorCorrespondence*.

Port is an abstract concept that is specialized in *OrientedPort* and *Parameter*. An *orientedPort* has as an attribute *direction*. *Direction* could be either *in* or *out*, to express the direction of the information flow. *OrientedPort* is specialized in *PropertyPort* and *ControlPort*. *PropertyPort* represents a *subViewElement* property that is shared with its environment. Properties are shared with other *subViewElements* of the same *StructuralSubView*. Properties can also be used by the *controlSubView* to take decisions in the control of the *structuralSubView*. For instance, if a robot reaches the limit of its running area, the position value is transmitted to the corresponding *controlSubView* to stop the robot movement. *PropertyPort* is an abstract component that is specialized to express

the nature of the property according to the specific domain, *e.g., PositionPort* could be a *propertyPort* that shares the position property of a *subViewElement*. *ControlPort* defines the control flow between a *controlSubView* and a *structuralSubView*. This flow is specified by events that change the behavior of the *subViewElements*. *PropertyPort* and *ControlPort* can be defined by *views*, *subViewElements*, *structuralSubViews* and *controlSubViews*. To expose parameter values in a *equationalSubView*, we specify *Parameter*. This port does not have any direction. The value of the connected equation parameters is equal, *i.e.,* the available parameter value of an equation is replaced in the associated equations.

We consider that the flow of information between *views* and between *subViews* through ports is a kind of *correspondence* and *subCorrespondence*, respectively. Therefore, *ArchitecturalDescription* and *View* share an abstract concept named *ConnectorCorrespondence* in the *PRISMSYS* framework. This concept inherits from *Connector* and represents the flow of information between *subViews*, between *views* and possibly between *views* and *subViews* through *ports*. *ConnectorCorrespondence* is specialized into three different concepts: *ControlConnector*, *DataConnector* and *ParameterConnector*.

In the *View* context, *ControlConnector* is the connection between *controlPorts* of *ControlSubView* and *StructuralSubView*. This connector transmits the control messages sent from the *controlSubView* to the corresponding *subViewElements*. However, in the *ArchitectureDescription* context, *ControlSubView* coordinates control actions among *views*. Therefore, we constrain the use of *ControlConnector* between views only to connect *controlPorts* of *ControlSubViews*.

DataConnector represents the connection between two *propertyPorts*. Such *propertyPorts* must be defined either in *structuralSubViews*, in *controlSubViews* or in *views*. The connector between *propertyPorts* of *subViewElements* is specified according to the domain. The connected *propertyPorts* must have the same type, *e.g.,* if a *propertyPort* expresses the torque of an electric motor, the *propertyPort* that receives this information must have the same torque nature. *ControlConnector* and *DataConnector* must connect two ports whose directions are in in the same direction, *i.e.,* these connectors can only bind two output ports or two input ports.

ParameterConnector is the connection between two *parameter* ports. It represents the shared parameter value between two *equationalSubViews*. Figure 3.4 summarizes the correspondences and sub-correspondences identified in the *PRISMSYS Framework*.

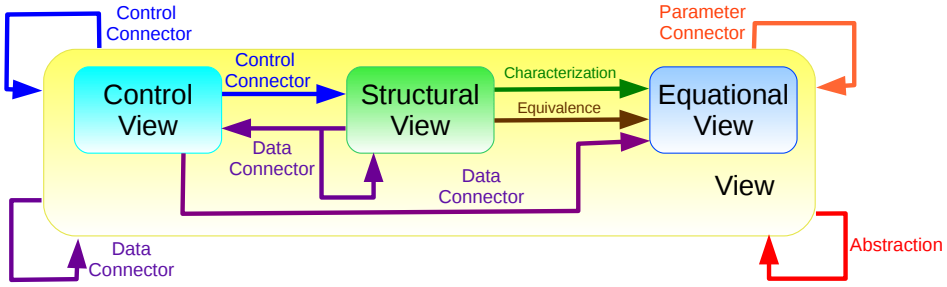


FIGURE 3.4: Correspondences and Sub-Correspondences in *PRISMSYS* Framework.

Correspondences and *sub-correspondences* are associated with the correspondences and the interpretations given by Clavreul [25]. A first identification is that the *PRISMSYS* correspondences and sub-correspondences are *model-based* correspondences. The *PRISMSYS* framework meta-model and the previous semantic description define the way they are employed. Nevertheless, their interpretations are diverse. *Abstraction* could have an *equivalence* interpretation, *i.e.,* the associated *subViewElements* are equivalent and in a merge process both *subViewElements* can be replaced by one *subViewElement* that has the properties of both merged *subViewElements*. *Equivalence* is another example of *equivalence* interpretation. In contrast, *Characterization* has an *interaction* interpretation. Once a *subViewElement* behavior changes the active equation, the new active equation must be evaluated. The same interpretation can be given to *Control, Data* and *Parameter Connectors*, once a *Parameter*, a *controlPort* or a *propertyPort* changes its value, the bound port also changes its value.

An *ArchitectureDescription* must contain at least one *view* that represents the functionality and structure of the system. If system experts add non-functional properties to the multi-view model, such as time, power or temperature, they add for each expert's domain a *view* and its corresponding *subViews* to represent their properties and the necessary elements that affect them. *PRISMSYS* can be extended with other kinds of *subViews* that do not follow the three sorts previously defined. Nevertheless, the designer must define the necessary *correspondences* and *subCorrespondences* of this new *subView* to keep the consistency of the multi-view model.

In the next subsections, we detail the definition of the *StructuralSubView*, *SubViewElement*, *EquationalSubView* and *ControlSubView*.

3.2.1. Structural SubView

StructuralSubView is a generic *subView* that can be specialized to represent expert domains. Adopting this *StructuralSubView* definition implies that, the structural representation of each view can be specified by domain experts and the relationship between views can also be expressed by using *abstraction*, *dataConnector* and *ParameterConnector* correspondences. Nevertheless, if a domain expert does not want to use *StructuralSubView* to represent his/her viewpoint of the system, this expert can specialize the *SubView* concept from the *PRISMSYS* meta-model to define his/her own *subView*, the *subCorrespondences* with the other *subViews* and the *correspondences* with other *views*.

An application of *StructuralSubView* is the representation of the thermal domain of an embedded system. One of the techniques used by thermal experts to represent the temperature evolution of the components is using electrical components, such as capacitors and resistances. The resulting Resistor-Capacitor circuit represents the temperature behavior among the junction points between the hardware components with the heat sink devices and the heat transmission among the components that are part of a system. This thermal representation of a system is known as Compact Thermal Model (CTM) [43]. Hotspot [44] is a tool that uses this modeling technique to represent the thermal layout of systems to analyze the temperature evolution of the components.

Figure 3.5 depicts an example of two views that define their *structuralSubViews*. *Execution Platform View* represents the hardware architecture of a system. *Thermal View* describes the thermal representation of the system. Each view has a *structuralSubView* where the structure of the domain is represented. We note that CPU is abstracted in the thermal view to specify the thermal properties and the thermal behavior that can be expressed using CTM. To define the association between the thermal representation and the hardware architectural representation of CPU, we use the *abstraction* correspondence. In the *structuralSubView* of the thermal view, there are also other elements that belong to the thermal domain. They are not included in the *structuralSubView* of the execution platform view, such as the heat sink and temperature source (T_{env}).

Finally, note that a *propertyPort* P is specified in the thermal view. This port represents the power consumption value of the CPU, used and evaluated in other views. The CPU power consumption value is needed to evaluate the CPU temperature. P port is connected by *DataConnector* correspondences to another view that characterizes the system power consumption.

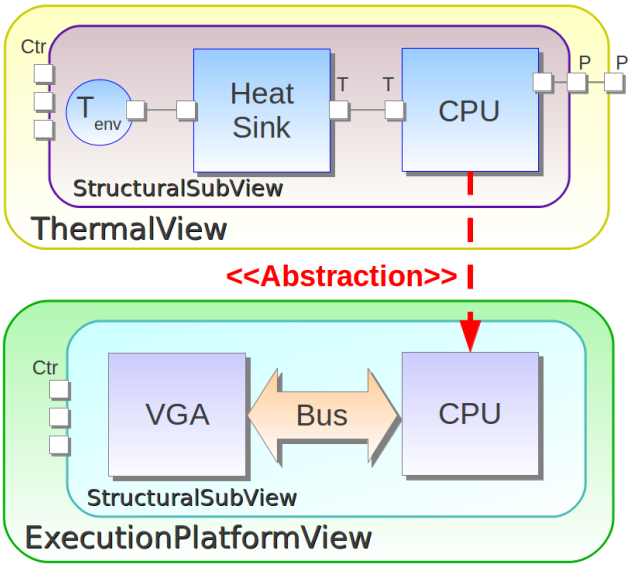


FIGURE 3.5: Example of *structuralSubViews* including the abstraction correspondence.

3.2.2. SubView Element

SubViewElement is the main concept of a *structuralSubView*. Such a concept has a specific role in the structural description of the concerning domain. *SubViewElement* defines the structure and the behavior of the *StructuralSubView* internal elements. Figure 3.6 presents the *SubViewElement* meta-model where the structure (on the right-hand side) and the behavior (on the left-hand side) of this concept are defined. *SubViewElement* follows the component approach, therefore we bring the component meta-model depicted in Figure 3.3 to define the *SubViewElement* structure. A *subViewElement* is a *Component* that contains *connectors*, *controlPorts*, *propertyPorts*, *properties* and possibly nested *subViewElements* (*ownedComponents*). *Property* represents an internal feature of *ViewElement*, *e.g.,* cost or size. *ControlPort* is sensitive to *Event* occurrences from the *controlSubView* that change the *subViewElement* behavior accordingly. Every *subViewElement* able to change its internal behavior must contain at least one *controlPort*. Note that *Property* and *State* are respectively associated with *Parameter* and

Equation, which are *EquationalSubView* concepts. The association is defined through *Equivalence* and *Characterization subCorrespondences*. We explain in details their use in Subsection 3.2.3.

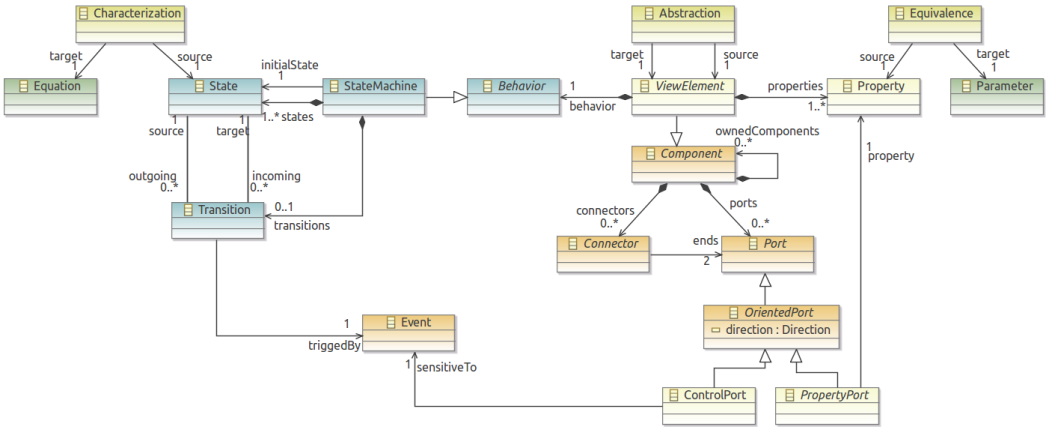


FIGURE 3.6: *SubViewElement* meta-model.

The behavioral definition of *SubViewElement* consists of a *Behavior* represented by a *StateMachine*. The behavior can be specialized in other kinds of behavioral descriptions such as Petri nets and synchronous data flow graphs, even though we only study here the case of *StateMachine*. According to the domain, the expert chooses which behavior definition fits better the domain description. For instance, a control expert may prefer to use state machines to describe the behavior of a thermal controller, whereas an image processing expert may choose a synchronous data flow graph to specify the face detection algorithm in a video stream. However, we consider that this definition must be homogeneous in all the domain specifications, *i.e.,* if *StateMachine* is chosen as a *subViewElement* behavior definition, every *subViewElement* in the specified *StructuralSubView* must be a *stateMachine*. This homogeneity helps to work with a single semantics of execution, easing the control specification defined in the *controlSubView*. Dedicated tools for heterogeneous composition might be used (see Chapter 2), however, this is not specifically supported by our methodology and tools at this level.

In the *SubViewElement* meta-model, a *StateMachine* contains *states* and *transitions*. The *StateMachine* has an *initialState*, which is the first state that is active when the *StateMachine* is executed. Each *state* represents a specific behavior mode according to the domain. For instance, to indicate the execution modes of a CPU, we can define two states: *running*, to express that the CPU is executing a task, and *halt*, when the CPU

stops. In Figure 3.6, *State* is associated with *Equation* through the *Characterization subCorrespondence*. This *subCorrespondence* means that when a *state* in a *viewElement* is active, the associated equation is activated, *i.e.,* the state is characterized by the associated equation. A state also represents the value change of a property defined in the *subViewElement*, which is specified by the associated equation. The *Equation* concept is part of the *EquationalSubView* definition detailed in Section 3.2.3. To change from one state to another, the corresponding transition is fired when the associated *Event* (see association *Transition-Event* in Figure 3.6) occurs on the *ViewElement controlPort* (see association *ControlPort-Event* in Figure 3.6). The execution semantics of the state machine is detailed in Section 3.4.1.

3.2.3. Equational SubView

EquationalSubView defines the evolution of non-functional properties of a view. This evolution is specified by equations that associate properties from a view with properties from other views in an acausal way. For instance, in classical mechanics, the equation that describes the force applied to an object in one dimension is represented by $F = m \bullet a$. The parameters of this equation are defined as properties, possibly, in different views. F could be defined in a force view where only force features such as torque, thrust, or drag can be described. In contrast, m could be specified in an object characteristic view, where mass, dimension and color features are represented.

We consider that the *EquationalSubView* meta-model is independent of the *StructuralSubView* and the *ControlSubView* meta-model, because the nature of the *EquationalSubView* elements is different from the elements of the *StructuralSubView* and *ControlSubView*. Such elements represent continuous behaviors through equations, while the *StructuralSubView* and the *ControlSubView* elements specify discrete behaviors. However, they share *Component* to define their concepts and the sub-correspondences with the other *subViews*.

Figure 3.7 presents the *EquationalSubView* meta-model. This meta-model is inspired on the SysML Parametric Diagram. An *equationalSubView* is a *subView*, *i.e.,* it is defined as a component. An *equationalSubView* contains *parameters* and a *clockPort* (*Port*

specializations), *bindings* (*Connector* specialization) and *equationalModels* (*ownedComponent* specialization). An *equationalModel* owns *equations* and its *Component* specialization is constrained to be associated with *parameters*. *Equation* is an acausal relationship among *parameters*. This relationship is given by the definition in form of a mathematical relation between parameters, *e.g.*, $v = d/t$ is an equation definition, where v, d and t are parameters. A single parameter value can be employed in various equations using *bindings*. *Binding* connects the parameters that share their values between two *equationalModels*. The *ClockPort* is employed to receive the events that execute the evaluation of equations. Every *equationalModel* have a parameter t to express the time dependence in the evolution of the non-functional properties. In fact, we only consider the case that the equations are time-dependent. It does not mean that equations of the *equationalModels* must include t as part of its definition. To transmit the events from *ClockPort* to the t parameters, we use *bindings*.

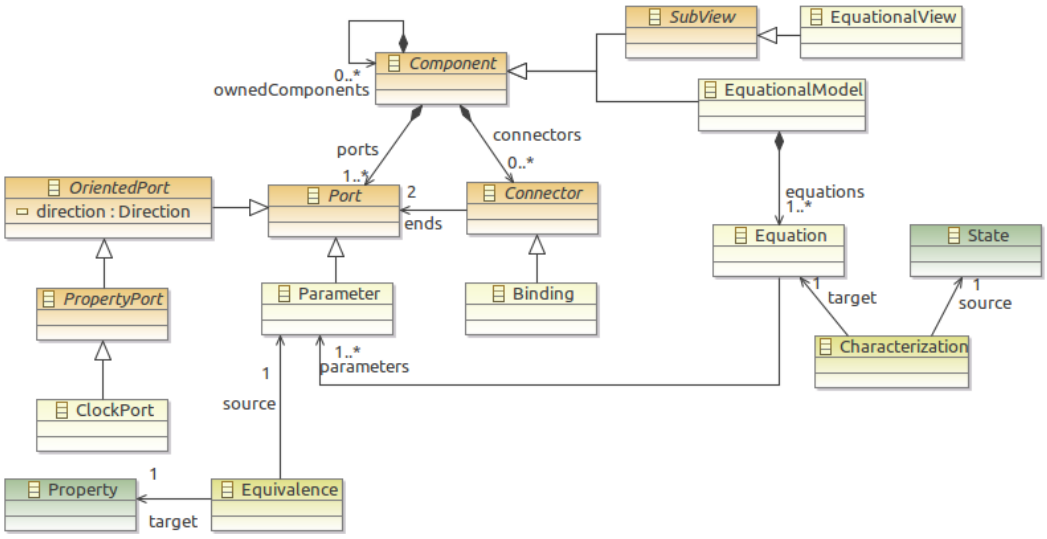


FIGURE 3.7: *EquationalSubView* meta-model.

Figure 3.8 presents an example of two views where their *equationalSubViews* are defined. In the figure, *Force View* describes its *equationalSubView* with two *equationalModels*: one defines a constant mass ($m = 1\,kg$) and the other one the force ($F = a \bullet m$). *Movement View* contains three *equationalModels* describing the acceleration ($a = dv/dt$) and the speed ($v = dx/dt$). In the same view, x is used to evaluate the speed, even though it is given by another view. Note that each *equationalModel* that defines a non-constant value equation (*e.g.*, $a = dv/dt$) contains a t parameter. Hence, these equations are evaluated

for each tick arrived to *step*. The equations that need the value of t to calculate the unknown value (*e.g.*, $v = dx/dt$), extract t from the specification of the clock signal that arrives to *step*. Usually, the clock is defined in another view where the time model of the system is its main concern. We describe in details the event specification in Subsection 3.4.2. We point out that the force equation does not have the t parameter. However, its *equationalModel* contains this parameter to evaluate the equation at each occurrence of *step*. We realize that the evaluation order of the equations depends on which value is known. In the example, we cannot evaluate $F = m \bullet a$ if we do not evaluate before $a = dv/dt$, and this last equation cannot be evaluated if $v = dx/dt$ is not calculated. The equation dependency and the evaluation order could be established by the *step* event specification. In the figure, we also present the *ParameterConnector* to bind parameters from one view to another. In the example, *ParamterConnector* connects the a parameters defined in *Force View* and *Movement View*.

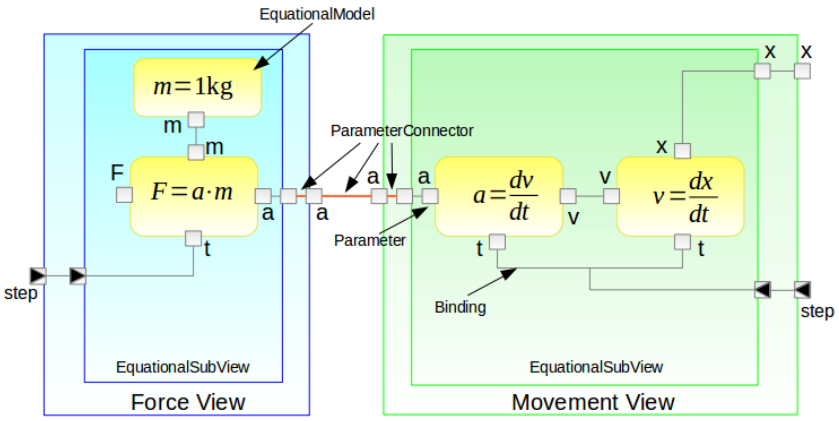


FIGURE 3.8: *EquationalSubView* Example

In the *EquationalSubView* meta-model, we also represent the *Equivalence* and *Characterization subCorrespondences* with their corresponding associations. By extracting a portion of the example depicted in Figure 3.8, we present the use of these *subCorrespondences*. In Figure 3.9, we define a *Mechanical View* that describes the mechanical structure of a system (a trailer hooked to a car) and its behavior according to the charge in the trailer. This view owns two *subViews*: a *structuralSubView* that defines the structure and behavior of the system, and an *equationalSubView* where the equations and values of the system physics are specified. In the *structuralSubView*, the trailer has two possible mechanical states: *charged* and *empty*. On the other hand, the car has only one state named *move* that represents the action to move the car by its engine. Trailer

has also a *mass* property whose value changes according to the *m* parameter value. In the *equationalSubView*, we specify the mass values of the trailer states associating such states with the corresponding equations by using *Characterization subCorrespondences*. By selecting a state, a mass value is assigned to the *m* parameter. At the same time, the value of the mass property defined in the *structuralSubView* of the trailer is equivalent to the *m* parameter value, because of the *Equivalence subCorrespondence*. In the *EquationalSublView*, we also define a force equation. This equation describes the required force that the car engine has to generate in order to move the trailer according to its mechanical states (charged or empty). In this example, we note that by using the *EquationalSubView*, we can study the impact of the behavior between *subViewElements* of the same *structuralSubView*, and it is possible to associate other behaviors from other views.

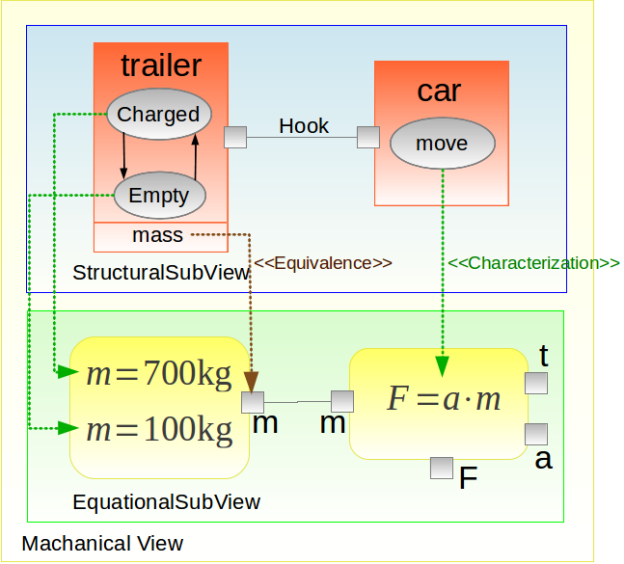


FIGURE 3.9: Example of the characterization and equivalence correspondences use.

3.2.4. Control SubView

ControlSubView synchronizes the execution of the *structuralSubView* according to the actions produced in its own view and from other views. *ControlSubView* also provides the events needed to evaluate the active equations in the *equationalSubView*. The goal of this *subView* is to coordinate the execution between views fulfilling the system requirements. For instance, the execution of a task in a CPU must satisfy a specific deadline defined in the system requirements. To achieve this deadline, we must set the frequency clock

of the CPU. This setting action is specified in the *controlSubView* of a time performance view.

The *subViewElement* execution is commanded by control events sent from a *controlSubView*. The *controlSubView* designers of each specific domain must specify the relationships among control events to ensure the correct coordination among *subViewElements*. Additionally, the designers have to synchronize the execution of the views guaranteeing the system requirements. These relationships can be defined in CCSL [3], which is a declarative language that specifies causal and temporal relationships among events. Using CCSL, we can generate a possible scenario that follows the event relationship definition using TIMESQUARE tool [6]. We can also generate observers that check the correctness of a hardware implementation [45].

The relationship between the events generated and received by *controlSubView* could directly be defined by CCSL expressions. However, we could also split the *controlSubView* structure in one or more sub-components named *controllers*. Figure 3.10 depicts the meta-model of *controller*. A *controller* is a *component* that owns *ports* (*controlPorts* and *propertyPorts*) and *connectors* (*controlConnectors*). These concepts are employed to send control events to *subViewElements* and to other views. Additionally, a *controller* can receive control events from other *controlSubViews* which may belong to different views, in order to synchronize the view execution. A *controller* can also receive property values from a *subViewElement* of its view. This value can be employed to take decisions in the *controller*.

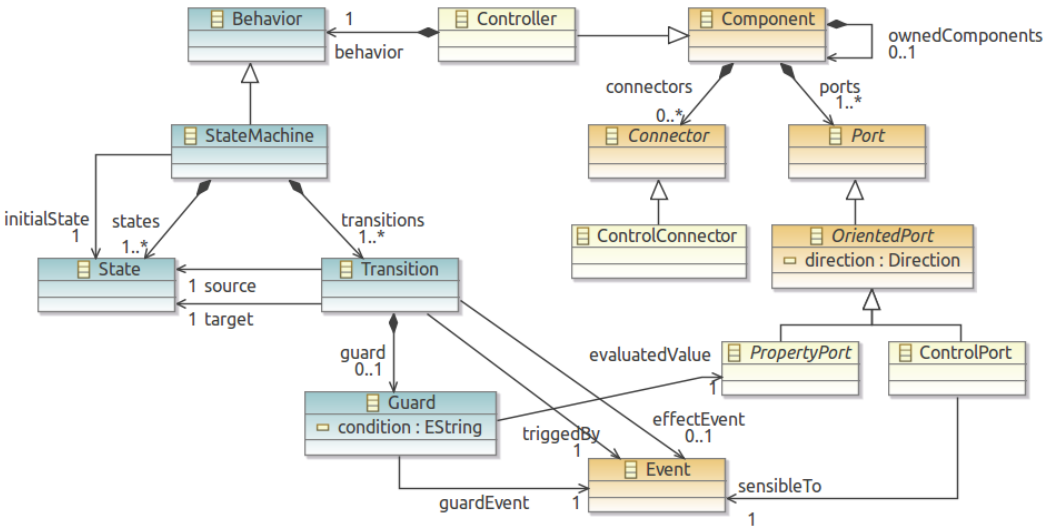


FIGURE 3.10: Controller meta-model

The behavior of a *controller* is expressed by a state machine. Contrasted with *subViewElement* state machine, the *controller* state machine transition contains a boolean condition to be able to fire it. UML state machine specifies this condition as *guard*. Nevertheless, instead of following the UML guard semantics, where the guard only enables the transition to be fired by a trigger event, we define that once the *guard* condition is true, the transition is fired. In our study, *guard* always evaluates a property value that is controlled, *i.e., guard* is true if the controlled property is higher or lower than a given value. In addition to the firing transition generated by the *guard* condition, the transition can directly be triggered by an event. This event arrives to the *controller* control ports coming from the other views. Once the transition is fired, an *effect* event is generated. This event is sent either to the corresponding *subViewElement* or to other views. The control event allows to change the active state of the *subViewElement* according to the changes of other views. As soon as a new state is active, one or more property values could change due to the transition of the associated equation. In consequence, the new values impact the controlled property value.

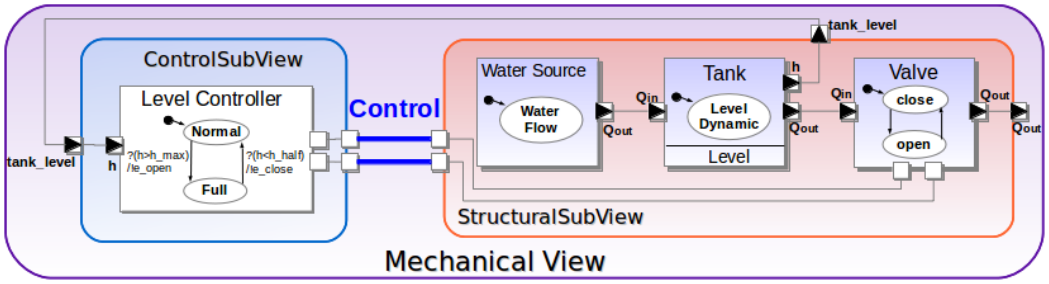


FIGURE 3.11: Example of the use of *ControlSubView* to control the water level of a tank.

In Figure 3.11, we present an example of a *controlSubView* by employing a *controller*. We depict a mechanical view of a system that controls the level of a water tank. This view contains a *controlSubView* and a *structuralSubView*. The *structuralSubView* defines two elements in the system: a *water source* and a *valve*. The *water source* supplies a flow of water to a *tank* and the *valve* controls the tank level by draining water from the *tank*. The *ControlSubView* is composed by a *level controller* that commands the valve actions according to the tank level. The behavior of *water source* and *tank* is specified as a state machine with a single state, *i.e.,* there is an associated equation that defines the water flow supplied by the water source and another equation that expresses the tank level dynamic. These equations are defined in an *equationalSubView*. The

controlled property is the tank level, therefore this property is sent to the *controlSubView* in order to take control decisions when the tank level arrives to the maximum or to the half of the tank. The behavior of *level Controller* reacts in two cases: when the tank level is higher than the maximum (h_max) or once it is lower than half of the tank (h_half). If the tank level reaches the maximum, *level controller* generates a control event (e_open) to open the valve reducing the tank level. In contrast, if the tank level is lower than half of the tank, *level controller* orders to close the valve, allowing the filling of the tank. We remark that there are *controlConnector subCorrespondences* between *ControlSubView* and *StructuralSubView*. This *subCorrespondence* allows to orchestrate the *structuralSubView* elements.

If we add more views to this example, *e.g.,* an electrical view or a time performance view, the actions of their *subViewElements* must be coordinated with the mechanical view execution to keep the execution consistency among views and to achieve the system requirements. The coordination is specified through the *controlConnector correspondences* among views. These correspondences transmit the control events among views and synchronize the execution of each view.

The behavior of *controllers* could be specified by using another model of computation, such as Petri nets. This behavior can also be defined by algorithms that optimize specific property values fulfilling certain restrictions, *e.g.,* reducing the time to fill the tank, taking into account the cross-sectional area of the *water sink*.

3.3. UML Profile for PRISMSYS

In Model-Driven Engineering, there are two branches for the developing of modeling languages. One branch defines specific languages adjusted to the terms and the way experts visualize their domains. This branch is the Domain Specific Modeling Languages (DSML). In contrast, the other branch defines a general language whose concepts give the necessary eloquence to represent a long range of domains. The main promoter of the later branch is the Object Management Group (OMG). The OMG defines the UML specification and has added other specific domains that use UML concepts as basis to represent their domain languages through UML *profiles*. Examples of these domains are

real-time systems with MARTE [5], systems engineering with SysML [4], or telecommunication with TelcoML [46].

There is an important UML community that uses this language to model their domains adopting the profile mechanism. Moreover, UML is implemented in recent modeling tools like Eclipse-Papyrus [47], UML Designer [48], MagicDraw [49], Modelio [50], Rational Software Architect [51] and Rhapsody [52].

To benefit from the UML development, we define a UML profile to represent the *PRISMSYS* framework. We use as much as possible the UML meta-classes including the stereotypes specified in SysML and MARTE to represent the *PRISMSYS* concepts. The concepts that are not included in UML or in the mentioned profiles, are defined by extending carefully selected UML meta-classes whose semantics are as close as possible to the expected *PRISMSYS* semantics.

3.3.1. UML Concepts for PRISMSYS

We represent part of the *PRISMSYS* framework meta-model concepts by using as basis the UML composite structures. We extend the composite structure meta-classes with the corresponding *PRISMSYS* concepts by defining stereotypes in the *PRISMSYS* profile. Table 3.1 lists the mappings between the PRISMSYS concepts and UML composite structures concepts.

PRISMSYS	**UML**
ArchitectureDescription	EncapsulatedClassifier
View	EncapsulatedClassifier
SubView	EncapsulatedClassifier
SubViewElement	EncapsulatedClassifier, BehavioredClassifier
Property	Property
Connector	Connector
StateMachine	StateMachine
State	State
Transition	Transition
Abstraction	Abstraction

TABLE 3.1: PRISMSYS - UML Mapping.

The main UML concept that we use to represent the structure of *PRISMSYS* is *EncapsulatedClassifier*. Figure 3.12 presents a simplified meta-model of this UML concept. We note that *EncapsulatedClassifier* inherits from *StructuredClassifier*, which contains *properties*, *connectors* and *parts*. *Parts* are instances of *StructuredClassifiers*. From the *PRISMSYS* point of view, these parts are the instances of *views*, *subViews* or *subViewElements* defined as *EncapsulatedClassifiers*. In Figure 3.12, we also observe that an *encapsulatedClassifier* not only has *properties*, but also *ports*, which are *property* specializations. In consequence, an *encapsulatedClassifier* contains *parts*, *properties*, *ports* and *connectors*, and that is the same structural definition specified for *ArchitectureDescription*, *View*, *SubView* and *SubViewElement* in the *PRISMSYS framework* meta-model.

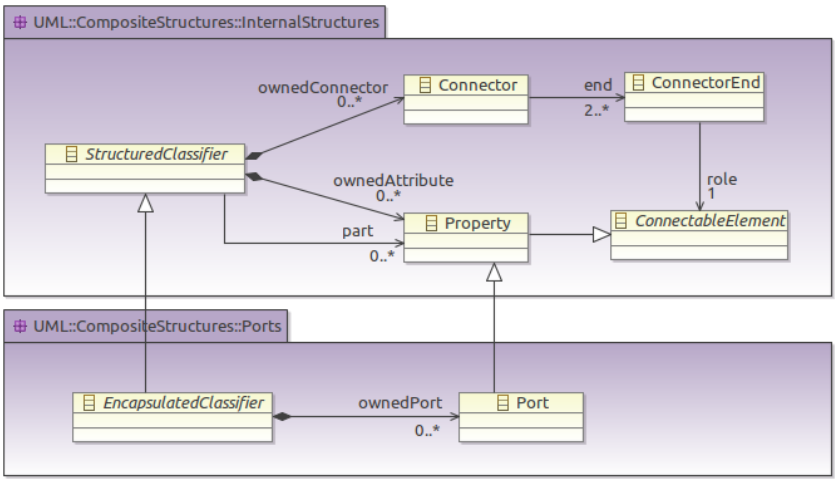


FIGURE 3.12: Simplified meta-model of *EncapsulatedClassifier*.

The *SubView* stereotype is specialized in *StructuralSubView*, *ControlSubView* and *EquationalSubView*. Therefore, these three kinds of *subViews* also specialize *EncapsulatedClassifier*. A *view* part is included in an *architectureDescription* and a *subViewElement* part is contained in a *structuralView*, following the *PRISMSYS* framework meta-model.

In the *PRISMSYS* framework meta-model, we also define that a *SubViewElement* contains a behavior specified by a *StateMachine*. Therefore, *SubViewElement* is also a *BehavioredClassifier* specialization. We constrain that the *SubViewElement* stereotype only owns a *StateMachine*. In the *StateMachine* definition, *Transition* keeps the UML definition. Nevertheless, *State* is extended to represent the *Characterization subCorrespondence* between state-equation defined in the *PRISMSYS* framework. Figure 3.13 presents the state extension. The *PRISMSYSState* stereotype contains the *equations* property whose type is *Constraint*, *i.e.,* a state stereotyped by *PRISMSYSState* must associate a *Constraint*, which is the way SysML recommends to specify equations in a *ConstraintBlock*.

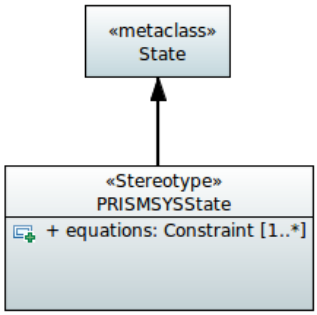


FIGURE 3.13: State stereotype.

The *Abstraction* correspondence of *PRISMSYS* is represented by the UML *Abstraction* relationship. According to the UML specification, an *Abstraction "is a relationship that relates two elements or sets of elements that represent the same concept at different levels of abstraction or* ***from different viewpoints****"* [21], which is the semantics that we want to give in *PRISMSYS*. To represent the abstraction of a *subViewElement* in a view, we specify that the abstracted *subViewElement* is a UML *reference* of the *subViewElement* defined in the original view. Figure 3.14 depicts the use of the *Abstraction* relationship and *reference* in *PRISMSYS* represented in UML. We define two views: a *layoutView* that represents the physical layout of the system, and a *hardwareView* that expresses the functionality of the system hardware components. In *LayoutView*, *CPU* is abstracted from *HardwareView* to give physical dimensions to *CPU*. We use the MARTE HW_Layout package, which is part of the MARTE HW_Physical package, to represent the physical components by using the *hwComponent* stereotype. *HwComponent* contains the necessary properties to describe the physical component specified in a circuit layout, such as dimension, position, number of pins. At the top of the figure, we depict the physical layout that is represented by the UML *LayoutView*. To indicate that the abstracted *CPU* is not a *part* of *LayoutView* (*i.e., CPU* is not owned by *LayoutView*), but a *reference* (*i.e.,* only shared), it is graphically represented with a dashed border in *CPU*. We also note the *abstraction* association between the *CPU* reference and the *CPU* part.

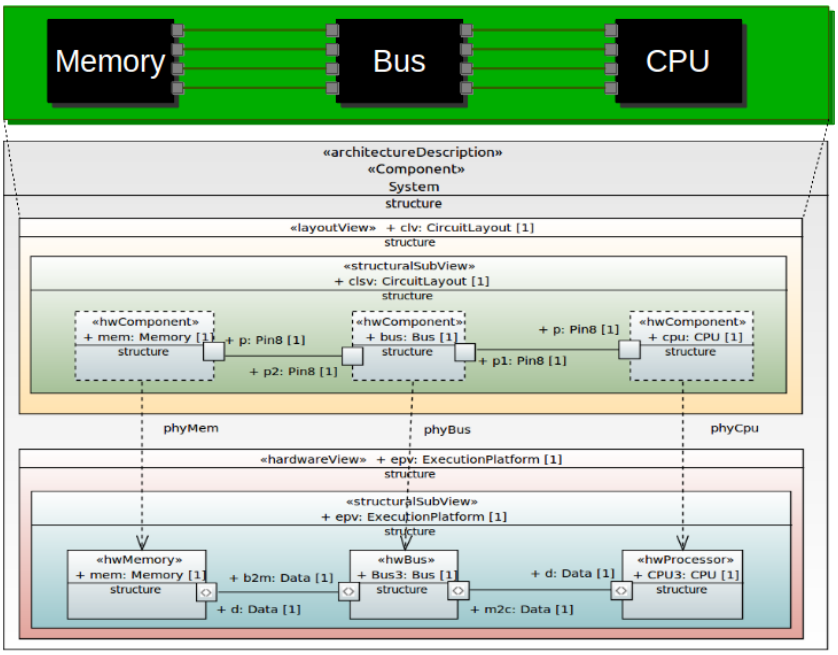


FIGURE 3.14: Abstraction of CPU in a layout component view.

3.3.2. MARTE Concepts for PRISMSYS

To represent the oriented direction of *OrientedPort* defined in the *PRISMSYS* framework meta-model, we use some MARTE concepts that are listed in Table 3.2.

PRISMSYS	**MARTE**
OrientedPort	FlowPort
ControlPort	Clock, FlowPort

TABLE 3.2: PRISMSYS - MARTE Mapping.

OrientedPort is an abstract concept in *PRISMSYS* that is represented by the UML *Port*. We add the MARTE extension *direction*, the property that represents the incoming or outgoing data flow in a port stereotyped by *FlowPort*. We have mentioned that *ControlPort* is a specialization of *Port* in *PRISMSYS*. This port is represented by the UML *Port* adding the MARTE *FlowPort* and *Clock* stereotypes. The *Clock* stereotype specifies that *ControlPort* is a set of instants, in this case, a set of control instants. This kind of clock is known as *LogicalClock* in MARTE. Other kinds of clocks can exist in specific domains of a system, such as the *EquationalView* that describes the physical

time domain. The physical time is represented by *ChronometricClocks* in MARTE. We explain the importance of *Clock* in the definition of the *PRISMSYS* execution semantics in Section 3.4.

3.3.3. SysML Concepts for PRISMSYS

EquationalSubView follows the component approach such as *StructuralSubView* and *ControlSubView*. Therefore, *EquationalSubView* is also an *encapsulatedClassifier* in UML. However, We use the SysML *ConstraintBlock* stereotype to represent this *subView* in order to apply the SysML parametric diagram. *ConstraintBlock* extends *Block* and this last stereotype extends the UML *Class* concept. A *Class* inherits from *encapsulatedClassifier*, when it contains an internal structure based on components. In fact, *EquationalSubView* stereotype extends *EncapsulatedClassifier*.

The *EquationalSubView* meta-model concepts are mapped to the elements that build the parametric diagram in SysML. Table 3.3 presents the mapping. In SysML, *ConstraintBlock* contains *constraintProperties*, *parameters*, *constraints* and *bindingConnectors*, such as they are shown in Figure 3.15. *ConstraintProperties* are instances of other *constraintBlocks* and play the role of *"parts"* in the internal definition of a *constraintBlock*. By observing the *EquationalSubView* meta-model (Figure 3.7) and the *ConstraintBlock* meta-model (Figure 3.15), we can distinguish that the *EquationalModel* concept is a generic SysML *ConstraintBlock*. In the *EquationalView* meta-model, we specify that an *equationalSubView* contains *equationalModels* that are not instances of other *equationalSubViews*. Due to the general use of *ConstraintBlock*, the separation between *EquationalSubView* and *EquationalModel* is not present in SysML. As a consequence, the way to represent these two concepts limits the usage of *ConstraintBlock* according to the *PRISMSYS* profile. The *ConstraintBlock* that is stereotyped by *EquationalSubView* only contains *bindingConnectors* (binding), *constraintProperties* (instance of *EquationalModels*) and *parameters*. On the other hand, the *ConstraintBlock* stereotyped by *EquationalModel* owns *parameters* and *constraints* (equations).

PRISMSYS	SysML
EquationalSubView	ConstraintBlock
EquationalModel	ConstraintBlock
Parameter	ConstraintParameter
Equation	Constraint
Binding	BindingConnector

TABLE 3.3: PRISMSYS - SysML Mapping.

The association between *Parameter* and *Property*, which is the *Equivalence subCorrespondence*, is mapped using the SysML path name dot notion to get a nested property in a block hierarchy. For instance, to use the *w* property defined in *viewElement1*, we can define a parameter using the following path name:

CircuitLayoutView.StructuralView.subViewElement1.w,

i.e., this parameter is a reference to the *w* property defined in *subViewElement1*, which is contained in the *structuralSubView* of *CircuitLayoutView*.

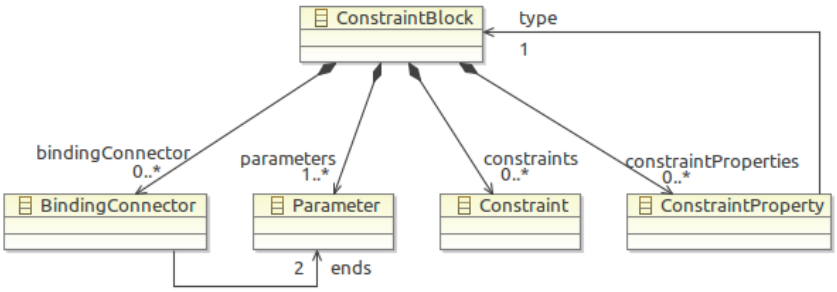


FIGURE 3.15: Simplified Constraint Block meta-model from the SysML specification.

3.4. Semantics of Execution

Once the syntax of *PRISMSYS* is specified, we define the way a *PRISMSYS* model is executed. In other words, we specify the execution semantics of *PRISMSYS*. It is based on the partial ordering of event occurrences, where each event represents a relevant change in the system. To achieve this goal, we use the Constraint Clock Specification Language (CCSL) [3].

CCSL is a formal declarative language to specify causal and temporal relationships between events. This language was firstly introduced in MARTE [5] to represent functional and extra-functional constraints over the time modeling of embedded systems. In MARTE, it is possible to define *Clock*s, which are an ordered set of instants. These clocks are used to represent the relevant changes in a system, on which constraints can be specified. For instance, a clock can represent the entering in a state, a function call, a data writing. Based on such clocks, relations can be specified to represent causalities or temporal aspects of the system. A clock can be of two types: *Chronometric* or *Logical*. Logical clocks represent functional time. For instance, based on clocks we can specify that the execution of an application is caused by touching the screen of a smart phone. In this example, the clock associated with the screen touching is in a causal relationship with the application execution. It is also possible to specify logical periodicity between clocks. For instance, specifying that a task is started every 100th cycle of a processor. Depending on the energy management in a computer, the start of the task can be periodic or not. When we want to specify something related to a physical dimension like the physical time or a distance, a chronometric clock is used. That is why, it is then possible to state that the CPU cycle is periodic every 3 ms.

Logical and chronometric clocks are employed in *PRISMSYS*. For example, a chronometric clock can express the physical time periodicity of a CPU cycle in a time description view. Furthermore, this clock can be used to define the instants when the equations in *equationalSubView* must be evaluated; *e.g.,* the temperature equation of a CPU is evaluated every 5 ms. On the other hand, a logical clock can describe the instant when a CPU starts to be busy (*i.e.,* once a task begins its execution on it). Logical clocks can also be used to define the execution semantics of Models of Computation (MoCs) [40]. In our case, we employ logical clocks to specify the behavior of the finite state machine (FSM) and the interactions that occur among *controlSubViews*, *controllers* and *subViewElements* (*i.e.,* the semantics of the sub-correspondence rules). Consequently, logical clocks are used to specify the coordination of the execution between MoCs of different nature. More precisely, in *PRISMSYS*, there are two behavior domains that have to be combined: a discrete event behavior represented by a set of finite state machines and a continuous time behavior, represented by a set of equations.

In this section, we first define the execution semantics of the finite state machine. Second, we specify the evaluation of the equations represented in *EquationalSubView*. Finally, the coordination between the finite state machine and the equation evaluation is described.

3.4.1. Finite State Machine Semantic Specification

In Section 3.2.2, we have chosen to specify the *SubViewElement* and *Controller* behavior by using a Finite State Machine (FSM). Sub-view elements and controllers do not use the same kind of FSM. The *SubViewElement* FSM changes from one state to another by the reception of a control event. In contrast, *Controller* reacts to either a guard condition or to the reception of a specific event. Additionally, *Controller* FSM can generate a control event (*effectEvent*) when a transition is fired. In this subsection, we define the FSM semantics by using *clocks* and relations defined in CCSL. First, we identify and specify the relevant clocks used to establish the FSM execution according to the concepts defined in the *SubViewElement* and *Controller* FSM meta-model. Second, we specify the relationship between clocks to describe the FSM semantics. In the following, we use the terms event and clock interchangeably.

3.4.1.1. Finite State Machine Clocks

In a FSM, there are various relevant events that occur during an execution. Most of the FSM concepts are associated with one or more events that describe a particular FSM change, *e.g.,* the entering in a state or the firing of a transition. We begin the definition of FSM clocks by representing the state activation. In a state, there are two possible events: Entering and leaving the state. For each of these events, we specify a clock in CCSL. To represent the entry into a state s, we define the clock s_{enter} and to express the leaving of this state, we define the clock s_{leave}.

The transition between two states is also represented by a clock. We name t_{ij} the clock that represents the firing of the transition between the two states s_i and s_j. A transition can be triggered either by an event representing the evaluation to true of the guard (*guardEvent*) or by the reception of a trigger event (*triggerEvent*). We designate $guard_{ij}$ the *guardEvent* of the transition t_{ij} and $trigger_{ij}$ its *triggerEvent*. *SubViewElement* FSM

transition is only sensitive to a *triggerEvent*, while *Controller* FSM can be sensitive to both events (*guardEvent* and *triggerEvent*). When one of these events occurs, the transition is fired instantaneously. Additionally, a *Controller* FSM can generate an *effectEvent* when a transition is fired. An *effectEvent* is a control event sent to either a *SubViewElement* to change its active state or to another view to synchronize the execution among views. We name $effect_{ij}$ the *effectEvent* of the transition t_{ij}.

Finally, we represent the event that initializes the state machine execution. We define the *init* clock that contains a unique instant. When *init* ticks, the FSM is entering simultaneously into the initial state.

Table 3.4 summarizes the clocks defined to represent the activity in the FSM of *subViewElement* and *controller*.

Clock	Action	FSM
$init$	initialization of the FSM	SubViewElement, Controller
s_{enter}	Entering into state s	SubViewElement, Controller
s_{leave}	Leaving from state s	SubViewElement, Controller
t_{ij}	Firing the transition from s_i to s_j	SubViewElement, Controller
$guard_{ij}$	Evaluation to true of the t_{ij} guard	Controller
$trigger_{ij}$	Reception of the trigger event of t_{ij}	SubViewElement, Controller
$effect_{ij}$	Event generated when t_{ij} is fired	Controller

TABLE 3.4: Clocks representing the relevant actions in a Finite State Machine for both *SubViewElement* and *Controller*.

3.4.1.2. Finite State Machine Clocks Relationship

Once the FSM clocks are defined, we identify the relationships of these clocks to describe the FSM execution semantics. We start defining the activation of a specific state, which is between the corresponding entering and leaving occurrences. Figure 3.16 presents a sequence of activations of the s state.

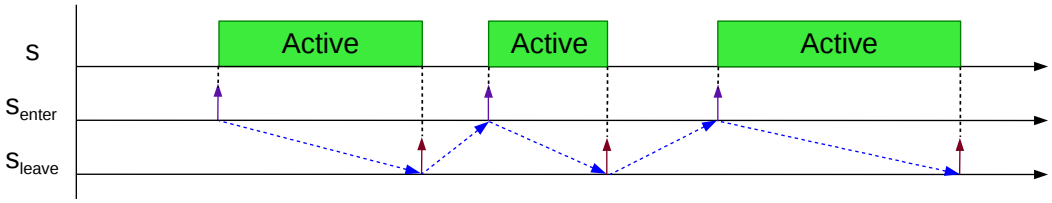


FIGURE 3.16: Representation of an active state by clocks

We specify that the s state is active when the s_{enter} clock ticks. The s state stops being active when s_{leave} ticks. We define that a state cannot be transitory, *i.e.,* the *enter* and *leave* events cannot be simultaneous. Moreover, a state can not be activated if it is already active. Consequently, we state an alternate relationship for all the states of FSM between s_{enter} and s_{leave} in CCSL as follows:

$$\forall s \in StateMachine.states,$$
$$s_{enter} \boxed{\sim} s_{leave} \tag{3.1}$$

where $StateMachine.states$ represents the set of states that belong to a FSM.

We have defined t_{ij} as the clock that represents the firing of a transition between two states s_i (source state) and s_j (target state). t_{ij} is formally specified as follows:

$$\forall i, j \text{ such that } s_i, s_j \in StateMachine.states,$$
$$t_{ij} = \{t \in StateMachine.transitions | t.source = s_i, \wedge, s.target = s_j\} \tag{3.2}$$

According to the execution semantics of FSM [53], a transition t_{ij} is fired if two conditions are achieved:

- s_i is active, and
- Either the $guard_{ij}$ occurs or $trigger_{ij}$ ticks.

We therefore study these conditions in the following items:

- **Transition fired by a guard:** Figure 3.17 depicts the transition between two states (s_i and s_j) caused by a *guardEvent* ($guard_{ij}$). Once s_i is active, *i.e.*, $s_{i_{enter}}$ ticks, it is possible to change to s_j. *eval* is a *chronometric clock* that commands the evaluation of the $guard_{ij}$ condition. Hence, if the evaluated condition is true, $guard_{ij}$ occurs. Considering that s_i is active and $guard_{ij}$ ticks, then the t_{ij} transition is fired.

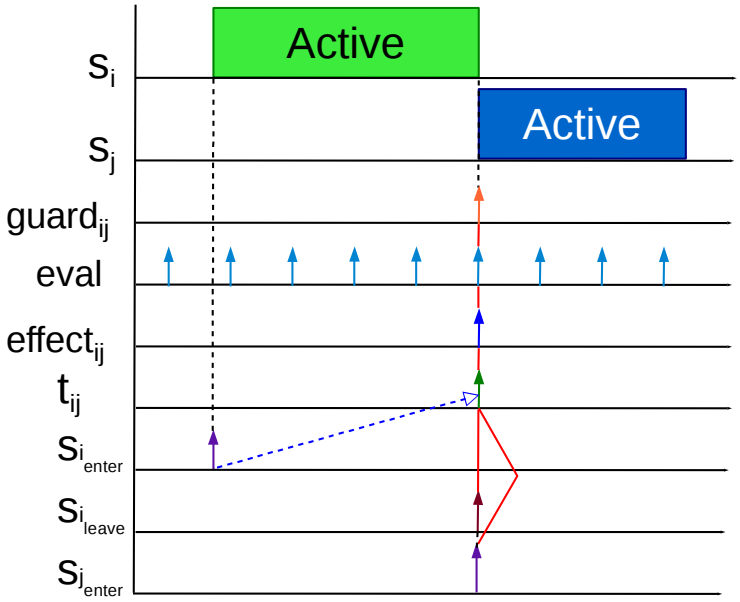


FIGURE 3.17: Representation of the clock ticks leading to a change between two states caused by a *guardEvent*.

We specify the relationship of these clocks by using CCSL expressions. We state the CCSL constraints to fire the t_{ij} transition by the following definition:

$$\begin{aligned} &\forall i,j \text{ such that } t_{ij} \in StateMachine.transitions, \\ &guard_{ij} <> null \textbf{ and } trigger_{ij} = null \textbf{ implies:} \\ &\text{let } t_{ik} = \{t \in StateMachine.transitions | t.source = S_i, \wedge, t <> tij\} \text{ and} \\ &\text{let } f_{ij} \triangleq [(s_{i_{enter}} \searrow guard_{ij}) \lightning \sum_{t \in t_{ik}} t] \bullet f_{ij} \text{ in} \\ &t_{ij} \boxed{=} f_{ij} \end{aligned} \tag{3.3}$$

this expression can be read as if $guard_{ij}$ occurs and not $trigger_{ij}$ then $s_{i_{enter}}$ is strictly sampled ($\searrow$) by $guard_{ij}$. Once $s_{i_{enter}}$ is sampled, if some transition fired from s_i occurs, different to t_{ij}, then f_{ij} is killed, *i.e.,* any other transition going out from s_i cannot be fired. The definition of the inability of s_i is represented by the CCSL relation *upto* ($\lightning$). The first part of Equation 3.3 ($[(s_{i_{enter}} \searrow guard_{ij}) \lightning \sum_{t \in t_{ik}} t]$) is only one occurrence of t_k, therefore each time s_i is active, the application of the first expression generates another f_{ij} occurrence. In consequence, we join the f_{ij} ticks by the CCSL *concatenation* operation ($\bullet$) in order to gather all the f_{ij} occurrences in one clock. Finally, t_{ij} coincides with f_{ij}.

Following the execution illustrated in Figure 3.17, s_i stops being active when t_{ij} occurs, *i.e.,* $s_{i_{leave}}$ ticks. The relationship between t_{ij} and $s_{i_{leave}}$ is specified by the CCSL *equality* relation ($\boxed{=}$):

$$\begin{gathered} \forall i \text{ such that } s_i \in StateMachine.states, \\ \text{let } t_{out} = \{t_{ij} \in StateMachine.transitions | t_{ij} = s_i.outgoing\} \text{ in} \\ s_{i_{leave}} \boxed{=} \sum_{t \in t_{out}} t \end{gathered} \tag{3.4}$$

we can interpret this specification as the leaving of s_i occurs when one of its outgoing transitions is fired, *i.e.,* the union of the occurrences of the outgoing transitions ($\sum_{t \in t_{out}} t$). The operator $\sum$ is derived from the union operator ($+$) in CCSL.

In Figure 3.17, we can also note that the t_{ij} clock coincides with the activation of s_j state, *i.e.,* $s_{j_{enter}}$ ticks. We specify this coincidence relationship by:

$$\begin{gathered} \forall j \text{ such that } s_j \in StateMachine.states, \\ \text{let } t_{in} = \{t_{ij} \in StateMachine.transitions | tij = s_j.incoming\} \text{ in} \\ s_{j_{enter}} \boxed{=} \sum_{t \in t_{in}} t \end{gathered} \tag{3.5}$$

this relation is read as the ticks of the fired incoming transitions of s_j (t_{in}) coincide with the $s_{j_{enter}}$ occurrences.

If the FSM belongs to a *controller*, then an *effect* can be generated, simultaneously with the transition firing, *i.e.,* $effect_{ij}$ occurs (see Figure 3.17). This relationship

is specified by:

$$\forall i,j \text{ such that } t_{ij} \in StateMachine.transitions,$$
$$effect_{ij} <> null \textbf{ implies :}$$
$$t_{ij} \boxed{=} effect_{ij} \tag{3.6}$$

- **Transition fired by an event:** A transition could be fired by an event according to the FSM meta-model. If t_{ij} is fired by $trigger_{ij}$, there is not synchronization with a *chronometric clock* to generate a t_{ij} tick. Figure 3.18 presents the t_{ij} firing case caused by $trigger_{ij}$.

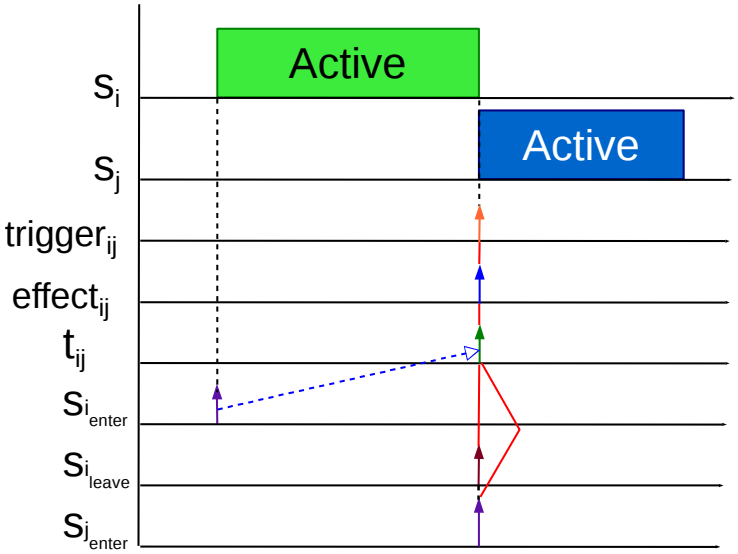


FIGURE 3.18: Representation of the clock ticks leading to a change between two states caused by a *triggerEvent*.

In the same way that $guard_{ij}$, the relationship between $s_{i_{enter}}$, t_{ij} and $trigger_{ij}$ is also specified in CCSL as follows:

$$\forall i,j \text{ such that } t_{ij} \in StateMachine.transitions,$$
$$guard_{ij} = null \textbf{ and } trigger_{ij} <> null \textbf{ implies:}$$
$$\text{let } t_{ik} = \{t \in StateMachine.transitions | t.source = S_i, \wedge, t <> tij\} \text{ and}$$
$$\text{let } f_{ij} \triangleq [(s_{i_{enter}} \searrow trigger_{ij}) \lightning \sum_{t \in t_{ik}} t] \bullet f_{ij} \text{ in}$$
$$t_{ij} \boxed{=} f_{ij} \tag{3.7}$$

- **Initial state definition:** The FSM must have at least one initial state to start its execution. We only consider the case that a FSM has only one initial state. We define a clock that begins the FSM execution activating the initial state. We have named this clock *init*. We only need a tick in *init* to active the initial state (see FSM mata-model - Figure 3.6). Therefore, we define *fsmClk*, which is a logical clock only used to specify *init*. Thus we state *init* in CCSL as follows:

$$init \boxed{=} fsmClk \blacktriangledown 1(0)^w \tag{3.8}$$

this equation means that *init* is the result of filtering $fsmClk$ with the binary periodic word $1(0)^w$. This word denotes that only the first tick of $fsmClk$ is taken.

The *init* clock must be associated with the initial state. Considering that s_{init} is the initial state of the FSM, we define its activation as follows:

$$\text{let } s_{init} = \{s \in StateMachine.states | s = StateMachine.initialState\}$$
$$s_{init_{enter}} \boxed{=} init \tag{3.9}$$

However, s_{init} is also activated during the FSM execution by its fired incoming transitions. Therefore, by using Equation 3.5 and 3.9, we complete the s_{init} specification by:

$$\text{let } s_{init} = \{s \in StateMachine.states | s = StateMachine.initialState\} \text{ and}$$
$$t_{in} = \{t \in StateMachine.transitions | t = s_{init}.incoming\} \text{ in}$$
$$s_{init_{enter}} \boxed{=} init + \textstyle\sum_{t \in t_{in}} t \tag{3.10}$$

we can interpret this equation as the initial state of the FSM (s_{init}) is active when either *init* occurs or an incoming transition to the initial state is fired.

3.4.2. Equational View Semantic Specification

In systems, the notion of time is always present in the evolution of non-functional properties. These properties are evaluated in a time instant and their values could be used to calculate other properties by using equations. For instance, the temperature evolution

of a *cpu* depends on the progression of its dissipated power. In *PRISMSYS*, *EquationalSubView* contains such equations and the active ones are evaluated through time. The *characterization subCorrespondences* allows to change the active equations according to the active *subViewElement* states. In this section, we formally specify the non-functional property evolution through equations. These equations are evaluated at discrete time and according to active states. To this end, we use CCSL to specify a *chronometric clock* to state the discrete time for the equation evaluation. CCSL is also employed to define the causal relationship between the active states and the associated equations to be evaluated.

We specify that the time notion in an *equationalSubView* follows the physical time specified in MARTE. This standard describes that physical time is *"a continuous and unbounded progression of physical instants"* [5]. Physical time can be modeled as a dense time base. Such a time base is an ordered set of instants where *"for a given pair of instants, there always exists at least one instant between the two"* [5]. *Dense clocks* could be defined from the dense time base. The MARTE *TimeLibrary* contains a dense clock called *idealClock*. This dense clock represents the physical time that describes physical laws. For instance, in the equation $a = dv/dt$, dt could be represented by *idealClock*. *IdealClock* has as time base unit *second*. By using *idealClock*, we define *chronometricClocks*. A *chronometricClock* represents the periodic occurrences of the physical time evolution. Therefore, we define *chronometricClocks* to mark the periodic time evolution of certain *subViewElements* that need the time notion. For instance, we could represent the measure of humidity by using a *chronometricClock* that ticks every 10 s. For each clock tick, the humidity is measured.

We specify a *chronometricClock* to evaluate the equations defined in *equationalSubView*. We name this clock *step*. At each occurrence of *step*, a new value is calculated according to the equations activated by the *subViewElement* states. The *step* clock can be specified by discretizing *idealClock* or it can be derived from the relationships with other *chronometricClocks* specified in other views. For instance, *step* occurrences could coincide with the ticks generated from the CPU clock source, clock that can be defined in a time performance view.

Figure 3.19 presents an example of a *PRISMSYS* model where the temperature evolution of a CPU is specified.

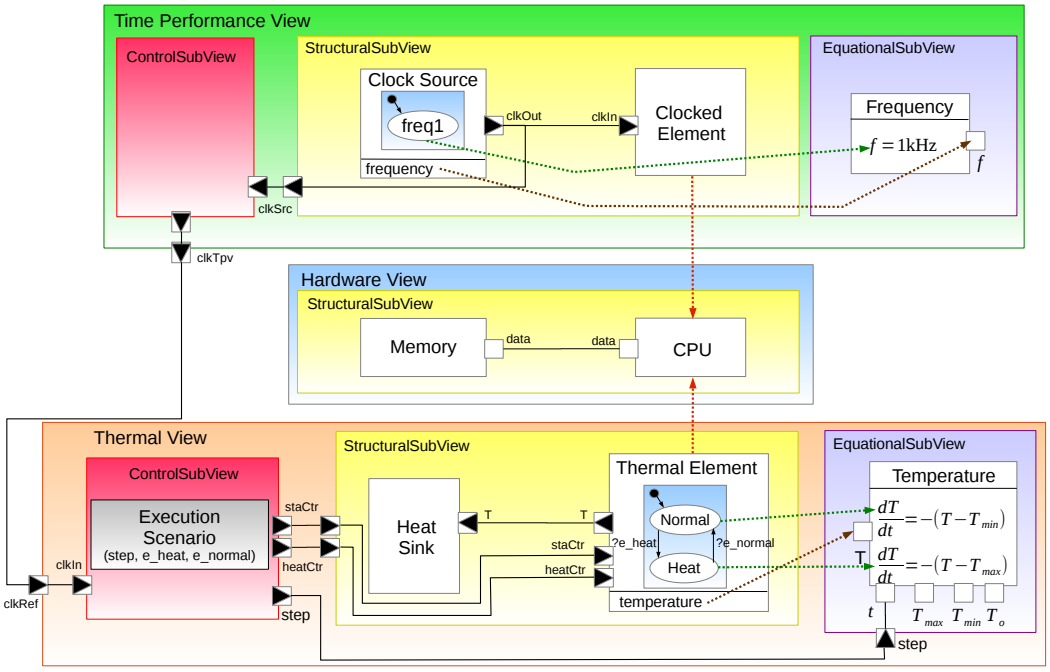


FIGURE 3.19: PRISMSYS model where the temperature of a CPU is characterized in the *equationalSubView*.

In the figure, three views are depicted. *Hardware View* is the view where the structure and the functional behavior of the system components are defined. *Thermal View* describes the thermal architecture of the system, including its thermal behavior and its equational representation. The thermal behavior corresponds to the CPU activity that is specified in *ThermalElement*, which is the CPU abstraction from the thermal point of view. We represent the CPU thermal activity by *states*. The transition between states is controlled by the *controlSubView*. In this example, we only recreate a possible execution scenario in the *ControlSuvView* of *ThermalView* to command the thermal states of the CPU. The thermal states of the CPU are two: *Normal* and *Heat*. The former expresses that the CPU maintains the typical temperature when it is not active. In contrast, *Heat* describes that the CPU temperature raises if it is active. Both states are associated by transitions that are sensitive to the *e_heat* and *e_normal* events generated from the *controlSubView*.

The thermal representation of the CPU also contains a temperature property whose value depends on the active thermal state. The temperature value is the result of the evaluation of the active thermal equation defined in the *equationalSubView*. The thermal equations belong to an *equationalModel* named *Temperature*. Such equations are

associated with the thermal states in *ThermalView*. The equations are first-order differential equations whose solutions are exponential functions. *Normal* state is associated with a temperature equation whose response is asymptotic to T_{min}, which is the minimum temperature that the CPU can achieve in halting state (*i.e.,* without activity). The *Heat* state is characterized by the second temperature equation whose response is asymptotic to T_{max}, the maximum temperature that CPU can support before burning out. The *Temperature equationalModel* also contains the parameters T, T_{min}, T_{max}, T_o and t. T is the temperature evaluated according to the active equation, T_{min} and T_{max} are constant values as well as T_o, which is the initial temperature at $t = 0$, *i.e.,* T_o is the environmental temperature.

The t parameter is the physical time of the equations. t is discretized by a *chronometricClock* defined in *TimePerformanceView*. Such a view defines the temporal features of the example system. We note that its *structuralSubView* contains a *ClockSource* that is a clock generator. The *ClockSource* owns a frequency property whose value is defined by the associated equation $f = 1\ kHz$. By using this definition, we specify the generated clock signal from *ClockSource* by the following CCSL expression:

$$clkOut \boxed{=} idealClk \textbf{ discretizedBy } 0.001 \tag{3.11}$$

where 0.001 is the period defined by the equation $f = 1kHz$. This generated clock signal is used to evaluate the active thermal equation. To share the *clkOut* signal, we send the generated clock signal to *controlSubView* of *Time Performance View* through *clkSrc* port. The connection between *StructuralSubView* and *ControlSubView* is a *DataConnector subCorrespondence*. The *controlSubView* retransmits the *clkSrc* clock signal to the *Thermal View* through the connection between the *clkTpv* and *clkRef* ports. This connection is a *DataConnector Correspondence*. Afterwards, *clkRef* port is connected to *clkIn*, which is an input port of *Thermal View controlSubView*. As a consequence, *controlSubView* can generate the temperature scenario synchronizing the *e_heat* and *e_normal* occurrences with the clock signal received on *clkIn*. Additionally, the received clock signal is shared with *equationalSubView* to mark the instants when the active equation of the *equationalModel* is evaluated. The received clock signal is sent through the *step* port to *equationalSubView*. *step* is associated with t by using the *binding* connector. This association specifies that the *step* clock evolution is equal to

the t progression. Consequently, for each tick of the *step* clock, the active equation is evaluated.

By using this example, we can specify the semantics of *DataConnector correspondence* and *subCorrespondence* in the specific case of the transmission of a clock signal. Additionally, we define the coordination between the active states (*i.e.,* the active equation) and the equation evaluation. We can specify in CCSL the relationship between *clkOut*, *clkSrc*, *clkTpv*, *clkRef*, *clkIn* and *step* as:

$$clkOut \boxed{=} clkSrc \tag{3.12}$$

$$clkSrc \boxed{=} clkTpv \tag{3.13}$$

$$clkTpv \boxed{=} clkRef \tag{3.14}$$

$$clkRef \boxed{=} clkIn \tag{3.15}$$

$$clkIn \boxed{=} step \tag{3.16}$$

these CCSL relations could be read as the instants generated by *clkOut*, *clkSrc*, *clkTpv*, *clkRef*, *clkIn* and *step* are coincidental, in other words, they tick at the same time instant. Therefore, the execution semantics of *DataConnector correspondence* and *subCorrespondence* is specified by an *equality* CCSL clock relation, in the case that the transmitted data is a clock signal.

In the *controlSubView* of *ThermalView*, we define an execution scenario to specify at which instant *e_heat* and *e_normal* occur. Figure 3.20 presents the temperature evolution through time according to an execution scenario. At the beginning of the simulation, *i.e.,* at $t = 0$, the state machines in *ClockSource* and *ThermalElement* enter into their respective initial states (*freq1* in *ClockSource* and *Normal* in *ThermalElement*). Therefore, the active equations in the *equationalSubViews* are $f = 1\ kHz$ in *Frequency equationalModel* and the first equation in *Temperature equationalModel*. At the same instant, the clock generated by *ClockSource*, *i.e., clkOut*, starts to tick. Following Equations 3.12, 3.13, 3.14, 3.15 and 3.16, for each *clkOut* occurrence, the first equation of *Temperature equationalModel* is evaluated. Note the coordination between the state machine execution (discrete time behavior) and the equation evaluation (continuous time behavior). Once an *e_heat* event occurs, the transition from *Normal* to *Heat* is fired and the *Heat* state is active. In consequence, the associated equation is activated and

the temperature value is evaluated at the next *step* tick. After producing the *e_heat* event, *step* ticks twice before *e_normal* ticks. This *e_normal* event fires the transition from *Heat* to *Normal* returning to the *Normal* state. In the figure, we note the change of the active equation by the new evaluated temperature value in the next *step* tick. This value is calculated by the first equation of the *Temperature equationalModel*.

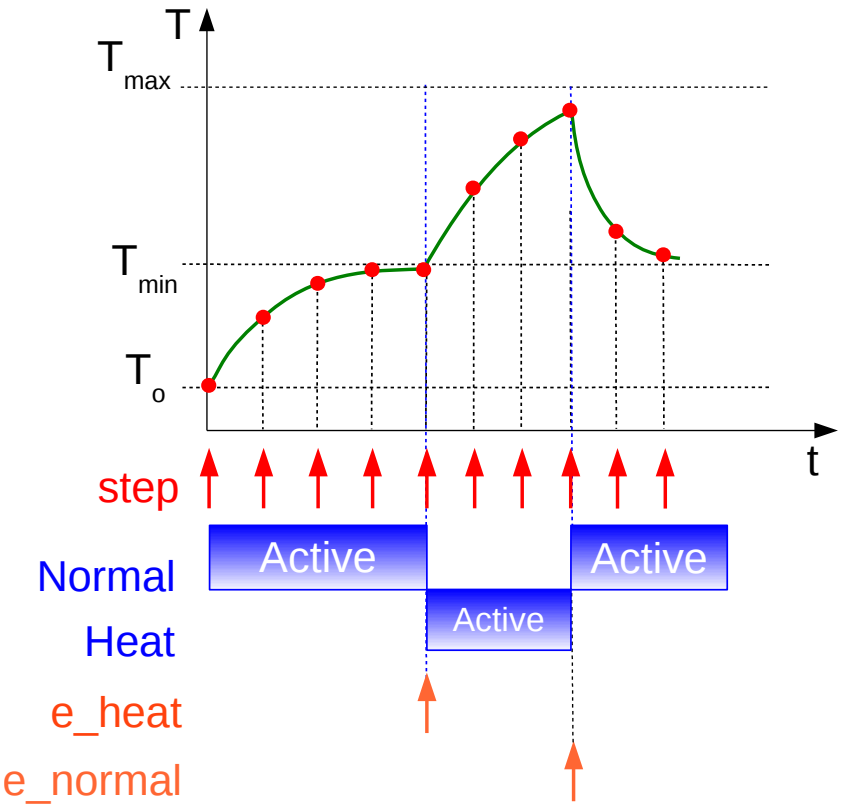


FIGURE 3.20: Temperature evolution through time according a predefined execution scenario.

We note in this example that the synchronization between heterogeneous behaviors (finite state machine and continuous time) is given by the time discretization and the relationship specification between the actions in the state machine and the instants where the equations are evaluated. This relationship is specified in CCSL.

3.5. Conclusion

In this chapter, we have presented the *PRISMSYS framework*. This framework is a language that allows the description of systems from different points of view. *PRISMSYS*

exposes different sub-views that must be specified in each view to describe a specific domain. *PRISMSYS* provides the basic sub-views to be extended in order to express the necessary views of the stakeholders' concerns. The *PRISMSYS* framework also defines the necessary correspondences to maintain the coherence among the views and to coordinate their execution. We also define the sub-correspondences between the predefined sub-views to keep the consistency among sub-views. Correspondences avoid the re-definition of domain elements, re-using elements and properties from other views. Additionally, correspondences expose the execution impact between views in a single system model. This impact is also projected in the achievements of system requirements.

We also propose a UML profile to represent a *PRISMSYS* model in UML by using as much as possible the concepts already specified in UML, SysML and MARTE. The designers that employ UML tools to describe systems, they could easily apply the *PRISMSYS* framework in a UML environment.

We define the execution semantics of *PRISMSYS* by using CCSL. Thanks to CCSL, we could define the execution of a discrete event model, *i.e.,* Finite State Machine, and the instants when the equations of a continuous time model are evaluated. The relationship definition between both models (discrete event and continuous time) allows the coordination of the execution of these models, through the use of another way to execute heterogeneous models.

In the next chapter, we present a use case that defines the necessary views to describe power consumption of an embedded system. We also illustrate the impact of other views in the system power consumption.

Chapter 4

Power Consumption Modeling

Contents

4.1. Introduction 72

4.2. Dynamic Power Consumption 73

4.3. Static Power Consumption 74

4.4. Characterization for Power Consumption 75

4.5. Power Management Techniques 77

4.5.1. Clock-Gating 78

4.5.2. Power-Gating 78

4.5.3. Dynamic Voltage-Frequency Scale 80

4.6. Power Design Specification 81

4.6.1. UPF, CPF and IEEE 1801 81

4.6.2. SystemC 84

4.6.3. UML 84

4.7. Discussion 85

4.8. Conclusion 86

4.1. Introduction

Nowadays, digital circuits are built using the CMOS technology. In figure 4.1, we depict the base gate of the CMOS technology whose behavior corresponds to a NOT logical function. From this gate, various logical functions can be built. In the figure, the CMOS gate contains a *PMOS* transistor and a *NMOS* transistor. These transistors have the same physical characteristics in order to have the same behavior when they are switched. V_{in} is the input signal that can be a logic *0* (a voltage close to ground) and *1* (a voltage close to V_{dd}). V_{out} is the output signal of the gate.

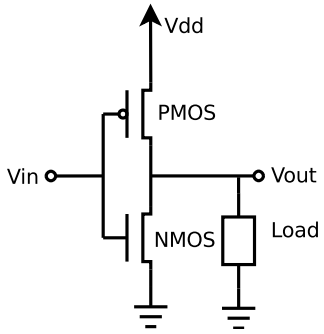


FIGURE 4.1: CMOS inverter circuit.

According to the V_{in} signal, V_{out} is obtained. Considering V_{in} is initially in *1*, *i.e.,* in V_{dd}, and we change the V_{in} value to *0*. Once the change is done, the *PMOS* transistor is closed and the *NMOS* transistor is open during a short period of time. If the *PMOS* transistor is closed, the current that circulates from V_{dd} to the charge *Load* is reduced to almost *0A*. In contrast, the *NMOS* transistor is opened, therefore there is a current that circulates from *Load* to ground though the *NMOS* transistor. This current is also generated for a short period of time; while the *Load* charge is discharged. During the state change, the produced current in both transistors generate power consumption. Once the circuit arrives to a stable state, the V_{out} value becomes a *0* logic. However, this *0* is not exactly a *0V*. There is a small current that circulates from V_{dd} to ground during the stable state, producing additional power consumption.

Various authors [54] [55] [56] [57] identify three sources of power consumption in digital CMOS circuits:

$$P_{total} = P_{short} + P_{switch} + P_{static} \tag{4.1}$$

where P_{short} is the power consumed when the *NMOS* and *PMOS* transistors are simultaneously active, *i.e.,* producing a short-circuit current from V_{dd} to ground. This power consumption is usually small compared to P_{switch} and P_{static}. P_{switch} is the power consumed during the period that the circuit is in constant activity, *i.e.,* the transistor are switching. The sum of P_{switch} and P_{short} is known as dynamic power consumption (P_{dyn}). In contrast, P_{static} is the power consumed when the digital circuit is in stand-by state, *i.e.,* when the transistor are not switching.

The power consumption that predominates among the mentioned powers is $P_{dynamic}$. However, in the last years, caused by the transistor size reduction, P_{static} is becoming an important source of power consumption.

In the next sections, we explain in more detail the dynamic and static power consumptions. We continue describing the power consumption estimation according to the abstraction description level of the system. Afterward, we present the main strategies to manage the power consumption. Finally, we expose the different approaches that specify power design for electronic systems.

4.2. Dynamic Power Consumption

Previously, we mentioned that the dynamic power consumption is defined by the following equation:

$$P_{dyn} = P_{short} + P_{switch} \tag{4.2}$$

where P_{short} is the power consumed during the period when both transistors are active, and P_{switch} is the power consumed during the switching period. We can express P_{switch} according to the following equation:

$$P_{switch} = \alpha C_L V_{dd}^2 f \tag{4.3}$$

Where α is the input transition activity factor of the CMOS gate, C_L is the capacitance of *Load*, V_{dd} is the voltage of the CMOS gate source and f is the transition frequency. *Load* represents the wires and other transistors that are connected to the CMOS output.

According to this equation, P_{switch} depends mainly on the voltage and the frequency, therefore there are certain techniques to reduce the power consumption at this point, for example Dynamic Voltage-Frequency Scale (DVFS) and clock-gating. We present these techniques in detail in Section 4.5.

4.3. Static Power Consumption

According to [54] and [58], static power consumption of a CMOS gate is due to various leakage currents that flow through the gate during the stable state. Figure 4.2, depicts a *NMOS* transistor with its main leakage currents. This transistor contains a p-type substrate, *i.e.,* this substrate contains excess of charge carries or "holes" and a n-type channel, *i.e.,* the channel transmits free-electrons from *Drain* (D) to *Source* (S) terminals. The *Gate* (G) terminal controls the electrons flow between *Drain* and *Source* according to the voltage applied. Finally, the *Body* terminal (B) is connected to the p-type substrate. Generally, *Body* is connected to ground in a *NMOS* transistor.

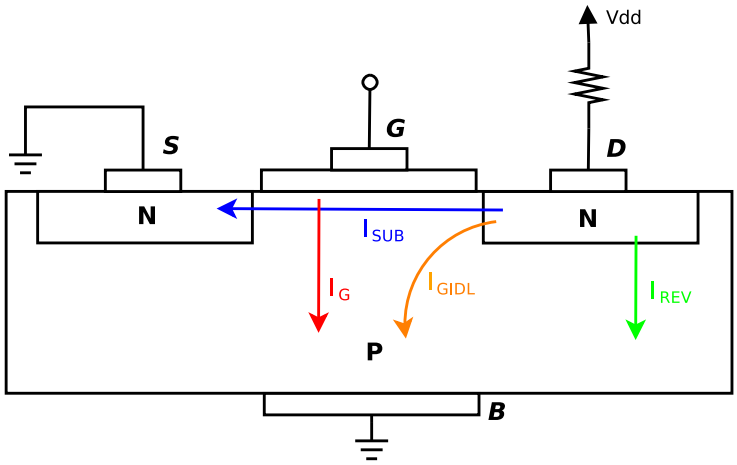


FIGURE 4.2: Leakage currents of a NMOS transistor.

In the figure, I_{REV} represents the *Junction Leakage* current. This current is produced by the reverse-biased junction. I_{GIDL} represents the *Gate-Induced Drain Leakage* current. This current is produced by the band-to-band tunneling effect in the gate-drain overlap

region. I_G depicts the current that flows from the gate terminal to the p-type substrate through the oxide insulation. I_{SUB} represents the *Substhreshold leakage* current. This current that is produced between *Source* and *Drain* terminals caused by working the transistor in the weak inversion region.

All these currents are affected by the transistor characteristics (size, voltage applied, etc.) and by the temperature. One of the most significant leakage current is I_{SUB}. This current can be modeled by the following equation:

$$I_{SUB} = KV_T^2 \left(\frac{W}{L}\right) e^{(V_{GS}-V_{th})/nV_T} \left(1 - e^{-V_{DS}/V_T}\right) \tag{4.4}$$

where K, W, L, n are transistor characteristics, V_{GS} is the *Gate-Source* voltage, V_{DS} is the *Drain-Source* voltage, V_{th} is the threshold voltage and V_T is the thermal voltage. V_T is directly proportional to the transistor temperature, therefore according to the equation, I_{SUB} exponentially increases in function of the temperature.

4.4. Characterization for Power Consumption

Power models characterize the power consumption of hardware components according to a functional execution. These power models are implemented in various tools using different abstraction levels. Ibrahim et al. [59] present a survey of the techniques used to estimate the power consumption of system components. They classify these techniques in the following levels:

- *Transistor-Level*: This level is a detailed description of the system components in circuits based on transistors. This level uses the physical transistor model, which is described in a continuous time domain, to get the component behavior and its characteristics such as time performance and power consumption. Generally, the power consumption is estimated by monitoring current and voltage of the analyzed circuit. This level is the most precise power consumption estimation technique because every characteristic of the transistor is defined. However, the simulation time is too long, moreover when designers want to simulate components

that have millions of transistors. Tools that use this technique are SPICE [60] and PowerMil [61].

- *Gate-Level*: In this level, the system components are described by logical gates. Therefore, the system simulation changes from a continuous-time domain to a discrete-time domain where each component is sensitive to events. According to the equation 4.3 from Section 4.2, α represents the input transition activity in a CMOS gate. In gate-level, this activity parameter can be estimated using different probabilistic methods. Chou and Roy [62] present a signal activity estimator based on Monte-Carlo experiments. Ding et al. [63] use probability waveforms to estimate the average switching activity.

- *Register Transfer-Level*: The register transfer models are interconnected blocks where each block has a specific functionality in a system. To characterize the power consumption of these models, their internal blocks are individually measured and analyzed from their physic implementation and their power properties are extracted. As gate-level, Register Transfer-Level estimation mainly works focused on extracting the activity information from the blocks and measure their power consumption response.

- *Architecture-Level*: This level uses a combination of the techniques mentioned before, mainly Gate-Level and Register Transfer-Level to estimate the power consumption of a system. For instance, *SimplePower* [64] employs transition-sensitive power models to estimate the power consumption of functional units. In contrast, SoftWatt [65] and Wattch [66] use a fixed-activity model. PowerSC [67] is a C++ library that extends SystemC [68] to specify power features and to estimate power consumption using different power modeling techniques.

  Another tool that is part of this level is Aceplorer [8]. They define the power consumption though the specification of voltage and current for each component of the system. These parameters are defined by equations and they can represent from the lower level power characterization, such as transistor-level, to the higher level, like instruction-level. However, this tool is commonly used to estimate power in the first phases of the system design. We detail this tool in Chapter 6. We use this tool to analyze the power consumption of the system specified in *PRISMSYS*.

- *Instruction-Level*: This level is exclusive to components that execute instructions. In this level, current measurements are taken when a sequence of instructions is executed. For each instruction a cost is assigned according to the measurements. An extra-cost is also assigned according to the transition from an instruction to another. Tiwari et al. [69] and Konstantakos et al. [70] present power consumption estimator models in this level. Tiwari was one of the first authors to propose this power estimation in processors. Konstantokos defines a power consumption model for an embedded system based on a microcontroller.

- *Functional-Level*: As the previous level, this level is also applied to processing components. Here, the studied component is split in different functional blocks. Thus, the application features that impact the power consumption of the functional blocks activity are defined, such as parallelism rate, clock frequency and data mapping. Once the parameters are specified, their values are changed according to an algorithm that individually stimulates the functional blocks. During the program execution, the current consumed by the component is measured. Regressions are applied to the current consumed according to the features variation thus obtaining the power model of the component. SoftExplorer [71] is a power estimation tool that follows this technique.

4.5. Power Management Techniques

Power management is the use of certain hardware elements to optimize the component power consumption; these can be switches, voltage sources and clock sources where properties such as current, voltage and frequency can be changed. There exist different techniques to reduce the power consumption of systems. Power experts combine these techniques to reduce power in each system state. The combination of such techniques is defined in a functional block called *power manager*. This block synchronizes the implemented control techniques to guarantee the system functionality and optimizing the power consumption. In this section, we describe three of the most important techniques: *Clock-Gating*, *Power-Gating* and *Dynamic Voltage-Frequency Scale*.

4.5.1. Clock-Gating

Clock-gating is one of the first techniques used to reduce dynamic power consumption when a processing component is not active. This technique consists in turning off the signal clock that is received by the component when it is not in use. The power reduction directly affects the registers that belong to the component. These registers are flip-flops with clock inputs. For each clock cycle, the flip-flops consume dynamic power, even when the data input is not changed.

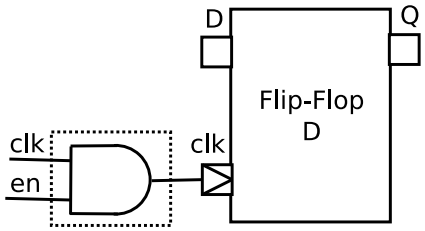


FIGURE 4.3: Example of a clock gating implementation.

Clock-gating can be implemented with a simple AND gate. Figure 4.3 presents a D-type flip-flop where the clock input is controlled by an AND gate. Such a gate allows passing the clock signal only when *EN* input has a logic *1*. This implementation can easily be described in RTL models using the *and* operator. Okuhira and Ishihara [72] report that around 40% of the total power consumption in microprocessors is caused by register circuits. In this percentage, more than 80% of the power consumption is caused by the clock signal transition in the register circuits. In consequence, applying this technique, a significant energy reduction can be made.

4.5.2. Power-Gating

Power-gating is a technique exclusively conceived to reduce static power consumption. This technique can be applied to every hardware component during the time periods when it is not in use. Whereas clock-gating only turns the clock input off, power gating turns the hardware component off when it is not active. The implementation of this technique uses a transistor as power switch to cut off the current supplied to the hardware component. Figure 4.4 presents a power gating implementation. The transistor is fixed between V_{dd} and the component to control the current flow. The switch can also be located from the component to ground or both.

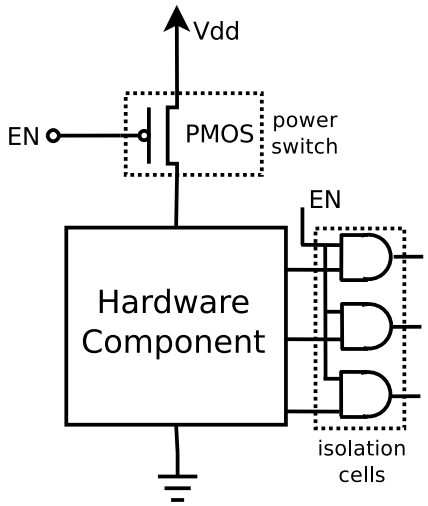


FIGURE 4.4: Example of a power gating implementation.

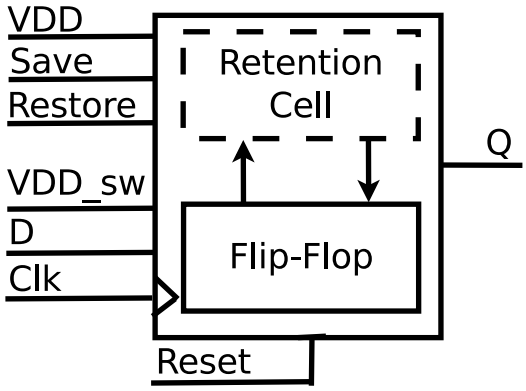


FIGURE 4.5: Example of a retention register.

.

Once the hardware component is turned off, the component outputs can generate undecided signals. Such signals could affect other components that are active during the period the component is gated. To solve this problem, power experts add an *isolation cell* for each component output. Before turning the component off, the isolation cells are activated producing a logic value to the interconnected components. These isolation cells can be implemented by AND gates. Figure 4.4 depicts the implementation of the isolation cells. Each output of the hardware component is connected to an isolation cell, as well as it is connected to the interconnected component inputs.

We can also add another functionality to a power gated component. This functionality is to save the current state of the internal registers before the component is turned off. Once the component is turned on, the saved state is restored and the component can

continue its execution from its previous state. To implement such a functionality, the internal register information can be charged in *retention cells*. Figure 4.5 depicts the retention register structure. This register contain two internal registers: a main register that is identified by a *Flip-flop* and a shadow register called *Retention Cell*. The main register is supplied by *VDD_sw*. In contrast, the shadow register is supplied by *VDD*. *VDD_sw* is the gated power supply. *D*, *Clk*, *Reset* and *Q* are connected to the main register. *Save* and *Restore* are bound to *Retention Cell*. The main register operation is made by the main internal register. Before the power gated component is turned off, an event is sent to *Save* in order to record the information of the main register in *Retention Cell*. Once the register information is saved, the power gated component is turned off and *VDD_sw* does not supply current to the internal main register. Nevertheless, *Retention Cell* is on, because *VDD* is not cut off. Once the gated component is turned on, an event is sent to *Restore* to return the saved information in the internal main register.

The retention functionality takes certain time to save and restore the gated component information. Therefore, this functionality is only used in certain cases.

4.5.3. Dynamic Voltage-Frequency Scale

According to Equation 4.3, the switching power depends on voltage and the transition frequency in a CMOS circuit. In a processing component, if we vary these values according to the component workload, we could significantly reduce its power consumption. However, we can not choose voltage and frequency values randomly. A specific frequency value must correspond to a specific voltage value. Technologically speaking, when we reduce the switching frequency, the voltage level can be reduced until a certain limit. This limit is given by the transistor characteristics and the voltage control implemented. Processors that implement this kind of technique called *operation points* the determined frequency/voltage values. For instance, OMAP3 [73], which is an application processor, has up to six operation points.

To optimally apply this technique, it is necessary to know the workload and the time constraints to be executed. Most of the works apply this technique, taking into account the task execution deadline given by the scheduling policy. According to this deadline,

the operation point is dynamically changed. For instance, Ejlali et al. [74] propose to use DVFS and power-gating techniques to reduce power consumption in redundant-hardware employed in real-time systems. They present a DVFS algorithm according to a common execution deadline for a task sequence, the operation points can be changed according to the time execution of each task that conforms the sequence. Genser et al. [75] propose an algorithm where the operation point changes to execute a task depending on the time execution of the previous one.

This technique can be applied in different zones of a system, so that the system can have multiple voltage level zones. Power experts called these zones *voltage domains*. To guarantee the communication between components of different voltage domains, power experts add *level shifters* to each connector that crosses the voltage domain border. Level shifters level the voltage of a logic signal from a voltage domain to another one.

4.6. Power Design Specification

The elements employed to reduce power consumption were initially designed at transistor-level. The power techniques impact the system functionality, which is usually specified at higher levels than transistor-one. Therefore, the validation of the correctness between power and functional execution is evaluated in the last stages of the system design. In consequence, such elements have begun to be implemented at a higher description level. In this section, we present various languages that have been conceived to define power architectures at three different description levels.

4.6.1. UPF, CPF and IEEE 1801

Hardware description languages (HDLs), like VHDL [76] and Verilog [77], were developed to model the functionality and the time performance of digital systems. However, these languages lack expressivity to implement all the elements that are involved in the power reduction techniques. In 2006, various semiconductor and electronics companies demand to the electronic design automation industry to define an open standard for power specification.

Responding to this need, Accellera Systems Initiative[1], with the support of Synopsys and Mentor Graphics companies, developed a standard named *Unified Power Format* (UPF) [78]. The aim of this standard is to define the elements needed to implement the predominant power reduction techniques at a register transfer level (RTL). The first UPF version was released in 2007 and, in same year, it was transferred to the IEEE in order to create a new IEEE standard. In 2009, IEEE publishes its first power specification standard named *IEEE-1801* [79].

Another power specification standard was also developed this time in 2007 by Cadence. This specification is named *Common Power Format* (CPF) [80]. Such a standard was also transferred to an independent organization called Silicon Integration Initiative (Si2)[2] to continue its development. This organization has produced two new versions. The last CPF version was released in 2011.

The two standards have many concepts in common, however the most notorious is the power intent description complexity. UPF describes the exact physical structure of the power intent in RTL, *i.e.,* it specifies the wires, the ports and the connection between the power elements. In contrast, CPF defines the power concepts that include the basic information to reduce the physical structure complexity. For instance, a *power domain* is associated to a voltage level (*nominal condition* in CPF) in a *power mode* [3]. IEEE-1801 is a new UPF version that unifies the concepts from CPF and UPF in a unique standard. The convergence between the two standards continues and a new IEEE-1801 version, whose release is available since the first semester of 2013, contains more Si2 contributions.

[1] http://www.accelera.org
[2] http://www.si2.org
[3] a *power mode* defines the voltage levels that each power domain must be.

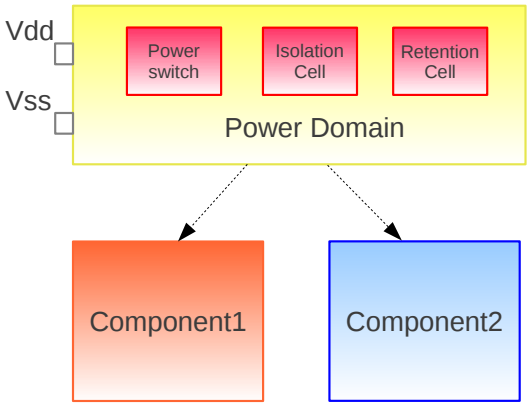


FIGURE 4.6: Example of Power Domain association.

The main concepts of these standards used to define a system power architecture are: *Power Domain*, *Power Switch*, *Level Shifter*, *Isolation Cell* and *Retention Cell*. We have mentioned in Section 4.5 that *Power Switch*, *Isolation Cell* and *Retention Cell* are elements employed to implement power gating technique. Additionally, we have commented that *Level Shifters* guarantee the logic level between voltage domain in DVFS.

The dynamic of the power elements is specified in a *Power State Table* (PST), where the voltage levels are coordinated with the states of *Power Switches*, *Retention Cells* and *Isolation Cells*. By using PST, the designer can verify the synchronization between the power and functional model execution. Nevertheless, not one of these standards specify a way to estimate the power consumption of the hardware components where the power modes are applied.

In IEEE-1801, *Power Domain* is the concept that gathers the elements of a system architecture where the power design is applied. For instance, Figure 4.6 depicts a *Power Domain* that contains a *Power Switch*, a *Retention Cell* and an *Isolation Cell* to provide the hardware elements needed to implement the power-gating technique. Assigning the *Power Domain* to one or more hardware components means that these components are supplied in function to the power domain mode. We remember that the associated hardware components are specified in RTL and these standards are only applied to RTL models.

4.6.2. SystemC

Transaction-Level Modeling (TLM) [81] is a system description level where the communication between components is realized by transactions through channels. SystemC [68] is a C++-based language that implements this modeling level [4]. Such as RTL, TLM has initially been developed to describe functionality and to analyze time performance. However, when the system designers had to model the power characteristics of their models, a new research area was open in TLM to implement these new characteristics to existing TLM models. Mbarek et al. [82] implement the power concepts defined in IEEE-1801 to describe a power architecture in SystemC. They define a framework called *PwARCH*. In this framework, the IEEE-1801 power control elements are defined in a C++ library and can directly be used in the SystemC system model. *PwARCH* also includes a test engine to validate the behavior constraints between power and functional architectures. For instance, if a component is turned off by the power architecture, this component cannot be executed in the functional architecture. Additionally, the authors add a power estimation analyzer that evaluates the power consumption, according to the system execution.

4.6.3. UML

Unified Modeling Language (UML) [21] is a graphical general purpose modeling language developed by the Object Management Group (OMG). UML was initially used mainly to specify object oriented software systems. Nevertheless, this language has been more and more employed to define various kinds of systems, like real-time systems, hardware platforms, control systems, etc. Such specific languages have been built by extending the UML concepts. This extension process is defined in a UML profile. For instance, Modeling and Analysis of Real-Time Embedded Systems (MARTE) [5] is a profile used to model and to analyze real-time systems, and System Modeling Language (SysML) [4] is another profile used in systems engineering.

UML is considered as a language that can be used to specify systems at a higher abstraction level than TLM. In UML, there are some works to specify power concerns: Hagner et al. [83] and Arpien et al. [84] defined UML profiles providing the modeling elements to

[4] SystemC can also implement RTL. This language eases the task to refine the model from TLM to RTL

represent power management techniques and to analyze power consumption. However, these two approaches abstract the elements involved in the power management techniques, without taking into account the impact that causes the control made by these elements on the system behavior.

4.7. Discussion

In the design of low-power systems, we note a clear separation of concerns: on one hand, a power design represented by power characterization and power management techniques, and, on the other hand, the functional design of the system. The power characterization is implemented in certain tools that hide their power models, forcing the user to employ their models and approaches. We also observe that the aim of the power design is to optimize the power consumption, which is one of several non-functional properties defined in a system. By the construction of a power architecture, which controls the power consumption of the system according to its activity, we can identify the impact of the power design on the functionality of the system. The power design alters the functionality of the system, therefore verification process must be applied.

Following the *PRISMSYS* approach, we provide a modeling framework that allows the separation of concerns through *views*. The structure and behavior of the functional design could be defined in a *view*, while the power design could be specified in another *view*. The tools that implement the power management techniques are generally different to the tools that estimate the power consumption. The *PRISMSYS equationalSubView* can be employed to specify the characterization of the power consumption defined by equations. A *StructuralSubView* can be used to define the structure needed to implement the power management techniques. This framework follows a white box approach, *i.e.,* the power design is freely defined and modified by the user. Finally, thanks to the *PRISMSYS correspondence*, we can state the relationship between power and functional design.

4.8. Conclusion

In this chapter, we have introduced a background of the existent concepts and approaches to model and characterize the power consumption in electronic systems. We have introduced the main sources of power consumption in systems that are based on the CMOS technology: dynamic and static power. Afterwards, we have presented how the power consumption is estimated in different abstraction levels. We have continued by describing the power management techniques, defining hardware elements that controls the energy supplied to the hardware components of the system. We have also showed that these power management techniques are represented in different abstraction levels and that the power community is looking for an adequate way to add power-related management in existing system models. We use this background to develop a case study where the *PRISMSYS* framework is employed.

We have pointed out the separation of concerns between power and functional design. Moreover, we have discussed about the division between power characterization and power management, being both parts of the power design, a single expert domain. Even though the power design is separated of the functional, they are associated and one design impacts on the other one.

In the next chapters, we use the power expert domain concepts and technologies to show how the architecture defined in the *PRISMSYS* framework can be used to deal with such problems. The *PRISMSYS* model describes the power expert domain and the other domains that affect the power consumption in a system.

Chapter 5

PRISMSYS Framework for Power-Aware Modeling

Contents

5.1. Introduction 88

5.2. Views 89

5.2.1. Hardware View 89

5.2.2. Application View 92

5.2.3. Power View 93

5.2.4. Clock View 99

5.2.5. Thermal View 102

5.3. Correspondences 105

5.3.1. Allocation 106

5.4. Sub-Correspondences 107

5.5. Conclusion 108

5.1. Introduction

To illustrate the use of *PRISMSYS* framework, we apply it to define the views that impact and characterize the power consumption in embedded systems. To this purpose, we specialize *View* and *SubViewElement* to represent the elements of specific domains according to the expert knowledge. We identify five *views* that are associated with power consumption: *HardwareView*, *ApplicationView*, *PowerView*, *ClockView* and *ThermalView*.

StructuralSubView, *ControlSubView* and *EquationalSubView* are integral parts of the identified views. As such we have explained in Chapter 3, the *controlSubViews* are specified to coordinate the *subViewElements* of each expert domain. Furthermore, they are employed to synchronize the execution between views. In the power-aware model, these *subViewElement* coordination and *view* synchronization rather than fulfilling the functional system requirements, such as executing a task in a processing element, they satisfy the system non-functional constraints, like the maximum system power consumption or the deadline to execute a certain application. These constraints are performed by the synchronization of each expert domain guaranteeing the preservation of the functional requirements. For instance, applying power management techniques, the power experts can reduce the power consumption, while the time performance of task execution and the system functionality are impacted in other expert domains. The *structuralSubView* concepts are specialized defining the concepts commonly employed by experts of each specific domain. The *equationalSubViews* state the equations needed to evaluate the power consumption and temperature of the system components, as well as the values of the non-functional properties employed to calculate such equations, such as frequency and voltage.

To represent the multi-view model for a power-aware system, we build a UML model of the system applying the *PRISMSYS* profile. *View*, *StructuralSubView* and *SubViewElement* stereotypes are specialized according to the specific domain. We also use other MARTE stereotypes to define *subViewElements* that are already specified in this profile.

By applying the *PRISMSYS* framework on this use case, we identify a specific *correspondence* commonly employed in the design of embedded systems. This correspondence

is named *Allocation* and associates *subViewElements* from the application domain (*ApplicationView*) to the execution platform domain (*HardwareView*). *Allocation* is not expressed by the semantics of *Abstraction*, therefore it must separately be specified, specializing the *correspondence* concept from the *PRISMSYS* meta-model.

In this chapter, we begin defining the views that describe the expert domains of the power-aware model. The first two views are the domains that specify the execution platform (*HardwareView*) and the application that is executed on it (*ApplicationView*). *HardwareView* is the backbone of the *PRISMSYS* power-aware model. Therefore, the other views are specified abstracting the elements of this view to define their non-functional properties and other domain elements. Between these derived views, we first specify *PowerView* that characterizes the power consumption properties of the *HardwareView* elements and the power control elements. We continue defining *ClockView* that states the *HardwareView* temporal properties and the control clock signal elements. Afterwards, we specify *ThermalView* that represents the thermal elements associated with the backbone model. This view also characterizes the temperature evolution of the *HardwareView* elements. Finally, we illustrate the use of *correspondences* and *sub-correspondences* for the views defined in the *PRISMSYS* power-aware model.

5.2. Views

In this section, we define the views that describe the expert domains of the power-aware model. For each view, we specify the concepts of its *subViews* specializing the *PRISMSYS* framework meta-model concepts. Afterwards, we represent the view elements with the *PRISMSYS* profile. The elements are specified in the profile either extending them or employing the MARTE stereotypes. Finally, each view is depicted in UML to describe a *PRISMSYS* power-aware model.

5.2.1. Hardware View

We define *HardwareView* as the platform execution of the system. This view plays the role of backbone of the *PRISMSYS* power-aware model. Figure 5.1 depicts the *HardwareView* meta-model. In this figure, the white meta-classes describe the *HardwareView*

concepts. *HardwareView* inherits from *View* and it contains a *structuralSubView* and a *controlSubView*. *HardwareView* does not include an *equationalSubView* because the non-functional properties are described in other views. *StructuralSubView* defines the concepts and relationships needed to describe the hardware architecture. *SubViewElement* is specialized by *HwComponent*, which represents any hardware component defined in the platform execution. For instance, a CPU can be a *HwComponent* whose functional modes (*Free* and *Busy*) are defined. The CPU modes are expressed by the states of a state machine. *ControlSubView* commands the states of the *hwComponents* synchronized with the execution of the other views. For instance, if a task, which is described in another view, *e.g.,* in an application view, is mapped to a CPU, the *controlSubView* of *HardwareView* must be notified when the task is executed. Once the *controlSubView* receives the notification, it sends a control event to the CPU to change its internal mode, *e.g.,* to *Busy* state. The communication between *hwComponents* is represented by the connection of *hwPorts*. A *hwPort* is a specialization of *PropertyPort*. *HwPort* transmits data between *hwComponents* through *wires*, a *Connector* specialization.

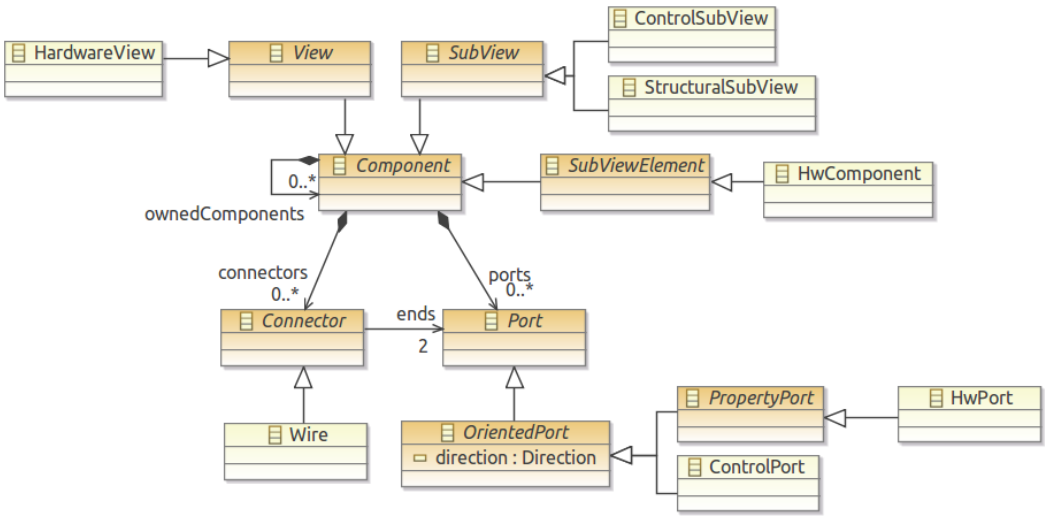


FIGURE 5.1: Hardware View meta-model.

Each new definition of a view is represented in UML by extending the *view* stereotype of *PRISMSYS*. In consequence, *HardwareView* extends the *View* stereotype. In the same way, we extend the other *PRISMSYS* stereotypes according to the expert domain. However, in *HardwareView*, we express *HwComponent* in UML by using the MARTE model

elements that state the hardware structure of a system. Such model elements are specified in MARTE HW_Logical package [5]. Similarly to *HwComponent*, *HwPort* is represented by the MARTE *flowPort* stereotype. The use of MARTE is a simple way to follow the component paradigm employed in *PRISMSYS* while reusing as much as possible concepts from MARTE instead of defining new ones.

Figure 5.2 presents the *HardwareView* of a *PRISMSYS* power-aware model. This view has a *structuralSubView* and a *controlSubView*. *StructuralSubView* includes three *parts* that are *CPU*, *Memory* and *Bus*. We identify each part with the corresponding MARTE stereotype. For instance, *CPU*, which is a *HwComponent*, is stereotyped by *hwProcessor*. The connection hub is a *bus*, so that *memory* and *cpu* can be communicated through *bus*. A *Data* type is assigned to each *HwPort* to define the nature of the data that is transmitted between *hwComponents*. Each *hwComponent* has one or more *controlPorts* to change the internal state of the *hwComponent* behavior. The modes of *cpu* are specified in a state machine. In the same way, the modes of *bus* and *memory* are defined. *ControlSubView* owns the control ports needed to coordinate the *hwComponent* modes, according to the execution of the other views. This *subView* also synchronizes the execution of the *Power* and *Clock* views according to the *ApplicationView* execution. In the figure, we depict that *HardwareView* receives control events from *ApplicationView* to inform that an action is executed. Therefore, *controlSubView* sends control events to its *structuralSubView* according to the events received and it also sends control events to *ClockView* and *PowerView* to synchronize their execution.

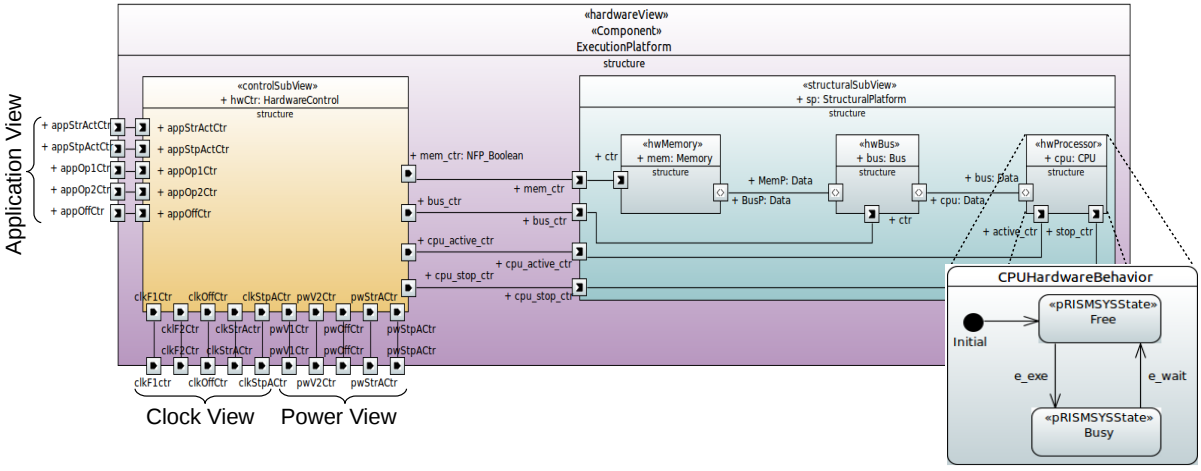


FIGURE 5.2: Hardware View of the *PRISMSYS* power-aware model.

5.2.2. Application View

ApplicationView represents and abstraction of the application that is executed on the execution platform specified in a *HardwareView*. Figure 5.3 depicts the *ApplicationView* meta-model. *ApplicationView* is a *view* that contains two *subViews*: a *controlSubView* and a *structuralSubView*. The *subViewElements* of *StructuralSubView* are specialized by *Actions*. We define that an action represents an atomic element of the application that cannot be refined. *PropertyPort* is specialized in *DataPort*, which means that the information transmitted between *actions* is data. Such ports are bond by *dependency-Connetors*.

ControlSubView coordinates the execution of the actions in the *structuralSubView*. This coordination could depend on control events received from the other views. For instance, if an *action* is executed in *cpu*, *ApplicationView controlSubView* must notify to *HardwareView controlSubView* that an *action* is in execution. Once the action is executed, *ApplicationView controlSubView* informs to *HardwareView controlSubView* that the action was executed. Nevertheless, *ApplicationView controlSubView* could be notified by *HardwareView* that the *hwComponent* where the action is executed has been stopped since the *hwComponent* temperature attained its maximum limit. The control event coordination defined in *controlSubView* is expressed by CCSL [3], which is detailed in Chapter 6.

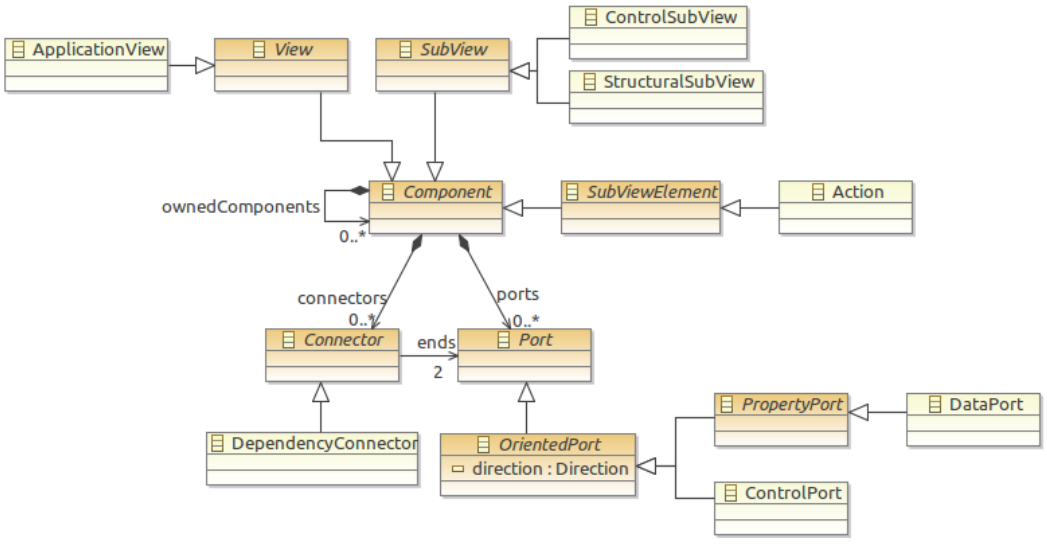


FIGURE 5.3: Application View Meta-model.

In UML, *actions* are defined as *components* that are *parts* of the *StructuralSubView*. *DataPorts* and *DependencyConnectors* are specified by MARTE *flowPorts* and UML connectors, respectively. Figure 5.4 presents the *ApplicationView* of a *PRISMSYS* power-aware model. In this figure, there are two actions: *t1* and *t2*. Each action behavior is represented by a state machine that contains two states: *Execute*, when the action is in execution, and *Stop*, when it finishes or is interrupted. There is a data flow dependency between these actions that is expressed by the connection between *d1* and *d2 flowPorts*. *ControlSubView* commands the execution of the *actions*. Once an action is executed, *HardwareView* is notified to coordinate its *subViewElements* and to inform the other views the performed actions.

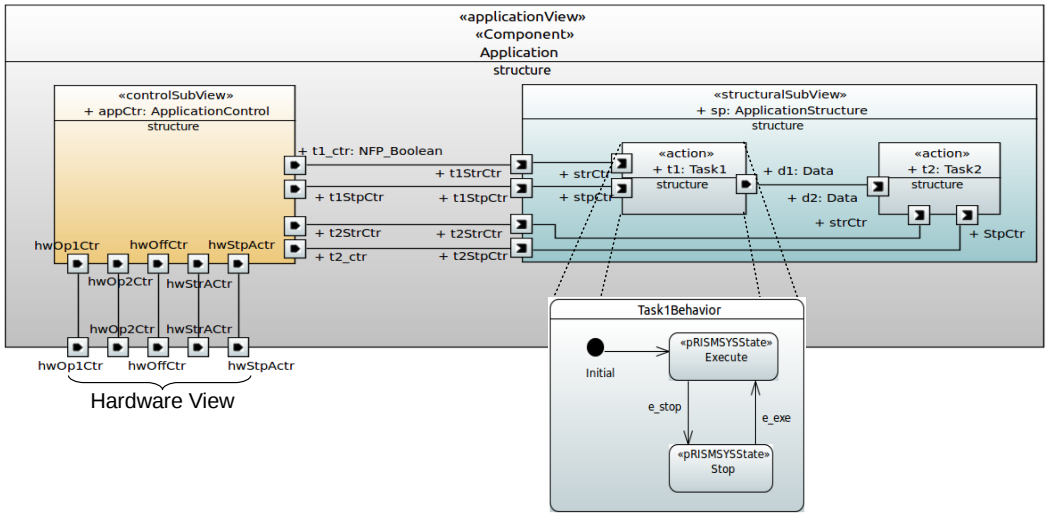


FIGURE 5.4: Application View of the *PRISMSYS* power-aware model.

5.2.3. Power View

The elements of this view intend to supply and control power properties of system components defined in *HardwareView*. These control elements implement the power management techniques that have been described in Chapter 4. Power experts build their power model without modifying *HardwareView*, which is the objective of the multi-view modeling approach. The elements from these views are inspired by the concepts defined in the IEEE-1801 [79] and CPF [80] languages.

Figure 5.5 depicts the specialization of the *PRISMSYS* framework concepts to define the power domain concepts. *PowerView* contains the three *subViews* previously defined in the *PRISMSYS* framework: a *structuralSubView*, an *equationalSubView* and a *controlSubView*.

The *StructuralSubView* owns the following *viewElements*: *voltageSources*, *powerDomains* and *poweredElements*. *PoweredElement* defines the power features of the *viewElements* specified in *HardwareView*. In other words, *PoweredElement* is the abstraction of a *HwComponent* from a power point of view. A *poweredElement* owns a *supplyPort*. This port receives a voltage value from a *powerDomain* or from a *voltageSource*. *SupplyPort* specializes *PropertyPort* to represent the transmission of voltage values, *i.e.,* a power-specific feature. A *poweredElement* also possesses *controlPorts* to change the active state of its state machine. Such a state machine expresses the power consumption modes of a *HwComponent*.

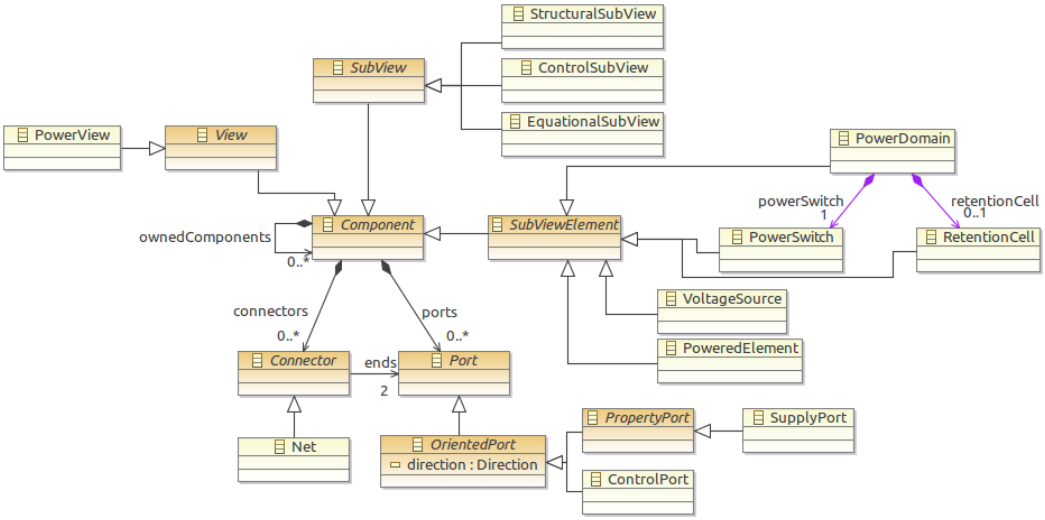


FIGURE 5.5: Power View Meta-model.

VoltageSource represents the functionality of a power source. This power source supplies current to the hardware components using different voltage levels. In the *PowerView* definition, *VoltageSource* generates different voltage values to implement a part of power management techniques such as DVFS [85]. A *voltageSource* owns a *supplyPort* to transmit the voltage values to a *powerDomain*, or directly to a *poweredElement*. Changes in voltage are specified by the *subViewElement* state machine. The states represent the different voltage levels provided by the voltage source. *VoltageSources* also have

controlPorts that receive events from the *controlSubView* to fire transitions between states, changing the generated voltage level.

A *powerDomain* controls the voltage value transmission from a *voltageSource* to a set of *poweredElements*. A *PowerDomain* owns two kinds of *subViewElements*: *PowerSwitch* and *RetentionCell*. A violet composite association is depicted in Figure 5.5 to illustrate which *subViewElements* are owned by *PowerDomain*. However, this association is not defined in the original meta-model, because it is explicitly defined in the *SubViewElement* definition (the self-contained association inherited from *Component*). *PowerSwitch* cuts the current that is supplied to a *poweredElement* when it is not in use,*i.e.,* the voltage applied to the target *poweredElement* is $0V$. A *PowerSwitch* contains two *supplyPorts* and two *controlPorts*. The first *supplyPort* receives voltage value from a *voltageSource* and this value is sent to the connected *poweredElements* according to its active state (On or Off) through the second *supplyPort*. *ControlPorts* receive the control events to change the active state. *RetentionCell* saves information of the *ViewElement* associated with the supplied *PoweredElement* before this element is turned off. Meanwhile the element is turned on, the *RetentionCell* restores the saved information. *PowerDomain* also owns *controlPorts* and *connectors* that transmit the control events sent from the *controlSubView* to its internal *subViewElements*. *Connector* is specialized in *Net* to be compatible with the power expert domain. Additionally, a *powerDomain* has *supplyPorts* to receive and to transmit voltage values. Using *powerSwitches* and *retentionCells*, we can implement the power-gating technique [56]. Low abstraction level elements from IEEE 1801 [79] and CPF [80], like *isolation cells* and *level shifters*, are not specified in this thesis because using the MDE transformation technique, they can be automatically generated from the *PowerView* model definition according to the *powerSwitches* and the *voltageSources* that supply the *poweredElements*.

Each *subViewElement* of the *structuralSubView* contains its *controlPorts* that are exposed on the *structuralSubView* (see Figure 5.6). These *controlPorts* are connected to the *controlSubView controlPorts*. Such a *controlSubView* coordinates the execution of the mentioned power *subViewElements* according to control events received from *HardwareView*. Additionally, *controlSubView* receives a clock signal (through *ctrStepCtr*) from *ClockView* to evaluate the active power consumption equation at each tick of this clock in the *equationalSubView*.

Whereas *HardwareView* has a predefined representation of their *subViewElements* in UML and MARTE, *PowerView* does not have it. In consequence, the *SubViewElements* of *PowerView* must specialize the stereotypes of the *PRISMSYS* profile. Similarly to the other specific domains, *PowerView* is specified as a stereotype that inherits from the *View* stereotype. *PowerDomain* and *VoltageSource* are also defined as stereotypes inheriting the *SubViewElement* stereotype. The *SupplyPort* nature and certain *PoweredElement* property types are specified by MARTE *NFP*[1] types. *NFP* follows the International System of Units standard (SI) [86]. For instance, a typical property in the power view is *voltage*. This property is expressed in function of the unit *Volt*, in short, *V* and its value.

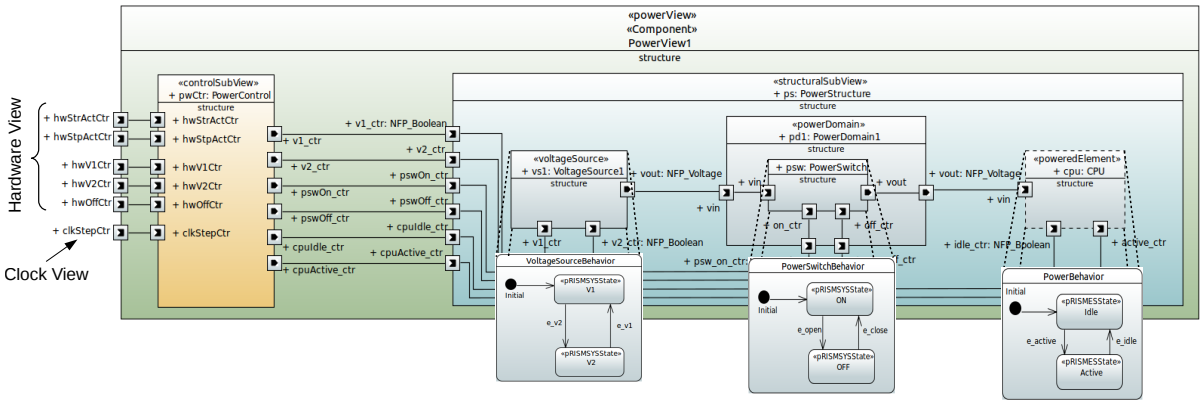


FIGURE 5.6: Power View of the *PRISMSYS* power-aware model without including its *equationalSubView*.

Figure 5.6 represents part of the *PowerView* of a *PRISMSYS* power-aware model in UML. The *structuralSubView* defines three parts that represent power *subViewElements*: *vs1*, *pd1* and *cpu*. *vs1* and *pd1* are respectively instances of *VoltageSource1* and *PowerDomain1* components. These components are stereotyped by *VoltageSource* and *PowerDomain* extending the *PRISMSYS SubViewElement* stereotype. *PowerDomain1* owns a *PowerSwitch* instance (*psw*) to control the current flow from *vs1* to *cpu*. In contrast to *VoltageSource* and *PowerDomain*, *PowerSwitch* is a component predefined in a UML *PRISMSYS* library that is imported to be reused in this model. This library also includes the NFP types that are not included in the MARTE library, like voltage, current and temperature. *Cpu* is a *poweredElement* whose stereotype also extends the *SubViewElement* stereotype. *SupplyPorts* are represented by MARTE *flowPorts* in the

[1]Non-Functional Property

figure. To specify their voltage nature, a *NFP_Voltage* type is assigned to these ports. Thanks to the flow port properties, the data flow direction is defined. For instance, the *vout flowPort* in *vs1* is configured as output, *i.e.,* the voltage value generated by *vs1* is shared with its environment, in this case with *pd1*.

Each *subViewElements* defined in the *PRISMSYS* power-aware model expresses its behavior by a state machine in Figure 5.6. *Cpu poweredElement*, which is a *HwComponent* in the *HardwareView*, owns a power behavior whose modes are: *Idle*, to express that *CPU* is consuming *static power*, and *Active*, to describe that *CPU* is consuming *dynamic power*. *VoltageSource* behavior (*vs1*) contains two states: *V1* and *V2*. Each state represents a specific voltage level that is defined in the *equationalSubView*. The *powerSwitch* behavior is expressed by two states that represent the powering on (state *ON*) and the cutting off (state *OFF*) of the current from *voltageSource* to the *cpu poweredElement*.

ControlSubView are also represented in Figure 5.6. This *subView* receives control events from *HardwareView* in order to coordinate the power *subViewElements* behavior defined in *structuralSubView* according to the *HardwareView* execution. *hwStrActCtr* and *hwStpActCtr* ports receives the events indicating that an *action* is executed or stopped. *hwV1Ctr*, *hwV2Ctr* and *hwOffCtr* collect the events to change the *cpu* operation points. According to the received events, the *subViewElement* control events are generated.

The execution of the *ControlSubView* must fulfill the system requirements. A system requirement focused on power consumption could be: *the CPU must be ON when an action is executed*. In this example, there are involved three views: *HardwareView*, where the CPU component is defined, *ApplicationView*, where the *actions* are executed in the CPU and *PowerView*, where the CPU power control is described. In this case, we only focus on the power control. To fulfill the mentioned system requirement, we must synchronize the execution to turn CPU on, if it is *OFF*, and the *actions* execution. Therefore, we can specify these executions through the following steps:

1. *PowerView ControlSubView* receives a control event from the *HardwareView ControlSubView* that *cpu* is executing an actions, *i.e.,* it is in mode *Busy*.

2. *PowerView ControlSubView* sends a control event to turn the *powerSwitch* on in order to supply current to the *cpu poweredElement*.

3. *PowerView ControlSubView* sends a control event to change the *cpu* power mode to *Active*.

These steps can be defined by the specification of the relationships among control events. Therefore, we can use CCSL [3] to this specification. Such specification is stated in Chapter 6.

We characterize the power consumption of the *poweredElements* by means of *equationalModels* defined in the *equationalSubView*. These *equationalModels* include the equations that define the power consumption according to the *poweredElement* behavior. We also specify other *equationalModels* that specify constant values. Such values are associated with the power consumption equations. We do not extend the concepts previously defined in the *EquationalSubView* meta-model of *PRISMSYS* framework, because the equation representation is used in multiple domains, and the power consumption domain is not an exception.

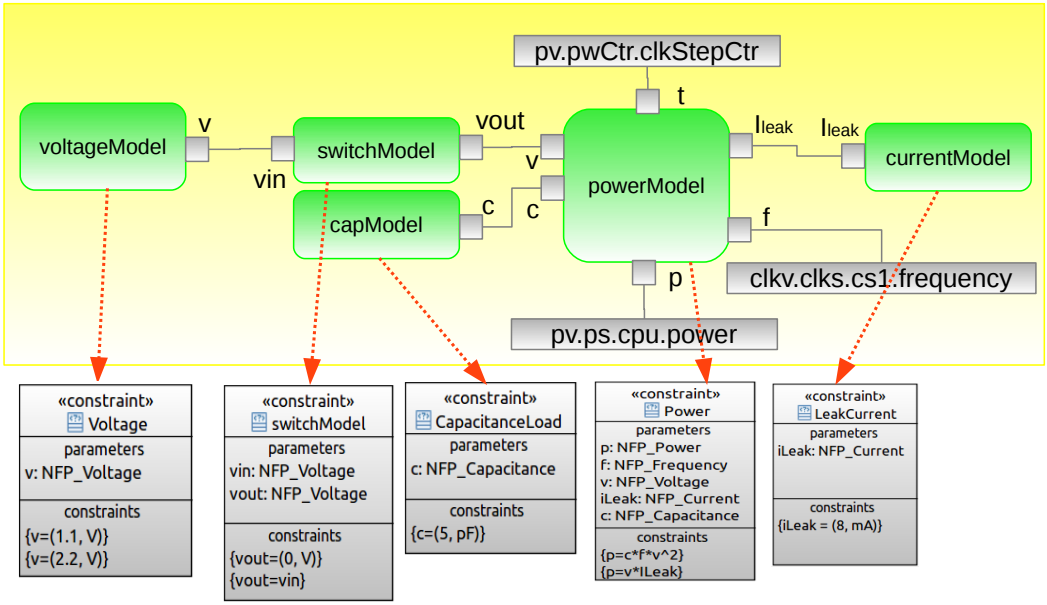


FIGURE 5.7: *EquationalSubView* of *PowerView*.

Figure 5.7 depicts an example of this representation to evaluate power consumption of *cpu*. We employ the SYSML parametric diagram to represent this *subView*. In this figure, there are five *equationalModels* defined by *constraintBlocks*: *voltageModel*, *switchModel*, *capModel*, *currentModel* and *powerModel*. Each *equationalModel* defines

its parameters and equations. For instance, *voltageModel* specifies a v parameter whose type is *NFP_Voltage*, *i.e.,* this parameter is a voltage type. This *equationalModel* also owns two equations that assign a constant value to the v parameter: $v = (1.1, V)$ and $v = (2.2, V)$. The *NFP* types follow the *Value Specification Language (VSL)* datatype syntax defined in MARTE. Such a datatype is a 2-tuple where the first element is the value and the second one is the *NFP* unit. For instance, in the first equation 1.2 represents the value and V the voltage unit.

In *PowerView*, the main *equationalModel* is *powerModel*. It characterizes the dynamic and static power consumption equations of the *cpu poweredElement*. This *equationalModel* depends on the values given by other *equationalModels* defined in this *subView*. Therefore, according to the active values in the other *equationalModels* and the active *powerModel* equation, the power consumption is evaluated. The evaluation of the active power equation is executed by the clock signal received on *clkStepCtr*. *PowerModel* is also relied on the *frequency* parameter. *Frequency* value is shared from the *ClockView equationalSubView*. *ClockView* specifies the temporal features of the system. The details of *ClockView* are described in the following section.

5.2.4. Clock View

ClockView specifies the elements that provide and control the clock signals. Such clock signals activate the *HardwareView* elements and give temporal properties to the actions executed in these elements. Likewise *PowerView*, we specialize the *PRISMSYS* framework concepts to define the *ClockView* elements. Figure 5.8 presents the meta-model of *ClockView*. *ClockView* has the three identified *subViews* of the *PRISMSYS* framework. Nevertheless, we only specify the *subView* elements needed to evaluate power consumption. The *structuralSubView* contains equivalent concepts to *PowerView structuralSubView*, but the nature of the non-functional properties specified and controlled is different. For instance, *ClockPort* and *PowerPort* are concepts derived from *ProperyPort*. Whereas *PowerPort* represents a power nature property, *ClockPort* expresses a timing nature, *i.e.,* the non-functional property transmitted by this port is a clock signal. Another example is *ClockSource* that is a clock signal generator. The *ClockSource* states identify the frequency of the clock signal transmitted by *ClockPort* instead of

a voltage value change such as *VoltageSource* performs. *ClockSwitch* and *ClockedElement* is the *ClockView* representation of *PowerSwitch* and *PoweredElement*, respectively. However, *ClockSwitch* controls the clock signal transmission from a *ClockSource* to a *ClockedElement*. *ClockedElement* is the abstract time performance representation of a *hwComponent* and defines the timing properties of the abstracted *hwComponent*.

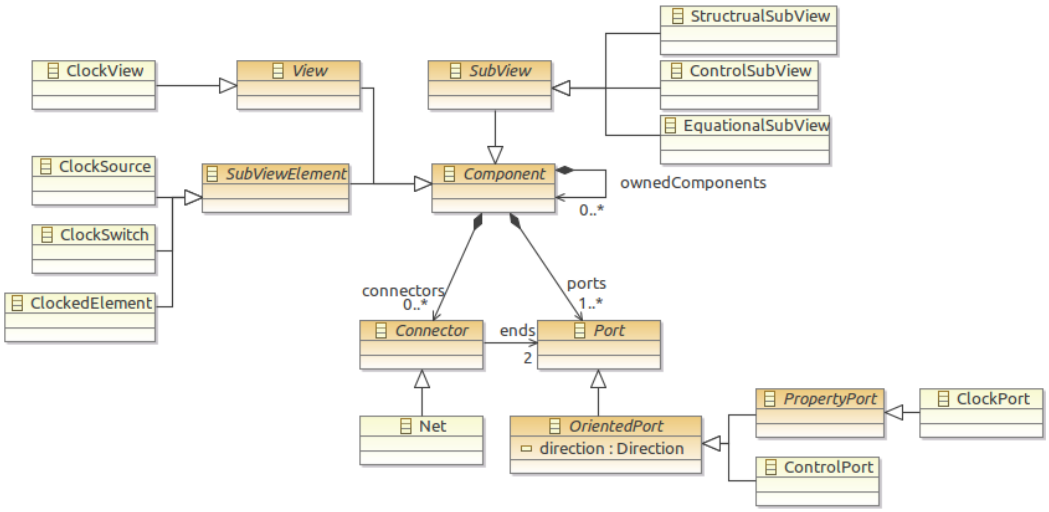


FIGURE 5.8: Clock View Meta-model.

ClockSources and *clockSwitches* affect the power consumption by changing the clock frequency or cutting the clock signal off. Therefore, there is a coordination between the *ClockView controlSubView* and the *controlSubViews* of the other views. For instance, if an *ApplicationView* action must be executed before a specific deadline, *ClockView controlSubView* could change the frequency clock in order to reach the required deadline. This frequency change depends on the voltage level, therefore the *PowerView controlSubView* must also be notified in order to change the voltage to the specified frequency. As well as other views, the *controlSubView* is specified by using CCSL. This specification is detailed in Chapter 6.

Similarly to *PowerView*, the *subViewElements* of *ClockView* are implemented in UML by specializing *SubViewElement* stereotypes of the *PRISMSYS* profile. Figure 5.9 depicts the *ClockView* of a *PRISMSYS* power-aware model represented in UML. This view contains two *subViews*: a *structuralSubView* and a *controlSubView*. The *structuralSubview* is composed by four parts: *ClockSource1* and *ClockSource2* instances (*cs1* and *cs2*), a *ClockSwitch* instance (*csw*), and a *ClockedElement* instance (*cpu*). *cs1* is a *ClockSource*

that supplies a clock signal through *clkout* port. This port not only is stereotyped by the MARTE *FlowPort*, but also by MARTE *Clock*. In consequence, the time properties of the clock signal, like frequency, can be specified by CCSL and the clock signal behavior can be simulated in TIMESQUARE. *cs2* is another *ClockSource* that generates a clock signal with a fixed frequency. This signal is shared with the other *PowerView* and *ThermalView* to coordinate the equation evaluation in their *equationalSubViews*. *cs2* sends the clock signal to *controlView*, and this sends two clock signals to *PowerView* and *ThermalView*, whose ticks are coincident with the *cs2* clock. *cpu* is the timing domain representation of the *HardwareView cpu*. The *structuralSubView* elements define their behavior by state machines. The *cs1* states represent the change of frequency of the generated clock signal to *csw*. *cs2* owns only one state where the clock frequency is fixed. The *csw* states specify the action to cut off or to transmit the clock signal to the *clockedElement*. The *clockedElement* state machine is specified by two states: *Run*, to express that *clockedElement* is executing a sequence of instructions per clock cycles, and *Stop*, to indicate that *cpu* stops the instruction execution.

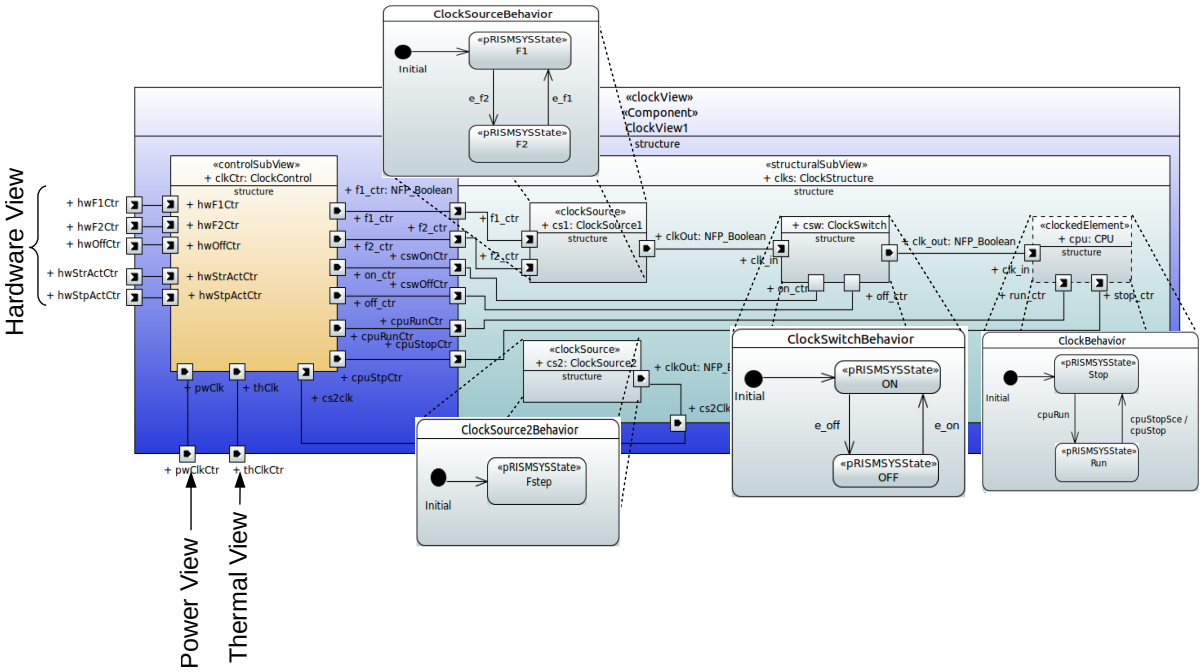


FIGURE 5.9: Clock View of the *PRISMSYS* power-aware model without including its *equationalSubView*.

The *equationalSubView* is also defined in *ClockView*. Figure 5.10 depicts a parametric diagram that represents the *equationalModels* of the *ClockView subViewElements* for

the *PRISMSYS* power-aware model. These elements are associated with the *cpu* power consumption defined in *PowerView*. Furthermore, we define a clock signal to evaluate the equations of the other *equationalSubViews*. In the diagram, there are three *equationalModels*: *frequencyModel1* and *frequencyModel2*, to respectively set the frequency of the *cs1* and *cs2* clock sources, and *switchModel*, to kill the clock signal or to transmit it to *cpu*. *frequencyModel1* and *switchModel* share a frequency parameter represented by the binding connection between *f* and *f_in*. *SwitchModel* is also connected to *clkv1.clks1.cs1.frequecy*, which is the frequency property defined in the clock source *cs1*. In the same way, *frequencyModel2* is linked with *clkv1.clks1.cs2.frequecy*. Afterwards, *cv.clks1.cs1.frequency* and *clkv1.clks1.cs2.frequecy* are shared with the *PowerView equationalSubView*. The former to provide a frequency value in order to evaluate the power consumption of the *cpu*. The latter to generate a clock signal whose instants causes the evaluation of the power consumption and the temperature progression.

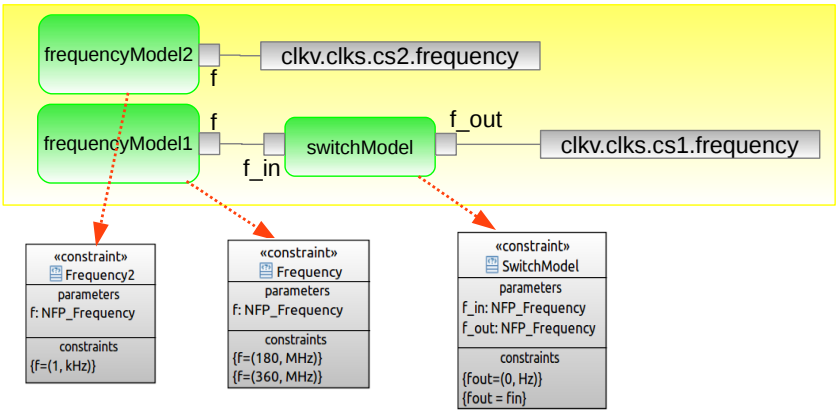


FIGURE 5.10: Equational Sub-view of Clock View.

5.2.5. Thermal View

ThermalView describes the domain specified by thermal experts to represent thermal features of the *HardwareView subViewElements* and to define *subViewElements* of this domain such as heat sinks. Figure 5.11 presents the thermal view meta-model. Similarly to *PowerView* and *ClockView*, *ThermalView* inherits from *View*. The *ThermalView structuralSubView* owns two types of *subViewElements*: *ThermalElement* and *HeatSink*. The former is the thermal abstraction of a *hwComponent*. The latter represents the element that helps to dissipate the heat. This heat dissipation causes a temperature decrease. A *heatSink* is connected to a *thermalElement* by a *junctionPoint*. *JunctionPoint*

is the specialization of *Connector* in *ThermalView*. *ThemperaturePort* inherits from *PropertyPort* to represent the temperature nature transmitted between *ThermalElement* and *HeatSink*.

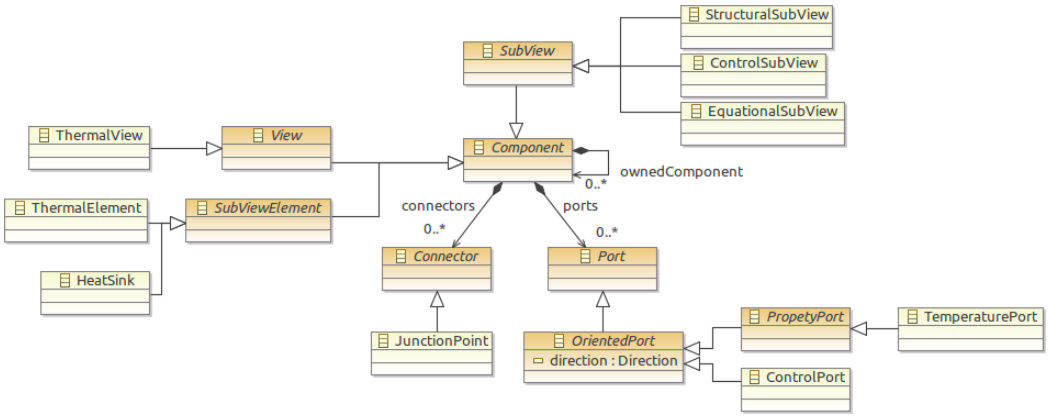


FIGURE 5.11: Thermal view Meta-Model.

ThermalView contains the three *subViews* specified in the *PRISMSYS* framework. *StructuralSubView* and *controlSubView* are depicted in Figure 5.12, which is a UML representation of *ThermalView*. In the *structuralSubView*, we define a *thermalElement* named *cpu*. It is the thermal abstraction of the *cpu* defined in *HardwareView*. The thermal behavior of *cpu* is specified by a state machine with a single state. This state represents the *cpu* temperature behavior. The *cpu thermalElement* transmits the temperature value to the *controlSubView* named *T*. Unlike the *controlSubViews* defined in the other views, *T* specifies its behavior by a state machine in a *controller*. Such state machine contains two states: *HIGH*, to represent that the *cpu* temperature rises to its limit, and *LOW*, to express that the temperature is in a typical operation temperature. The transitions between states contain *guards*, where the *cpu* temperature is evaluated in order to fire the transition and to change the control state. Once a guard is fired, an event is sent to the *controlSubView* of the *PowerView*. This event commands to turn *cpu* off to fall its temperature. When the temperature descends to $50°C$, *ThermalView controlsubView* allows to *PowerView* turning *cpu* on sending an event to turn *cpu* on. To evaluate the temperature property, a clock signal is sent from *ClockView* to *ThermalView controlSubView*. This clock is received on the *clkIn* port.

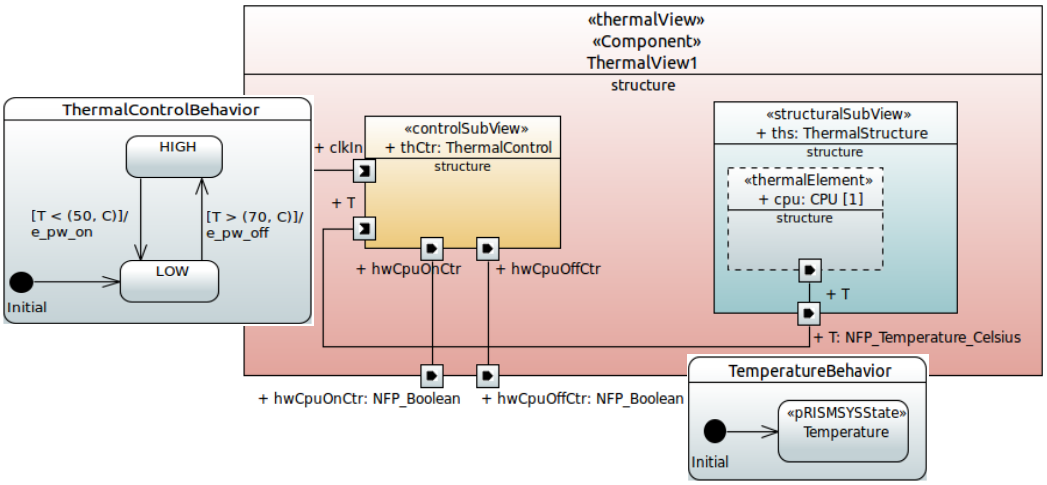


FIGURE 5.12: Thermal view of the *PRISMSYS* power-aware model.

The *Temperature* state defined in the *cpu thermalElement* is characterized by an equation in the *equationalSubView*. We use the Compact Thermal Model (CTM) [43] to express the thermal equation of the *HardwareView* elements. Figure 5.13 depicts the *equationalSubView* of the *ThermalView*. In this figure, *TempModel* defines the temperature evolution through time. This *equationalModel* owns a first-order differential equation whose parameters are thermal properties of the hardware component (*cTh* and *rTh*), *temp_env* is a constant temperature, *p* is evaluated in *powerView* and imported through *ParameterConnectors* (*pv.ps.cpu.power*) and *t* is generated from *ControlView*, transmitted through *DataConnectors* to the *controlSubView* of *ThermalView* (*thv.thCtr.clkIn*).

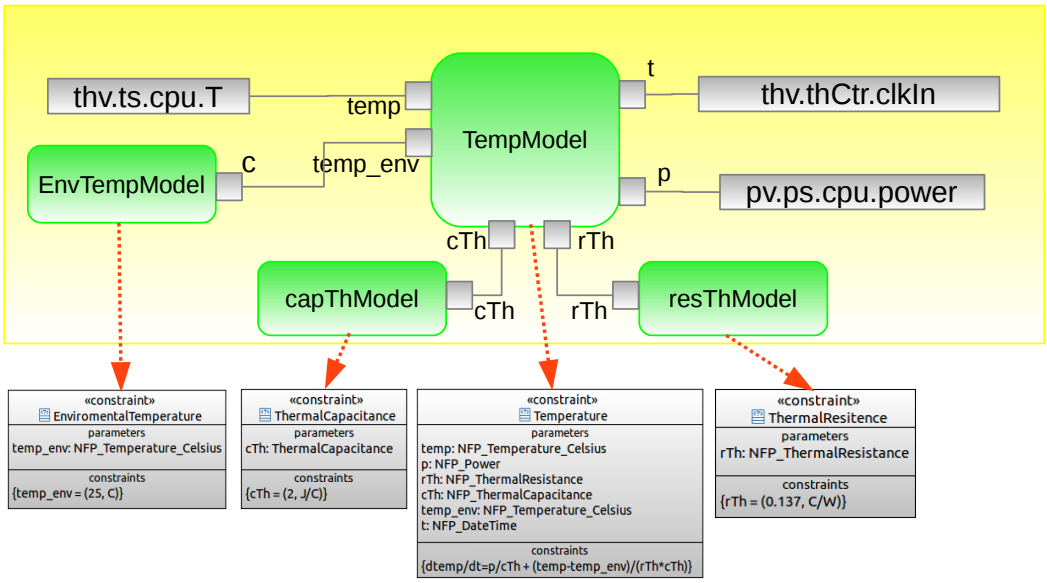


FIGURE 5.13: Equational Sub-View of Thermal View.

5.3. Correspondences

In the specification of the *PRISMSYS* power-aware model, we use the *correspondences* defined in the *PRISMSYS* framework to state the relationships between views. *Abstraction* is one of the first correspondence that we can identify. Figure 5.14 presents an example of the *abstraction* use. *cpu*, which is a *hwElement* defined in *HardwareView* is abstracted by the *cpu poweredElement*. In this example, the *cpu* power representation specify the properties and behavior associated with *PowerView*. Similar correspondence use is defined for *clockedElement* and *thermalElement*.

In the same figure, we depict the *ControlConnector* Correspondence. This correspondence is specified between the *hwV1Ctr*, *hwV2Ctr* and *hwOffCctr controlPorts* and the *pwV1Ctr*, *pwV2Ctr* and *pwOffCtr controlPorts*, respectively. For instance, if the *cpu HwElement* enters to *Busy* mode, *controlSubView* sends a control event to *PowerView* in order to inform that the *cpu* power abstraction must change is power mode (to *Active*).

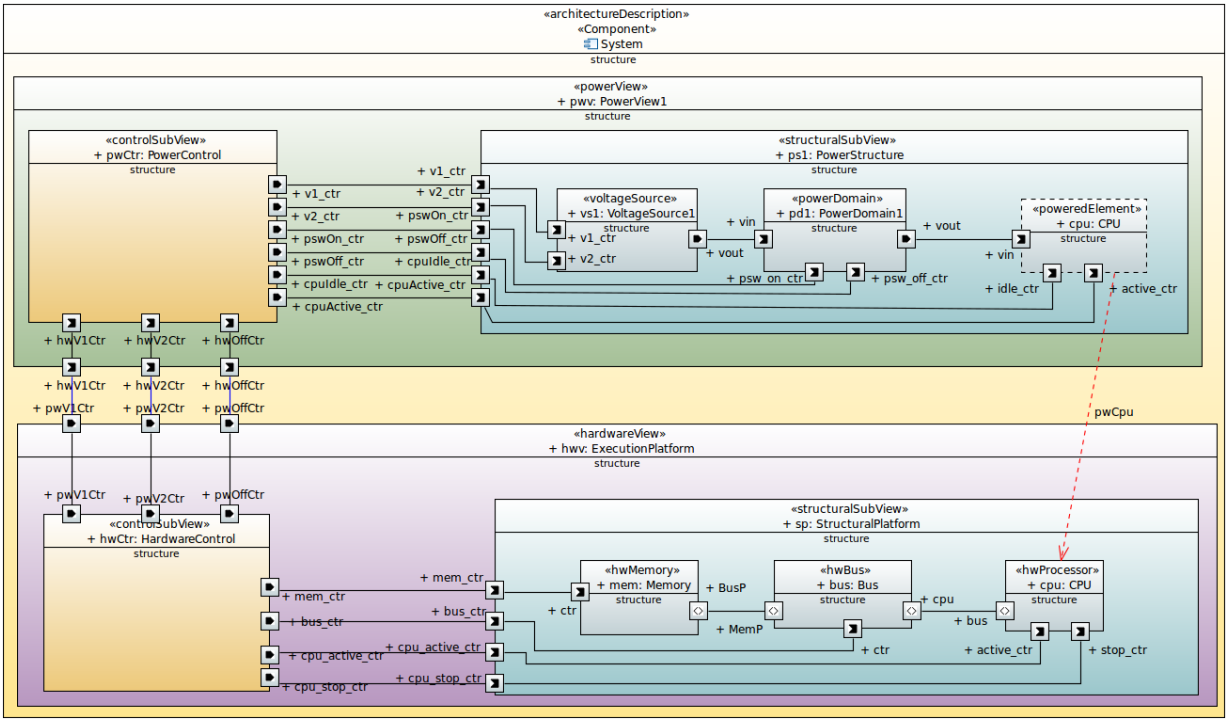


FIGURE 5.14: Example of the *Abstraction* and *ControlConnector* correspondences between *PowerView* and *HardwareView*.

We also employ the *parameterConnector correspondence* to import the property value evaluated in other expert domain. For instance, in Figure 5.13, *TempModel* needs the *power* value that is evaluated in *PowerView*. Therefore, by using the SysML path name dot notion (see *pv1.ps1.cpu.power* parameter), we import the power parameter from the *PowerView equationalSubView*. This imported parameter represents a *parameterConnector correspondence* between *PowerView* and *ThermalView*.

5.3.1. Allocation

We identify a *correspondence* that is commonly employed to associate an *action* from *ApplicationView* to a *hwComponent* in *HardwareView*. This association is named *Allocation*. This correspondence is only used between *application* and *hardware* views. The semantics of *Allocation* is to map *actions* to an *hwComponents*. The mapping type is a spatial distribution, *i.e.,* an *action* is executed in the associated *hwComponent*.

Figure 5.15 depicts an example of allocation representation in UML between *ApplicationView* and *HardwareView*. In *ApplicationView*, *t1* and *t2* are allocated to *cpu*, *i.e.,* the

execution of *t1* and *t2* is performed in *cpu*. This correspondence also gives the possibility to assign multiple *hwComponents* to execute and store an *ApplicationView* action. We reuse the *Allocate* association defined in MARTE to represent this correspondence. The nature property employed in *Allocate* is *spatialDistribution* to maintain the defined correspondence semantics.

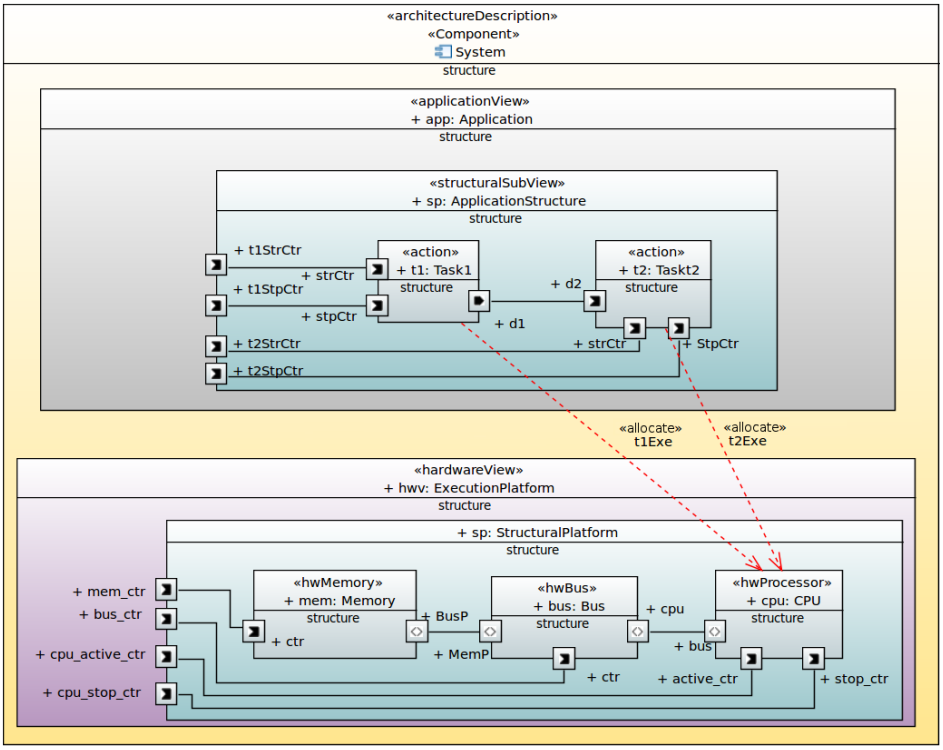


FIGURE 5.15: Example of *Allocation* correspondence between *ApplicationView* and *HardwareView*.

5.4. Sub-Correspondences

The *PRISMSYS* power-aware model also applies *subCorrespondences* specified in the *PRISMSYS* framework. Figure 5.16 presents the use of *characterization* and *equivalence subCorrespondence* in *PowerView*. Each state of the *subViewElements* are associated with one or more equations. For instance, the *idle* state is associated with the static equation $p = v * Ileak$. This state is also associated with $Ileak = (8, mA)$ in order to activate the static current employed in the static equation. The *equivalence subCorrespondence* is expressed by a parameter that import a property from a *subViewElement* by using the SYSML path name dot notion, such as *pv.ps.vs1.vout* and *pv.ps.pd1.psw.vin*

parameters. The *binding* among *pv.ps.vs1.vout*, *pv.ps.pd1.psw.vin*, *v* and *vin* expresses the *equivalent subCorrespondence* between the parameters defined in *equationalSubView* and *properties* of *subViewElements*.

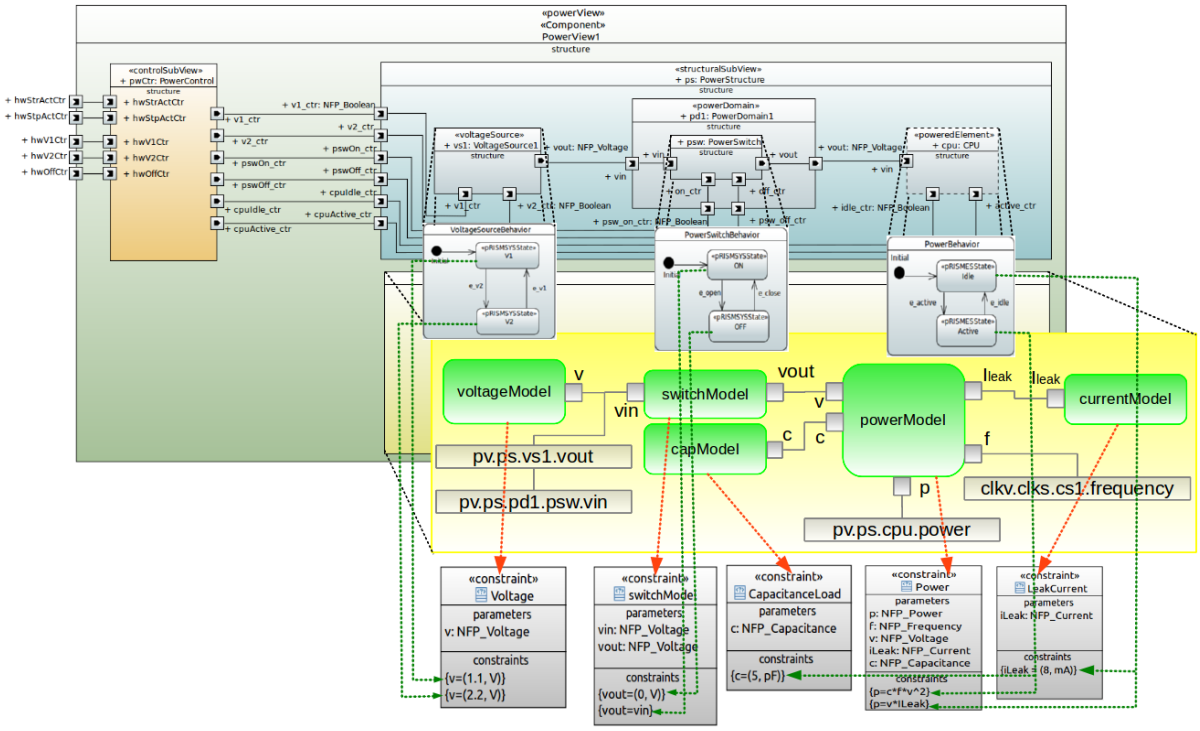


FIGURE 5.16: Example of *Characterization* sub-correspondence in *PowerView*.

Summarizing the *PRISMSYS* power-aware model, Figure 5.17 presents the big picture of its five defined views.

5.5. Conclusion

In this chapter, we have presented the syntax definition of the *PRISMSYS* power-aware model extending the *PRISMSYS* framework concepts. We have identified the expert domains that evaluate and impact the power consumption of a system. For each domain, we define a meta-model where the concepts commonly employed are represented. We depict the views by using the UML representation.

In the next chapter, we implement the execution semantics of the *PRISMSYS* power-aware model to be simulated. Such a simulation allows observing the evolution of the

system power consumption and temperature through time. We also propose an power consumption analysis by transforming the *PRISMSYS* power-aware model to an specific analysis tool, such as *Aceplorer* [8].

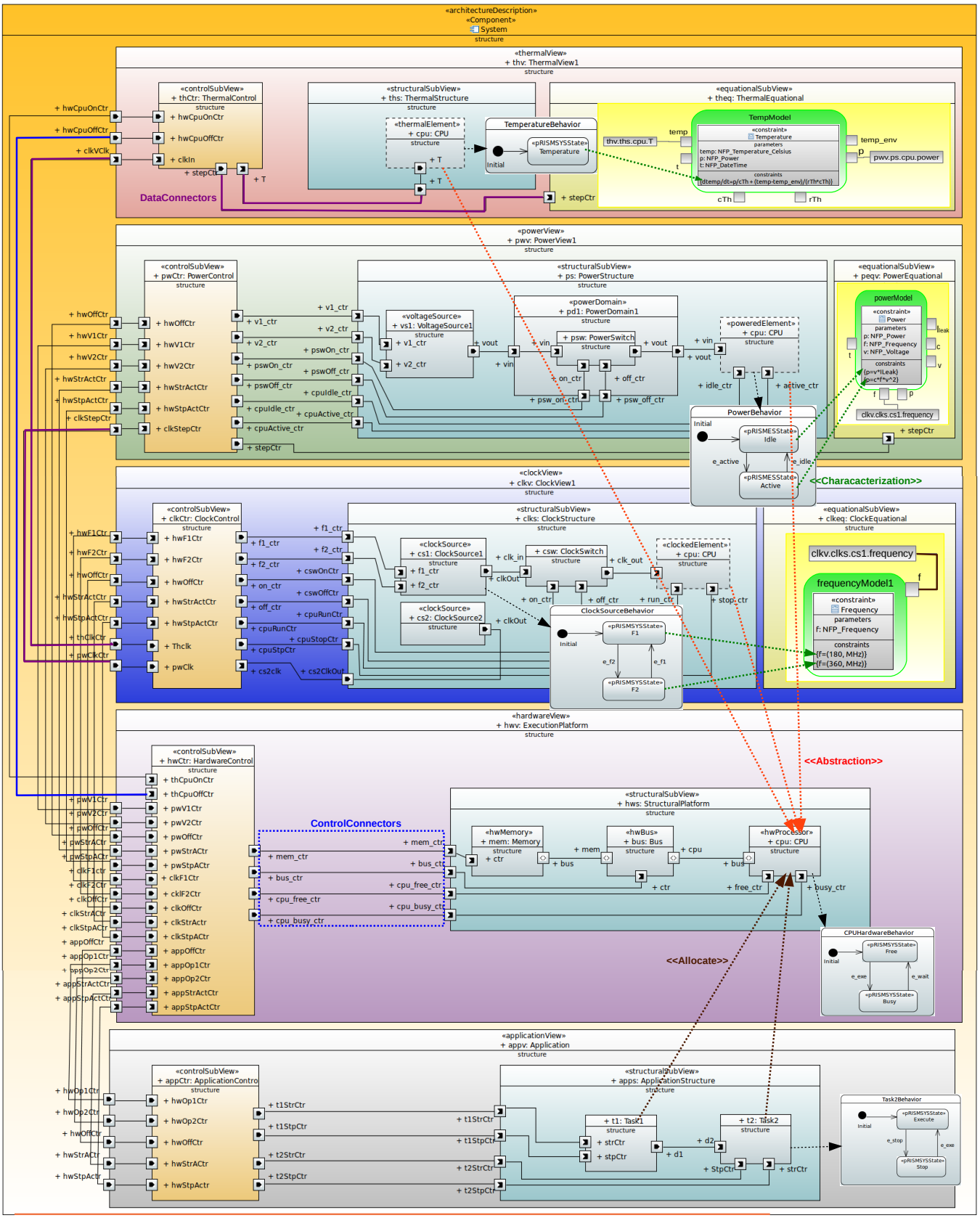


FIGURE 5.17: *PRISMSYS* Power-Aware Model Overview.

Chapter 6

PRISMSYS Power-Aware Model Analysis

Contents

6.1. Introduction 112

6.2. PRISMSYS Power-Aware Model Simulation 112

6.2.1. Scilab Solver 113

6.2.2. The PRISMSYS Power-Aware Model Scenario 115

6.3. PRISMSYS Power-Aware Model Analysis in *Aceplorer* . . 135

6.3.1. Transformation Overview 136

6.3.2. *Aceplorer* Domain Model 137

6.3.3. PRISMSYS to *Aceplorer* Transformation 139

6.3.4. *Aceplorer* Code Generation 140

6.3.5. Test Scenario Generation 140

6.4. Conclusion 144

6.1. Introduction

The specification of the *PRISMSYS* power-aware model is completed by the definition of the execution semantics. Such a semantics allows the analysis of the non-functional properties defined in the model through time. This analysis is possible, once the model is simulated and the properties are evaluated through time.

We specify the execution semantics of the *PRISMSYS* power-aware model by employing the *PRISMSYS* execution semantics defined in Chapter 3. We additionally define the *controlSubView* execution semantics of each views by only using CCSL expressions. The *controlSubView* execution definition is bound with the clocks described in the *PRISMSYS* execution semantics. Moreover, The *controlSubView* execution expresses the scenario to synchronize the execution of the views. We support the *controlSubView* execution specification by employing the UML sequence diagram to define the interactions among the *controlSubViews* and among their *subViewElements*. For each view, we define a sequence diagram to illustrate the *controlSubView* interaction. Afterwards, we specify the CCSL expressions that specify the interactions represented in the sequence diagrams.

Once the semantics of the *PRISMSYS* power-aware model is defined, it is simulated in TIMESQUARE. Nevertheless, the evaluation of the equations (*e.g.,* power and temperature equations) must be performed in another tool. We choose as equation solver *Scilab* [7], an open source tool for numerical computation. Thus, we develop a *"connector"* between TIMESQUARE and *Scilab* to evaluate the active equations, regarding TIMESQUARE simulation. We named *Scilab Solver* to this *connector*.

In this section, we simulate the evolution of power consumption and temperature in a *cpu* specified in the *PRISMSYS* power-aware model. In addition to the simulation, we propose to analyze the *cpu* power consumption by transforming the *PRISMSYS* power-aware model to *Aceplorer*.

6.2. PRISMSYS Power-Aware Model Simulation

In this section, we explain how *Scilab Solver* works. Thereafter, we describe the interaction between the different software components (*i.e.,*, *PRISMSYS* Model, *Scilab*

Solver and *Scilab*) supporting us on a sequence diagram. This interaction is employed to simulate the continuous time behavior of the *PRISMSYS* power-aware model.

6.2.1. Scilab Solver

The definition of the *PRISMSYS* execution semantics is specified in order to be simulated or to verify the results of the implementation in lower abstraction levels. We know there are two kinds of execution behaviors to simulate a *PRISMSYS* model: discrete event and continuous time. The former is represented by the state machine behavior and the event constraints that could be defined in *ControlSubView* by using CCSL. The latter is expressed by equations in *equationalSubViews*. The tools used to run each execution domain are different. To simulate the CCSL specifications, we use TIMESQUARE. To resolve the equations, we choose *Scilab*. Both tools, TIMESQUARE and *Scilab*, provide an application programming interface (API) that allows the implementation of a *"connector"* that interacts with the services that offer these tools.

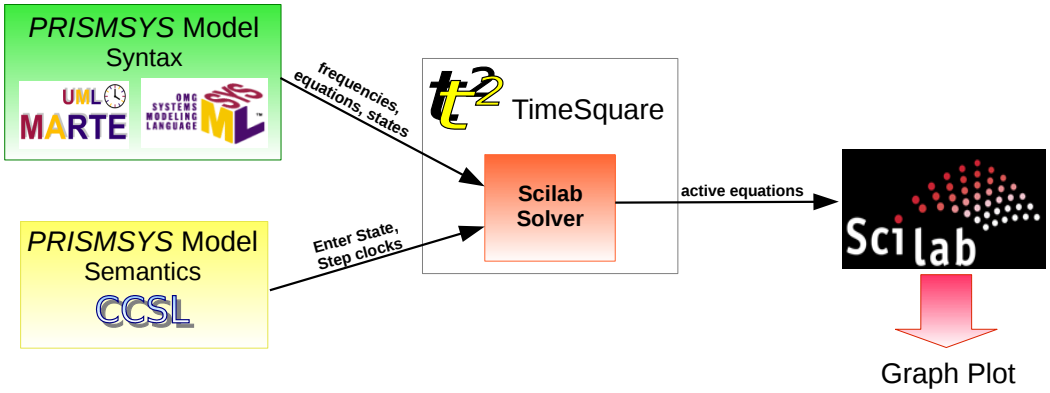


FIGURE 6.1: Overview of the *PRISMSYS* framework co-simulation implementation.

Figure 6.1 presents an overview of this implementation. TIMESQUARE is a module application based on the Eclipse plug-in approach. In consequence, we implement *Scilab Solver* as an Eclipse Plug-in to connect the TIMESQUARE solver module with the evaluation of the *PRISMSYS* model equations. From the CCSL specification, *Scilab Solver* extracts the clocks that are associated with entering states in the *PRISMSYS* Model. Next, *Scilab Solver* extracts the equations that characterize the states from the *PRISMSYS* Model. In the TIMESQUARE solver, once an event occurs in some of the entering

state clocks, the associated equation is sent to *Scilab* in order to evaluate it and generate the graph plot of the property evolution. In the *PRISMSYS* model, a chronometric clock is assigned to manage the equation evaluation. This clock has been named as *step* in Chapter 3. As soon as *step* ticks, a new value is generated in *Scilab*.

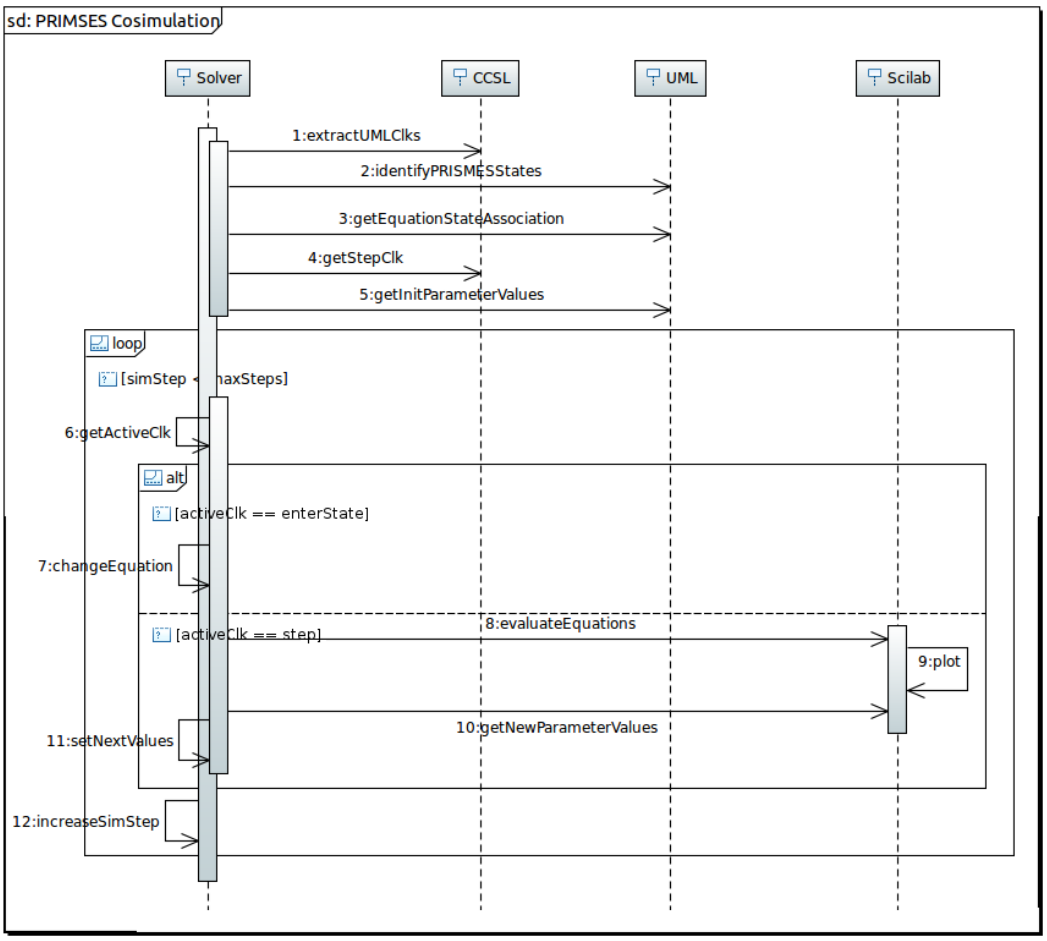


FIGURE 6.2: Sequence diagram of the PRISMSYS model Simulation.

Figure 6.2 depicts a sequence diagram that summarizes the *PRISMSYS* model execution. The *Solver* lifeline represents the *Scilab Solver*. Once the simulation starts, *Scilab Solver* extracts the clocks that represent the entering UML states associated in the CCSL specification. The UML states are filtered by their stereotype in the UML model, *i.e.,* having the clocks associated with UML states, *Scilab Solver* only searches the states stereotyped as *PRISMSYSState*. In the UML model, *Scilab Solver* also identifies and extracts the equations associated with the stereotyped states and the initial values of the equation parameters. The *step* clock is also extracted from the CCSL specification.

This clock is identified by the *clockPort* that are bound to the *t* parameters in *equationalSubViews*.

Once the TimeSquare simulation starts, *Scilab Solver* observes the extracted clocks. When an event occurs in some of these entering state clocks, *Scilab Solver* changes the equation associated with the active state. If the *step* clock ticks, the active equations are evaluated in *Scilab* with the initial parameter values. The result of the evaluation is marked in a *Scilab* plot window. After the equation evaluation, the new parameter values are gotten by *Scilab Solver* and it updates the initial parameter values. This execution continues up to the last *step* occurrence in the TimeSquare simulation.

Scilab Solver is employed to simulate the *PRISMSYS* Power-Aware Model. This simulation exhibits the evolution of non-functional properties defined in the model, such as power consumption and temperature.

6.2.2. The PRISMSYS Power-Aware Model Scenario

The scenario of *PRISMSYS* power-aware model allows to stimulate the execution of the views and the definition of the execution coherence among views. In order to specify the scenario, we state the *controlSubView* interaction with its *subViewElements* and with other *controlSubViews*. These interactions are represented in uml sequence diagrams. A sequence diagram identifies which control events are sent from and received to different elements of the *PRISMSYS* power-aware model. Once the diagrams are finished, its execution semantics is described in ccsl. The *controlSubView* specification is added to the ccsl constraints that express the behavior of the *subViewElements* and then to have a complete ccsl specification of the *PRISMSYS* power-aware model. Such a ccsl specification is simulated in TimeSquare in order to activate the *subViewElement* states. Additionally, the equations associated to the active states are processed by *Scilab Solver*. The equations are evaluated and traced in *Scilab*.

6.2.2.1. Application View

ApplicationView starts the coordination of the other views. This view defines the way as the *actions* are executed. Once an *action* begins its execution, the *controlSubView* of this

view informs to *HardwareView* that an *action* is been executed. In order to determine the instant that an action starts or stops, the *controlSubView* defines a *chronometric clock* whose ticks coincide with the clock occurrences generated by *cs2* in *ClockView*. We name this clock *appCtrPhysClk_ms*.

The *applicationView controlSubView* sends five control events to the *HardwareView*: *exeAction*, *stopAction*, *cpuOp1*, *cpuOp2* and *cpuOff*. *ExeAction* announces to *HardwareView* that an *action* starts its execution. In contrast, *StopAction* informs that an *action* stops. *CpuOp1* and *cpuOp2* command that the *cpu* runs in operation point 1 or 2, respectively. An operation point is the selection of a specific frequency and voltage to execute an *action*. The use of operation points is a strategy to reduce the power consumption tuning the performance time when an *action* is executed in the *cpu*. *CpuOff* requests to turn the *cpu* off.

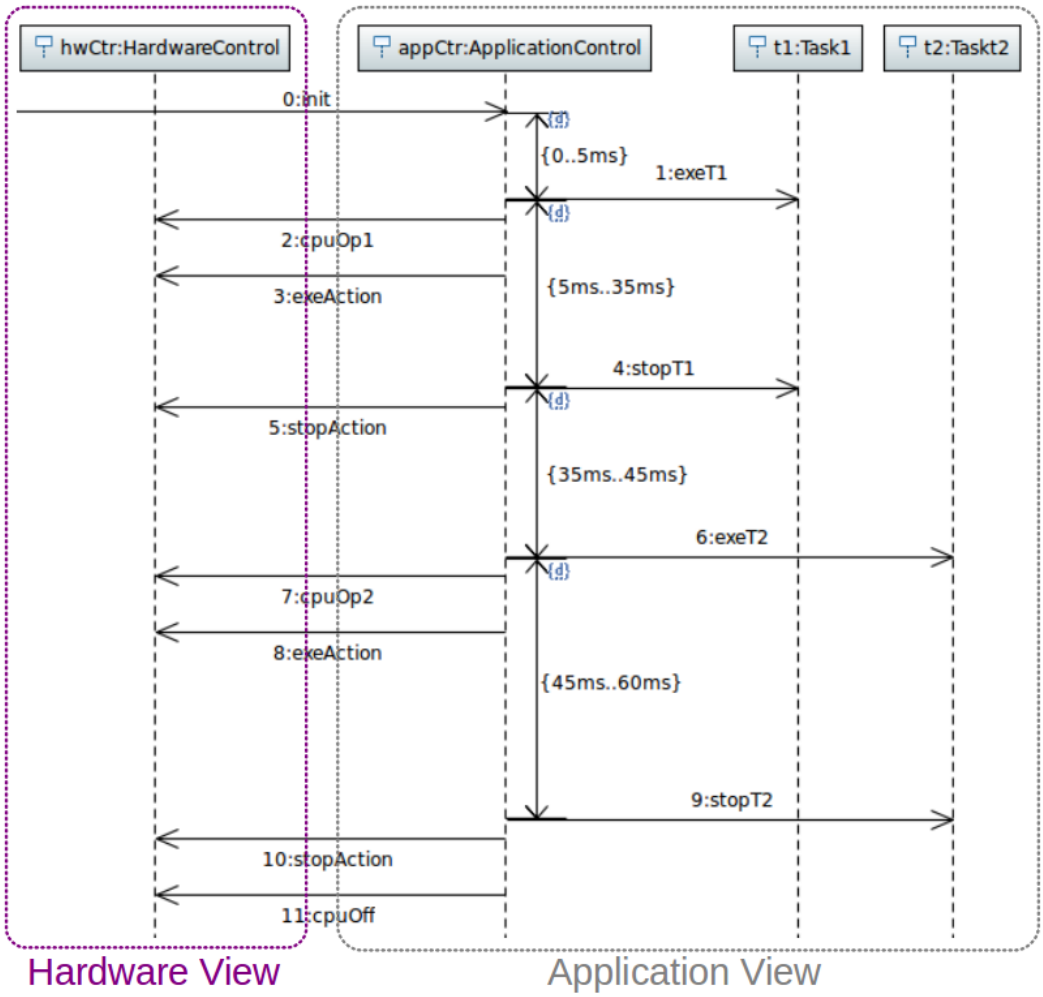


FIGURE 6.3: Execution of *ApplicationView* and its interaction with *HardwareView*.

Figure 6.3 presents a sequence diagram that specifies the way as the *T1* and *T2* actions are executed in *ApplicationView*. This diagram depicts the control events sent to the other views in order to synchronize their execution regarding the *ApplicationView* execution. The *ControlSubView* of *ApplicationView* (*appCtr*) sends an *exeT1* event to *t1* in order to change the *t1* state from *Stop* to *Run*. This event is sent *5ms* after starting the model simulation. *AppCtr* also sends an control event (*exeAction*) to announce to *HardwareView* that an *action* is being executed in *ApplicationView*. *HardwareView* coordinates the execution of *ClockView* and *PowerView* according to the control events received from *ApplicationView*. *ThermalView* does not receive any event from the other views. This view only evaluates the *cpu* temperature evolution depending on the power dissipated.

Following the *ApplicationView* sequence, *appCtr* configures the *cpu* operation point to execute the action. In the *T1* execution case, *appCtr* sends a *cpuOp1* event to configure *Operation Point 1*. We detail the frequency and voltage selected for the operation points in Section 6.2.2.2. At 35ms of the *appCtr* execution, *T1* is stopped. *stopT1* event is sent to *t1* in order to change its state to *Stop*. Next, *HardwareView* is informed that the *action* was stopped by sending an *stopAction* event. This event is received by the *HardwareView ControlSubView* (hwCtr). In the same way, *T2* is executed. However, *Operation Point 2* is configured to execute *T2* (cpuOp2). *T2* starts at 45ms and stops at 60ms. Finally, *appCtr* commands to turn the *cpu* off by sending *cpuOff* event.

The relationships among the control events sent from *appCtr* is specified in CCSL. We consider each control event as ticks of a *clock* in CCSL. Therefore, we define a clock for each interaction with the *controlSubView*. To express that *T1* starts at *5ms* and finishes at *35ms*, we define periodic clocks that tick once in a predefined period. These clocks are synchronized with the *chronometric clock appCtrPhysClk_ms*. Hence, we define as period *60ms*, *i.e.,* the periodic clocks tick once each *60ms*. We also define the instant that the periodic clocks tick. We name this instant *offset*. To specify the instant when the *T1* action starts, we represent this instant by a periodic clock that ticks in the fifth occurrence of *appCtrPhysClk_ms*, *i.e.,* at *5ms*. This periodic clock repeats this occurrence each *60ms*, *i.e.,* at *65ms*, *125ms*, etc. In CCSL, we specify *exeT1* clock as follows:

$$exeT1 \textbf{ isPeriodicOn } appCtrPhysClk_ms \textbf{ period } 60 \textbf{ offset } 5 \tag{6.1}$$

these specification are read as *exeT1* occurs in the fifth tick of *appCtrPhysClk_ms* each *60ms*. Once the *exeT1* ticks, *exeAction* and *cpuOp1* are generated. The relationships between these clocks are specified by:

$$exeT1 \boxed{=} exeAction \tag{6.2}$$

$$exeT1 \boxed{=} cpuOp1 \tag{6.3}$$

These two CCSL relations mean that once *exeT1* occurs, an event in *exeAction* and *cpuOp1* ticks simultaneously.

In the same way *exeT1* is specified, we state the instants when *T1* stops:

$$stopT1 \textbf{ isPeriodicOn } appCtrPhysClk_ms \textbf{ period } 60 \textbf{ offset } 35 \tag{6.4}$$

The relationship between *stopAction* and *stopT1* is specified as well as *exeT1*:

$$stopAction \boxed{=} stopT1 \tag{6.5}$$

Once *T1* stops, the time continues running. After *10ms* (at $45ms$), *appCtr* sends an *exeT2* to starts the *T2* action. To define when *T2* starts its execution, we state the following CCSL specification:

$$exeT2 \textbf{ isPeriodicOn } appCtrPhysClk_ms \textbf{ period } 60 \textbf{ offset } 45 \tag{6.6}$$

which means that *exeT2* occurs in the 45th tick of *appCtrPhysClk_ms* each *60ms*.

As soon as *exeT2* is sent, *appCtr* commands to *HardwareView* to change the operation point sending a *cpuOp2* event. *AppCtr* also informs that an new action starts. Therefore, *appCtr* sends an *exeAction* to *hwCtr*. Similarly to the CCSL specification of the *t1Start* relationships, the *t2Start* relations are defined by:

$$exeT2 \boxed{=} exeAction \tag{6.7}$$

$$exeT2 \boxed{=} cpuOp2 \tag{6.8}$$

To specify the end of *T2*, which occurs at *60ms*, we define the following periodic clock:

$$firstappCtrPhysClk_ms \textbf{ isPeriodicOn } appCtrPhysClk_ms \textbf{ period } 60 \textbf{ offset } 0 \tag{6.9}$$

and then, we filter this clock deleting the first tick:

$$stopT2 \boxed{=} firstappCtrPhysClk_ms \blacktriangledown 2(1)^w \tag{6.10}$$

where $\blacktriangledown$ is the CCSL operator that filters *appCtrPhysClk_ms* and the word $2(1)^w$ means that the fist occurrence of $firstPhysClk_ms$ is filtered, *i.e.,* this clock starts to tick at $60ms$.

Finally, once *stopT2* occurs, a *stopAction* is sent to *hwCtr*. the relationship between these clocks is specified in CCSL as:

$$stopAction \boxed{=} stopT2 \tag{6.11}$$

Figure 6.4 depicts the simulation of the *ApplicationView* specified in CCSL by using TimeSquare. In this figure, we presents the state machine behavior reacting to the control events from *controlSubView*. Each *action* state is represented by a start and finish event, *e.g., t1StopStat* and *t1StopFinish*. At the begin of the simulation, the *T1* and *T2* are in *Stop* state. Once the *controlSubView* commands to execute an *action*, the states of *T1* and/or *T2* change. In this simulation, the sequence *T1*, *T2* and *T1* is executed. The relationship between events are depicted by blue arrows (precedence) and red lines (coincidence).

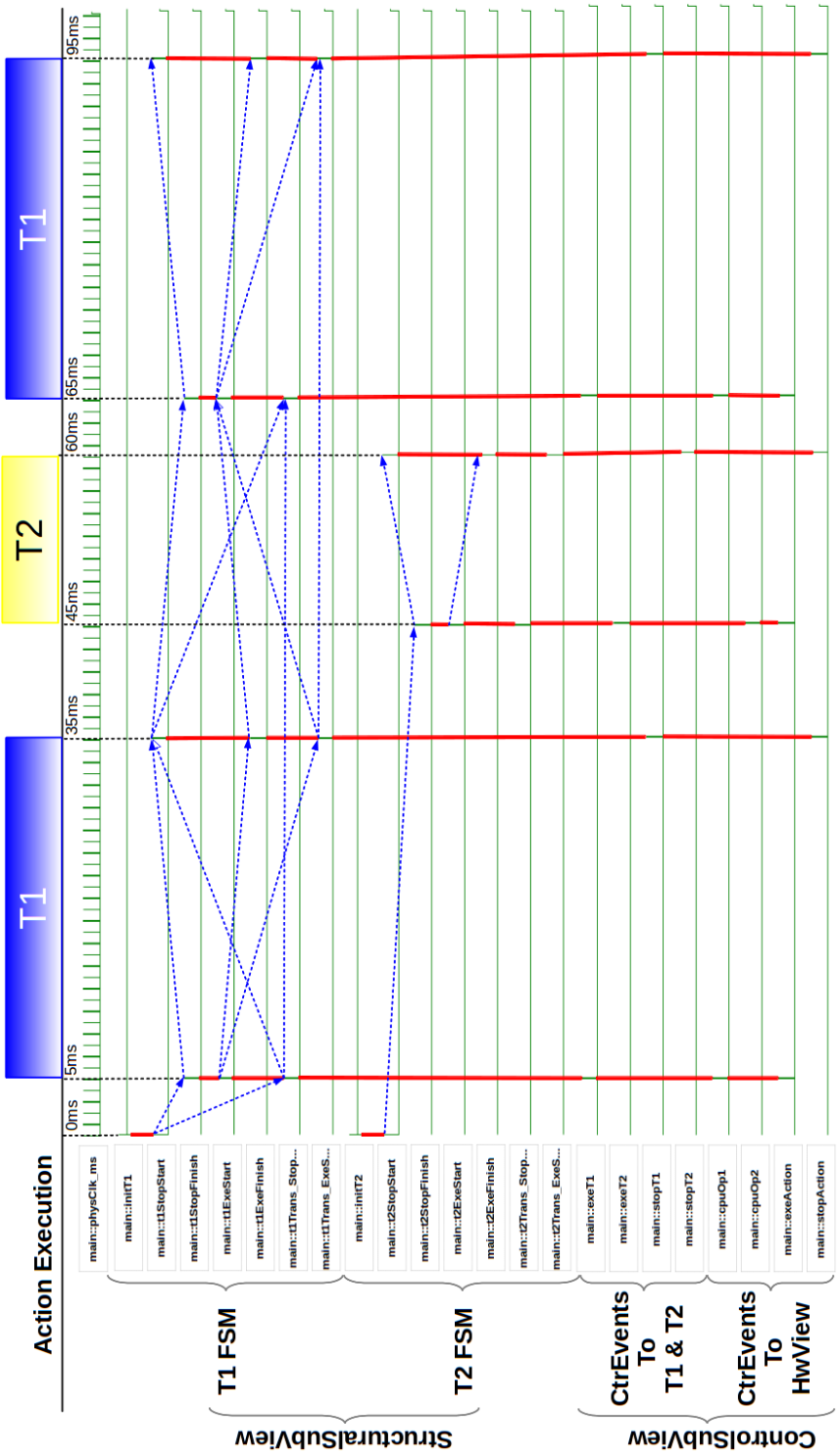


FIGURE 6.4: *ApplicationView* simulation in TIMESQUARE.

6.2.2.2. Hardware View

Once *ApplicationView* is in execution, *HardwareView* receives control events to coordinate its *subViewElements* and to synchronize the *PowerView* and *ClockView* execution. Figure 6.5 presents the sequence diagram of the interaction among *HardwareView*, *ApplicationView*, *ClockView* and *PowerView* from the *HardwareView* point of view. At the beginning of the execution sequence, *hwCtr*, which is the *controlSubView* of *HardwareView*, receives two events: *cpuOp1* and *exeAction*. The former commands to *hwCtr* to configure *Operation Point 1*. Usually, the *cpu* manufacturers give the possible operation points where their *cpus* could works. Therefore, in this example, *hwCtr* sends an *actV1* event to *PowerView* and an *actF1* event to *ClockView* to configure the operation point. These events active *V1* and *F1* states in the corresponding views, if they are not already in these states. *ExeAction* causes that *hwCtr* changes the *cpu* state to *Busy*, *i.e.*, *cpu* is executing an *action*, and it sends *pwExeAction* and *clkExeAction* to *PowerView* and *ClockView*, respectively, to change the abstracted *cpu* states. Thanks to the *allocation* correspondence, *HardwareView* can know which action (*T1* or *T2*) is in execution according to the action active state.

We specify in CCSL that *actV1* and *actV2* are caused by *cpuOp1* as:

$$cpuOp1 \;\boxed{=}\; actV1 \tag{6.12}$$

$$cpuOp1 \;\boxed{=}\; actF1 \tag{6.13}$$

these CCSL relations mean that once *cpuOp1* ticks, *actV1* and *actV2* occur. Similar specification is defined to the relationship among *exeAction*, *pwExeAction* and *clkExeAction*:

$$exeAction \;\boxed{=}\; pwExeAction \tag{6.14}$$

$$exeAction \;\boxed{=}\; clkExeAction \tag{6.15}$$

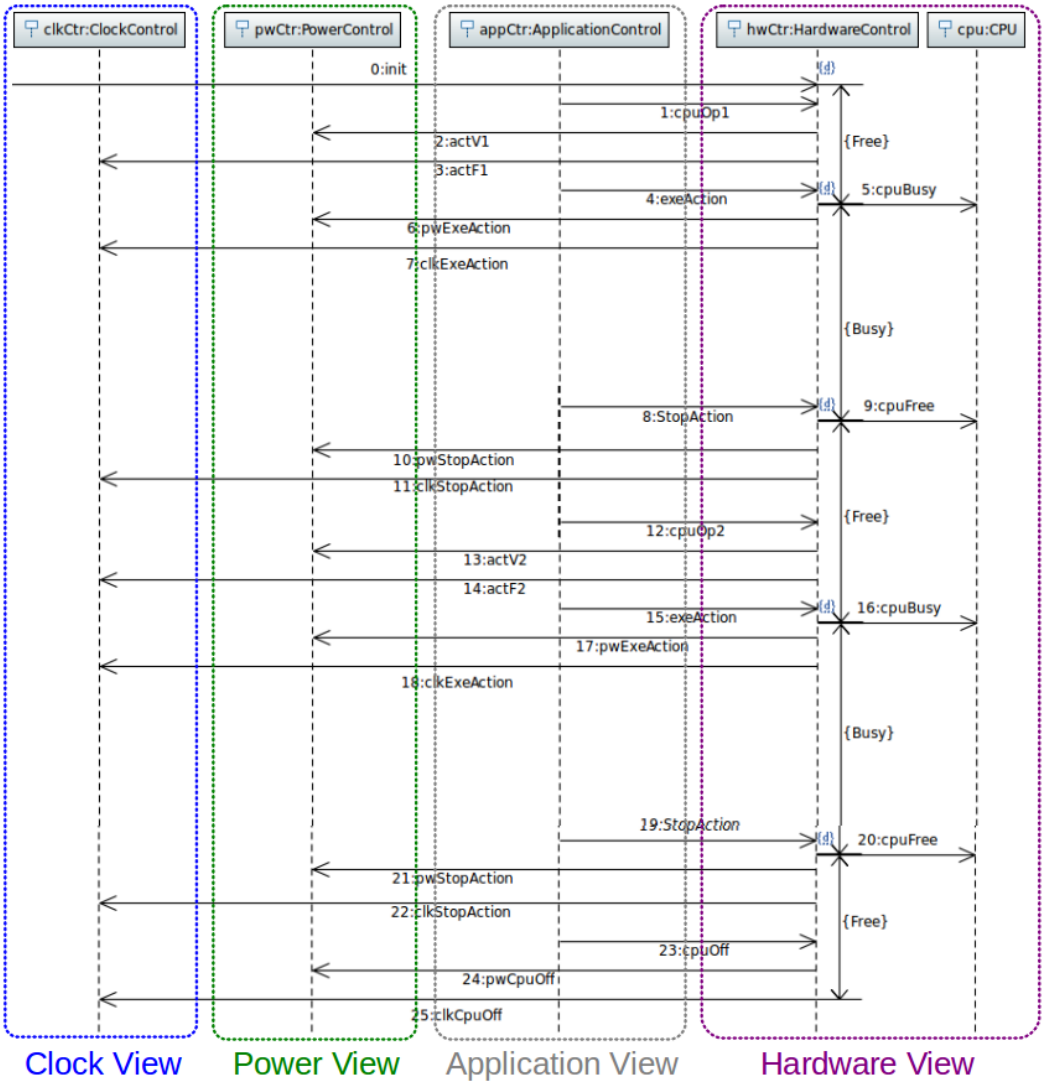


FIGURE 6.5: Execution of the *HardwareView controlSubView* and its interaction with *ApplicationView*, *PowerView* and *ClockView*.

In the figure, we note that *ApplicationView* is which decides when the actions finish their executions. As we have explained in Chapter 5, the notion of time is specified in *ClockView*. This time notion is shared with *ApplicationView* to specify the scenario. By employing the abstraction of *cpu* represented in *ClockView*, we can know the number of clock cycles needed to execute each action.

Once the *T1* action is stopped in *ApplicationView*, this view sends the *stopAction* event to indicate that the action was executed. When *hwCtr* receives the control event, it commands to *cpu* to change to *Free* state, and sends *pwStopAction* and *clkStopAction* to change the abstraction of *cpu* state in *PowerView* and *ClockView*, respectively.

The relationship specification among *stopAction*, *cpuFree*, *clkStopAction* and *pwStopAction* is expressed in CCSL as:

$$stopAction \boxed{=} cpuFree \tag{6.16}$$

$$stopAction \boxed{=} clkStopAction \tag{6.17}$$

$$stopAction \boxed{=} pwStopAction \tag{6.18}$$

therefore, *stopAction* coincides with *cpuFree*, *clkStopAction*, and *pwStopAction*.

After some milliseconds, whose evolution is continued in *ApplicationView*, *appCtr* sends *cpuOp2* and *exeAction* to *hwCtr* in order to execute the *T2* action. *cpuOp2* causes two control events: *actV2* and *actF2* to configure *Operation Point 2*. In the same way that *cpuOp1*, the CCSL specification of *cpuOp2* is defined as:

$$cpuOp2 \boxed{=} actV2 \tag{6.19}$$

$$cpuOp2 \boxed{=} actF2 \tag{6.20}$$

The *exeAction* relationship is already defined. The difference is the action to execute. Finally, when *T2* is executed, *appCtr* commands to turn the *cpu* off (*cpuOff*). Hence, *pwCpuOff* and *clkCpuOff* events tick. The CCSL specification among these clocks are:

$$cpuOff \boxed{=} pwCpuOff \tag{6.21}$$

$$cpuOff \boxed{=} clkCpuOff \tag{6.22}$$

Figure 6.6 depicts the simulation of the *HardwareView controlSubView* and the interaction with the other views. The figure presents the behavior of the *cpu* state machine according to the received control events. The *cpu* states are represented by *busy* and *free* activity periods in the top of the figure to express the state changes. Additionally, the simulation represents the relationship between control events of *ApplicationView*, *HardwareView*, *PowerView* and *ClockView*.

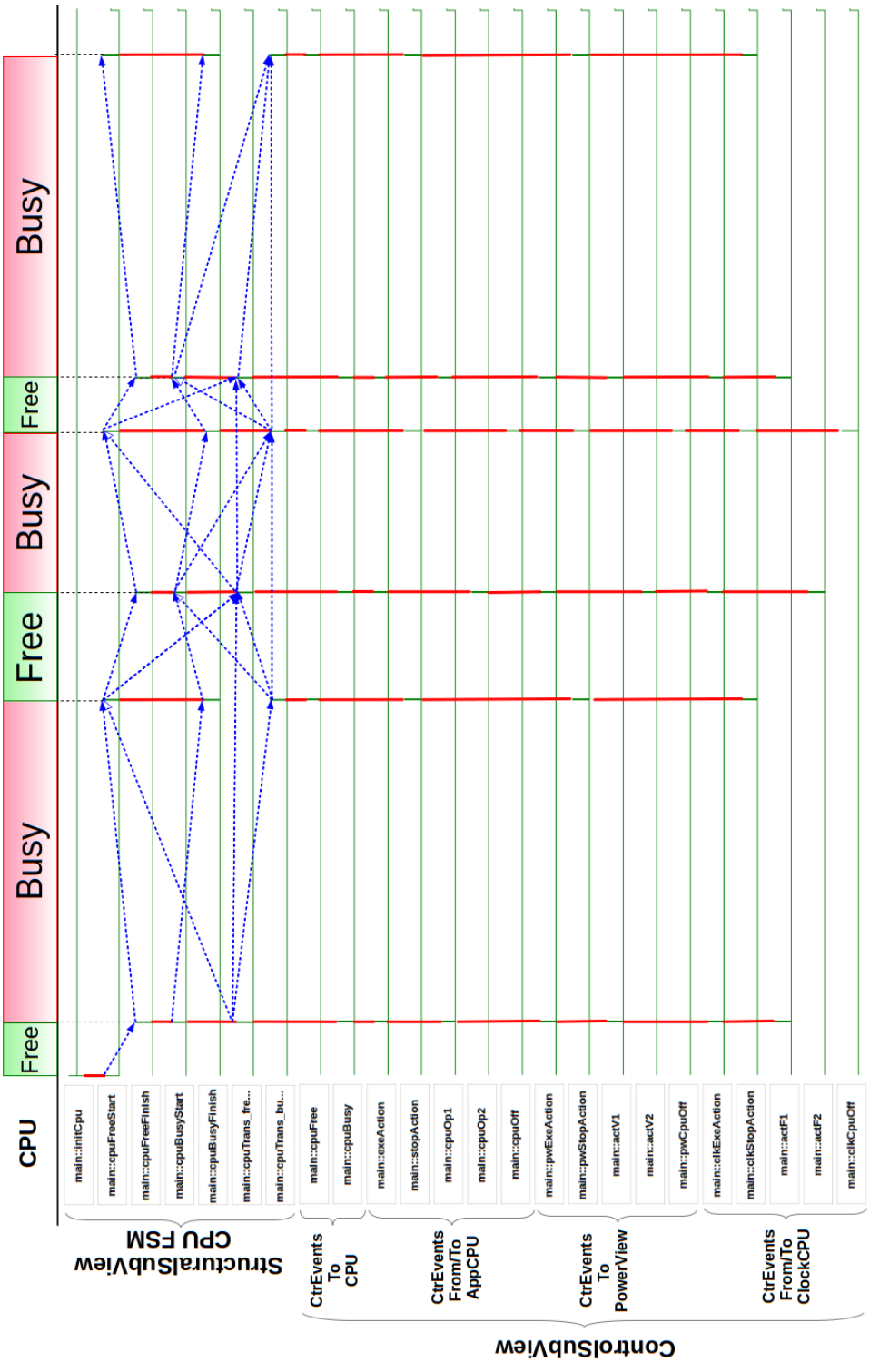


FIGURE 6.6: *HardwareView* simulation in TIMESQUARE.

6.2.2.3. Clock View

The evolution of time is expressed in *ClockView*. To describe this evolution, we must specify a base *chronometric clock* in CCSL. There are two clock frequencies defined in *cs1* (Chapter 5) that active the *cpu* when *T1* and *T2* are executed. Such clock frequencies specify a clear evolution of time, *e.g.,* for each *cpu* cycle, a time step is executed according to the selected frequency. However, the time step for each *cpu* cycle is too small ($5.56ns$ for *F1* and $2.78ns$ for *F2*). In consequence, we can choose that the time evolution in the simulation is either a multiple of the possible *cpu* cycle frequencies or a common clock whose frequency could represent the two *cpu* clocks in a low frequency. For the sake of simplicity, we choose a common clock to specify the simulation time step in this *PRISMSYS* model example. This clock specifies the clock signal of *cs2* of the *ClockView* defined in Chapter 5. Such a clock is a *chronometric clock* that ticks each millisecond. It is specified in CCSL as:

$$physClk_ms \boxed{=} idealClk \textbf{ discretizedBy } 0.001 \tag{6.23}$$

where *idealClk* is a *DenseClock* that ticks each seconds (see Chapter 3).

The duration of the action execution (*T1* and *T2*) in *ApplicationView* and the waiting time between the two action executions is synchronized with $physClk_ms$. In fact, we define the correspondence between $physClk_ms$ and $appCtrPhysClk_ms$ as:

$$appCtrPhysClk_ms \boxed{=} physClk_ms \tag{6.24}$$

By using these clocks, we can evaluate the clock cycles employed by *T1* and *T2* to be executed in *cpu*. Table 6.1 presents the clock cycles spend by *T1* and *T2* and the duration of the action execution by using $physClk_ms$ and $appCtrPhysClk_ms$. We remark the clock cycles of both actions are exactly equal. Nevertheless, the time execution of *T1* is twice *T2*. This variation is caused by the operation point configuration. While *T1* is executed at *180MHz*, *T2* is performed at *360MHz*. Although the difference of time performance is notable, these operation points affect the power consumption. We explain this power concern in Subsection 6.2.2.4.

Action	*Clock Cycles*	*Time (ms)*
T1	5400000	30
T2	5400000	15

TABLE 6.1: Action execution in *cpu* clock cycles and time.

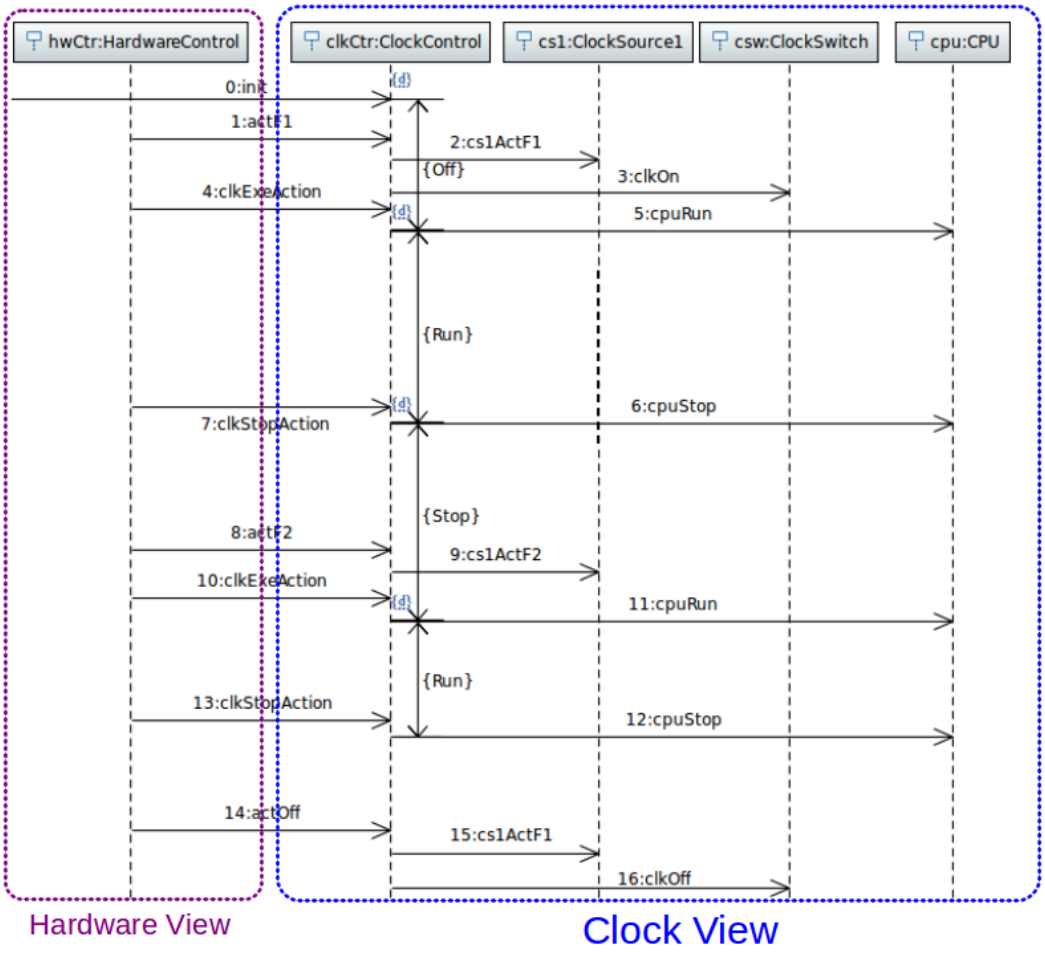


FIGURE 6.7: Execution of the *ClockView controlSubView* and its interaction with its internal *subViewElements* and with *HardwareView*.

Figure 6.7 depicts the internal interactions into *ClockView* and the communication with *HardwareView*. In *ClockView*, *clkCtr*, which is the *controlSubView*, receives an *actF1* event from *HardwareView*. Thus, *clkCtr* sends a *cs1ActF1* to *cs1* to change the frequency state to *F1* (*180Mhz*, according to the *characterization subCorrespondnece* with *equationalSubView*). Additionally, *clkCtr* sends a *clkOn* to *csw* in order to allow passing the clock signal generated by *cs1* to *cpu*. *clkCtl* also receives a *clkExeAction* to change

the abstract *cpu* state in *ClockView*. Therefore, *clockedElement cpu* changes to *Run* state. Once this action stops, *hwCtr* sends a *clkStopAction* to *clkCtr*. Thus, *clkCtr* sends a *cpuStop* event to the *clockedElement cpu* and it changes to *Stop* state.

The relationship among *actF1*, *cs1ActF1* and *clkOn* are specified in CCSL as:

$$actF1 \boxed{=} cs1ActF1 \tag{6.25}$$

$$actF1 \boxed{=} clkOn \tag{6.26}$$

In the same way, the relationship between *clkExeAction* and *cpuRun* is defined in CCSL:

$$clkExeAction \boxed{=} cpuRun \tag{6.27}$$

Once the action finishes, *hwCtr* sends a *clkStopAction* to *clkCtr* changing to *Stop* state of the *cpu clockedElement*. In consequence, we express this causality between *clkStopAction* and *cpuStop* as:

$$clkStopAction \boxed{=} cpuStop \tag{6.28}$$

Certain cycles of clocks later, an *actF2* event is received. This event causes a *cs1ActF2*, which is specified as:

$$actF2 \boxed{=} cs1ActF2 \tag{6.29}$$

The execution of the second action is the same as the first one, therefore *clkExeAction*, *clkStopAction*, *cpuRun* and *cpuStop* events are generated. Finally, *clkCtr* receives an *actOff* event from *hwCtr*. Hence, *clkCtr* commands to change the frequency to $180MHz$ (*cs1ActF1*) and *csw* is closed (*clkOff*). The CCSL specification of these control events are:

$$actOff \boxed{=} cs1ActF1 \tag{6.30}$$

$$actOff \boxed{=} clkOff \tag{6.31}$$

Figure 6.8 depicts the *ClockView* simulation. In this view, there are three *subViewElements* (*cs1*, *clkSw* and *clkEleCPU*) whose behavior is represented by state machines. The execution of these *subViewElements* is coordinated according to the control events sent by the *controlSubView*. In the top of the figure, we represents the change of states

of each *subViewElements* according to the TimeSquare simulation. We also depicts *physClk_ms* in the simulation where each tick occurs each millisecond.

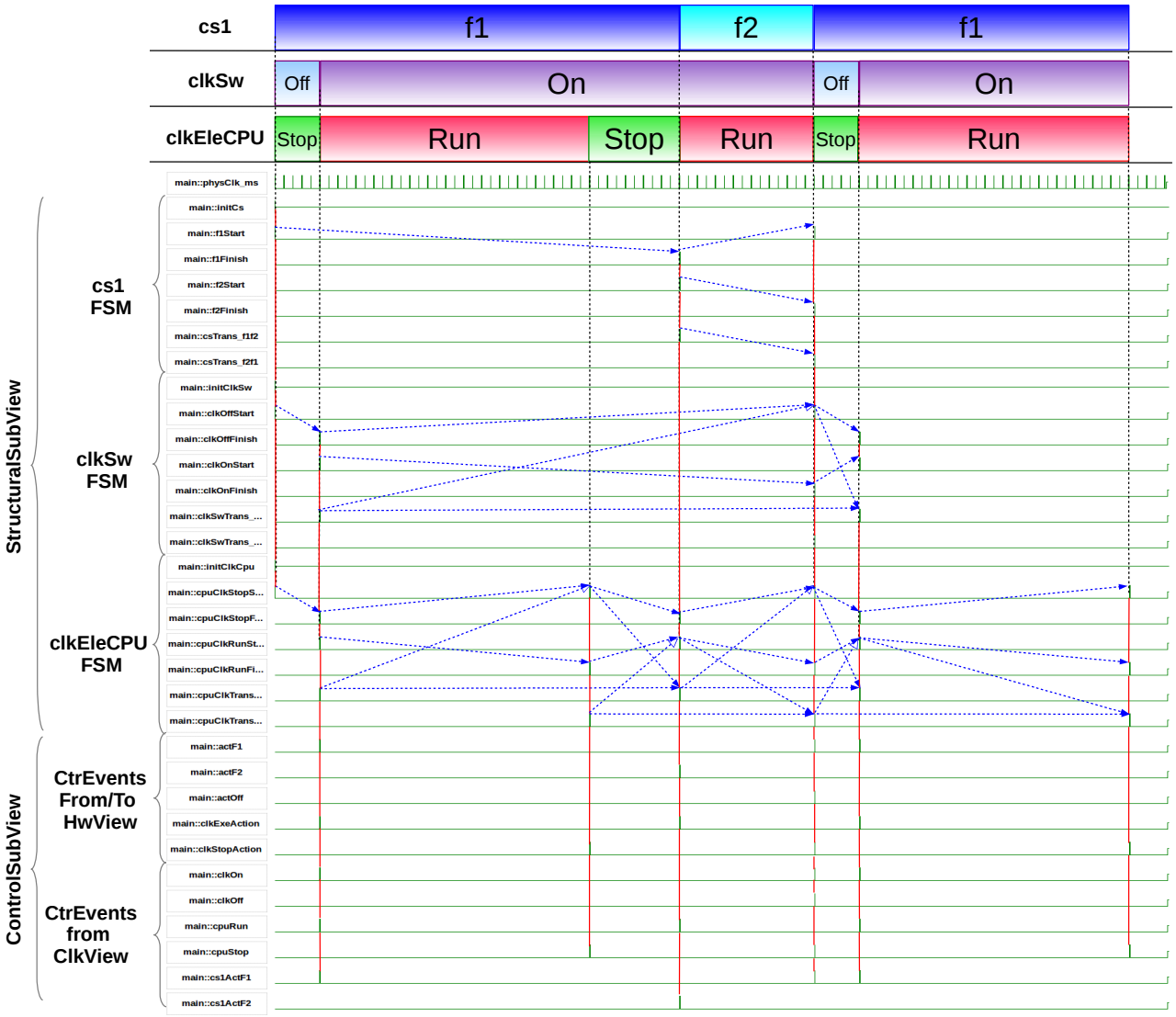


Figure 6.8: *ClockView* simulation in TimeSquare.

6.2.2.4. Power View

Once the evolution of time is defined in *ClockView*, we can use the occurrences of *physClk_ms* to evaluate associated equations with the active states. As we have done with *ApplicationView*, we define a *chronometric clock* in the execution semantics of *controlSubView* that coincides with *physClk_ms*. We name this new clock *pwCtrPhysClk_ms*. This clock is the *step* occurrence that arrives to *clkStepCtr* of *pwCtr*, which is transmitted to the *peqv* (see Figure 5.17) in order to evaluate the power consumption of the system.

Figure 6.10 presents the control events sent from the *PowerView controlSubView* to its *subViewElements* (*vs1*, *psw* and *cpu*). This figure also depicts the interaction between *PowerView* and *HardwareView*. The interaction is analogous to the *ClockView* interaction (Figure 6.8). Although, in this case, the voltage values, the power switch actions and the *cpu* power abstraction states are controlled. Similarly to the other views, the relationship among control events are specified in CCSL as follows:

$$actV1 \boxed{=} vs1ActV1 \tag{6.32}$$

$$actV1 \boxed{=} pwOn \tag{6.33}$$

$$actV2 \boxed{=} vs2ActV2 \tag{6.34}$$

$$actOff \boxed{=} vs1ActV1 \tag{6.35}$$

$$actOff \boxed{=} pwOff \tag{6.36}$$

$$pwExeAction \boxed{=} cpuActive \tag{6.37}$$

$$pwStopAction \boxed{=} cpuIdle \tag{6.38}$$

The simulation of *PowerView* is depicted in Figure 6.10. This simulation is split in three parts: The state change representation in *PowerView* and *ClockView* (top), the discrete simulation executed in TIMESQUARE (middle) and the continuous simulation evaluated in *Scilab* (Bottom) by employing the connector *Scilab Solver*. We remark that each time that *pwEleCPU*, which is the *cpu* power abstraction, is in the *Active* state, the equation that characterizes it is the dynamic power equation ($P = Cfv^2$). Therefore, according to the configured operation point ((180MHz, 1.1V) or (360MHz, 2.2V)), the dynamic power is evaluated. Once *pwEleCPU* is in the *Idle* state, the static power equation is evaluated ($P_{leak} = I_{leak} * V$). In this simulation, the static power is a constant value

because the voltage value and the leakage current always have the same values ($v = 1.1V$ and $I_{leak} = 8mA$).

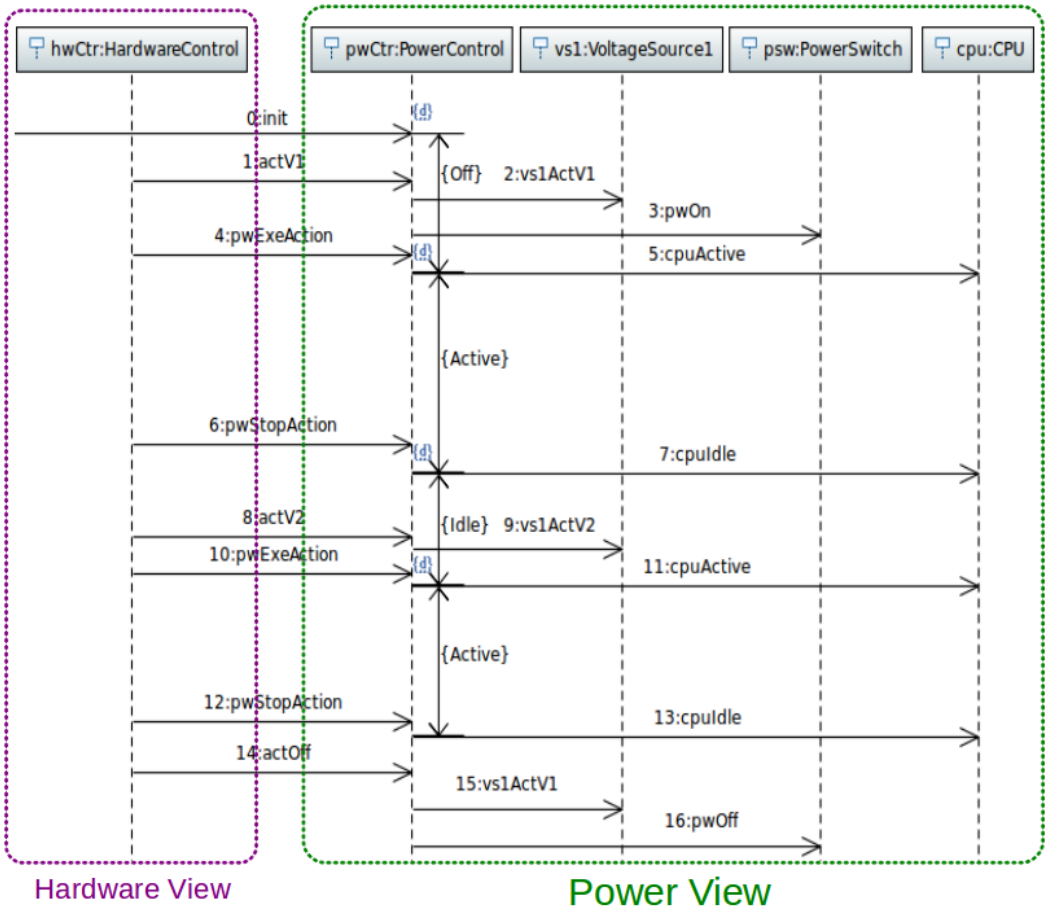


FIGURE 6.9: Execution of the *HardwareView controlSubView* and its interaction with *ApplicationView*, *PowerView* and *ClockView*.

Scilab solver is executed each time that *pwCtrPhysClk_ms* ticks. Therefore, the associated equations that characterize the active states are evaluated. We note in the first period (*0-5ms*) that the power consumed is *0W*. This power value is due to *pwSw* is *Off*. In the second period (*5ms-35ms*), *T1* is executed. The configured operation point is *(180MHz, 1.1V)* and the dynamic power is evaluated giving as result *87mW*. This power consumption is kept constant during the execution of *T1*. In the third period (*35ms-45ms*), any action is executed, *i.e., cpu* is *Free*. However, *cpu* is *on* consuming static power during this period (Idle state in *PowerView*), whose result is $880\mu W$. In the fourth period (*45ms-60ms*), *T2* is executed. The operation point is also changed to *(360MHz, 2.2V)*. As we have mentioned in Subsection 6.2.2.1, the time to execute

T2 is shorter than *T1*, even though the clock cycles between *T1* and *T2* are equals. Nevertheless, *T2* consumes more power than *T1*, such as it is illustrated in the *Scilab* simulation. Its power consumption is *0.697W*. By using this simulation, we demonstrate that reducing the time performance, the power consumption rises. The next period is the repetition of the first four periods.

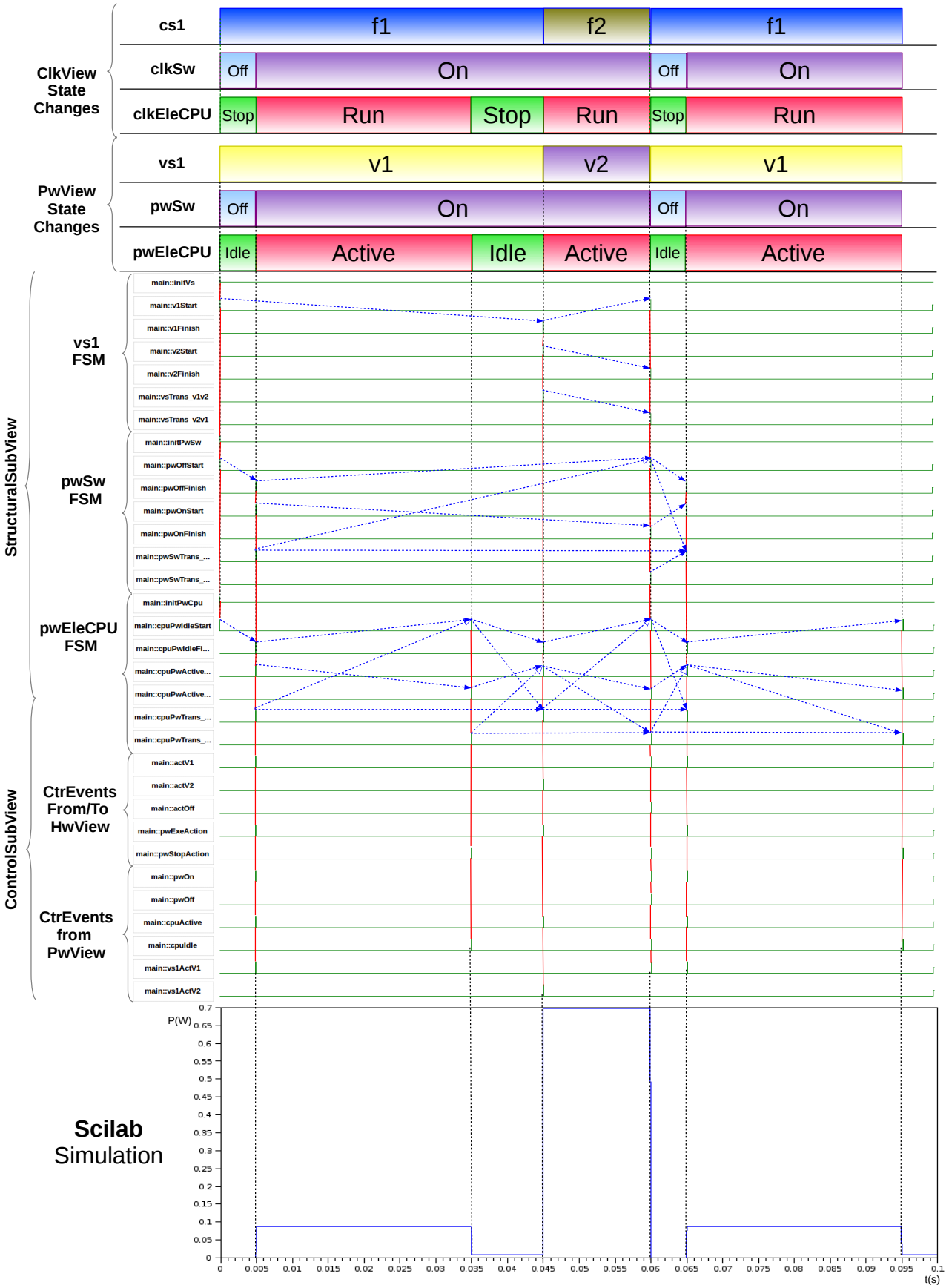


FIGURE 6.10: *Power View* simulation in TIMESQUARE.

6.2.2.5. Thermal View

ThermalView only contains a *subViewElement* in the *structuralSubView*. Such an element is the *cpu* thermal abstraction. The behavior of this element owns a state that is characterized by the thermal equation (extracted from Figure 5.13):

$$\frac{dT}{dt} = \frac{P}{C_{th}} + \frac{1}{R_{th}C_{th}}(T - T_{env}) \tag{6.39}$$

Where T is the *cpu* temperature, C_{th} and R_{th} are respectively the thermal capacitance and resistance of *cpu*, T_{env} is the *cpu* environmental temperature and P is the *cpu* power consumption. In this equation, the parameters that change their value through time are P and T. Moreover, T depends on P to evaluate its value at a specific instant. Therefore, the temperature evolution of *cpu* relies on its power consumption evolution. Figure 6.11 presents the *cpu* temperature simulation. The temperature value is evaluated according to the active states of the other views and the power consumption evaluation. We note that the temperature value rises when the power consumption increases and it falls once the power consumption decreases.

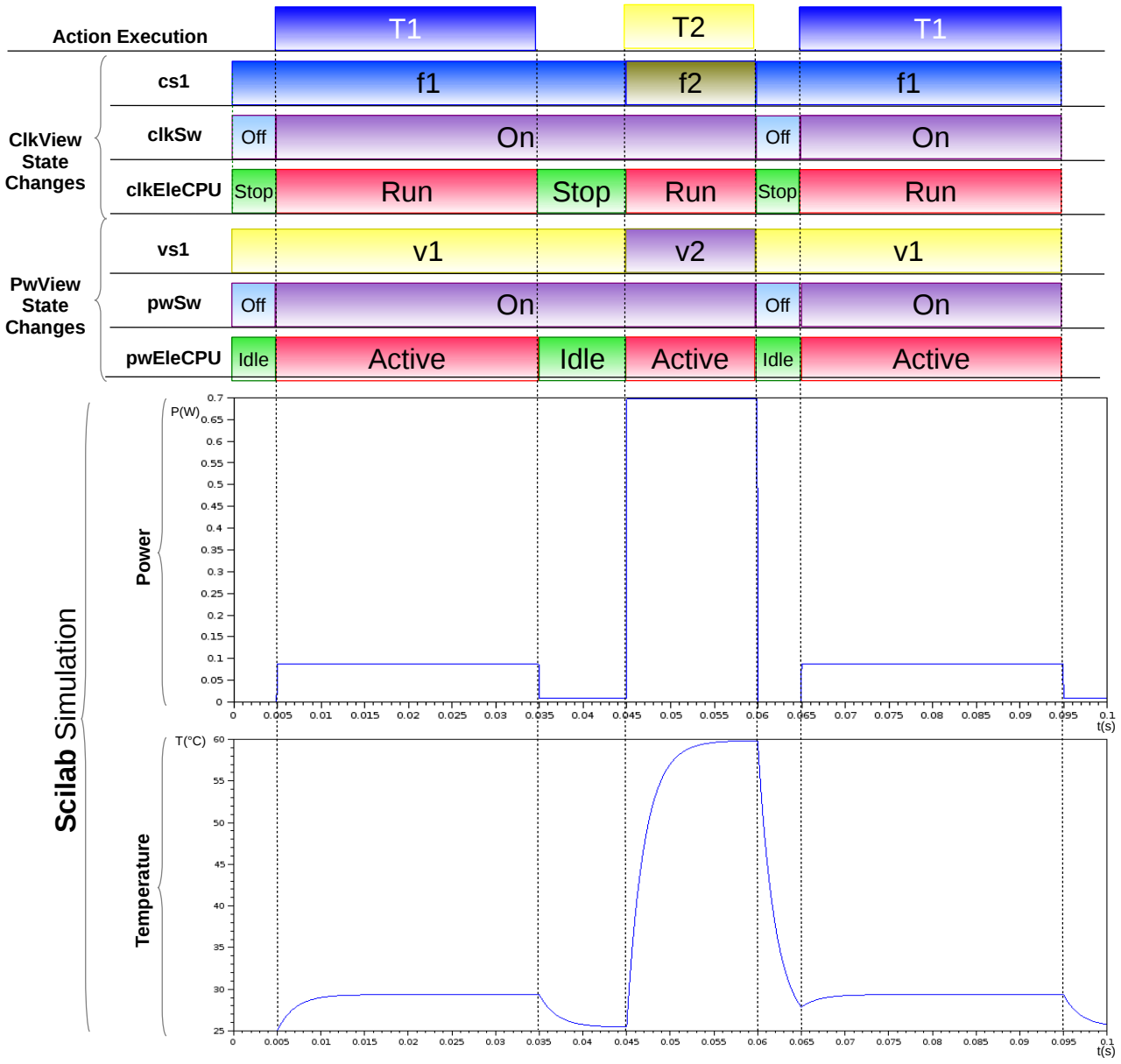


FIGURE 6.11: *Thermal View* simulation in TIMESQUARE.

6.3. PRISMSYS Power-Aware Model Analysis in *Aceplorer*

Non-functional properties of embedded systems are modeled and analyzed either by abstracting the execution of the elements that belong to a system, or by using dedicated tools. On one hand, the definition of the execution semantics of *PRISMSYS* states a sort of abstract analysis by representing the actions of the view elements by clocks. A different abstract analysis approach is proposed by Abdallah et al. [87]. They specify the execution of the elements of application and execution platform by logical and physical clocks, respectively. The analysis consists in exploring potential allocations between the application and various execution platforms, in order to achieve the functional requirements and the time deadline restriction; meanwhile reducing the system power consumption. This exploration is stated by the relation of logical (application) and physical (execution platform) clocks. They analyze time as a non-functional property, but unfortunately it is not possible for them to quantify in a precise manner the impact of the time on other non-functional properties, such as power consumption and temperature. *PRISMSYS* execution semantics together with *Scilab Solver* could help to automate the cited exploration process, adding the quantifiable evaluation of the power consumption and temperature.

On the other hand, dedicated tools use concepts and languages commonly defined by domain experts to specify systems from their points of view. Nevertheless, these tools have to redefine the elements specified in other domains to build their own models. The redefinition produces elements redundancy among analysis tool models. For instance, *Aceplorer* [8], a power consumption analysis tool, represents the system from a power point of view redefining its elements already represented in other tools or languages. For instance, by using the *Aceplorer* modeling process, a memory, specified in a hardware architecture language such as VHDL [76] or SystemC [68], is redefined in *Aceplorer* with power properties to evaluate its power consumption.

To avoid the redundancy between tools, the *PRISMSYS* power-aware model abstracts the elements that are defined in a domain to be used in another one. In the case of the memory example, it is represented as a *SubViewElement* in *HardwareView*. This view can specify the hardware architecture model implemented in SystemC. Such a memory

is abstracted by a *PoweredElement* in *PowerView* defining its power properties. Taking these two views, an *Aceplorer* model can be generated.

Thanks to these element abstractions and the possibility to generate specialized models from the *PRISMSYS* power-aware model, we can extract the information needed by a specific analysis tool to evaluate a non-functional property of the system, but also to feed the *PRISMSYS* power-aware model with the result of specific analyses. For instance, the worst case execution time (WCET) of a task allocated on a CPU can be estimated by a WCET analysis tool and this estimation can be injected into the model in order to be used for the power consumption analysis.

The scenario employed to execute an *Aceplorer* model is the functional simulation output of the system. For instance, to analyze the power consumption of a system whose functional model is implemented in SystemC, we must transform the SystemC simulation output to an *Aceplorer* scenario. This transformation is manually recreated or the VCD file generated from the SystemC simulation can be imported by *Aceplorer* to generate the test scenario. However, to import this file, the architecture defined in SystemC must be the same architecture in *Aceplorer*.

In order to ease the transmission of the system model execution between tools, we propose to use the *controlSubView* behavior of the selected views to build the scenarios that are employed to execute the models in each tool. By Using these scenarios, we have the needed elements to analyze the non-functional properties using different tools.

In this section, we present a transformation overview to generate analysis tool models from the *PRISMSYS* power-aware model. This transformation allows to evaluate non-functional properties specified in our model by using various analysis tools. We detail this transformation for the study of the power consumption in *Aceplorer*. We also describes how we can generate a scenario from the *controlSubView* specifications to be used in *Aceplorer*.

6.3.1. Transformation Overview

We define the transformation from the *PRISMSYS* power-aware model to an analysis tool in two steps as illustrated on Figure 6.12. The first step transforms the UML *PRISMSYS* power-aware model to a *PRISMSYS* power-aware domain model that we

name *PRISMSYS pivot model*. This transformation reduces the UML model navigation complexity creating a model that is conformed to the *PRISMSYS* power-aware meta-model. Such a model eases the transformation from the *PRISMSYS* power-aware model to analysis tool models, but it is transparent to designers. The second step transforms the *PRISMSYS* power-aware domain model to an analysis tool model. To define this transformation, we propose two options to be implemented. The first one is to define an analysis tool meta-model. This meta-model helps to specify the transformation rules between meta-models and to generate the code that will be executed in the tool. The second one is to generate directly the code to be executed in the analysis tool.

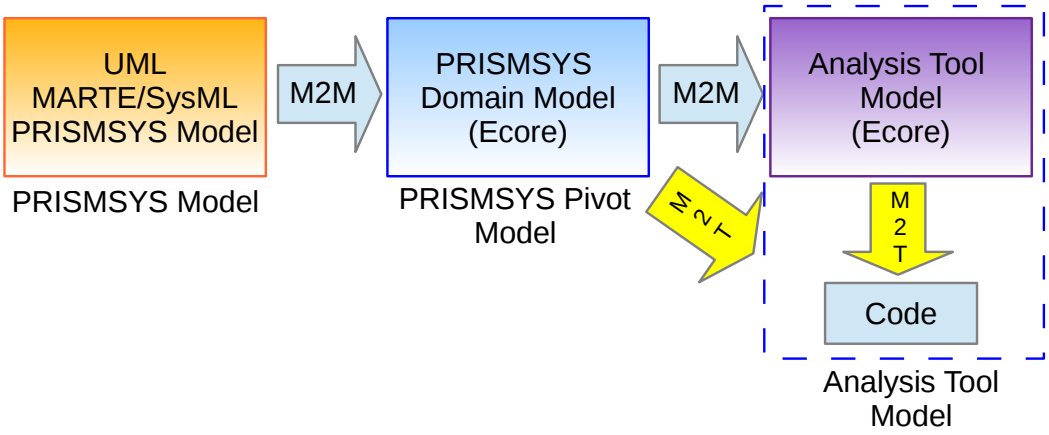


FIGURE 6.12: Transformation Overview.

To give an example of this transformation, we chose the first option to transform the *PRISMSYS* power-aware model to *Aceplorer*. First, we introduce the *Aceplorer* meta-model to present the main concepts contained in its model. Second, we point out the main features of the *PRISMSYS* - *Aceplorer* transformation.

6.3.2. *Aceplorer* Domain Model

Aceplorer uses its own language to create the power model of a system evaluating the power consumed by each system component. However, this language is implemented without using the MDE techniques to define DSMLs. Therefore, we extract the concepts and relationships used in *Aceplorer* to specify a system power model and we represent them in a meta-model. Figure 6.13 depicts a simplified *Aceplorer* meta-model. An *Aceplorer* model has three main elements: *modules*, *links* and *types*. *Module* is an abstract

element whose specification follows the component approach, *i.e., Module* is a structural element that contains *interfaces* (ports), *properties* (attributes) and a behavior definition represented by *states*. *Link* connects *Input* to *Output* interfaces to bind the shared data between *Modules*. *Type* is a type definition to specify a value and a unit of *typedElements* defined in *Modules*. *Property* can be *Static* or *VariableElement*. *Static* is a typed constant, such as number of gates and component load capacitance and *VariableElement* is a typed variable such as voltage (*VoltageVariable*), current (*CurrentConsumption*) and frequency (*Variable*). *State* contains *variableEquationElements*, comprising equation definitions associated with *parameters*. *Parameter* is the unknown element in the equation definition and it can be a *property* or an *interface*. *VariableEquationElement* is specialized in three equation types: *CurrentConsumptionEquation*, *VoltageVariableEquation* and *VariableEquation*. *CurrentConsumptionEquation* specifies the current consumed by a *module*. *CurrentConsumptionEquation* is associated with a *currentConsumption* to express that this equation defines the module property. *VoltageVariableEquation* contains *voltageStates* where a voltage value is specified. *VoltageVariableEquation* is connected to *VoltageVariable*, which represents that each time a voltage state changes; the voltage value assigned to a *voltageVariable* is changed. *VariableEquation* is employed to express property equations that are not current or voltage, such as frequency. *VariableEquation* is connected to *Variable*.

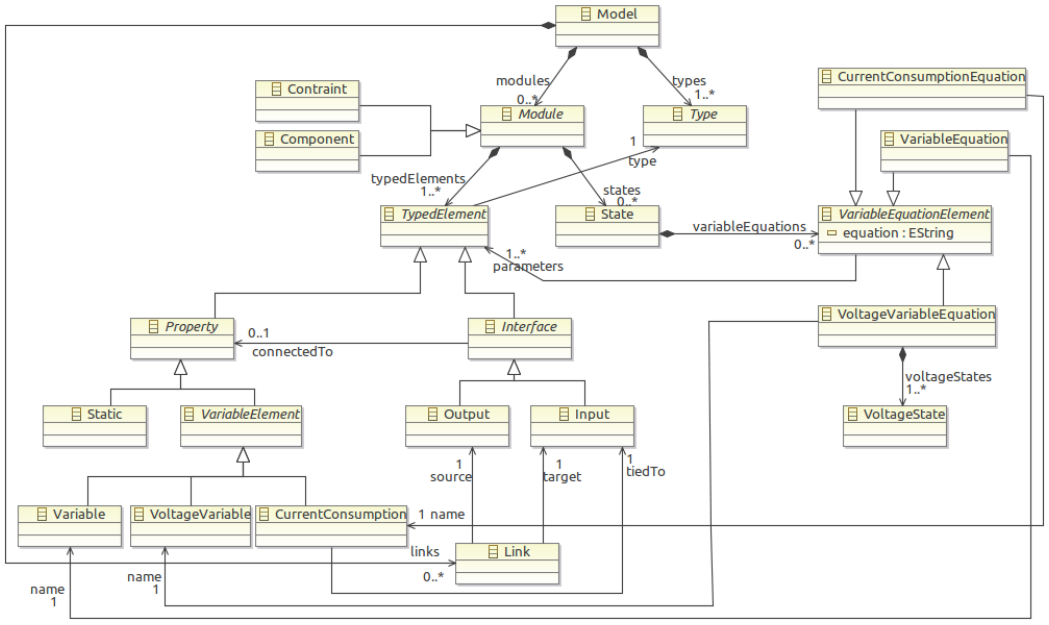


FIGURE 6.13: Simplified *Aceplorer* meta-model.

Module is specialized in *Constraint* and *Component* elements. The former represents a value generator, *e.g.,* a voltage generator and a clock generator. The latter defines power features of a system component. *Component* uses generated values from linked constraints to evaluate the component power consumption. *Component* contains *currentConsumptions* that is the current drained by the *Component* from a *Constraint*. *CurrentConsumption* is associated with *Input* that is the *interface* that supplies the current to the *Component*.

6.3.3. PRISMSYS to *Aceplorer* Transformation

In order to define the transformation rules, first, we identified the views that are involved in the system power consumption and then, we map the elements from these views to the corresponding *Aceplorer* elements. *PowerView*, *ClockView*, *HardwareView* and *ApplicationView* are selected to build a complete *Aceplorer* model. *PowerView* and *ClockView* define the elements that control non-functional properties that impact the power consumption of system components, such as voltage and frequency. *ClockView* also specifies the clock cycles of the actions executed in *ApplicationView*. *HardwareView* represents the hardware architecture of the system we want to analyze. *ApplicationView* specifies the scenario to evaluate the power consumed by the system components.

View	**SubViewElement**	**Aceplorer**
PowerView	VoltageSource	Constraint
ClockView	ClockSource	
PowerView	PoweredElement	Component
ClockView	ClockedElement	
StructuralSubView	PropertyPort	Input or Output
StructuralSubView	Connector	Link
StructuralSubView	State	State
PowerView	PowerSwitch	State
EquationalSubView	Equation	VariableEquationElement
EquationalSubView	Parameter	VariableElement

TABLE 6.2: Multi-View - *Aceplorer* Mapping.

Table 6.2 presents the main elements to map from the *PRISMSYS* power-aware model to *Aceplorer*. This table lists *Views* or *SubViews*, their *SubViewElements* and their corresponding *Aceplorer* concepts. We identify *VoltageSource* and *ClockSource* are transformed to *Constraint* in *Aceplorer*. These two *SubViewElements* supply a value to other *SubViewElements* that corresponds to the *Constraint* definition of value generator. We also observe the abstractions of *HardwareView* elements, *PoweredElement* and *ClockedElement*, are mapped to *Component*. These abstractions define non-functional properties used to estimate power consumption, *e.g.,* voltage, current and frequency as well as *Component* in *Aceplorer*. Other elements such as *PropertyPort*, *Connector* and *State* are transformed to their equivalents in *Aceplorer* (*Input* or *Output* interface, *Link* and *State*, respectively). We want to point out in *PowerSwitch* that is a current control element in *PowerView*. This *SubViewElement* can be transformed to a simple *Aceplorer State*. This state represents the *Off* state of a hardware component when it is turned off through the power switch. We note the power architectural designer, who defines *PowerSwitches* in the system hardware architecture, has a different vision to the power consumption analysis expert. Finally, *Equation* and *Parameter* are respectively mapped to *VariableEquationElement* and *VariableElements*.

6.3.4. *Aceplorer* Code Generation

Once the transformation between *PRISMSYS* power-aware model and *Aceplorer* model is done, we generate the analysis tool model in Python code, by using the *Aceplorer* library. This code is charged in *Aceplorer* and it is executed in order to create the analysis tool model on the *Aceplorer* environment. This model contains the structure, states, variables and equations that are needed to evaluate the system power consumption.

6.3.5. Test Scenario Generation

Aceplorer tool needs a scenario to evaluate the power consumption of the modeled system. An *Aceplorer* scenario is composed by a sequence of *steps*. An *step* defines the active state in each module of the model, during a period of time. For instance, a step could active the states: *V1* in *vs1*, *F1* in *cs1* and *Active* in *poweredElement cpu* in the

transformed *PRISMSYS* power-aware model. Additionally, this step is executed during $5ms$.

To generate this scenario, we use the change of the *subViewElement* states during the simulation generated by TimeSquare [6]. Moreover, we only extract the state changes of the *subViewElements* that affect the power consumption. In the *PRISMSYS* power-aware model, these elements are: *cs1*, *vs1*, *clkSw*, *pwSw* and the *cpu poweredElement*. The top of Figure 6.14 presents the state changes of the mentioned *subViewElements* simulated in TimeSquare.

clkSw and *pwSw* are respectively merged to *cs1* and *vs1* in the *Aceplorer* model. Therefore, their state changes must also be joint. Firstly, we specify the clocks that represent the *cs1* states in *Aceplorer* by using the following CCSL expressions:

$$cs1OffStart \boxed{=} clkOffStart \tag{6.40}$$

$$cs1OffFinish \boxed{=} clkOffFinish \tag{6.41}$$

$$cs1F1Start \boxed{=} (f1Start \searrow clkOnStart) + (f1Start - clkOffStart) \tag{6.42}$$

$$cs1F1Finish \boxed{=} f1Finish \tag{6.43}$$

$$cs1F2Start \boxed{=} (f2Start \searrow clkOnStart) + (f2Start - clkOffStart) \tag{6.44}$$

$$cs1F2Finish \boxed{=} f2Finish \tag{6.45}$$

where *cs1OffStart* and *cs1OffFinish* represent the *Off* state, *cs1OffFinish*, *cs1F1Start* and *cs1F1Finish* express the *F1* state, and *cs1F2Start* and *cs1F2Finish* define the *F2* state in *cs1*. The 6.42 and 6.44 expressions could be read as *cs1F1Start* occurs either once *clkOnStart* ticks or when *f1Start* occurs removing the ticks that coincidence with *f1Finish*. In this way, we distinguish when *cs1* is in *Off* state and when it is in *F1*.

Similarly, we specify the merge of states in *vs1*:

$$vs1OffStart \boxed{=} pwOffStart \tag{6.46}$$

$$vs1OffFinish \boxed{=} pwOffFinish \tag{6.47}$$

$$vs1V1Start \boxed{=} (v1Start \searrow pwOnStart) + (v1Start - pwOffStart) \tag{6.48}$$

$$vs1V1Finish \boxed{=} v1Finish \tag{6.49}$$

$$vs1V2Start \boxed{=} (v1Start \searrow pwOnStart) + (v2Start - pwOffStart) \tag{6.50}$$

$$vs1V2Finish \boxed{=} v2Finish \tag{6.51}$$

Figure 6.14 presents the scenario simulated in TimeSquare. Such a scenario is used to evaluate the power consumption in *Aceplorer*. In the the figure, first, we depict the *ClockView* and *PowerView* states that we use to define the *Aceplorer* scenario. Next, these states are merged and their CCSL specification is simulated in TimeSquare. This tool generates a VCD file that is transformed to a VCD file that follows the *Aceplorer* model specification. The VCD file is imported by *Aceplorer* and the tool generates a scenario to execute its model. Once completed the scenario and the model in *Aceplorer*, the power consumption of each system component can be analyzed. We depict the power consumption of the *cpu* evaluated in *Aceplorer* (Bottom).

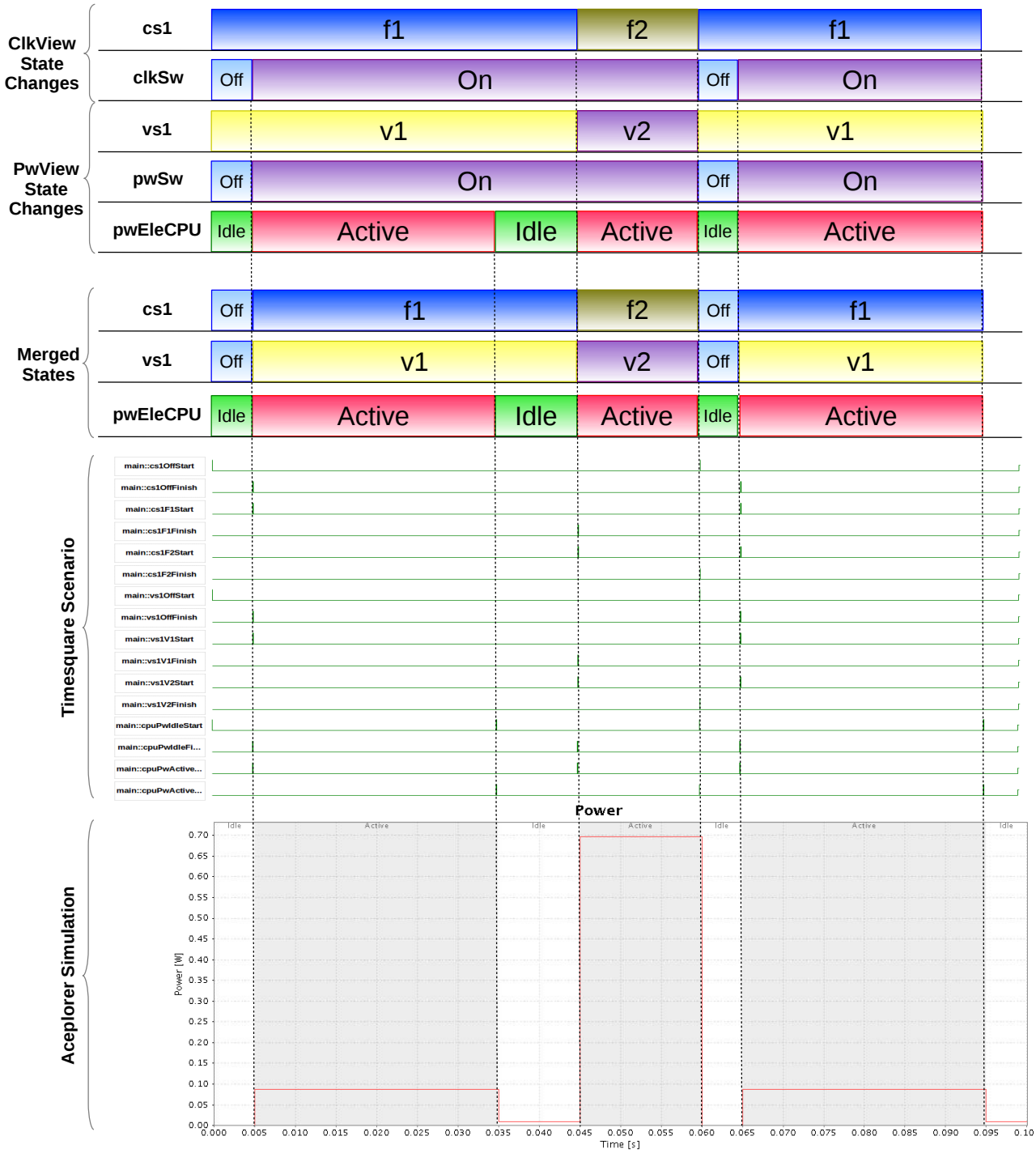


FIGURE 6.14: Control View Scenario generated by TIMESQUARE (above) and the power consumption response in *Aceplorer* (below).

6.4. Conclusion

In this chapter, we have presented the simulation of the *PRISMSYS* power-aware model. We have defined the interaction between *controlSubViews* of different *views* by using the UML sequence diagram. The semantics of this diagram has been specified by clocks and relations in CCSL. By using the semantics defined in Chapter 3, we have specified the *subViewElement* behavior. We have simulated the scenario defined and coordinated by the *controlSubView* to change the *subViewElement* states in TIMESQUARE. We have developed and employed *Scilab Solver* to evaluate the power and thermal equations according to the TIMESQUARE simulation. *Scilab Solver* follows the execution semantics defourfined in Chapter 3. *Scilab Solver* complements the equation evaluation that is not supported by TIMESQUARE, interpreting the TIMESQUARE execution to coordinate the *Scilab* evaluation of the equations. We have presented the results of these simulations in *Scilab*.

We have also introduced another way to analyze the power consumption of the *PRISMSYS* power-aware model by using the MDE transformation technique. We transform the *PRISMSYS* model to *Aceplorer*, which is a power consumption analysis tool. The VCD file generated by TIMESQUARE is employed to build the scenario to evaluate the power consumption in *Aceplorer*.

In the next chapter, we summarize the most important contributions of this thesis and we propose various perspective paths that could be a guide to continue this work.

Chapitre 7

Conclusion (Version en Français)

Dans cette thèse, nous avons montré que la complexité de la conception d'un système peut être gérée en utilisant une approche multi-vue. Une telle approche sépare l'architecture d'un système en différents domaines spécifiques où chaque expert définit le système selon son point de vue. Le standard IEEE-42010, propose une façon générale de spécifier l'architecture d'un système. Cependant, l'architecte du système est libre de définir les vues et les relations parmi eux. En plus, il n'y a pas une manière clairement définie pour spécifier le comportement de l'architecture du système ni son modèle d'exécution.

Nous avons proposé un environnement nommé *PRISMSYS* qui fournit les éléments nécessaires pour décrire l'architecture d'un système en suivant une approche multi-vue. Les concepts de *PRISMSYS* sont inspirés par les concepts définis en l'IEEE-42010. Cependant, *PRISMSYS* spécifie plus en détails la définition d'une vue, ses éléments internes et les associations possibles qui existent entre les différents éléments des vues d'un système. De plus, *PRISMSYS* spécifie la manière dont un modèle multi-vue peut être exécuté de façon cohérente. La structure de *PRISMSYS* suit une approche à composants.

PRISMSYS définit une *vue* en trois types de domaines spécifiques, dénommés *sous-vues* : *contrôle*, *structure* et *équation*. Chacune de ces sous-vues a un rôle spécifique dans la définition d'une *vue*. La sous-vue *structurelle* définit la structure de la vue, la sous-vue de *contrôle* commande les actions des éléments internes de la sous-vue structurelle, et la sous-vue *équationnelle* caractérise les propriétés non fonctionnelles établies dans une vue à travers des *équations*.

PRISMSYS fournit aussi une sémantique spécifique au concept *correspondance*, qui est l'association parmi les éléments de l'architecture du système en accord avec le standard IEEE-42010. La sémantique de *correspondance* est utilisée pour représenter l'abstraction des éléments d'une vue par rapport aux autres éléments d'autres vues (la correspondance d'*Abstraction*), ou pour allouer une action sur un composant hardware (la correspondance d'*Allocation*). *PRISMSYS* spécifie aussi un autre type de correspondance dénommée *sous-correspondance*, qui est l'association entre les *sous-vues* (Celle-ci sont les sous-correspondances d'*Équivalence* et de *Caractérisation*).

PRISMSYS considère la *correspondance* définie par l'IEEE-42010 comme une simple association entre éléments de différentes vues. Deux associations spécifiques sont proposées : l'*Équivalence* et la *Caractérisation*. Entre une vue et une sous-vue, d'autres types de *correspondances* sont proposées : les connecteurs de *contrôle*, de *données* et de *paramètres*. Ces connecteurs partagent des informations qui impactent l'exécution des *vues* ou *sous-vues*. Ces correspondances assurent la cohérence structurelle et sémantique entre les vues et leurs sous-vues.

En utilisant l'Ingénierie Dirigée par les Modèles, la syntaxe de *PRISMSYS* est définie par des *méta-modèles*. La sémantique d'exécution des modèles *PRISMSYS* est décrite dans le language CCSL. Cette définition sémantique permet la simulation du modèle *PRISMSYS* avant l'implantation dans un langage de description de plus bas niveau, comme SystemC ou VHDL. *PRISMSYS* est représenté en UML en spécifiant un profil. Le profil de *PRISMSYS* utilise autant que possible les concepts définis dans les profils OMG SysML et MARTE.

Deux types de comportements d'exécution sont définis en *PRISMSYS* : l'événement discret et le temps continu. Les deux comportements doivent coordonner leurs exécutions afin d'évaluer les propriétés non fonctionnelles définies dans le modèle du système. Grâce à CCSL, la coordination entre ces comportements hétérogènes pourrait être spécifiée dans un environnent homogène. CCSL définit les relations logiques et temporelles pour exécuter le modèle et ne pas s'occuper de la manipulation des données. Cette manipulation est contrôlée par *Scilab*. *Scilab Solver* est un connecteur spécifique qui a été développé afin de gérer la co-simulation entre TimeSquare et l'évaluation des équations en *Scilab*.

PRISMSYS offre une structure pour capturer et unifier la spécification d'un système. Celui ci peut alors être utilisé par les experts des domaines pour transformer une partie

du modèle *PRISMSYS* vers un modèle d'un outil de domaine spécifique. Nous avons illustré cet aspect en transformant le modèle *PRISMSYS* dédié à la consommation de puissance vers le format interne de *Aceplorer* afin d'analyser la consommation de puissance.

7.1. Perspectives

PRISMSYS et son cas d'étude (le modèle *PRISMSYS* dédié à la consommation de puissance) fournissent quelques perspectives pour élargir et améliorer le travail développé dans cette thèse. Nous listons les propositions que nous considérons comme essentielles pour la continuité de ce travail.

- **Employer *PRISMSYS* dans un autre type de systèmes :** Cette thèse définit une structure pour la modélisation multi-vue qui permet de spécifier l'architecture du système et son exécution. *PRISMSYS* peut être étendu en cohérence avec les experts des domaines et le système à concevoir. En conséquence, *PRISMSYS* initialise la construction d'un chemin qui peut être adapté à des autres domaines. Nous illustrons l'utilisation de l'approche *PRISMSYS* dans la modélisation de la consommation de puissance avec l'inclusion des informations de temps. Cependant, cet approche peut être appliquée pour plusieurs types de systèmes, tel que l'automatique, la construction et les systèmes de software.

- **Étendre le comportement du concept *subViewElement* :** Les experts utilisent les machines à états fini et les équations afin de spécifier le comportement de chaque domaine. Cependant, ils/elles emploient aussi d'autres types de modèles, comme les réseaux de Petri et les graphes flots de données. Pour supporter ces autres comportements, le concept *Behavior* du méta-modèle *subViewElement* doit être spécialisé afin de définir la syntaxe du comportement (structure) et alors la sémantique d'exécution en CCSL.

- **Améliorer la spécification de la vue thermique et son impact sur les autres vues :** Nous avons défini une simple vue thermique pour simuler la propriété de température, qui a un comportement non linéaire. Néanmoins, il y a

d'autres concepts qui appartiennent à cette vue. En outre, l'évolution de la température impacte la consommation de puissance statique. Celle-ci est une des caractéristiques qui a le plus d'effet sur la consommation de puissance pour les nouvelles technologies.

- **Généraliser l'interaction parmi différents types de comportements :** Dans *PRISMSYS*, nous spécifions la sémantique d'exécution d'un comportement à événements discrets, modélisé par une machine d'états fini et un diagramme de séquences ; ainsi que le comportement en temps continu, représenté par les équations dans le diagramme de paramètres. La sémantique d'exécution des deux comportements a été formellement spécifiée en CCSL. La coordination entre ces comportements est également définie en CCSL (activer un état, activer une équation afin d'être évaluée). Cependant, plutôt que d'avoir un mécanisme ad-hoc pour implanter la composition hétérogène, nous pourrions dépendre d'environnements plus génériques, tel que *Ptolemy* ou *ModHel'X*, qui s'intéressent explicitement à la composition de modèles de calcul hétérogènes. Implanter PRISMSYS à travers ce type d'environnement permettrait de prendre en compte une sélection plus large de MoC. Cela pourrait se traduire par la définition d'un directeur ou d'un connecteur PRISMSYS.

Chapter 7

Conclusion

In this thesis, we have demonstrated that the complexity of the system design can be managed by using a multi-view approach. Such an approach splits the architecture of a system in various specific domains where experts define the system from their points of view. The IEEE-1470 and IEEE-42010 standards, propose a general way to specify a system architecture. Nevertheless, the system architect is free to define the views and the relationships between them. Moreover, there is not a clear standard way to specify how the behavior of the system architecture and its execution model could be specified.

We have proposed a framework named *PRISMSYS* that provides the elements needed to describe the system architecture following a multi-view approach. The *PRISMSYS* concepts are inspired by the concepts defined in IEEE-42010. Nevertheless, *PRISMSYS* specifies in more details the definition of a view, its internal elements and the possible associations that exist between the various view elements of a system. Furthermore, *PRISMSYS* defines the way a multi-view model can be coherently executed. The structure of *PRISMSYS* follows a component approach, *i.e.,* the system architecture is a modular design whose elements transfer information to each other through ports.

PRISMSYS defines a *view* in three kinds of specific domains named *subViews*: *controlSubView*, *structuralSubView* and *equationalSubView*. Each one of these *subViews* has a specific role in the definition of a *view*. *StructuralSubView* states the structure of the view, *ControlSubView* commands the actions of the internal elements of the *structuralSubView* and *EquationalSubView* characterizes the non-functional properties defined in the view by means of *equations*.

PRISMSYS also provides a specific semantics to *correspondence*, which is the association between the system architecture elements according to IEEE-42010. The framework states a specific semantics to *correspondence* to represent the abstraction of the elements from one view to another (*Abstraction* correspondence), or to map an *action* on a *Hardware Component* by using the *Allocation* correspondence. *PRISMSYS* also specifies another kind of correspondence named *sub-correspondence*, which is the association between *subViews* (*Equivalence* and *Characterization* sub-correspondences).

PRISMSYS identifies that *correspondence* is not the only association between its concepts, there are also *Abstraction*, *Equivalence* and *Characterization*. But *correspondence* can also be a connection between *views* and *subviews*, like *ControlConnector*, *DataConnector* and *ParameterConnector*. These connections share a certain information that impacts the execution of the *views* and *subViews*. Since *correspondences* and *subCorrespondeces* are employed, each *view* or *subViews* can identify the structural and behavioral impact of their elements on other *views* or *subViews*, allowing the right syntactic (structural) and semantics (behavioral) coherence of the *PRISMSYS* model.

By using Model-Driven Engineering, the syntax of *PRISMSYS* is defined by *metamodels*, allowing the reuse of their concepts to build multiple models. The *PRISMSYS* syntax is accompanied by the specification of the execution semantics described in CCSL. This semantics definition allows the simulation of the *PRISMSYS* model before being implemented in a lower-level description language, such as SystemC or VHDL. *PRISMSYS* is represented in UML specifying a profile. The *PRISMSYS* profile uses as much as possible the concepts defined in SysML and MARTE.

Two kinds of execution behaviors are defined in *PRISMSYS*: a discrete event behavior, and a continuous time behavior. Both behaviors have to coordinate their execution in order to evaluate the non-functional properties defined in the system model. Thanks to CCSL, the coordination between these heterogeneous behaviors could be specified in a homogeneous environment. CCSL defines the logical and temporal relations to execute the model and do not deal with data manipulations. This latter aspect is addressed by *Scilab*. A specific connector *Scilab Solver* has been developed to run a co-simulation with TimeSquare and evaluate equations in *Scilab*.

PRISMSYS offers a framework to capture and unify the specification of a system. It can then be used by domain experts to transform part of the *PRISMSYS* model into

a specific domain tool model. We have illustrated this aspect by transforming the *PRISMSYS* power-aware model to *Aceplorer* in order to analyze the system power consumption.

7.1. Future works

PRISMSYS and its case study (the *PRISMSYS* power-aware model) provide some perspectives to extend and to improve the work carried out in this thesis. We list the propositions we consider essential to the continuity of this work:

- **Employing *PRISMSYS* in other kind of systems:** This thesis states a basic multi-view framework that formally allows to specify the system architecture and its execution. This framework can be extended according to the expert domain and the system to design. Therefore, *PRISMSYS* initiates the construction of a path that can be tailored to other domains. We illustrate the use of the *PRISMSYS* approach in a power-aware model with time-related information. However, this approach can be applied to different sorts of systems, such as control, construction and software systems.

- **Extending the *subViewElement* behavior:** Experts use Finite State Machines and equations to specify the behavior of their domains. However, they also employ other kinds of behaviors, such as Petri nets and Synchronous Data Flow graphs. To support these other behaviors, the *subViewElement Behavior* concept must be specialized to define both the syntax of the behavior and then the execution semantics in CCSL.

- **Enhancing the *ThermalView* specification and its impact on the other views:** We have defined a simple thermal view to simulate the temperature property, which has a non-linear behavior. However, there are more concepts that belong to this view and the temperature evolution impacts the static power, and this is one of the features that has more effects in the power consumption for new technologies.

- **Generalizing the interaction between various kinds of behaviors:** In *PRISMSYS*, we specify the execution semantics of a discrete event behavior, modeled by Finite State Machine and Sequence Diagram; as well as a continuous time behavior, represented by equations in a Parametric Diagram. The execution semantics of both behaviors have formally been specified in CCSL. Furthermore, the coordination among them is also defined in CCSL (activating a state, activate an equation to be evaluated). However, rather than having an ad-hoc mechanism to implement the heterogeneous composition, we could rely on more generic environments, such as *Ptolemy* and *ModHel'X*, in which the heterogeneity is addressed explicitly by directors and MoC connectors. This would allow the use of a larger choice of MoCs.

Bibliography

[1] IEEE recommended practice for architectural description of software-intensive systems. *IEEE Std 1471-2000*, pages i –23, 2000.

[2] Systems and software engineering – architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pages 1 –46, 2011.

[3] Charles André, Julien DeAntoni, Frédéric Mallet, and Robert de Simone. *The Time Model of Logical Clocks available in the OMG MARTE profile*, chapter 7, pages 201–227. Springer Science+Business Media, LLC 2010, July 2010.

[4] OMG. Systems Modeling Language (SysML). *Object Management Group*, v1.2, June 2010.

[5] OMG. UML profile for MARTE. *Object Management Group*, v1.1, October 2010.

[6] Julien DeAntoni and Frédéric Mallet. Timesquare: treat your models with logical time. In *Proceedings of the 50th international conference on Objects, Models, Components, Patterns*, TOOLS'12, pages 34–41, Berlin, Heidelberg, 2012. Springer-Verlag.

[7] Scilab Consortium. Scilab. http://www.scilab.org/. [Sep. 10, 2013].

[8] Docea Power. Aceplorer. http://www.doceapower.com/products-services/aceplorer.html. [Mar. 7, 2012].

[9] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[10] Cécile Hardebolle and Frédéric Boulanger. ModHel'X: A component-oriented approach to multi-formalism modeling. In Holger Giese, editor, *Models in Software Engineering*, volume 5002 of *Lecture Notes in Computer Science*, pages 247–258. Springer Berlin Heidelberg, 2008.

[11] Systems and software engineering system life cycle processes. *ISO/IEC 15288:2008(E) IEEE Std 15288-2008 (Revision of IEEE Std 15288-2004)*, pages 1–84, 2008.

[12] U.S. Department of Defence. Department of Defence Architecture Framework (DoDAF). http://dodcio.defense.gov/dodaf20.aspx, 2010. [Jan. 20, 2013].

[13] UK Ministry of Defence. Ministry of Defence Architecture Framework (MODAF). https://www.gov.uk/mod-architecture-framework, 2012. [Jan. 20, 2013].

[14] The Open Group. The Open Group Architecture Framework (TOGAF). http://www.opengroup.org/togaf/, 2008. [Jan. 20, 2013].

[15] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, November 1995.

[16] J.F. Sowa and J.A. Zachman. Extending and formalizing the framework for information systems architecture. *IBM Systems Journal*, 31(3):590–616, 1987.

[17] Elif Demirli and Bedir Tekinerdogan. Software language engineering of architectural viewpoints. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *Software Architecture*, volume 6903 of *Lecture Notes in Computer Science*, pages 336–343. Springer Berlin Heidelberg, 2011.

[18] OMG. MDA guide. *Object Management Group*, v1.0.1, June 2003.

[19] OMG. Meta object facility (MOF) 2.0 core specification. *Object Management Group*, v2.0, October 2003.

[20] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[21] OMG. OMG unified modeling language. *Object Management Group*, v2.4.1, August 2011.

[22] *Architecture Analysis and Design Language (AADL).* SAE, September 2012. AS5506B.

[23] F. Jouault and I. Kurtev. Transforming models with atl. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138, Berlin, 2006. Springer Verlag.

[24] Rich Hilliard, Ivano Malavolta, Henry Muccini, and Patrizio Pelliccione. Realizing architecture frameworks through megamodelling techniques. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 305–308, New York, NY, USA, 2010. ACM.

[25] Mickael Clavreul. *Model and Metamodel Composition: Separation of Mapping and Interpretation for Unifying Existing Model Composition Techniques.* PhD thesis, Université de Rennes, 2011.

[26] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[27] Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools.* Addison-Wesley Professional, first edition, 2004.

[28] Franck Fleurey, Benoit Baudry, Robert France, and Sudipto Ghosh. A generic approach for automatic model composition. In Holger Giese, editor, *Models in Software Engineering*, volume 5002 of *Lecture Notes in Computer Science*, pages 7–15. Springer Berlin Heidelberg, 2008.

[29] M. Nassar. VUML: a viewpoint oriented uml extension. In *Int. Conf. on Automated Software Engineering*, pages 373–376, oct. 2003.

[30] Antonio Cicchetti, Federico Ciccozzi, and Thomas Leveque. Supporting incremental synchronization in hybrid multi-view modelling. In *W. on Models in Software Engineering*, MoDELS Workshops, pages 89–103, Berlin, Heidelberg, 2011. Springer-Verlag.

[31] Frédéric Boulanger, Christophe Jacquet, Cécile Hardebolle, and Elyes Rouis. Modeling heterogeneous points of view with ModHel'X. In *W. on Models in Software Engineering*, MoDELS Workshops, pages 310–324, Berlin, Heidelberg, 2009. Springer-Verlag.

[32] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[33] C. A. R. Hoare. Viewpoint - retrospective: an axiomatic basis for computer programming. *Commun. ACM*, 52(10):30–32, 2009.

[34] Lars-Åke Fredlund, Bengt Jonsson, and Joachim Parrow. An implementation of a translational semantics for an imperative language. In *CONCUR*, pages 246–262, 1990.

[35] Antoon Goderis, Christopher Brooks, Ilkay Altintas, Edward A. Lee, and Carole Goble. Heterogeneous composition of models of computation. *Future Generation Computer Systems*, 25(5):552 – 560, 2009.

[36] Modelica Association. Modelica. http://www.modelica.org. [Jan. 14, 2013].

[37] Mathworks. Matlab. http://www.mathworks.com/products/matlab/, 2013. [Jan. 14, 2013].

[38] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *SEFM*, pages 3–12, 2006.

[39] E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(12):1217–1229, 1998.

[40] Benoit Combemale, Julien Deantoni, Matias Vara Larsen, Frédéric Mallet, Olivier Barais, Benoit Baudry, and Robert France. Reifying concurrency for executable metamodeling. In *the 6th International Conference on Software Language Engineering (SLE 2013)*, oct. 2013.

[41] *EAST-ADL Domain Model Specification.* EAST-ADL Association, May 2013. v2.1.11.

[42] Perry Alexander and Cindy Kong. Rosetta: Semantic support for model-centered systems-level design. *IEEE Computer*, 34(11):64–70, 2001.

[43] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. Archit. Code Optim.*, 1(1):94–125, March 2004.

[44] Wei Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M.R. Stan. Hotspot: a compact thermal modeling methodology for early-stage vlsi design. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(5): 501 –513, may 2006.

[45] Charles André, Frédéric Mallet, and Julien Deantoni. VHDL Observers for Clock Constraint Checking. In *Symposium on Industrial Embedded Systems*, trento, Italie, April 2010. IEEE Computer Society.

[46] OMG. UML Profile for Advanced and Integrated Telecommunication Services (TelcoML). *Object Management Group*, Beta1, January 2012.

[47] Eclipse. Papyrus. http://www.eclipse.org/papyrus/, 2013. [Jan. 20].

[48] Obeo. Uml designer. http://www.obeodesigner.com/, 2013. [Jun. 20].

[49] No Magic. Magic draw. http://www.nomagic.com/. [Jan. 20, 2013].

[50] Modeliosoft. Modelio. http://www.modelio.org/, 2013. [Jun. 20].

[51] IBM. Rational software architect. http://www-03.ibm.com/software/products/us/en/ratisoftarch/, 2013. [Jun. 20].

[52] IBM. Rhapsody. http://www-03.ibm.com/software/products/us/en/ratirhapfami/. [Jun. 20, 2013].

[53] Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[54] D. Helms, E. Schmidt, and W. Nebel. Leakage in CMOS circuits – an introduction. In Enrico Macii, Vassilis Paliouras, and Odysseas Koufopavlou, editors, *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, volume 3254 of *Lecture Notes in Computer Science*, pages 17–35. Springer Berlin / Heidelberg, 2004.

[55] W. Zhang, J. Williamson, and L. Shang. Power dissipation. In *Low-Power Variation-Tolerant Design in Nanometer Silicon*, pages 41–80. Springer Berlin / Heidelberg, 2010.

[56] Miltos D. Grammatikakis, George Kornaros, and Marcello Coppola. Power-aware multicore SoC and NoC design. In *Multiprocessor System-on-Chip*, pages 167–193. Springer, 2011.

[57] D. Flynn, R. Aitken, A. Gibbons, and K. Shi. *Low Power Methodology Manual for System-On-Chip Design*. Integrated Circuits and Systems. Springer, 1 edition, 2007.

[58] Farzan Fallah and Massoud Pedram. Standby and active leakage current control and minimization in cmos vlsi circuits. *IEICE Transactions*, 88-C(4):509–519, 2005.

[59] Mostafa E. A. Ibrahim, Markus Rupp, and Hossam A. H. Fahmy. A precise high-level power consumption model for embedded systems software. *EURASIP J. Embedded Syst.*, 2011:1:1–1:14, January 2011.

[60] Laurence W. Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. PhD thesis, EECS Department, University of California, Berkeley, 1975. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/1975/9602.html.

[61] Charlie X. Huang, Bill Zhang, An-Chang Deng, and Burkhard Swirski. The design and implementation of powermill. In *Proceedings of the 1995 international symposium on Low power design*, ISLPED '95, pages 105–110, New York, NY, USA, 1995. ACM.

[62] T.-L. Chou and K. Roy. Accurate power estimation of cmos sequential circuits. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 4(3):369–380, 1996.

[63] Chih-Shun Ding, Chi ying Tsui, and M. Pedram. Gate-level power estimation using tagged probabilistic simulation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(11):1099–1107, Nov 1998.

[64] W. Ye, N. Vijaykrishnan, M. Kandemir, and M.J. Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *Design Automation Conference, 2000. Proceedings 2000*, pages 340 –345, 2000.

[65] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mary Jane Irwin, N. Vijaykrishnan, Mahmut Kandemir, Tao Li, and Lizy Kurian John. Using complete machine simulation for software power estimation: The softwatt approach. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, page 141, Washington, DC, USA, 2002. IEEE Computer Society.

[66] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. *SIGARCH Comput. Archit. News*, 28(2):83–94, May 2000.

[67] F. Klein, G. Araujo, R. Azevedo, R. Leao, and L.C.V. dos Santos. An efficient framework for high-level power exploration. In *Circuits and Systems (MWSCAS)*, pages 1046–1049, Aug 2007.

[68] IEEE standard for standard SystemC language reference manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, 9 2012.

[69] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445, 1994.

[70] V. Konstantakos, A. Chatzigeorgiou, S. Nikolaidis, and Th. Laopoulos. Energy consumption estimation in embedded systems. In *Instrumentation and Measurement Technology Conference, 2006. IMTC 2006. Proceedings of the IEEE*, pages 235 –238, april 2006.

[71] J. Laurent, N. Julien, E. Senn, and E. Martin. Functional level power analysis: an efficient approach for modeling the power consumption of complex processors. In *Design, Automation and Test in Europe*, volume 1, pages 666–667, Feb 2004.

[72] Takumi Okuhira and Tohru Ishihara. Unified gated flip-flops for reducing the clocking power in register circuits. In JoséL. Ayala, Braulio García-Cámara, Manuel Prieto, Martino Ruggiero, and Gilles Sicard, editors, *Integrated Circuit and System Design. Power and Timing Modeling, Optimization, and Simulation*, volume 6951 of *Lecture Notes in Computer Science*, pages 237–246. Springer Berlin Heidelberg, 2011.

[73] *OMAP35x Applications Processor Technical Reference Manual*. Texas Instruments, Apr 2010.

[74] A. Ejlali, B.M. Al-Hashimi, and P. Eles. Low-energy standby-sparing for hard real-time systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(3):329 –342, march 2012.

[75] A. Genser, C. Bachmann, C. Steger, R. Weiss, and J. Haid. Power emulation based dvfs efficiency investigations for embedded systems. In *System on Chip (SoC), 2010 International Symposium on*, pages 173 –178, sept. 2010.

[76] IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, 2009.

[77] IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 2006.

[78] Accellera. Unified power format 1.0. http://www.accellera.org/activities/p1801_upf, 2007.

[79] IEEE. IEEE standard for design and verification of low power integrated circuits. *IEEE Std 1801-2009*, pages C1–218, 2009.

[80] Silicon Integration Initiative. *Common Power Format Specification 2.0*. Silicon Integration Initiative, Inc., Feb 2012.

[81] Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '03, pages 19–24, New York, NY, USA, 2003. ACM.

[82] Ons Mbarek, Alain Pegatoquet, and Michel Auguin. A methodology for power-aware transaction-level models of systems-on-chip using upf standard concepts. In *PATMOS*, pages 226–236, 2011.

[83] Matthias Hagner, Adina Aniculaesei, and Ursula Goltz. UML-based analysis of power consumption for real-time embedded systems. In *Int. Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1196–1201, Nov. 2011.

[84] Tero Arpinen, Erno Salminen, Timo D. Hämäläinen, and Marko Hännikäinen. MARTE profile extension for modeling dynamic power management of embedded systems. *Journal of Systems Architecture*, 2011.

[85] Dongsheng Ma and R. Bondade. Enabling power-efficient DVFS operations on silicon. *Circuits and Systems Magazine, IEEE*, 10(1):14–30, 2010.

[86] Bureau International des Poids et Mesures. The International System of Units (SI). pages 1–180, 2006.

[87] A. Abdallah, A. Gamatie, and J. Dekeyser. Correct and energy-efficient design of socs: The h.264 encoder case study. In *System on Chip (SoC), 2010 International Symposium on*, pages 115–120, 2010.