UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE NANTES

THESE DE DOCTORAT DE

L'UNIVERSITE DE NANTES
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Electronique

Par
Tien Thanh NGUYEN

Model-Driven Architecture Exploration for Fault Tolerance Improvement

Thèse présentée et soutenue à Nantes, le 15 Novembre 2019
Unité de recherche : IETR UMR CNRS 6164

Rapporteurs avant soutenance :

Alexandre Nketsa	Professeur, Université Toulouse 3
Samy Meftali	Maître de conférences, HDR, Université de Lille 1

Composition du Jury :

Président :	Alexandre Nketsa	Professeur, Université Toulouse 3
Examinateurs :	Florence Maraninchi	Professeur, INP Grenoble
	Samy Meftali	Maître de conférences, HDR, Université de Lille 1
	Olivier Pasquier	Maître de conférences, Université de Nantes
Invité :	Anthony Mouraud	Chef de projets, CEA Tech Nantes
Dir. de thèse :	Sébastien Pillement	Professeur, Université de Nantes
Co-dir. de thèse :	Mathieu Thévenin	Chercheur, CEA - Paris Saclay

Projet de thèse :

Cette thèse est principalement préparée au sein du laboratoire IETR (Institut d'Électronique et de Télécommunications de Rennes), UMR CNRS 6164 à l'Ecole polytechnique de l'université de Nantes, avec des séjours réguliers aux CEA Tech à Nantes et au CEA Saclay.

La thèse est financée par un projet HOLISTAR de ressourcement CEA Tech en région Pays de la Loire.

Acknowledgments

I would like to start my thesis to talk about what is important in my 3-year journey. It was not a long time but it was enough for me to discover not only knowledge but also a process of hearing and identifying myself. It was also a journey that I met and worked with new people in France, a beautiful land far from Vietnam where I was born.

First, I would like to thank my advisors, Mathieu THÉVENIN, Researcher at CEA Paris Saclay, and Sébastien PILLEMENT, Professor at Université de Nantes. They gave me dedicated instructions that played a key role in my research process. Mathieu has given me good advice for one year I worked at CEA. And Sébastien, whose clear insights into the direction of the job, has challenged me with his full-of-ideas questions for two years at Polytech Nantes.

Next, I would like to thank my co-advisors, Anthony MOURAUD, Project Manager at CEA Tech Nantes, and Olivier PASQUIER, Lecturer at Université de Nantes. The interesting debates with Anthony showed me meaningful ideas when I was working at CEA Tech Nantes. Meanwhile, Olivier's caution kept me on track in my works at IETR Polytech. They helped me lead my work further and more fully.

I want to express my sincere gratitude to all the members of the jury. I begin by thanking Mr. Alexandre NKETSA, Professor at Université Toulouse 3, and Mr. Samy MEFTALI, Lecturer, HDR, Université de Lille 1, for having accepted the responsibility of rapporteur. I express my gratitude to MM. Florence MARANINCHI, Professor at INP Grenoble, to have participated in the jury of my thesis.

Special thanks to the secretaries, Sandrine CHARLIER at IETR Polytech Nantes and Nathalie FEIGUEL at LCAE/DM2I CEA for taking care of all my administrative issues with their ultimate goodness.

I also want to express my gratitude to my friends, brothers, colleagues that I can hardly name them all. They are an integral part of my life.

And last but not least, I want to send my love to my family, my father Trong Hoang NGUYEN, my mother Thi Ngan HOANG, my brother and sister-in-law Duc Canh NGUYEN and Thi Thao DO, and my nephew Hoang Duong NGUYEN. They all are my motivation, my love, and my spiritual support even though they stay thousands of kilometers away from me.

Finally, I would like to conclude this section with a verse in the poem "Cáo tật thị chúng" of Zen Master Man Giac (1052-1096), a famous Zen Master in Vietnamese history:

"*Mạc vị xuân tàn hoa lạc tận*

Đình tiền tạc dạ nhất chi mai."

("*Do not say that the spring is over, all the flowers fall off*

Last night in the front yard there was a branch of apricot blossom.")

Thank you all,

Tien Thanh NGUYEN

Contents

List of figures 9

List of tables 11

Acronyms 15

Résumé 17

1 Introduction 21
- 1.1 Embedded system 22
- 1.2 Fault model 25
- 1.3 Tolerance strategies 27
  - 1.3.1 Spatial redundancy 27
  - 1.3.2 Temporal redundancy 28
  - 1.3.3 Check-pointing 28
  - 1.3.4 Error correction code 29
- 1.4 Problem statement and contributions 30

2 Literature reviews 31
- 2.1 Model driven engineering 32
  - 2.1.1 Model-Based Testing 33
  - 2.1.2 Model-driven development 33
- 2.2 Static and dynamic exploration 39
  - 2.2.1 Design-time exploration 40
  - 2.2.2 Run-time exploration 43
  - 2.2.3 Fault-tolerance and reliability-based exploration 45
  - 2.2.4 Analysis 48
- 2.3 Summary 52

3 Fault-tolerant platform meta-model 53
- 3.1 Platform meta-model 54
  - 3.1.1 Proposed meta-model 54
  - 3.1.2 Fault tolerance in the meta-model 58
- 3.2 Supporting tool 64
- 3.3 Summary 67

4 Design space exploration **69**
4.1 Design space 70
4.1.1 Initialization 71
4.1.2 Mapping process 73
4.1.3 Solution evaluation 86
4.2 Optimization process 95
4.2.1 Objective function 95
4.2.2 Search strategies 96
4.3 Summary 97

5 Experimental evaluation **99**
5.1 Case-study description 100
5.1.1 Sobel filter 100
5.1.2 Harris detector 102
5.2 DSE results 104
5.2.1 Sobel filter 104
5.2.2 Harris detector 109
5.3 Summary 117

6 Conclusions and perspectives **119**
6.1 Conclusions 120
6.2 Perspectives 121

Bibliography **122**

Publications **133**

Appendices **135**

A Code implementation **137**
A.1 Sobel filter code implementation 137
A.2 Harris conner dectector code implementation 138
A.3 Latency measurement on Microblaze and ARM 141

List of Figures

1.1 An example of target application with 4 functions and their properties. 22
1.2 Overview of a heterogeneous Multi-Processor System-on-Chip (MPSoC). 23
1.3 Abstraction levels of design space exploration [4]. 24
1.4 Relation between fault, error and failure. 26
1.5 A transient fault appears and disappears on a Processing Element (PE) component. . 26
1.6 An instance of spatial redundancy: Triple Modular Redundancy (TMR) (Triple Modular Redundancy). 28
1.7 An instance of temporal redundancy: Triple Re-execution Redundancy (TReR) (Triple Re-execution Redundancy). 28
1.8 Subsystem without/with checkpoint. 29
1.9 Memory with correction code. 29

2.1 System, models and technical space [26]. 32
2.2 Model Driven Engineering (MDE) classification. 33
2.3 Hardware architecture meta-model proposed by Graphical Array Specification for Parallel and Distributed Computing (GASPARD) [34] describes the hardware architecture at a) the clock cycle level and b) the timed programmer's view level. 35
2.4 Application and platform meta-model for the heterogeneous multiprocessor architectures with time-triggered execution paradigm [38]. 36
2.5 Platform Application Model (PAM) metamodel [40]. 37
2.6 Meta-model proposed by ModES [41]. 38
2.7 Classification of exploration methodology. 39
2.8 Design-time exploration chart. 41
2.9 Run-time exploration flow. 43
2.10 Review the reliability-aware Design Space Exploration (DSE) studies. 49

3.1 architectural part related to the component. Gray elements refer to the Model-driven Embedded System (ModES) platform meta-model [41]. White elements are defined in this thesis. 54
3.2 an example of an MPSoC platform. 56
3.3 proposed meta-model of subsystem. Gray and bold elements refer to Figure 3.1. Violet and italic elements represents the subsystem level. 57
3.4 an example: a *compCommunication* in a *subPPE*. 58
3.5 fault tolerance strategies on the proposed platform meta-model. 59

3.6 relation between the simplified architectural part and the fault tolerance part of the proposed platform meta-model. The left part (Architectural part) refers to Figure 3.8. The right part (Fault-tolerance part) refers to Figure 3.7. 60
3.7 fault tolerance part of the proposed platform meta-model. 61
3.8 the simplified architectural part related to the fault tolerant. The $platform$, $subDPE$, $subPPE$, $subCommunication$, $subMemory$ and $compHardware$ elements refer to Figure 3.3. 62
3.9 overall view of the platform meta-model. The left part (Architectural part) refers to Figure 3.1, 3.3 and 3.8. The right part (Fault-tolerance part) refers to Figure 3.5. . . . 63
3.10 illustration of the workbench for the modeling of a 6-subsystem platform. 64
3.11 four big buttons in the left have changed from the gray to pink, meaning the DSE process is complete and the result has been found. The best solution is found shown in the workbench. 65
3.12 workbench inside a subsystem diagram; the centre is the Subsystem Diagram; the palette on the right is composed of the tools that allow to modify, customize the subsystem, the box in the bottom shows parameters of the Subsystem. 66
3.13 table generated automatically in terms of fault tolerance and cost of the platform according to the best solution found in a DSE process. 66

4.1 proposed design solution generator flow. 70
4.2 a platform template and a component list prepared before a DSE process. 73
4.3 Description of $cM_{sub_i}^{compE_j}$. 75
4.4 Description of $fM_{F_j}^{sub_i}$. 75
4.5 Description of $dM_{d_{j_k}}^{sub_i}$. 75
4.6 Description of $tM_{sub_i}^{k}$. 75
4.7 Illustration of component list, platform template, and component mapping. 76
4.8 Application graph, platform, and function mapping. 79
4.9 An example of the component mapping does not respect Constraint 4.3. 84
4.10 Connection between two functions. 87
4.11 An example of function mapping and scheduling. 88
4.12 A transient fault appears and disappears on a PE component. 89
4.13 Multi functions are mapped on the same PE subsystem with TMR. 90
4.14 Multi functions are mapped on the same PE subsystem with TReR. 92

5.1 the Directed Acyclic Graph (DAG) of the Sobel-filter application 100
5.2 platform template for the case study-1 is composed of 6 subsystems: 4 $subPE$s (in blue), 1 $subCommunication$ (in green), and 1 $subMemory$ (in white) 101
5.3 the DAG of the 10-function application in the case study-3. 102
5.4 platform template in the case study-2 is composed of 15 subsystems. 103
5.5 Impact of the number of steps on the probability to find the best solution. 106
5.6 result found and shown on the DSE tool by all three search strategies (a) with $\{\alpha = 0.5, \beta = 0.5, \gamma = 0.0\}$. (b) with $\{\alpha = 0.5, \beta = 0.0, \gamma = 0.5\}$. 107
5.7 initial solution in which each PE subsystem only performs one function, "INITIAL 1", 110
5.8 initial solution in which all functions are mapped into a single processor, "INITIAL 2", 111
5.9 good platform found with $\{\alpha = 1.0, \beta = 0.0, \gamma = 0.0\}$ by Simulated-Annealing (SA), . 112
5.10 good platform found with$\{\alpha = 0.5, \beta = 0.25, \gamma = 0.25\}$ by SA. 113

5.11 platform template used in the DSE process with $\{\alpha = 1.0, \beta = 0.0, \gamma = 0.0\}$ by Genetic Algorithm (GA). 113
5.12 platform template used in the DSE process with $\{\alpha = 0.5, \beta = 0.25, \gamma = 0.25\}$ by GA. 114
5.13 result found and shown on the DSE tool with $\{\alpha = 1.0, \beta = 0.0, \gamma = 0.0\}$, ... 114
5.14 result found and shown on the DSE tool with $\{\alpha = 0.5, \beta = 0.25, \gamma = 0.25\}$, ... 115

List of Tables

1.1 Targets for the probabilistic metric for random hardware failure [9]. 25

2.1 MDE studies for embedded system development. 39
2.2 summary of design-time exploration studies on multicore/multiprocessor architecture. . 42
2.3 Summary of run-time & hybrid exploration studies on multicore/multiprocessor architecture. 44
2.4 Summary of reliability-based application mapping studies on multiprocessors architectures. 50

3.1 quality of service and properties specializations. 55

4.1 Description and notation of parameters and variables used in ILP formulation. 74
4.2 List of equations to calculate the execution time for each subsystem type depending fault tolerance strategy. 87
4.3 List of equations used to calculate the cost of fault tolerance strategies of a subsystem C_{sub_i}. 94

5.1 details of functions and data in the Sobel application. 100
5.2 component parameter list in the case study-1. 101
5.3 functions and data in the Harris application. 102
5.4 component parameter list in the case study-2. 103
5.5 parameters in the objective function of Case-study 1. 104
5.6 GA parameters in Case-study 1. 105
5.7 SA parameters in Case-study 1. 105
5.8 reliability, execution time and cost of found solutions in the case study-1. 108
5.9 failure rate of the system. 108
5.10 found solutions in the case study-1 with $\{\alpha = 0.99, \beta = 0.0, \gamma = 0.01\}$. 109
5.11 three search strategies in the DSE process of Case-study 1. 109
5.12 parameters in the objective function of Case-study 2. 110
5.13 reliability, execution time and cost of solutions found by DSE process with $\{\alpha = 1.0, \beta = 0.0, \gamma = 0.0\}$. 115
5.14 reliability, execution time and cost of solutions found by DSE process with $\{\alpha = 0.5, \beta = 0.25, \gamma = 0.25\}$. 116
5.15 improvement of the solution found by DSE process compared to the "naive" initial solutions with $\{\alpha = 1.0, \beta = 0.0, \gamma = 0.0\}$. 116
5.16 improvement of the solution found by DSE process compared to the initial solutions with $\{\alpha = 0.5, \beta = 0.25, \gamma = 0.25\}$. 116

Acronyms

ACG Application Characteristic Graph.

ADAG Annotated Directed Acyclic Graph.

ASIC Application-Specific Integrated Circuit.

ASIL Automotive Safety Integrity Level.

CABA Cycle-Accurate Bit-Accurate.

CIM Computational Independent Modes.

COMM Communication.

CS Comprehensive Search.

DAG Directed Acyclic Graph.

DP Dynamic Programming.

DPE Dedicated PE.

DSDE Domain Specific Development Engineering.

DSE Design Space Exploration.

DVFS Dynamic Voltage/Frequency Scaling.

EA Evolutionary Algorithm.

FIFO First In, First Out.

FIT Failure In Time.

FPGA Field Programmable Gate Array.

GA Genetic Algorithm.

GASPARD Graphical Array Specification for Parallel and Distributed Computing.

GDSE Gradient-based Design Space Exploration.

GPP General Purpose Processor.

GPPs General Purpose Processors.

GUI Graphical User Interface.

ILP Integer Linear Programming.

IP Intellectual Property.

IP-cores Intellectual Property cores.

ISPs Instruction Set Processors.

LUT Look-Up Table.

MBT Model-Based Testing.

MDA Model Driven Architecture.

MDD Model-Driven Development.

MDE Model Driven Engineering.

MDSD Model-Driven Software Development.

ME Memory.

ModES Model-driven Embedded System.

MOF Meta-Object Facility.

MOST Multi-Objective System Tune.

MPSoC Multi-Processor System-on-Chip.

MPSoCs Multi-Processor System-on-Chips.

MTTF Mean Time To Failure.

NoC Network on Chip.

NSGA-II Non-dominated Sorting Genetic Algorithm-II.

OCL Object Constraint Language.

OMG Object Management Group.

PAM Platform Application Model.

PE Processing Element.

PEs Processing Elements.

PIM Platform Independent Model.

PPE Programmable PE.

PSM Platform Specific Model.

RTL Register Transfer Level.

SA Simulated-Annealing.

SDF Synchronous Dataflow.

SDFG Synchronous Data-Flow Graph.

SER Soft Error Rate.

SEUs Single-Event Upsets.

SIL Safety Integrity Level.

SOFT Software.

SUS System Under Study.

SUT System Under Test.

TMR Triple Modular Redundancy.

TPV Timed Programmer's View.

TReR Triple Re-execution Redundancy.

UML Unified Modeling Language.

wcet worst-case execution time.

XMI XML Model Interchange.

XML Extensible Markup Language.

Résumé

Dans le processus de conception d'un système MPSoC hétérogène (MPSoC : Multi Processor System on Chip), un large espace de solutions de conception émerge de différentes solutions alternatives. La fiabilité est la probabilité qu'un système ou un composant remplisse les fonctions requises sans défaillance dans les conditions spécifiées pendant une période donnée. Par conséquent, la fiabilité est devenue l'une des propriétés les plus importantes des systèmes embarqués. Outre les efforts visant à améliorer la fiabilité du matériel, il est nécessaire d'utiliser des stratégies de tolérance aux fautes afin de réduire l'impact des fautes au niveau du système. La tolérance aux fautes est la capacité d'une unité fonctionnelle à continuer à exécuter une fonction requise en présence de fautes ou d'erreurs. Il existe plusieurs stratégies de tolérance aux fautes développées dans la littérature comme redondance triple modulaire, checkpoiting, code de correction d'erreur. Par conséquent, les concepteurs ont besoin d'une méthodologie qui intègre l'évaluation de la fiabilité dans le processus d'exploration de l'espace de conception (DSE: Design Space Exploration). Notre objectif est d'établir un cadre permettant de trouver la meilleure solution pour une application donnée dans des contraintes de tolérance aux fautes. Cet objectif s'inscrit dans le contexte de la prise en compte de l'impact des stratégies de tolérance aux fautes (permanentes et transitoires) sur la fiabilité et le temps d'exécution du système et le coût de la plate-forme. De plus, le mapping des composants, le mapping des données et le mapping des fonctions sont des aspects importants qui affectent la fiabilité du système. Nos contributions et travaux en cours peuvent être résumés par les points suivants:

1) Nouveau méta-modèle intégrant la tolérance aux fautes pour les systèmes embarqués. Le méta-modèle est au cœur de la méthode d'ingénierie dirigée par les modèles. Ceci propose de couvrir la tolérance de fautes; sert de pont entre les différents outils, entre les différents langages de programmation et les différentes étapes de conception, permettant ainsi aux concepteurs de disposer d'une vue cohérente et unifiée d'une plateforme MPSoC. Avec des modèles, les concepteurs peuvent configurer l'exploration en fonction de leur propre niveau d'expertise, tout en faisant abstraction de la complexité et des spécificités de l'architecture cible. Cela favorise la portabilité et la réutilisation des conceptions en fournissant des modèles tout en résumant les détails de bas niveau du système considéré.

2) La nouvelle méthode DSE comprend la génération, l'évaluation et l'optimisation d'espaces de conception de la tolérance aux fautes. Dans la spécification utilisateur, les dimensions explorées incluent le choix du matériel, le mapping des tâches, le mapping des données et le choix de la stratégie de tolérance aux fautes. Une nouvelle solution est générée et évaluée en matière de temps d'exécution, de coût et de niveau de fiabilité. Ensuite, un processus d'optimisation explore la meilleure solution parmi les espaces de conception.

3) Evaluation de la plate-forme MPSoC hétérogène sous l'impact des fautes transitoires et permanentes. Cette évaluation est une partie très importante de la DSE pour aider les concepteurs à choisir la stratégie de tolérance aux fautes appropriée en ce qui concerne un compromis avec les exigences de l'application.

4) Un nouvel outil avec une interface utilisateur graphique permet de modéliser et d'exécuter le processus DSE. Il simplifie le processus en interagissant avec l'utilisateur via l'interface graphique et en automatisant le processus d'exploration de l'espace de conception.

L'ingénierie dirigée par les modèles (MDE: Model Driven Engineering) peut fournir des moyens efficaces pour résoudre les besoins de la DSE. Un modèle représente une abstraction d'un système ainsi que des éléments de ce système d'un point de vue de la conception. Les mécanismes de construction de modèles valides sont spécifiés dans des méta-modèles. De nombreux travaux ont fourni des méta-modèles pour les structures et le comportement des plates-formes MPSoC. Cependant, ces méta-modèles sont conçus pour des objectifs spécifiques tels que les architectures avec paradigme d'exécution déclenché par le temps, l'estimation des performances ou la génération de codes pour la simulation ou à un niveau d'abstraction bas, ce qui les rend difficiles à réutiliser. Dans tous les cas, les méta-modèles présentés dans la littérature ne sont pas développés pour l'évaluation de la fiabilité. En outre, dans l'exploration de l'espace de conception, il y a de nombreux efforts de recherche qui combinent le mapping des composants, des fonctions ou des données avec l'évaluation de la fiabilité. Cependant, ces travaux considèrent souvent ces objectifs individuellement ou sont conçus pour des architectures spécifiques (processeurs homogènes ou uniquement).

Il existe de nombreuses études DSE basées sur la fiabilité. L'objectif commun est d'améliorer la fiabilité d'une plate-forme face aux exigences d'une application multifonction. Différentes approches à différents niveaux du système ont été étudiées. Ces études utilisent des modèles d'application et de plate-forme en tant qu'entrée de leur processus DSE, puis fournissent une solution optimale avec la cartographie. Plusieurs stratégies de tolérance aux fautes sont utilisées, telles que la réplication de ressources, la réplication de tâches, le point de contrôle, la réexécution. Différentes stratégies de recherche ont été utilisées (Recuit Simulé, Algorithme Génétique). Cependant, il reste des points dans le processus d'exploration qui n'ont pas été examinés ou considérés séparément, telles que l'impact des deux types de faute, l'impact de différents types de composants et l'impact de la tâche ou de la fonction et la cartographie des données sur la fiabilité globale.

Nous avons développé un nouveau méta-modèle de plateforme intégrant la tolérance aux fautes. Le méta-modèle est construit en utilisant la syntaxe UML. Un niveau de représentation intermédiaire est créé appelé "sous-système". Par le biais des "sous-systèmes", les parties de tolérance aux fautes sont connectées à la partie architecturale dans le méta-modèle. Dans la partie architecturale, une plate-forme MPSoC hétérogène est composée de composants (matériels et logiciels) configurés pour fournir un ensemble de services (mémorisation, exécution, etc.). Un composant peut être une mémoire, un composant d'interconnexion ou un processeur. La plate-forme peut contenir plus d'un type de processeur ou d'unité de traitement (terme: hétérogène). La partie de la tolérance aux fautes déclare des stratégies de tolérance aux fautes comme la redondance de temps, la redondance de composants. Un sous-système est composé d'un type de composant matériel et éventuellement de plusieurs versions de composants logiciels. La description des sous-systèmes prend en charge la modélisation de la redondance utilisée dans les stratégies de tolérance. Sur une plate-forme, si des composants individuels sont observés séparément, il sera difficile d'évaluer la fiabilité d'une fonction et, par conséquent, il est difficile d'évaluer la fiabilité d'une plate-forme pour une application. En attendant, l'utilisation du concept de sous-système n'affecte pas l'unité de la plate-forme entière. Ainsi, le "sous-système" constitue non seulement le niveau intermédiaire entre le niveau de la plate-forme et celui des composants, mais également un pont entre un modèle de plate-forme et un modèle d'application. Évaluer une

plate-forme sans tenir compte d'une application n'est pas utile, mais si le modèle de plate-forme et le modèle d'application sont trop étroitement liés, la capacité de réutilisation et l'extensibilité de la plate-forme ne sont que considérablement limitées dans la cadre de l'application. Le niveau de sous-système garantit à la fois la liaison du modèle de plate-forme avec le modèle d'application, mais également une indépendance par rapport au modèle de plate-forme, notamment en termes d'évaluation de la fiabilité.

Les trois parties principales du flux DSE sont composées des processus d'initialisation, de mapping et d'évaluation de la solution. L'initialisation est l'étape de préparation d'un processus DSE. À cette étape, une application considérée, une plate-forme et les composants disponibles sont initialisés avec leurs paramètres. Les méta-modèles de plate-forme sont la base du processus DSE.

À partir des modèles d'entrée, un processus de mapping est effectué pour trouver une solution de mapping. Le problème peut être transformé en un problème de programmation linéaire entier. Le mapping comprend un ensemble de règles, des contraintes pour allouer des ressources et pour mapper les fonctions requises, des données sur les ressources. Une règle est un guide pour définir la relation entre les éléments (application, plate-forme) dans un système MPSoC, qui est stricte et impossible à modifier dans un cadre DSE spécifique. Une contrainte est une limitation pour une ou plusieurs fonctionnalités (quantité ou qualité) d'un système MPSoC qui peut être modifiée ou définie par les concepteurs dans un cadre DSE spécifique. Dans la deuxième étape (processus de mapping), il y a quatre règles: mapping de composant, mapping de fonction, mapping de données, choix des stratégies de la tolérance aux fautes. Ces règles sont implémentées sous forme d'algorithmes permettant de créer une conception dans la forme la plus élémentaire sans aucune contrainte. Ensuite, la conception est vérifiée sous contraintes. Les contraintes sont moins obligatoires que les règles et peuvent être modifiées et remplacées en fonction des objectifs des concepteurs, tels que la limitation des ressources, de la technologie et du temps. Dans nos travaux, il existe quatre contraintes: limitation de la capacité (de l'élément de traitement), limitation de la taille des données, limitation de la quantité et limitation du chemin de connexion entre les composants d'une plate-forme. S'il répond aux contraintes (solution valable), la solution passe à la troisième partie (processus d'évaluation de la solution); sinon (une solution non valide), nous devons revenir en arrière pour créer un autre design.

Ensuite, il est possible de passer par une série d'évaluations. Le processus d'évaluation permet aux concepteurs d'examiner la solution de manière quantitative, ce qui constitue la base du choix de la meilleure solution lors des prochaines étapes de la DSE. Il y a trois évaluations en matière de temps d'exécution, de fiabilité et de coût. Avec l'évaluation du temps d'exécution d'une application, tout d'abord, le temps d'exécution de chaque fonction est estimé. Ensuite, une stratégie de planification est appliquée à l'application et enfin, le temps d'exécution de l'application donnée est estimé. La fiabilité d'un système pendant une exécution est la probabilité qu'aucune défaillance ne survienne sur le résultat du système pendant toute la durée.

L'évaluation de la fiabilité est réalisée à partir d'un modèle de fautes permanentes et transitoires. Une faute est un événement qui cause un défaut dans le composant, tel qu'un bogue logiciel, un blocage, un circuit cassé. Une erreur fait référence à une différence entre la sortie réelle et la sortie attendue lorsqu'une opération requise est exécutée. Une défaillance apparaît lorsqu'un système ou un sous-système ne parvient pas à exécuter une fonction requise conformément à ses spécifications. Il y a deux catégories de la faute dérivées de ses conséquences pour des composants: permanente et transitoire. Les fautes permanentes découlent d'une destruction matérielle telle que le vieillissement, des circuits électroniques cassés ou bloqués. Une faute permanente reste active jusqu'à ce qu'une intervention de correction soit effectuée. Les fautes transitoires peuvent être causées par une seule particule ionisante frappant un nœud sensible ou une interconnexion d'un circuit. Contrairement aux

fautes permanentes, une faute transitoire ne reste active que pendant une courte duration et disparaît à la prochaine duration de fonctionnement. Taux de défaillance (FIT : Failure In Time) est l'inverse du temps moyen de fonctionnement avant une défaillance. Cette valeur représente la possibilité d'une faute entraînant une défaillance lors de l'opération d'un composant. Ces paramètres reflètent les expériences existantes des processus d'injection, de simulation, de prévision des défauts et d'expérimentation sur les composants utilisés. Avec le taux de défaillance, nous avons utilisé des distributions de probabilité pour modéliser l'impact des fautes permanentes et transitoires sur le système. Dans cette thèse, deux stratégies de tolérance aux fautes sont considérées: Redondance Triple Modulaire (TMR), Redondance Triple Ré-exécutions (TReR). Sur la base du modèle de probabilité des fautes, nous proposons des équations mathématiques pour évaluer la fiabilité de chaque sous-système, puis la fiabilité globale d'une plate-forme.

Le coût d'une plate-forme est le coût total de tous les composants actifs utilisés dans cette plate-forme. L'évaluation des coûts vise à assurer l'équilibre entre les ressources matérielles et les objectifs de performance et de fiabilité lors de la conception d'un système. Chaque composant a une valeur de coût.

Il y a trois critères de performance d'une solution de mapping: temps d'exécution, niveau de fiabilité et coût. L'objectif est de maximiser le niveau de fiabilité (maximum = 1), de minimiser les coûts et de minimiser le temps d'exécution. Fondamentalement, avec l'optimisation d'un problème multi-objectifs, une formule est configurée pour comparer deux solutions basées sur la méthode de métriques pondérée. Une fonction mathématique objectif est proposée et elle permet d'évaluer simultanément trois sous-objectifs d'une solution de mapping. La stratégie la plus classique de l'optimisation est la recherche exhaustive (CS: Comprehensive Search). Trois stratégies de recherche telles que la recherche complète, le recuit simulé et l'algorithme génétique consistent à trouver la meilleure solution parmi l'espace de conception possible. Nous allons calculer à partir de la première solution jusqu'à la solution finale, sans manquer aucune solution possible dans l'espace de conception. Le recuit simulé est un algorithme d'optimisation basé sur la simulation d'un processus de refroidissement du métal, du verre ou du cristal. L'algorithme génétique est basé sur le paradigme néo-darwinien de simulation de l'évolution naturelle des systèmes biologiques.

Deux cas d'étude sont présentés, le filtre Sobel et le détecteur de coin Harris. Le filtre Sobel est une application simple et très répandue qui permet de détecter les contours des images. De plus, le détecteur de coin Harris est utilisé dans le traitement d'image pour extraire les coins d'une image, qui s'appuie sur le filtre Sobel. Le détecteur de coin Harris contient plus de fonctions que le filtre Sobel et est appliqué pour augmenter la complexité des études de cas pour notre framework. Les résultats expérimentaux ont montré que le cadre DSE permet une exploration efficace d'un grand espace de conception et des résultats proches ou équivalents d'une approche de la recherche complète. De plus, la fiabilité des solutions trouvées est supérieure à celle des solutions construites sans notre Framework.

Les modèles construits par notre framework sont basés sur le méta-modèle proposé. L'outil est construit sur l'environnement Sirius avec l'espace de travail appelé PolarSys, qui est un projet Eclipse. Il simplifie le processus en interagissant avec l'utilisateur via l'interface graphique et en automatisant le processus d'exploration de l'espace de conception.

Chapter 1

Introduction

Contents

1.1 Embedded system **22**

1.2 Fault model **25**

1.3 Tolerance strategies **27**

1.3.1 Spatial redundancy 27

1.3.2 Temporal redundancy 28

1.3.3 Check-pointing 28

1.3.4 Error correction code 29

1.4 Problem statement and contributions **30**

1.1 Embedded system

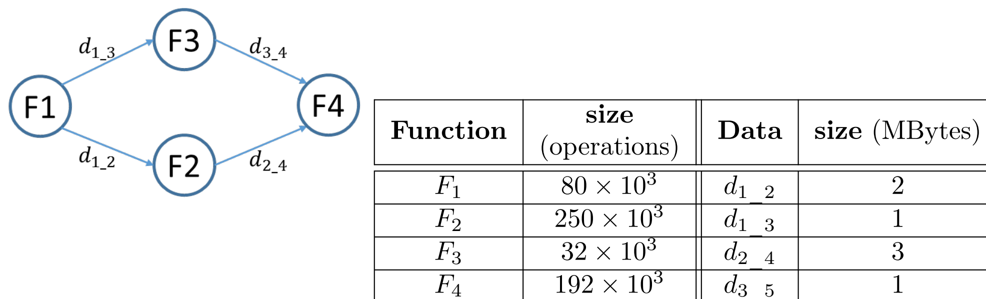
Nowadays, embedded systems become a integral part of most domains such as vehicle, airplane, industrial machines. An embedded system is composed of two parts: an embedded application and a platform. **Embedded applications** can take many forms as signal processing, distributed control system, etc. An application is composed of multi functions/tasks. The complexity of an application is increasing with different requirements such as performance, power consumption, cost, and reliability. Any application need to be executed on hardware and software components. This set of components is called a platform.

Embedded applications

The DAG is very widely used in the DSE literature to describe an application with functions or tasks. It describes effectively the characteristics of applications and is easy to use for modeling, mapping objectives as well as programming [1]. Therefore, we introduce the following concept to describe an application (Definition 1.1) used in this thesis:

Definition 1.1. An application G is represented by a DAG $G = (V, E, D)$, where: each node in $V = \{F_1, F_2, ..., F_k | k \in \mathbb{N}\}$ represents a function of G; E is the set of directed arcs that represent precedence constraints and connect the functions with each other as well as indicate their data dependencies; $D = \{d_{1_2}, d_{1_3}, ..., d_{j_k} | j, k \in \mathbb{N}\}$ specifies the amount of communicated data associated to each link. d_{j_k} represents the data block which is sent from F_j to F_k. These functions have to be executed in a given order to produce desired outputs from the input functions to the output ones. An input function is a function where no arc goes in and an output function is a function where no arc goes out. A **period** is an execution duration from the earliest start time of input functions to the latest ending time of output functions that gives expected results in the output functions. An application can be executed iteratively through many periods.

Figure 1.1 illustrates an example of an application with four functions and only four data blocks between these functions. In this application, there is one input function (F_1) and one output function (F_4). The pre-defined size corresponds to a number of operations of each function. The number of operations is obtained from the algorithm of the application. An operation can be an addition (+), a subtraction (−), a multiplication (×), a division (÷), or an assignment (=). Besides, the data size is also predefined in the 2 rightmost columns. The data is the variables shared between 2 functions.



Function	size (operations)	Data	size (MBytes)
F_1	80×10^3	d_{1_2}	2
F_2	250×10^3	d_{1_3}	1
F_3	32×10^3	d_{2_4}	3
F_4	192×10^3	d_{3_5}	1

Figure 1.1: An example of target application with 4 functions and their properties.

Heterogeneous MPSoC platforms

A heterogeneous/homogeneous MPSoC (Multi-Processor System-on-Chip) is a system in which all components as processing units, memories, buses are put on a same chip. The term "homogeneous" means that all processing elements are identical and interconnected through a dedicated communication infrastructure. Meanwhile, the term "heterogeneous" means that there are different types of processing elements, such as general purpose processors, digital processors, accelerators, that are interconnected through a communication infrastructure as shown on Figure 1.2. The homogeneous architectures are commonly used for certain architectures such as servers, desktop computers, video game consoles. The heterogeneous Multi-Processor System-on-Chips (MPSoCs) are used for embedded systems as they use heterogeneity for optimisation purpose [2, 3].

Increasing the complexity of applications and platforms makes many potential solutions appearing, especially for **heterogeneous** systems, because the different properties of components can make more difficult to designers to choose an optimal solution. In that context, exploration of the design space and optimization of design solutions are practical needs.

The exploration process evaluates design points in a design space and gives a set of best solutions in terms of execution time of an application, reliability of a platform, cost of a system, energy under design constraints. At different levels of abstraction, the exploration strategies corresponds to different accuracy levels (Figure 1.3). An accuracy approach is often used at a low level and the system-level architecture features are fully defined. At this level, designers can use precise simulators (instruction, Register Transfer Level (RTL) levels) or implement prototypes to accurately evaluate a solution. The exact approach is suitable for exhaustive exploration when a several design points are already characterized. However, design spaces are often large ($> 10^{10}$ design points - a point represents a design solution) and the exhaustive exploration is impossible. Moreover, the exploration in low level also takes a lot of time to evaluate a solution. If there has not been an optimal process and evaluation on the whole design space before, it is likely that designers are wasting time evaluating solutions that are not good.

Meanwhile, the higher the abstraction level (cycle-approximate estimation) is, more design solutions can be explored, and more design alternatives can be chosen to satisfy design constraints or achieve a better performance. The higher abstraction level often takes place at the system level. The DSE (Design Space Exploration) focuses on the the application and platform models. If the application model is given and fixed, the exploration process is mainly on the platform building process and mapping process. The platform building defines the resource allocation, the interconnection, etc. The

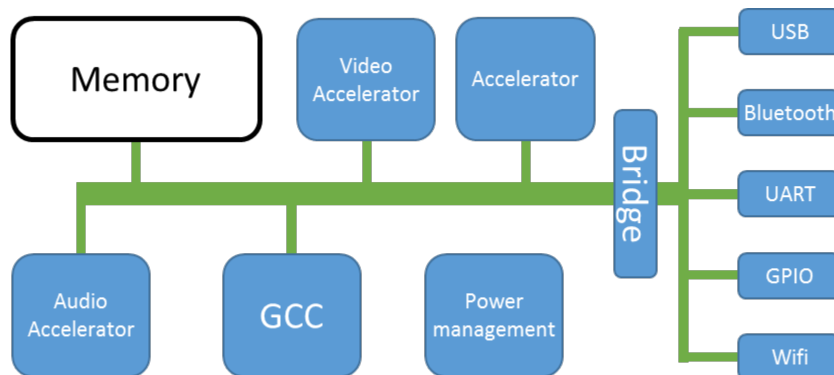


Figure 1.2: Overview of a heterogeneous MPSoC.

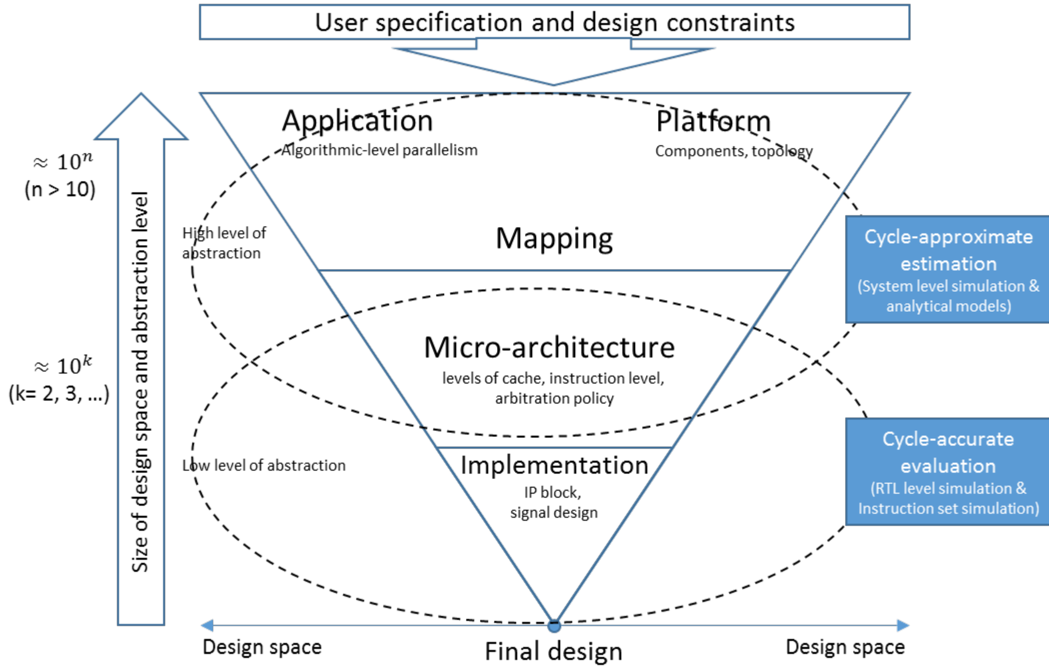


Figure 1.3: Abstraction levels of design space exploration [4].

mapping process defines the function/task allocation. After these processes, designers can find several solutions and they can bring them into the next design steps at a lower abstraction level to evaluate them pore accurately. Definitions 1.2, 1.3, 1.4 describe concepts of an MPSoC platform.

Definition 1.2. A heterogeneous MPSoC platform is composed of (hardware and software) components configured to provide a set of services (memorization, execution, etc.). The platform contains more than one type of processor or processing unit (term: *heterogeneous*). The connections between components form the platform topology.

Definition 1.3. A hardware component can be:

- a PE (Processing Element) that can be hardwired (such as an IP, an Application-Specific Integrated Circuit (ASIC)), thus called a Dedicated PE (DPE) (no software can be executed on it) or a general purpose processor (it can execute software as a hard-core or a soft-core), called a Programmable PE (PPE). PE components are used to execute functions of a given application;
- a Memory (ME) component used to store data and source codes;
- a Communication (COMM) component used to transfer data, signals between others hardware components.

Each component provides at least one service to implement requirements of a given function. A service is a component-specific capability. A service is represented by metrics such as delay, cost, computing capacity and will be discussed in more details in the next Section.

Definition 1.4. A Software (SOFT) component is a software implementation of a function. Its source code is stored in a memory and a SOFT component runs on top of a PPE [5].

Table 1.1: Targets for the probabilistic metric for random hardware failure [9].

ASIL	Random hardware failure rate values
D	$< 10^{-8}$ per hour
C	$< 10^{-7}$ per hour
B	$< 10^{-7}$ per hour
A	$< 10^{-6}$ per hour

Reliability and system safety

In the process of exploration, a special concern can be the fault tolerance to ensure system safety. Fault tolerance of a system can be assessed through its reliability level in the range from 0% to 100%. Higher is better. The analysis of the predictive reliability of the electronic components is studied and standardized in FIDES [6] and MIL-HDBK-217 [7].

However, the comprehensive evaluation of reliability of an embedded system requires the oversight of all behaviors of the system such as the time to perform functions/tasks, the time of receiving/sending data, the interconnection time, etc. Besides, many designers want to use fault-tolerance strategies in their design. Assessing reliability needs them to consider the impact of these strategies on the system performance and cost. And, the DSE process also need to explore solutions with different fault-tolerance strategies.

Reliability and the safety are not the same, but the probabilistic risk for a random failure has created a relationship between these two aspects [8]. For example, Table 1.1 shows the failure rate used in the quantitative safety assessment in the safety standard ISO26262. ISO26262 is a safety standard released in November 2011 to measure and document the safety level of automotive electronic systems [9]. A designer can use this standard to build their system as a constraint. Each safety goal is given as Automotive Safety Integrity Level (ASIL) that determines what ISO 26262 safety requirements that apply to the goal.

The evaluation and optimization of reliability of a heterogeneous MPSoC also need to be relevant in relation to other needs of the system such as processing time, system construction costs, energy etc.

Thus, in this whole context, designer will have to solve the problems for embedded systems such as: modeling the heterogeneous MPSoC architecture, exploring mapping solutions, exploring fault-tolerant strategies, evaluating system and eventually finding optimal solutions.

1.2 Fault model

In order to tolerate faults, we need to understand the object that we need to resist: that is the fault consequence, fault kind, and fault model.

A **fault** is an event that causes a defect within the component such as a software bug, a bit stuck, a broken circuit, etc. An **error** refers to a difference between the actual output and the expected output when a required operation is executed. A **failure** appears when a system or a subsystem fails to perform a required function according to its specification. The relation between fault, error, and failure is depicted in Figure 1.4. A fault may lead to an error, and an error might cause a failure in the whole system or a subsystem, or a component [10] [11]. A goal of safety-critical systems is that errors should not lead to any failure. More generally, designers expect that the system should not have a fault that results in failure.

A fault can be divided into two categories derived from its consequences for components: permanent and transient. Permanent faults are derived from a hardware destruction such as aging, broken

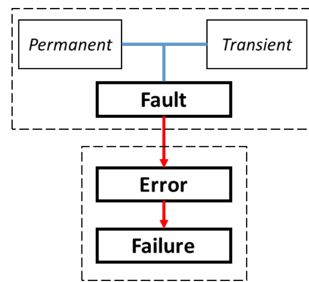


Figure 1.4: Relation between fault, error and failure.

electronic circuits, stuck-at-bit [10]. A permanent fault remains active until there is an intervention for correction. Transient faults can be caused by a single ionizing particle striking a sensitive node or an interconnection of a circuit [10]. Unlike permanent faults, a transient fault only remains active in a short period and disappears on the next operating period. For example in Figure 1.5, a transient fault appears in the execution duration of F_2 but it disappears in the execution duration of F_3. As consequences of faults, the effects of errors and failures on the component are also either temporary or permanent.

The failure rate is often expressed in Failure In Time (FIT), which is the number of failures that can be expected in one 10^9 device-hours of operation. This value represents the possibility of a fault causing a failure during the operation of a component. The failure-rate evaluation of transient and permanent faults in every single component is not trivial. These parameters reflect the existing experiences from complex and time-consuming processes of injection, simulation, prediction of faults, and experiment on components [12, 13, 14, 15, 16]. This work uses the failure rates as the inputs of the design exploration in order to evaluate the reliability of components. The reliability is the probability that the component or system will not cause a failure for a specified time under specific conditions.

Permanent fault There are four main wear-out effects for integrated circuit components: electromigration, time-dependent dielectric breakdown, stress migration, and thermal cycling [17]. For this work, electromigration related wear-out failures are assumed, however, any other effects can be included either standalone or using sum-of-failure-rate model for any combination of the above failure effects. We use a Weibull distribution to describe the wear-out effects, Equation 1.1 gives the evaluation of the probability that permanent faults cannot cause any failure on the component during the interval between 0 and the time moment τ, where $alpha(T)$ is the scale parameter, that is a function of temperature depending to the wear-out failure model, β is the slope parameter shown to be nearly independent of temperature [18].

$$R_{\text{PF}}(\tau) = e^{-\left(\frac{\tau}{\alpha(T)}\right)^{\beta}} \tag{1.1}$$

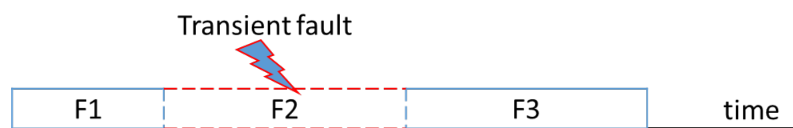


Figure 1.5: A transient fault appears and disappears on a PE component.

To model the useful life of the product, we choose $\beta = 1$ as in [19]. So, in that case, Equation 1.1 becomes Equation 1.2:

$$R_{\mathrm{PF}}(\tau) = e^{-\frac{\tau}{\alpha(T)}} \quad (1.2)$$

We have Equation 1.3, where $\lambda_{\mathrm{comp}_{PF}} = \frac{1}{\alpha(T)} = \frac{1}{MTTF}$ is the failure rate of the component caused by permanent faults, $MTTF$ is Mean Time To Failure, as following:

$$R_{\mathrm{PF}}(\tau) = e^{-\lambda_{\mathrm{comp}_{TF}} \cdot \tau} \quad (1.3)$$

For example, 1 component has a lifetime of 10 years, so it means that the average time for a permanent error to appear is 10 years ($MTTF = 10$ years). So, $\lambda_{\mathrm{comp}_{PF}} = \frac{1}{\alpha(T)} = \frac{1}{MTTF} = \frac{1}{10 \times 365 \times 24}$ failure/hour $= \frac{10^9}{10 \times 365 \times 24}$ failure/(10^9 hours) = 11415 FIT.

Transient fault Assuming the failure from transient faults arrival on a component follows Poisson distribution [20], Equation 1.4 gives the evaluation of the probability that transient faults cannot cause any failure on the component during the interval between 0 and the time moment τ, where $\lambda_{\mathrm{comp}_{TF}}$ is the failure rate of the component caused by transient faults.

$$R_{\mathrm{TF}}(\tau) = e^{-\lambda_{\mathrm{comp}_{TF}} \cdot \tau} \quad (1.4)$$

Given that the events of the permanent faults and the transient faults are independent, we have the reliability of a component during the interval between 0 and the time moment τ as Equation 1.5, where $R_{\mathrm{PF}}(\tau)$ and $R_{\mathrm{TF}}(\tau)$ is the reliability considering the permanent fault and the transient fault, respectively.

$$R_{\mathrm{comp}}(\tau) = R_{\mathrm{PF}}(\tau) \times R_{\mathrm{TF}}(\tau) \quad (1.5)$$

1.3 Tolerance strategies

There are several fault-tolerance strategies developed in the literature [10]. Herein, we describe the strategies that are popular and effective strategies.

1.3.1 Spatial redundancy

The k-out-of-n (also known as N-modular redundancy) is one of the most popular strategies of the passive redundancy [10]. The passive redundancy strategies have no need to detect faults but mask them. With the spatial redundancy, designers use multiple hardware components in the same subsystem. The components are multiplied to perform the same computation in parallel. The majority voting is used to determine the correct output from these components, except for $N < 3$.

Figure 1.6 gives the concept of a popular instance of the spatial k-out-of-n with $k = 2$ and $n = 3$. In the TMR subsystem, if one of the components fails, the voter masks the fault by comparing the outputs among the faulty module and the remaining two fault-free modules. In addition to the time that the component implements the function (τ), the system has to take into account the time that the voter handles the outputs (τ_v). Depending on the purpose, the redundant components can be processors, memories, buses, network connections, etc. This strategy can mask both permanent and transient faults. However, it requires high overheads in terms of area, price, energy, etc.

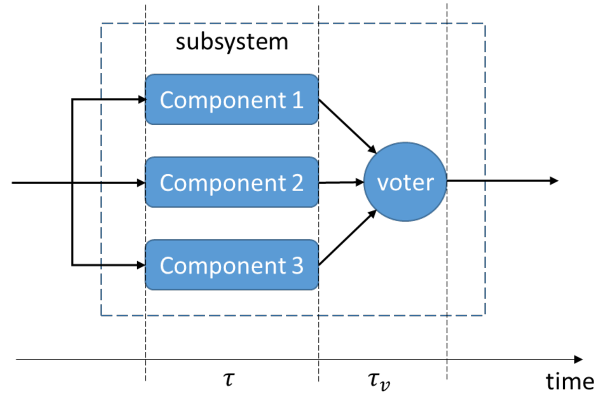


Figure 1.6: An instance of spatial redundancy: TMR (Triple Modular Redundancy).

1.3.2 Temporal redundancy

If the size, weight, power consumption and cost of a system is a big constraint, designers can use extra time rather than extra hardware to tolerate faults. The temporal redundancy includes the repetition of the computation between two or more times and compares the results with the previous computations [10]. This repetition is usually performed on only one hardware component.

The Figure 1.7 describes the TReR strategy as an instance of the temporal redundancy. The execution time in this concept is 3 times longer than the no-redundancy. It assumes that the function F_1 is mapped to the subsystem, at each computing cycle, this subsystem executes F_1 three times. If a permanent fault occurs and causes an error on this component, all the computations on this component are faulty and the resultant output is wrong. Therefore, this strategy can not mask permanent faults. However, if a transient fault occurs in one of three occurs and disappears in the remaining two times, this strategy can produce the correct output. A TReR subsystem can mask only one transient fault among 3 executions. The temporal redundancy is suitable for processors.

1.3.3 Check-pointing

The check-pointing technique is a recovery mechanism that is based on check-points and restart mechanisms [10, 21]. Most software faults are design faults that resemble hardware intermittent faults: they appear in a period and disappear, then appear again. Hence, simply restarting the subsystem is usually sufficient to successfully complete its execution, it also masks transient faults or software bugs.

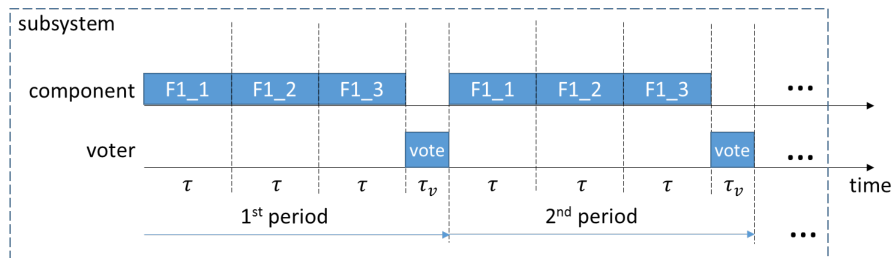


Figure 1.7: An instance of temporal redundancy: TReR (Triple Re-execution Redundancy).

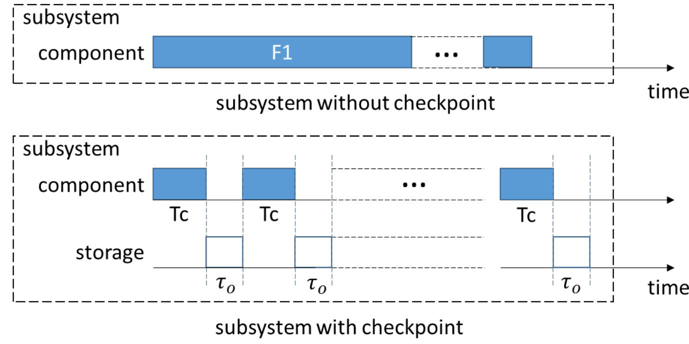


Figure 1.8: Subsystem without/with checkpoint.

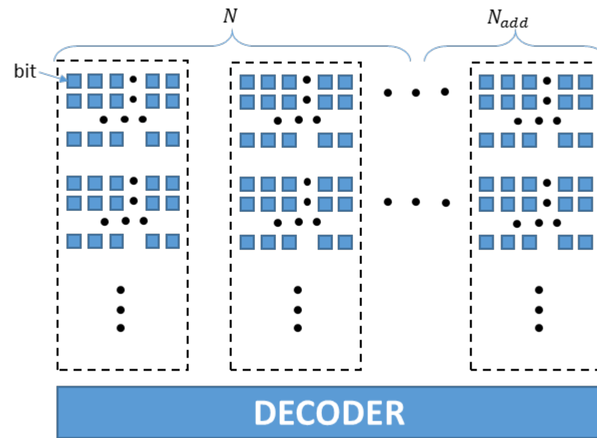


Figure 1.9: Memory with correction code.

The Figure 1.8 describes the concept of this strategy. In a normal period, the subsystem that is composed of a component executes a mapped function without checkpoints during τ - the total execution time of the mapped function on the subsystem. If designers apply the strategy with N checkpoints, checkpoints are created between N points during the execution. T_c is the execution time of the function in each check-point, $T_c = \frac{\tau}{N+1}$. At each checkpoint, the subsystem saves the current state. τ_o is the time for checkpoint capture and storage. If a fault is detected, the subsystem returns to the last checkpoint state and resumes the execution. The time for recovery from a checkpoint is τ_r. Fault-detection checks need to be embedded in the code and executed before the checkpoints are saved. This strategy is usually applied to processors by the ability to save theirs states, and rebooting can be easily done through programming in the code.

1.3.4 Error correction code

The code correction is a famous technique to tolerate faults for the memory [22]. The main idea is that some redundant bits are used to detect whether errors in the bits are stored in the memory and/or correct the errors.

Figure 1.9 shows a typical model of a memory with correction code. A data word stored in the memory owns initially N bits. Depending on the error correction requirements (capacity), there are

N_{add} encode bits to be added to data words. A single error can appear in any bit on $N + N_{add}$ bits. In order to correct at least one error in a data word, the number of additional bits must satisfy the inequality $N + N_{add} + 1 < 2^{N_{add}}$.

1.4 Problem statement and contributions

In this thesis, we present a framework that explores the design space of embedded systems that incorporates fault-tolerance strategies. The main contributions are:

1. New meta-model integrating the fault tolerance for embedded systems. The meta-model is the core of the Model-Driven Engineering method. This proposes to cover the fault tolerance; serves as a bridge between the different tools, between different programming languages, and different design stages, allowing designers to have a unified and coherent view of an MPSoC platform. With the model, designers can configure the exploration according to their own level of expertise, while also abstracting the complex and the specifics of the target architecture. This favors design portability and reuse by providing models while abstracting low-level details of the considered system.

2. New DSE method is composed of fault-tolerance design space generation, evaluation and optimization. From the user specification, explored dimensions include hardware choice, task mapping, data mapping, and fault-tolerance-strategy choice. A new solution is generated and evaluated in terms of execution time, cost and, reliability level. Then, an optimization process will explore the best solution among the design space.

3. Evaluation of heterogeneous MPSoC platform under the impact of transient and permanent faults. This evaluation is a very important part of the DSE to help designers choosing the appropriate strategy fault tolerance in regard to a compromise with the requirements of the application.

4. New tool with a graphical user interface allows to model and run the DSE process. It simplifies the process by interacting with the user through the graphical interface and automating the process of exploring design space.

The remains of this manuscript is as follows:

- Chapter 2 gives an overview of the state-of-the-art, classification of existing approaches of Model-Driven Engineering and Design Space Exploration;
- Chapter 3 introduces the meta-model of the MPSoC platform integrating the fault-tolerance and the modeling tool with graphical user interface;
- Chapter 4 introduces our DSE algorithms;
- Chapter 5 presents the evaluation and validation of the method through the use of three case-studies;
- Chapter 6 concludes this thesis by summarizing its contributions and providing perspectives and future work.

Chapter 2

Literature reviews

Abstract: In the previous chapter, we introduce the context of MPSoCs. Design space exploration must meet the needs for fault tolerance of embedded applications. In addition, modeling of high-level system components helps reuse existing modules and can reveal many potential solutions. Therefore, in this chapter, we present the works that are related to the same context. In order to clarify the high-level model-based designs, we review existing MDE (Model Driven Engineering) studies for the MPSoCs as well as platform meta-models in the first Section. Next, we will have a deeper look into different approaches of the DSE (Design Space Exploration). It should be noted that in this section we will devote most of the time to analyzing the DSE studies related to the use of fault tolerance or the reliability-awareness.

Contents

2.1 Model driven engineering 32
2.1.1 Model-Based Testing 33
2.1.2 Model-driven development 33
2.2 Static and dynamic exploration 39
2.2.1 Design-time exploration 40
2.2.2 Run-time exploration 43
2.2.3 Fault-tolerance and reliability-based exploration 45
2.2.4 Analysis 48
2.3 Summary 52

2.1 Model driven engineering

MDE (Model-Driven Engineering) addresses the application and platform complexity and expresses domain concepts effectively to alleviate this complexity.

A model is a structure that represents a design artifact as a relation schema, an interface definition, a domain specific language (such as Extensible Markup Language (XML), Unified Modeling Language (UML)) or a hypermedia document) [23]. In [24], George E. P. Box wrote: *"All models are wrong but some are useful"*. A model is not the considered entity and does not carry all characteristics of an entity. Designers just choose which properties of an entity are important and put them into their models to reduce the complexity of the interest entities (herein such as application, platform, and components). Each model has to conform to a meta-model. A meta-model describes the various types of model elements and how they are connected, arranged and constrained.

Figure 2.1 describes the relationship between the systems, models and technical space. A technical space [25] is an environment with associated concepts, tools, required skills based on some algebraic structures (tree, graphs, categories, etc.). Each technical space is based on a meta-meta-model and a set of meta-models. Several companies and organizations (Object Management Group (OMG), IBM, and Microsoft) are proposing several environments to support MDE.

For instance, OMG proposed Model Driven Architecture (MDA) as one of vision of MDE with a set of OMG standards like Meta-Object Facility (MOF), XML Model Interchange (XMI), Object Constraint Language (OCL), UML, etc [27]. In OMG MDA, the MOF and the collections of standard meta-models and UML profiles play the role of a technical space.

A meta-model is the core of an MDE methodology. As such, there are many supports for the model/meta-model building. The demand is to have a mechanism for the construction of valid models *i.e.* the meta-model needs to be defined in a suitable way and abstraction level to be easily reused, maintained and operated.

Figure 2.2 describes the classification of MDE. We classify according to the general purpose of applying MDE according to literature studies [27, 28, 29]. However, this classification is not really clear due to the overlap between the classes. But this classification makes it easier to see the MDE application in the literature.

Accordingly, MDE is classified into two main categories: Model-Based Testing (MBT) and Model-

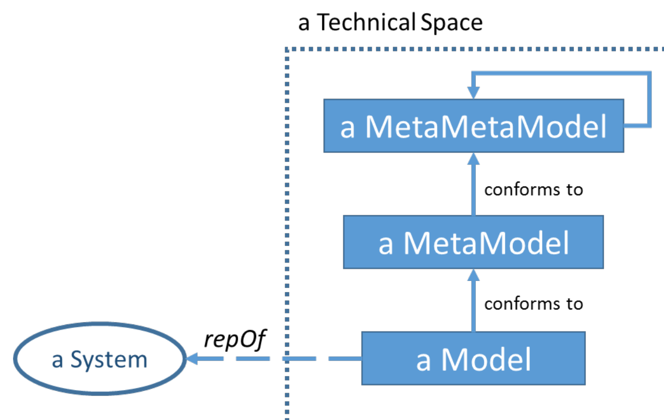


Figure 2.1: System, models and technical space [26].

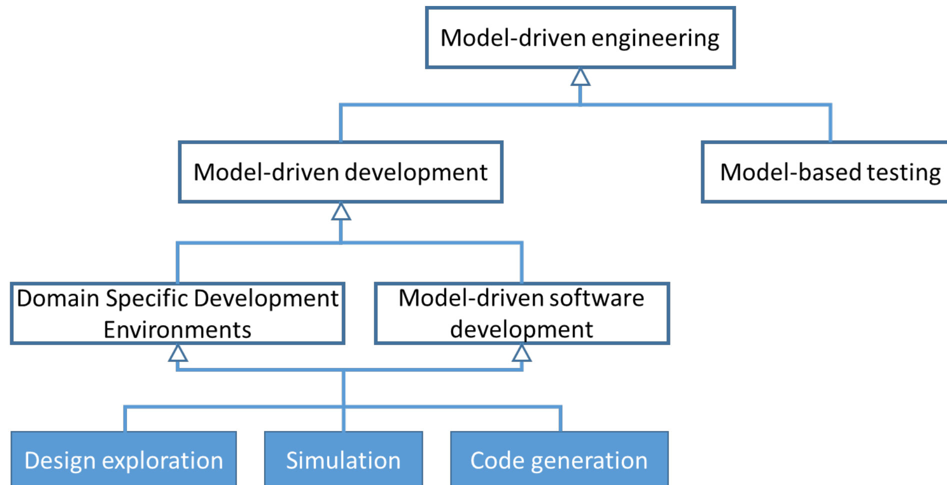


Figure 2.2: MDE classification.

Driven Development (MDD). In MDD, two classes were identified. One is called Model-Driven Software Development (MDSD) and the other is Domain Specific Development Engineering (DSDE). After that, the MDE methodologies can serve different purposes such as design exploration, simulation, code generation.

2.1.1 Model-Based Testing

MBT (Model-Based Testing) is a technique for automatic generation of test cases using models [30]. MBT is defined as the automation of the design of black-box tests. Testing models are used to represent the considered domain for the test input data, the desired behavior, testing strategies, and the testing environment of the System Under Test (SUT). A testing model is usually an abstract, partial representation of the desired behavior of the SUT. These models capture some of the requirements. Then model-based testing tools are used to automatically generate test-cases from that model. Test cases are functional tests and might then be mapped into executable tests that can communicate directly with the SUT by the specific testing tools and frameworks.

As such, we can see that MBT is an interesting area of MDE but not in the goal we are aiming at in the context of this thesis. Therefore, we will not continue to discuss this issue.

2.1.2 Model-driven development

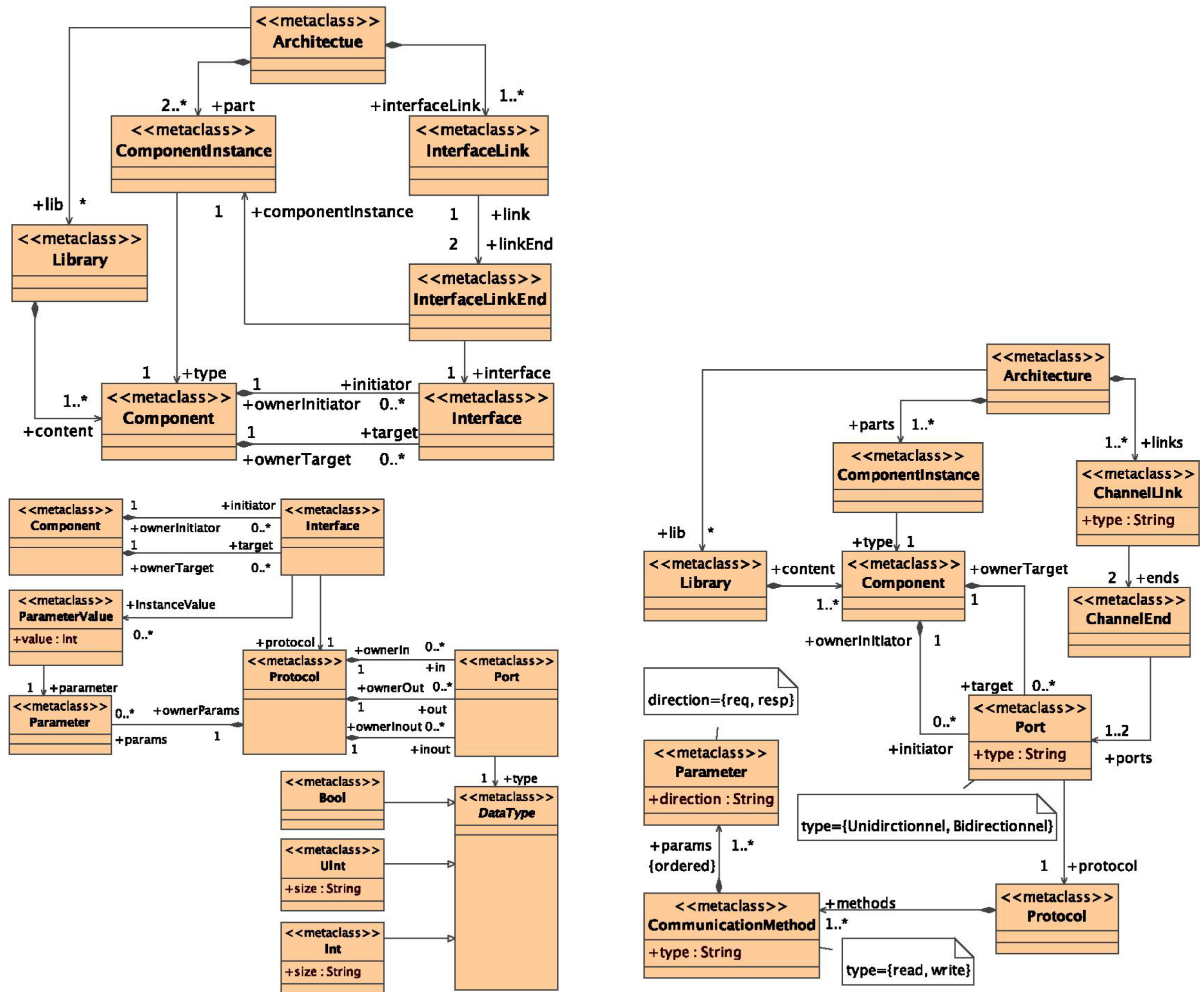
The second category of MDE is Model-Driven Development. MDD focuses on the requirements, analysis, design, and implementation disciplines [31, 32]. The MDD approach's problems are around the System Under Study (SUS) at different levels of abstraction. The underlying motivation for MDD is to improve the productivity and quality of the design process and of the final design. There are two ways to achieve this goal: in the short-term, designers try to increase a product's value in terms of how much functionality it delivers; and in long-term, they try to reduce the obsolescence rate of the product. As such, MDD uses the modeling and models for the design, development, and optimization of products.

MDD is used extensively in software design, called MDSD, where a typical representative of it is Model-Driven Architecture. MDA [32] is a standard proposed by OMG and aims at software design, development, and implementation. It provides guidelines and tools for structuring software specifications, models, and model transformations. MDA provides tools for defining and processing models, namely Computational Independent Modes (CIM), Platform Independent Model (PIM), and Platform Specific Model (PSM). CIM is built as a business or domain model that presents the required features of the SUS and in the environment where it must operate. It hide all information technology specification to remain independent from the system implementation or the system deployment. PIM is the model of the functionality and behavior, which enables the mapping to one or more platforms. PSM, as its name, suggests a model that combines the specification in the PIM with platform-specific detail to determine how a SUS is deployed on a particular platform.

DSDE (Domain Specific Development Engineering) is another class of MDD, which supports domain-specific knowledge to define relations between models and how these models could be refined. Basically, the way the DSDE works is quite similar to MDSD, and even some of the MDA techniques have been applied to DSDE studies. However, unlike MDSD, DSDE does not rely on standards and DSDE can be used not only for softwares, but also for hardware, electrical, mechanics parts, and embedded systems. At a more detailed level, DSDE studies can be used for purposes such as design exploration, simulation, code generation, which corresponds to the research objectives of this thesis. A meta-model plays a crucial role in any MDE method. Therefore, the meta-model review, especially platform meta-models, of the following research is one of our priorities. This subsection presents some DSDE studies for MPSoC development. These works produced different meta-models suitable for different objectives on different MPSoC architectures.

Metropolis [33] is an environment for electronic system design, in which tools are integrated through an API and a common meta-model. The meta-model of Metropolis represents the function of a system being designed at levels of abstraction, represent the architectural targets, allow mapping between different platforms. The Metropolis infrastructure captures application, architecture, and mapping that supports the **simulation** and the **synthesis** manually. The meta-model allows a unified view of both functional and platform aspects. This can be an advantage when integrating and mapping functionalities into the platform but is also a barrier to develop these two distinct aspects.

Within the GASPARD framework [34, 35], the authors propose hardware meta-models corresponding to several design abstraction levels. The first one describes the hardware architecture at the Cycle-Accurate Bit-Accurate (CABA) level (Figure 2.3a) and the second is at the Timed Programmer's View (TPV) level (Figure 2.3b). The CABA level meta-model is described at the clock cycle level. In this meta-model, a given architecture is composed of hardware components from the appropriate library. These hardware components communicate with the others through initiator and target interfaces. These interfaces are described at the signal level and define a communication protocol. This model allows for high-precision **simulation**. A micro-architectural simulator gives a number of cycles for the performance estimation of a given platform. The TPV meta-model is at an higher abstraction level. Details related to the computation resources (such as cache, processor control unit) and to the communication are omitted. At this level, the performance simulation is implemented by counting each component activity such as the number and types of executed instructions for the processors; hits and misses for the caches; the number of transmitted/received packets for the interconnection network; the number of read and write operations for the shared memory modules, etc. The execution time estimation in this level needs to count from the execution time of each activity. The execution time of each activity is estimated from the CABA platform. Therefore, the meta-models are quite detailed



a) at the cycle-accurate bit-accurate level. b) at the timed programmer's view level.

Figure 2.3: Hardware architecture meta-model proposed by GASPARD [34] describes the hardware architecture at a) the clock cycle level and b) the timed programmer's view level.

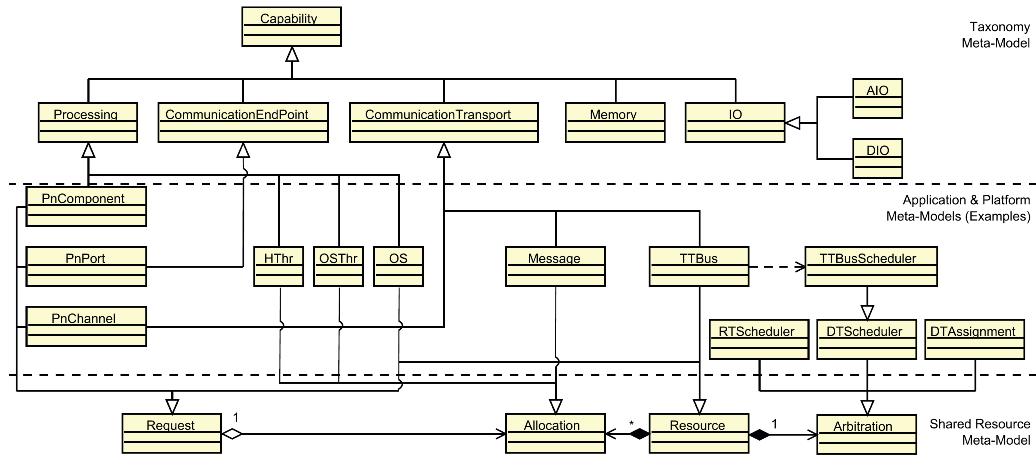


Figure 2.4: Application and platform meta-model for the heterogeneous multiprocessor architectures with time-triggered execution paradigm [38].

and effective for the performance **simulation** of a hardware architecture. The proposed meta-models are oriented towards SystemC **code generation** and **simulation**. This MDA-based study has some similarities with our approach in terms of modeling. However, due to their meta-models defined at the signal level and the instruction level, models may be quite complex and time-consuming to evaluate a solution for exploration purposes.

COMPLEX [36] is another methodology using MDE for the **design space exploration** of embedded systems. This methodology models applications, platform architecture, scenarios, environments, physical information by the UML/MARTE models. Then the SCoPE+ **simulation** infrastructure [37] in SystemC environment supports the solution evaluation. Finally, an optimization process is deployed on the Multi-Objective System Tune (MOST) tool, which enables discrete optimization specifically designed. This methodology does not propose its own meta-model for modeling but uses UML/MARTE modeling. This is on advantage because it reduces the development time of the modeling process. They build a graphical tool that allows to produce models of the system. Their methodology did not mention the fault tolerance in the embedded system development. However, this work also gave an idea on integrating the exploration process into a search engine available in the literature and building a modeling graphical tool.

In [38], the authors present a framework that provides a design flow for fault-tolerant embedded system design. Their approach focuses on heterogeneous multiprocessor architectures with time-triggered execution paradigm. A merged class-diagram of application and platform meta-models is described and shown in Figure 2.4. *Resource* classes in the bottom provide services to other components in the system. The access policy to a resource is defined on an arbitration object. There are 3 types of arbitration: design-time assignment, design-time scheduler, run-time scheduler. The *Allocation* class describes the fraction of a resource that is used to satisfy a given request (such as a time slot on a time-triggered bus or a storage amount in a memory component). *Request* class represents the entities that need to use services from the resources. The principle of the considered system is based on a time-triggered bus. They also propose a meta-model used for the **design exploration** and **code-generation** objectives. Thus, the meta-model focus on the communication description through *CommunicationEndPoint*, *CommunicationTransport*, and many classes derived from these two classes.

In a global perspective, this study shares our concerns, but their implementation way is relatively

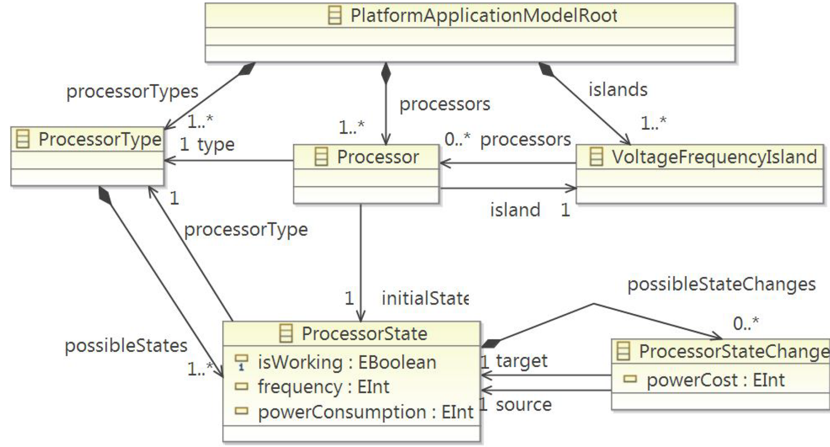


Figure 2.5: PAM metamodel [40].

different. Firstly, the combination of application and platform into one meta-model allows building mechanisms for mapping and linking between application elements and platform components. It is useful for modeling the dependencies between the application and the execution platform, but this limits the reuse of platform models for other applications. Especially, if designers want to have an in-depth look at the platform aspect, this meta-model can be difficult to reuse because it is strongly dependent on the application model. Secondly, the fault-tolerance aspect is not integrated into their meta-model, which is difficult to extend to many other strategies.

The HIPAO2 [39], stands for Hardware Image Processing system based on model-driven engineering and Aspect-Oriented modeling version 2, is a methodology that performs hardware/software partitioning and generates PSMs. The HIPAO2 methodology takes advantage of MDE for the development of image processing algorithms. However, due to limitations in the considered application, reusing HIPAO2 to different types of applications is not easy. In addition, HIPAO2 does not handle the fault tolerance. This study shows the advantages of MDE in an embedded system discovery, nevertheless it is not really our concerns.

In [40], authors proposed a model-driven hardware-software co-design framework that allows mapping a Synchronous Dataflow (SDF) application on a multiprocessor hardware platform. The framework proposes a meta-model for PAM (Platform Application Model) as described in Figure 2.5. The *PlatformApplicationModelRoot* class on top represents a platform as a whole. A platform is composed of a set of *Processors*. The processors are partitioned into groups of voltage/frequency islands. In a same voltage/frequency island group, processors run at a common voltage/frequency. Moreover, the characteristics of a processor are associated with its type (*ProcessorType* class). A processor is associated with the working or idle states (*ProcessorState* class). From the processor state and its frequency, the roof can estimate the power consumption. *ProcessorStateChange* class describes the transition overhead between frequencies for each processor *i.e.* the transition between a pair of *source* and *target* of processor state. This framework focuses on energy-optimal scheduling objective and the meta-model is used in the model-checking **simulation**. The processor's state is suitable for **simulations**, but it is not necessary to explore the space of mapping solutions. Moreover, it lacks information about memory and communication components.

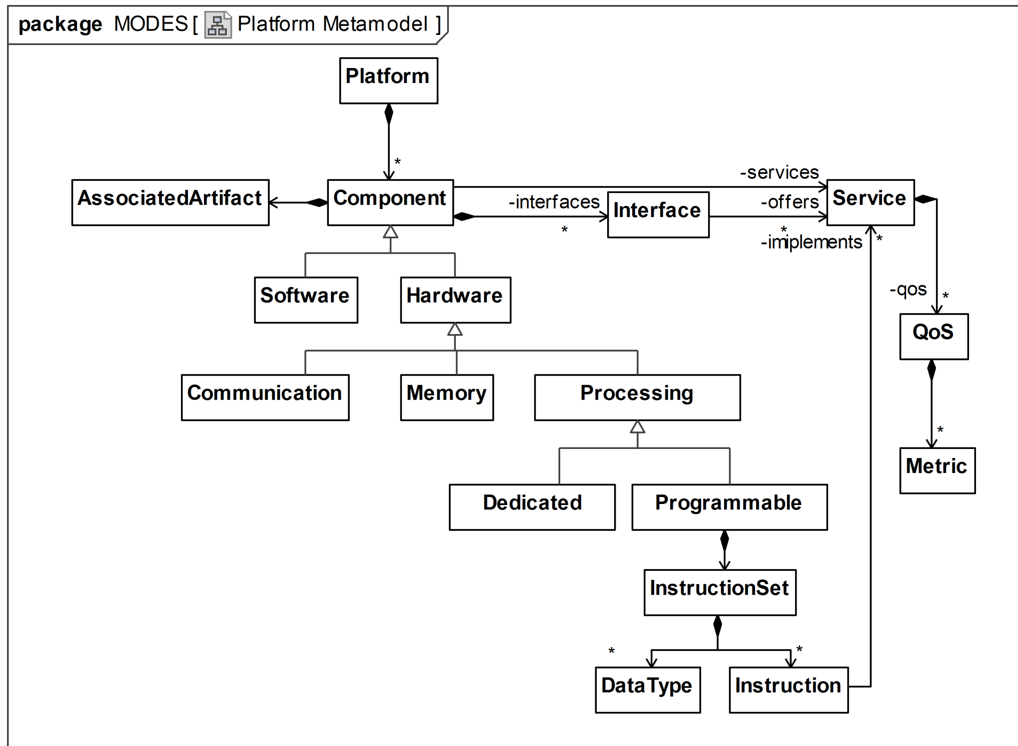


Figure 2.6: Meta-model proposed by ModES [41].

The ModES design framework [41] provides high abstraction meta-models representing MPSoC system concerns in specific aspects – application, platform, mapping, and implementation. The framework aims at **design exploration** and **code generation**. Figure 2.6 shows one of the platform meta-models proposed by ModES. They have several variants of this meta-model depending on the specific purpose of the discovery. There are available hardware and software components in a *Platform*. The software component characterization is obtained when the component code is compiled for a target architecture. The hardware component specification is obtained from the synthesis process and execution cycles. A *Hardware* type is classified as *Communication*, *Memory* or *Processing* corresponding to communication, storage, and processing resources respectively. Moreover, a *Processing* component can be a dedicated component (no software can be executed on), or a programmable component (needs a software). These components offer *Services* for an application through a set of interfaces. In general, these meta-models can partially fit into our goals because they are used to move towards the goal of exploring design space. Especially, these meta-models are built to allow the reuse of components and hardware, and are abstract enough to explore as many alternatives as possible. They also try to create platform-independent models of applications. As a consequence, this platform meta-model has a certain degree of independence, suitable for expansion or reuse. However, we can see in Figure 2.6, that they look at the system up to the *Instruction* and *DataType* levels. This results in their evaluation process on a solution being more accurate but longer lasting. In fact, the DSE process ran 3 hours for 5000 solutions meanwhile there is an estimate of 5.89×10^{41} solutions. So, to explore all this space, it takes a lot of time. This can be a disadvantage as we can miss the good solution while spending too much time on less attractive solutions. Moreover, no fault-tolerance is proposed in the ModES framework.

Table 2.1: MDE studies for embedded system development.

Name/authors	Objectives	Meta-model	Level	Fault-tolerance	Ref
Metropolis	simulation, synthesis	Yes	Multi	No	[33]
GASPARD	code generation, simulation	Yes	Multi	No	[34]
J. Huang *et al.*	DSE	Yes	data-transmission	Yes	[38]
COMPLEX	simulation, DSE	No	data-transmission	No	[36]
PAM	simulation	Yes	state, voltage	No	[40]
HIPAO2	DSE, code generation	Yes	system	No	[39]
ModES	DSE, code generation	Yes	instruction, system	No	[41]

This Section presented the existing MDE proposals in the literature. It can be seen that works have been developed for a variety of purposes. Table 2.1 summarizes the interesting studies of MDE in the literature. However, none of them proposed a meta-model integrating the fault tolerance to develop a highly reliable MPSoC platform. These MDE frameworks mainly focus on building software systems or software products. Elements in these frameworks on the platform or hardware architecture play only a role on the execution, almost fixed, or make them as independent of the software as possible. The primary goal is the code generation or the simulation of the system. Not many studies use MDE for the purpose of developing the architecture aspect in an MPSoC system.

2.2 Static and dynamic exploration

In this section, we discuss the existing works of design space exploration for MPSoCs. The MPSoC exploration process starts with a given application, available resources and it ends when a solution is found. The solution points out how an application is mapped to a platform such as the mission of each component in the system, assignment and ordering of tasks or functions of the given application and their communications onto the platform resources with regard to some optimization criteria (energy consumption, compute performance, reliability or combination of the criteria (hybrid)).

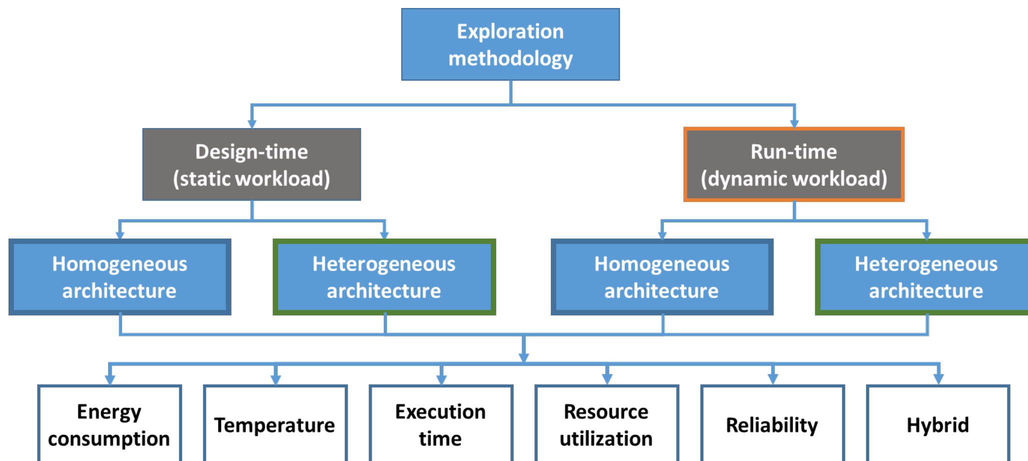


Figure 2.7: Classification of exploration methodology.

Based on the workload scenarios, the mapping can be classified in design-time or run-time methodology [1, 42, 43]. As the mapping process is the core of the DSE process, this classification can also be generalized to the DSE as the second level (from the top) of Figure 2.7. These methodologies target either homogeneous or heterogeneous multi-processor systems.

The design-time methodologies are suitable for pre-defined applications and static platforms. They do not allow the insertion of new elements into the system at run-time. The resources on which the given application is executed is decided before its execution and is not changed thereafter. As the approach is performed with known computation and communication behaviors and a static platform; makes it easy to analyze, explore, and select a design solution; thus optimizing the overall exploration time of the solution space. But this approach is not flexible in case of applications that presents many fluctuations in the execution process.

Conversely, a run-time methodology maps a task or functions of an application on resources by observing the workload of a system at run-time. The assignment of tasks on a platform can be changed anytime during execution of the application if the user requirements change or a new application has entered into the system. This kind of methodology can satisfy the performance requirements incurred at the run-time. The methodology can use system information to make decisions as compared to the design-time mapping methodologies. However, this advantage becomes a barrier to the overall execution time of the system because it takes time to make a decision. In some scenarios, some run-time decisions force to move some tasks/functions from one processor to another. This may require to reconfigure processors and/or the whole system. The reconfiguration depends on a reconfiguration module that can cause errors and reduce the overall reliability of the system.

The main objectives of the exploration are usually energy consumption, temperature management, performance (execution time), resource utilization and reliability. In particular, the studies on the reliability as well as the fault tolerance of the system account for only a small portion of the MPSoCs' DSE research. In the two next subsections, we discuss some exploration studies of design-time and run-time with the popular-objective studies and then in the Subsection 2.2.3, a state of the art of fault tolerance DSE is presented. We briefly describe the literature studies of interest with regard to our work. At the end of each section, we will highlight results and strategies proposed by the literature studies and we provide comments about their applicability to our work and how we position this thesis in the field.

2.2.1 Design-time exploration

In the last decade, the static exploration works account for a large proportion of DSE's research. This shows that this technique is still effective in looking at the characteristics of the system as well as allowing for finding and exploring potential design solutions. It has a global view of system at the early-design phase which does not take much time to make a decision.

Figure 2.8 describes the general processing flow of the design-time exploration. All the system settings are ready before the system is implemented and runs. The dashed line from the *Optimization* block to the *Mapping* block indicates that the mapping, evaluation and optimization processes can be performed simultaneously or separately *i.e.* designers can place the mapping and the evaluation into the optimization process, then each mapping solution is evaluated, improved and optimized immediately; or they can give a set of output solutions from *Mapping* and use them as inputs of the *Optimization*.

One of the optimization goals is about **energy consumption**. Various indirect or direct means are used to achieve this goal. In [44], the minimization of energy consumption on a 2D Network on Chip (NoC) architecture needs to meet the real-time requirements for a given embedded application. The

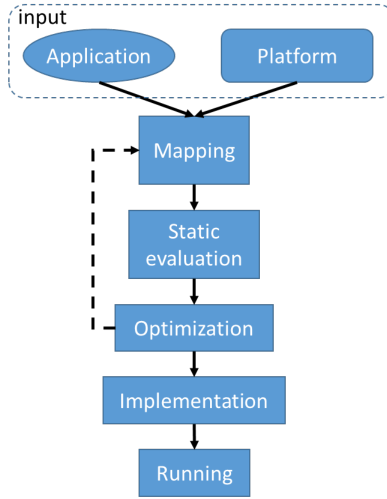


Figure 2.8: Design-time exploration chart.

system energy can be estimated directly through the predefined energy consumption of a task running on a PE, the predefined energy consumption of communication. The exploration method considers the mapping (a PE on a NoC tile, a task on a PE) and the scheduling under a time constraint. They use SA (Simulated Annealing) for solving their combinatorial optimization problems.

S. Cao *et al.* considered the thermal problem in the design space [45]. They proposed a multi-application task mapping scheme to balance workloads in order to reduce peak temperature on homogeneous NoC many-core systems. The temperature and energy model is based on the mechanism of frequency and voltage scaling of each core. The mapping is then improved by repeatedly scaling tasks to lower voltage/frequency levels and migrating tasks.

Besides, Dawei *et al.* [46] also proposed a method in which the voltage scaling and the frequency tuning of a processor are combined to manage overall system power consumption with a two-step approach. The application mapping is done in the first step to minimize the total energy consumption for communications. With the mapping achieved, the second step derives a scheduling for the mapped application and simultaneously set the voltage levels for tasks, set the link frequencies for communications and satisfy the precedence constraints of the given application.

Thus, we can see that energy consumption depends on the power consumption per component. The power consumption per component is predetermined and the studies can minimize the overall energy consumption by optimizing the component utilization or reducing connections/messages exchanging inside a system. Besides, the energy is also a mathematical function of frequency and voltage. The studies can reduce the energy consumption by looking for a solution with a best configuration of frequency or voltage.

The popular criteria in the DSE are the **performance** and **resource utilization**. J. Gonzalez-Dominguez *et al.* [47] used Servet which is a suite of benchmarks to obtain the important hardware parameters to support the automatic optimization of parallel applications on multi-core clusters. A set of parameters (message-passing, shared memory, and partitioned global address space) is detected by the Servet benchmarks. Then, the impact of the information on the mapping is analyzed to minimize the communication cost and maximize the memory access throughput.

Z.J. Jia *et al.* [4] examined the impact of mapping of a DAG-formalized application on a variety of issues of heterogeneous MPSoC such as: efficiency in the utilization of Processing Elements (PEs), the

Table 2.2: summary of design-time exploration studies on multicore/multiprocessor architecture.

Authors	Ref	Architecture	Objective	Optimization	Year
Y.-J. Chen et al.	[44]	2D NoC	Energy	SA	2009
J. Huang et al.	[49]	2D NoC		Timing Adjustment heuristic	2011
Haytham et al.	[50]	3D NoC		GA	2014
Ke Pang et al.	[51]	2D NoC		Not used	2015
Amin et al.	[52]	3D NoC		Not used	2016
S. Cao et al.	[45]	NoC		Heuristic algorithm	2016
Dawei et al.	[46]	NoC-based MPSoC		Earliest Task First & GA	2016
P. Mehrvarzy et al.	[53]	2D NoC		Not used	2016
B. Ouni et al.	[54]	MPSoC		Dynamic Slack Reclamation	2017
J. Gonzalez-Dominguez et al.	[47]	Multi-core	Performance and Resource utilization	Not used	2012
ASAM project	[55]	Heterogeneous MPSoC		Not defined	2013
M. Arjomand et al.	[56]	NoC		MOGA	2013
A. Bonfietti et al.	[57]	MPSoC		Tree search	2013
Z.J. Jia et al.	[4]	Heterogeneous MPSoC		GA	2014
A. Cilardo et al.	[58]	Heterogeneous MPSoC		Answer Set Programming	2014
X. An et al.	[48]	MPSoC		Exhaustive & NSGA-II	2015
M. Norazizi et al.	[59]	Homogeneous 2D NoC		GA	2015
R. Brillu	[60]	MPSoC		Tabu search	2015
A. Aravindhan et al.	[61]	cluster-based NoC		Depth First Search (DFS)	2016

load unbalancing in PEs, the number of memory accesses and/or the system load in network elements. Their two separated phases are: the first one is the pruning phase which uses genetic algorithms to explore the design space associated to the sub-problems of partitioning, scheduling and mapping; and the second one is the simulation phase in which a simulator evaluates more accurately each potential solution of mapping obtained by the first step. To the best of our knowledge, this is one of the few studies considering the effect of mapping of data into memory on the system behavior.

X. An *et al.* [48] presented a high-level DSE framework for the design of adaptive data-intensive applications on MPSoCs to find out a set of design points optimizing the time and energy consumption. Exhaustive and Non-dominated Sorting Genetic Algorithm-II (NSGA-II) techniques are used to support the exploration. A multiclock modeling of both software and hardware has been considered by exploiting the notion of abstract clocks. An application is represented according to its event occurrences with their precedence relations. A set of parameters is used to evaluate a design such as execution time of a task, usage ratio of a PE (ratio of the number of busy and idle cycles of clock), energy consumption, and used local memory space communicating between tasks.

In literature, the performance and resource utilization are often considered together when exploring a multi-core system. Designers often want to use the least resources but achieve the highest performance in terms of time, throughput, or communication cost. Therefore, we had a brief look at the design-time exploration studies with the popular objectives except reliability (which we discuss in more detail in the last Section). Table 2.2 summarizes the design-time studies on multi-core architecture. These studies are very interesting to give us ideas in terms of characteristics to explore a design solution in the design-time:

- many different platform structures are considered such as 2D-Mesh NoC, 3D-Mesh NoC, homogeneous/heterogeneous MPSoCs. However, by solving problems in 2D Heterogeneous MPSoC, we can all expand to the other types of architecture;

- several objectives are primarily considered such as energy, performance, resource utilization. To evaluate these objectives, designers need to qualifies these objectives This quantification process is mostly deployed indirectly through parameters such as: temperature, voltage, frequency, time, failure rate, resource quantity, workload;
- many supporting optimization algorithms are used for the exploration process such as heuristic algorithm, branch and bound, exhaustive, SA, etc;
- almost problems are described as Integer Linear Programming (ILP).

2.2.2 Run-time exploration

Unlike the static exploration, the run-time exploration has an online viewpoint which allows the system to be sensitive to the real-time requirements of the given application but it is time-consuming to make decisions as well as uncertainty as compared to the static exploration. Figure 2.9 describes the general view about the run-time exploration. A static *Mapping* solution can be initial; then the system starts running the application (solid line). The dynamic evaluation and the optimization/re-mapping are performed to modify parameters of a running application. The dashed line describes that the dynamic evaluation and the mapping can be performed from the first phase of the design and then wait for a new application into the system, and reconfigure the system; so each decision on the system operation is made online.

The **performance** is one of the issues that are of great interest in dynamic exploration. This is understandable because this aspect is necessary when performance requirements of applications often change in real time.

There are four communication-aware mapping algorithms based on the packing strategy presented in [62, 63], A.K. Singh *et al.* performs a task mapping and the four run-time task mapping heuristics for a heterogeneous MPSoC with an application modeled as a DAG. The initial tasks are implemented with software tasks (static aspect) so these are mapped onto software processing elements which are Instruction Set Processors (ISPs). The run-time upcoming tasks may be hardware tasks that need to be in re-configurable areas or in dedicated Intellectual Property cores (IP-cores).

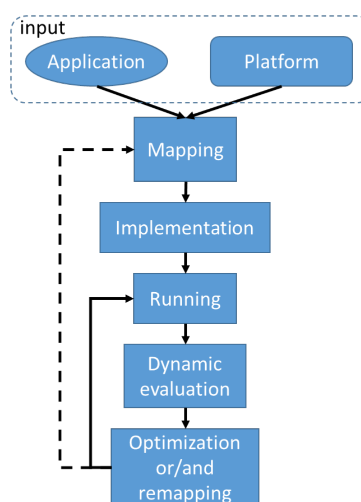


Figure 2.9: Run-time exploration flow.

Table 2.3: Summary of run-time & hybrid exploration studies on multicore/multiprocessor architecture.

Authors	Ref	Architecture	Objective	Mapping/optimization algorithms	Year
A.K. Singh et al.	[63]	Heterogeneous MPSoC	Performance	Heuristic algorithm	2010
A.J. Page et al.	[64]	Heterogeneous processor		Heuristic algorithm	2010
T. Maqsood et al.	[69]	NoC-based MPSoC		Nearest neighbor heuristic	2015
X. Zhou et al.	[70]	NoC		2 level task mapping	2017
C.-H. Huang et al.	[71]	NoC-based re-configurable system		Elastic Superposition Mapping	2017
G. Castilhos et al.	[65]	NoC	Energy	Hierarchical heuristic algorithm	2016
Navonil C.	[72]	NoC		heuristic algorithm	2017
M.F. Reza et al.	[73]	Heterogeneous NoC		heuristic algorithm	2018
Giovanni M. et al.	[66]	Multi-core	Combination of run-time and design-time	NSGA-II	2013
B. Khodabandeloo et al.	[68]	MPSoC		Robust optimization stochastic optimization	2014
W.Quan et al.	[74]	MPSoC		Probability distribution prediction	2016

A.J. Page *et al.* [64] proposed a GA-based methodology to schedule a set of heterogeneous tasks on to a set of heterogeneous processors in an effort to minimize the total execution time. To reduce the probability of processors becoming idle while waiting for a schedule to be generated, they use 8 heuristics such as max-min, min-min, max lightest loaded, min lightest loaded and four variants of the four previous heuristics. Each of them is suitable for different situations decides how a task is mapped on a PE.

Energy is also a concern in dynamic exploration. With [65], G. Castilhos *et al.* proposed a hierarchical heuristic to make a task mapping decision at run-time for NoC-based many-core systems. The hierarchical task mapping is similar to the cluster-based mapping technique in static exploration. It consists of 3 steps: 1) define a cluster to map a required application; 2) select processing elements to map the application initial tasks inside the cluster, initial tasks do not have dependencies on other tasks to start the execution; 3) select processing elements to map non-initial tasks. This hierarchical task mapping makes a trade-off between the energy consumption and the communication volume between tasks.

Some studies combined the **static and dynamic methodologies** to solve the exploration problems, as in [66], Giovanni M. *et al.* proposed a combined design-time/run-time framework. The design-time methodology uses NSGA-II to determine a set of Pareto optimal configurations representing the best trade-offs in terms of power consumption and throughput then a geometric average function is used to choose the final hardware configuration. The run-time methodology manages the resource utilization to maximize the performance under a specified power constraint by the a resource-allocation policy derived from [67], which is done on a periodic-basis. It means that allocated resources are reserved to applications within a time window with a defined size and the resource are not reallocated to other applications within the window.

Unlike Giovanni M. *et al.* with two quite independent design-time and run-time parts, B. Khodabandeloo *et al.* [68] presented a temperature-aware quasi-static task mapping-scheduling framework for the performance maximization and temperature peak minimization. In the quasi-static methodology, more than one mapping-scheduling candidates are generated at design time and then one of them is selected based on online parameters at runtime. Robust optimization algorithm is used to generate the candidates.

Thus, we have looked at the main points of the dynamic DSE in the last decade. Table 2.3 summarizes the DSE studies of run-time and hybrid mapping on multi-core/processor architectures. The foundation of dynamic exploration is based on predictive models to make decisions at run-time. Evolutionary and heuristic algorithms for making decisions are widely used in this field. Similar to the static DSE side, applications are modeled with familiar tools such as DAG, Application Characteristic Graph (ACG), Synchronous Data-Flow Graph (SDFG), etc. However, these dynamic DSE studies have a limit about guarantee to find a solution at the run-time. For example, when a problem occurs on a platform, run-time mapping strategies may be overwhelmed when problems exceed the expectations of conventional heuristics. This is quite worrying that the application requires a high reliability.

2.2.3 Fault-tolerance and reliability-based exploration

After having a comprehensive view of static and dynamic DSE studies with various common goals, this subsection is devoted to reviewing studies on the fault-tolerance and reliability-based exploration. This kind of DSE is just one of the exploration purposes as shown in Figure 2.7 but we want to dedicate this section to take a closer look at the field onto which our works will focus hereafter.

Lin Huang *et al.* in [75] introduced a task allocation and scheduling process methodology for the lifetime (aging problem) maximization during a number of application periods under a timing constraint. To determine a periodical task allocation and schedule that is able to optimize the execution time of tasks on processors and therefore, maximize the lifetime of the MPSoC embedded system. The lifetime is measure by Mean Time To Failure (MTTF). SA is used to perform the optimization process. The mapping is implemented at **design-time**. Moreover, each task must be finished before a predefined deadline. The deadlines can be relaxed by 0%, 5%, 10% respectively. Their MPSoC platforms are composed of the number of processor cores ranging from 2 to 8. Relaxing the deadline by 5%, the MTTF improvement on heterogeneous 6-processor platform is maximum 63.31%. With relaxing by 10%, the MTTF improvement on the same platform is up to 81.81%. This result is really meaningful, but it does not mention the reliability of memory components. To take an example, suppose we have an MPSoC platform containing one processor and one memory. Suppose MTTF of the processor is 10 years, MTTF of the memory is 1 year. That is, even if we are able to improve the reliability of the processor, the memory will be the lifetime bottleneck of the MPSoC because it ages much faster than processor. Moreover, the transient faults are not mentioned. A platform can have a very long lifetime, but during its operation it is constantly experiencing transient errors, the system outputs are also less reliable. However, the use of SA in this research showed the effectiveness to explore a large solution space (up to 10^{10} solution).

The failure recovery mechanism used in [76] states that the tasks on a permanently faulty processor is moved to any other available processor. The study consider streaming applications with a processor-pool based multi-core system. Dynamic Programming (DP) is used to find the optimal solution in terms of throughput performance at compile-time. All possible scenarios of remapping are stored in memory at **compile-time**. When a permanent fault occurs on a processor at **run-time**, a new task mapping among the stored mapping scenarios is chosen by a specific migration-cost function. The study used an interesting fault tolerance mechanism. However, if a temporary fault is present, this mechanism can be disturbed and ineffective and the storage overhead of the proposed technique is significant. As they mentioned, the numbers of tasks and processors in the system, T and N respectively, there are a total of $T \times (N!)$ fault scenarios. if $T = 30$ and $N = 10$, there are about 108×10^6 solutions, the storage requirement is really big. In fact, as they showed in the research, the algorithm fails for more than a 23-processor platform since the memory requirement outgrow the physical limit of the host machine.

In the scope of the context we have introduced, these study does not have much in common with our objectives except for the idea of permanent fault.

R.A. Shafik *et al.* [77] proposed an optimization technique which is composed of 3 step: 1) power minimization; 2) soft error-aware application task mapping; 3) and iterative assessment. Firstly, they use the Dynamic Voltage/Frequency Scaling (DVFS) technique [78] to find a minimization solution for a given Soft Error Rate (SER) and real-time constraint. As mentioned in the researches, soft errors have been observed that the rate of these faults increases exponentially as supply voltage decreases, because the number of particles with low energy, which can cause error in small critical charges, is much more than the number of particles with relatively higher energy. Reliability is a function of the supply voltage. Therefore, a proposed task mapping technique minimizes transient faults experienced for the chosen voltage scaling solutions. Finally, the resulting power consumption and Single-Event Upsets (SEUs) experienced are iteratively assessed until an optimized design in terms of minimized power consumption and improved reliability meeting the real-time constraint. These studies explore the design space at **design-time**. Their technique is evaluated by a case study of MPEG-2 video decoder (11 tasks) with four processing cores. The design produced in a task mapping algorithm gave 24.7% less number of SEUs experienced than the design produced in a soft error-unaware optimization. This reduction of SEUs is achieved at the expense of 5% higher power consumption compared to the design produced in in the soft error-unaware optimization. The results proved the impact of task mapping on reliability of an MPSoC application. These studies share the same objectives with us about the impact of task mapping on the overall reliability of a system. But their direction is the reliability improvement by voltage modification without using any fault tolerance strategy on a NoC platform. Changing the voltage of a system is not easy, and sometimes it is only possible to apply voltage of components at the manufacturing stage. The reliability of memory is not considered in these studies. Furthermore, just considering transient faults may not be comprehensive in the context in which we are looking.

In [79], Bolchini and Miele present a transient-fault tolerance DSE approach that allows meeting fault management requirements. The given application is composed of a set of tasks with different types of fault management constraints: fault tolerance, fault detection, and fault ignore. Their DSE methodology uses multiple fault detection and tolerance techniques and tries to achieve the optimal execution time. In the general context, this study wants to improve reliability by using fault tolerance strategies on an architecture at **design-time**. A reliability level is not considered, but they care about performance of an application when applying fault-tolerance strategies to tasks with different scenarios such as 100%, 75%, and 50% of tasks tolerated faults. The case-studies are invested such as 20, 40, 60, or 80 tasks mapped on maximum 6 processing units. The execution time of the 70% scenario improves 4 to 17 percent than the 100% scenario; and the execution time of the 50% scenario improves 8 to 33 percent than the 100% scenario. Clearly, the more fault-tolerance strategies we use, the more performance we have to lost. Therefore, the performance, especially the time execution, needs to be considered in the exploration process. However, in this study, to choose which strategy, the application needs to provide fault-management requirements. It implies that each task in the application must have a label that states the type of fault tolerance. This is not impossible, but in the early stages of design, the designer does not have much information. In addition, such limits may miss out potential solutions that meet the requirements of the application.

Shin-Haeng *et al.*'s study [80, 81] looks for the optimal solution minimizing the average power consumption under bandwidth, schedulability, and reliability constraints. The considered fault tolerance strategies are the re-execution and the replication (active and passive) that address the case of transient faults. The active replication always forces all replicated tasks executed simultaneously

at runtime, in passive replication, on voter requests, the cloned tasks are activated. These passive techniques may cause uncertain behaviors due to the accidental occurrence of incidents, thus this may cause reducing the accuracy of the DSE and optimization process. The study adopts an Evolutionary Algorithm (EA) based optimization. The static mapping is applied at **design-time**. This study use the Safety Integrity Level (SIL) standard as a constraint. SIL are defined in IEC 61508 [82], is used to specify the criticality levels of an application. In case-studies, the different level of SIL are applied. When the system is maximally enhanced for reliability with all applications having SIL 4, the power dissipation increased by 17% compared to different SIL levels. It is very significant to apply critical standards to evaluate an application and this is a suggestion for our research. However, in [81], the low-criticality tasks are dropped to give place to the high-criticality ones during the mapping process. This can never be allowed in real life when the application requires the full implementation of the functions, especially for critical applications. Moreover, this work focuses on implementing the fault tolerance of processors but not memory nor interconnections.

The work in [83] focuses on the fault tolerance DSE on Time-trigger NoC-bus multi-processor architectures. The DSE is employed at **design-time**. The processors directly exchange messages, so the data mapping process is not considered. Memory is set by default within each PE. That means the fault tolerance is not considered on memories, nor interconnections. Redundancy replicates tasks into multiple copies (replicas). The replicas can be executed on the same component (temporal redundancy) or distributed to several components (spatial redundancy). The reliability analysis focuses on computing the system-level reliability of a given design under the impact of transient faults. The estimation of the overall reliability is summarized as the occurrence probability of all transient fault scenarios. It can take a lot of time to find a solution if the number of tasks and processors in the system increases. The permanent faults are tolerated by migrating a task to another processor if its initial processor fails. The study considers the permanent fault tolerance as a hard constraint instead of an extra optimization objective. Their DSE mainly revolves around setting up scenarios for moving tasks on microprocessors.

There is a serie of studies of A. Das *et al.* that present a fault-tolerance task-mapping DSE process on MPSoCs at **design-time**. In [84, 85], the application remapping is considered. When one or more cores fail, the system restarts and a mapping with a reduced set of resources is fetched in such a way that the MTTF is maximized. All fault-scenarios mappings are pre-stored in memory at **design-time**. If it is acceptable, one can have a very large memory space to store all of these scenarios, this approach may not be suitable for applications that require continuity because the reboot phase for remapping can cause critical interruptions. The study in [21] is among the few studies examining the effects of both types of fault on a homogeneous MPSoC accompanied a shared re-configurable area. The check-pointing approach is used to eliminate the impact of transient faults for the software tasks (software tasks run on General Purpose Processors (GPPs), while hardware tasks run on Field Programmable Gate Array (FPGA)). No tolerance strategy is applied for permanent faults on General Purpose Processor (GPP) processors. Moreover, the re-configurable area is indicated to manage the fault by replicating the hardware implementation. However, there is no evaluation on the area to show how to use this technique and how it affects the system. The role of the re-configurable area in this study involves only the task mapping. A Gradient-based Design Space Exploration (GDSE) algorithm is proposed to solve the optimization problem. For experimentation, ten different applications on a homogeneous multiprocessor systems consisting of four cores with a fixed reconfigurable area are explored in terms of the mean time to permanent failure. At each application, the permanent fault aware mapping and transient fault aware mapping are used for comparison. The result showed that the transient fault-tolerant technique (checkpointing) lead to a reduction in mean time to failure

(permanent) as compared to the permanent fault preventive technique (task migration). This confirm the tradeoff between two type of faults. Because the reliability considering transient faults increases with an increase in the number of checkpoints, that is predicted by Equation in Equation 1.4 and Subsection 1.3.3. But, the reliability considering permanent faults decreases due to an increase in the expected execution time which increases processor aging, as predicted in Equation 1.3. The relation between permanent and transient faults is used as a reference in our experiments. In the studies, many fault-tolerance strategies are taken into consideration. Both types of faults (transient and permanent) were taken into account. However, the studies focus on PE have not mentioned other types of elements in the system such as memory, bus. And so, the impact of data allocation on the overall reliability has not yet been evaluated.

In [86], the authors proposed a DSE methodology to analyze three transient fault tolerance strategies (active and passive replication, and check-pointing with replication) in terms of execution time, deadline and energy consumption. In active replication, the replicas are executed along with the original task. After the completion of the tasks, the outputs from all the tasks are fed into a voter to check errors. With passive replication, the replicas are not executed at the same time. First the original task and one of its replica task are executed, then a checker is used for error detection. If any error is detected, another replica is executed. This study focuses on the specified NoC architecture which has a communication infrastructure containing routers and links that helps in transmitting messages between the source and destination PEs. Each PE owns a fault-detection sensor. There is a manager core that hosts the real-time operating system and manages the fault events from the general PEs. For experimentation, applications with different topologies and number of tasks varying between 5 and 20 are mapped in a target platform of 8×8 2D Mesh NoC. As shown in the results, with the active replication, number of tasks meeting their deadline is very high (almost 100%). Meanwhile, the fault-tolerance strategy integrated check-pointing has $\approx 70\%$ of tasks satisfying the deadline. It is because, the execution time increases by checkpoiting a task. And in the event of occurrence of faults, the execution segments affected by faults are re-executed. This study does not evaluate reliability level however it does give us a view into the dealine satisfaction of fault tolerant strategies in presence of transient faults.

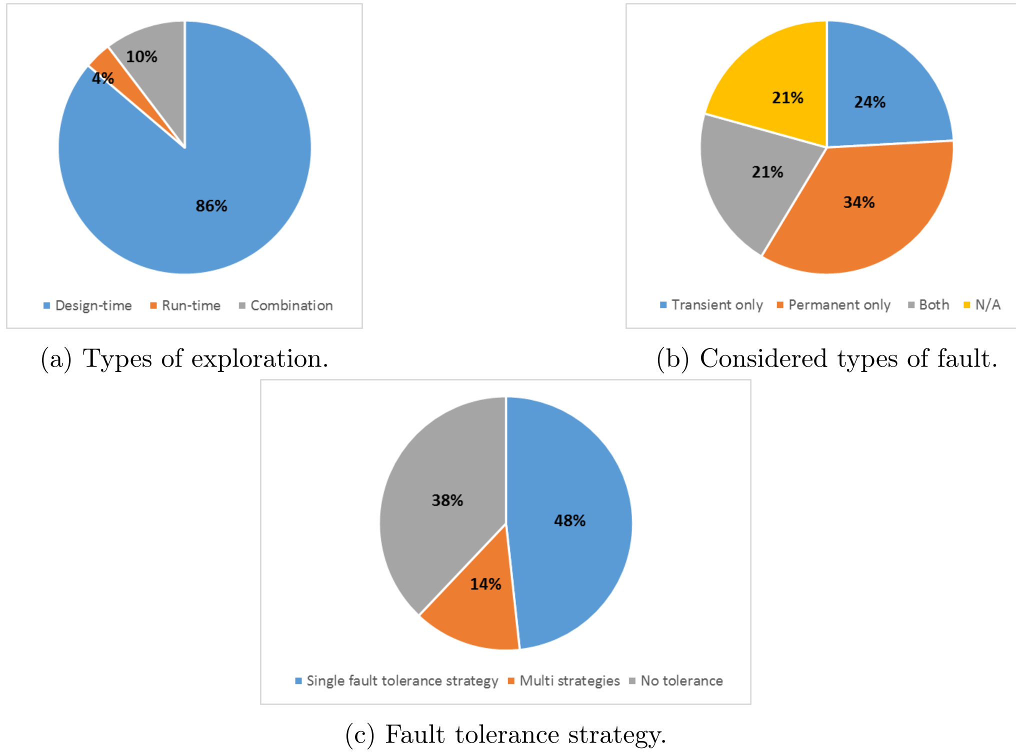
2.2.4 Analysis

In Section 2.2.3, we looked at each specific study related to improving reliability and fault tolerance. Now, herein, we will analyze the overall trend in this field.

Figure 2.10a describes the ratio of DSE types of studies among the reliability-aware studies we reviewed. The run-time accounts for only 4% of all considered DSE studies. Although this direction has certain benefits (such as sensitivity to real-world situations), the barrier of time and effectiveness makes it less applicable. It is easy to see that the Design-time has always dominated the field of system-level DSE because of the flexibility, the less-time-consuming and the acceptable accuracy.

Figure 2.10b describes the ratio of different types of fault models among the reliability-aware studies we reviewed. There is a small note here that "N/A" means that some authors do not specify the fault type they focus on in their study. It can be seen that there are not too many differences between the numbers of studied fault models. Each type of fault affects different behaviors on the system. However, in our view, evaluating the reliability of a system should pay attention to both types to have a more comprehensive view.

Figure 2.10c describes the using of fault tolerance strategies among the reliability-aware studies we reviewed. There are two general trends for improving reliability: the use of intermediate parameters



(a) Types of exploration. (b) Considered types of fault.

(c) Fault tolerance strategy.

Figure 2.10: Review the reliability-aware DSE studies.

and the use of fault tolerance strategies. The reliability of a processor/processing units is a function of some intermediate parameters such as temperature, voltage, frequency, execution/communication time. The functions can be found in FIDES [6] and MIL-HDBK-217 [7]. 38% of the studies tries to configure, set up these parameters in the design phase to maximize the reliability of an MPSoC system. More than 60% of studies use at least one fault-tolerance strategy to increase reliability, but almost studies concern only one of the two fault types. Exploring a design space with different fault-tolerance strategies for different fault types will be integrated in our DSE methodology.

Table 2.4 summarizes the reliability-based DSE studies in the last decade. To the best of our knowledge, no DSE study has addressed the comprehensive effect of the mapping (task to process, data to memory) on the reliability of a heterogeneous MPSoC system. And no one has consider the fault tolerance for memory and connection components in the MPSoC DSE process.

Table 2.4: Summary of reliability-based application mapping studies on multiprocessors architectures.

Authors	Ref	Platform type	DSE	Application Model	Fault	Fault tolerance	Optimization technique	Mapping	Year
Lin Huang *et al.*	[75, 87]	Homogeneous/ heterogeneous MPSoC	Design-time	DAG	Permanent (aging)	No	SA	Task into PE	2009 2010
Chanhee Lee *et al.*	[76]	Heterogeneous processor-pool based multi-core system	Design-time and quasi-dynamic	Streaming application DAG	Permanent	Task remapping	DP	Only one task into one PE	2010
Onur Derin *et al.*	[88]	Heterogeneous NoC multiprocessor	Design- and run-time	Kahn Process Network	Permanent	Remapping	ϵ-constraint method	At most one task onto each tile	2011
Brett H. *et al.*	[89]	Homogeneous NoC-based MPSoC	Design-time	Directed graph	Permanent	Task remapping	Critical Quantity Slack Allocation	Only one task into one resource	2010
	[90]	Homogeneous NoC-based MPSoC	Design-time	Directed graph	Permanent	No	Ant Colony Optimization	Only one task into one resource	2010
Chen-Ling C. *et al.*	[91]	Homogeneous NoC-based multiprocessor	Run-time	ACG	Permanent and transient	Task migration	N/A	At most one task onto each tile	2011
Philip A. *et al.*	[92]	MPSoC	Design-time	Set of independent tasks	Transient	Check-pointing	N/A	Replicating tasks into cores	2011
Cristinel A. *et al.*	[93]	Homogeneous NoC-based multicore	Design-time	ACG	N/A	No	Branch and bound	Task into tile	2011
Ivan Ukhov *et al.*	[94]	Heterogeneous multicore	Design-time	data dependencies task graph	Permanent	No	GA	Task into core	2012
R.A. Shafik *et al.*	[77]	Homogeneous NoC-based MPSoC	Design-time	DAG	Transient	No	DVFS	Task into core	2012
Suleyman T.	[95]	Heterogeneous MPSoC	Design-time	Set of independent tasks	N/A	Task duplication	DVFS and EDF	Task into core	2012
Hamid R. *et al.*	[96]	Heterogeneous Distributed Computing System	Design-time	Set of dependent tasks	N/A	No	SA & Tabu Search	Task into processor	2012
Fatemeh K. *et al.*	[97, 98]	Homogeneous NoC-based multiprocessor	Design-time	Directed core graph	Permanent and transient	Task migration	Ant Colony Optimization	Core (of task) into tile	2012 2013
F. Bolanos *et al.*	[99]	Heterogeneous NoC	Hybrid	ADAG	N/A	Task relocation	Population-based Incremental Learning	Task to core & core to network	2013
Cristiana B. *et al.*	[79]	Heterogeneous MPSoC	Design-time	DAG	Transient	Multiple techniques	GA	Task into processor	2013
A. Mahabadi *et al.*	[100]	Heterogeneous NoC-based MPSoC	Design-time	Task Graph	Transient	No	Heuristic algorithms	Multiple mapping	2013

Authors	Ref	Platform type	DSE	Application Model	Fault	Fault tolerance	Optimization technique	Mapping	Year
Shin-Haeng K *et al.*	[80, 81]	NoC-based MPSoC	Design-time	Kahn Process Network	Transient	Re-execution Replication	EA	Task into processor	2014
Jia Huang *et al.*	[83]	Heterogeneous time-triggered execution paradigm NoC-based MPSoC	Design-time	Kahn Process Network	Transient & permanent	temporal & spatial redundancy	EA	Task into processor	2014
Anup Das *et al.*	[101]	Homogeneous MPSoC	Design-time run-time	SDFG	Permanent	Task migration	Matlab optimization toolbox	Task into processor	2012
	[84, 85]	Homogeneous/ Heterogeneous MPSoC	Design-time	DAG SDFG	Permanent	Application remapping	CVX solver Matlab/ heuristic algorithms	Task into processor	2013
	[21]	Homogeneous MPSoC with a shared reconfigurable area	Design-time	DAG SDFG	Transient & permanent	Check-pointing	GDSE	SW task into GPP HW task into FPGA	2013
	[102]	Homogeneous MPSoC	Design-time	DAG	Transient & permanent	Replication	NSGA-II	DVFS & task into processor	2014
	[103, 104]	Homogeneous MPSoC	Design-time	SDFG	Permanent	No	Gradient-based heuristic	task into processor	2014 2016
L. Zhang *et al.*	[105, 106] [107]	Heterogeneous multi-core	Design-time	DAG	Transient	No	DVFS-base heuristic	Heterogeneous task into processor	2015-2017
Namazi *et al.*	[108, 109]	Homogeneous NoC-based many-core	Design-time	DAG	Transient & permanent	k-out-of-n	Mixed Non-Linear Programming	DVFS & task into processor	2017
N. Beechu *et al.*	[110, 111] [112]	Homogeneous NoC-based many-core	Design-time	ACG	Permanent	Task migration	Nodes Average Distance	Only one application core into processing core	2017
Chatterjee *et al.*	[113]	NoC-based multicore	Design-time	ACG	N/A	No	Particle Swarm Optimization	Only one application core into NoC router	2018
	[86]	NoC-based multicore	Design-time	DAG	Transient	Check-pointing, Replication, Hybrid	N/A	Task into processor	2018
W. Gao *et al.*	[114]	Tile-based 3D NoC multiprocessor	Design-time	ACG	N/A	No	Simulated Allocation	Only application core into one tile	2018

2.3 Summary

This chapter 2 has reviewed the literature studies related to our research: the MDE approach and the fault-tolerance DSE for the MPSoCs.

There are many MDE studies using meta-models as well as models for many different purposes such as simulation, code generation, DSE. Several MDE studies propose meta-models at different design levels as CABA, TPV, frequency, RTL etc. However, the application of MDE to explore solutions for MPSoC platform is very limited. Furthermore, there is a lack of a meta-model that could enable designers to model an MPSoC platform integrating fault tolerance strategies.

There are a lot of studies which look for the optimal designs for systems of multi-task applications and multi-core platforms in both dynamic and static aspects. Many algorithms are used to support this DSE processes such as mapping algorithms, optimization algorithms. These studies use application and platform models as the input of their DSE process and then return an optimal solution with the mapping. The principal optimization objectives of these processes are the energy consumption, the resource management, the temperature management, the reliability, etc. Reliability is a very important issue in critical applications, but it does not have adequate proportions in this field yet.

We have also reviewed the studies that referred to the reliability-awareness or/and the fault-tolerance consideration. There are a lot of existing reliability-based DSE studies. The common goal is to improve the reliability of a platform under the requirements of a multi-function application. Different approaches at different levels of the system have been studied. There are two types of techniques that can be used to increase reliability: (1) through some intermediate parameters and (2) fault tolerance strategy. The intermediate parameters may be voltage, frequency, execution/communication time, temperature, which affect failure rate through a mathematical function. There are several fault-tolerance strategies used such as resource replication, task replication, checkpointing, re-execution. Different search strategies have been used (SA, GA, EA, DVFS, etc). However, there remain gaps in the exploration process that have not been considered or considered separately, such as the impact of the two types of errors, the impact of different component types and the impact of the task/function and data mapping on the overall reliability.

In summary, there are five main conclusions of this review: 1) a model-driven DSE methodology for an MPSoC system with the fault-tolerance is very rare in literature; 2) there is a lack of a platform meta-model that can capture the fault-tolerance; 3) the design-time DSE still predominates and is effective in the studies; 4) no DSE study addresses the comprehensive effect of the mapping (task to processor, data to memory) on the reliability and the fault tolerance for memory/communication component of a heterogeneous MPSoC system; 5) a lack of exploration that examines the effects of both types of faults (permanent and transient) and provides tolerance strategies for these two types of faults. We intend to address these five points in the rest of this work.

Chapter 3

Fault-tolerant platform meta-model

Abstract: As presented in the previous chapter, the prerequisite to implement automated support for DSE is an appropriate models to represent problems and solutions. Meanwhile, there is a lack of a meta-model that combines performance exploration with the fault-tolerance problem. In this chapter, we first propose a new meta-model and discuss the issues surrounding the proposed meta-model. The meta-model provides the basis of models used in the DSE process in the following chapters. In the first Section, a meta-model enabling the construction of a heterogeneous MPSoC platform with support features for fault tolerance is presented. The next Section describes a Graphical User Interface (GUI) that allows modeling an MPSoC platform according to our proposed meta-model.

Contents

3.1 Platform meta-model **54**
3.1.1 Proposed meta-model 54
3.1.2 Fault tolerance in the meta-model 58
3.2 Supporting tool **64**
3.3 Summary **67**

3.1 Platform meta-model

In this section, we present a platform meta-model integrated the fault-tolerance which is built on the UML syntax. Finally, we introduce the fault-tolerance aspect in our meta-model.

3.1.1 Proposed meta-model

The platform meta-model of the ModES [41] framework partially matches our objectives as well as our definitions. However, their MPSoC platform meta-model does not consider fault tolerance. Therefore, we develop our new platform meta-model derived from the proposition of the ModES meta-models.

The Figure 3.1 depicts the architectural part of the proposed meta-model described below. In ModES, two basic classes are used to build an MPSoC platform meta-model: *platform*, and *component*. A platform contains many components.

Components are classified as hardware (*compHardware*) (Definition 1.3) and software (*compSoftware*) (Definition 1.4). *compHardware* can be a communication component (*compCommunication*), a memory component (*compMemory*) or a processing component (*processingElement*). *processingElement* may be a *comDPE* (dedicated, an accelerator, an Intellectual Property (IP) on FPGA or ASIC) or a *compPPE* (programmable, a GPP (General Purpose Processor)). In our proposed model, we add the element *portWire* to describe the interface of these component to the external environment. All of

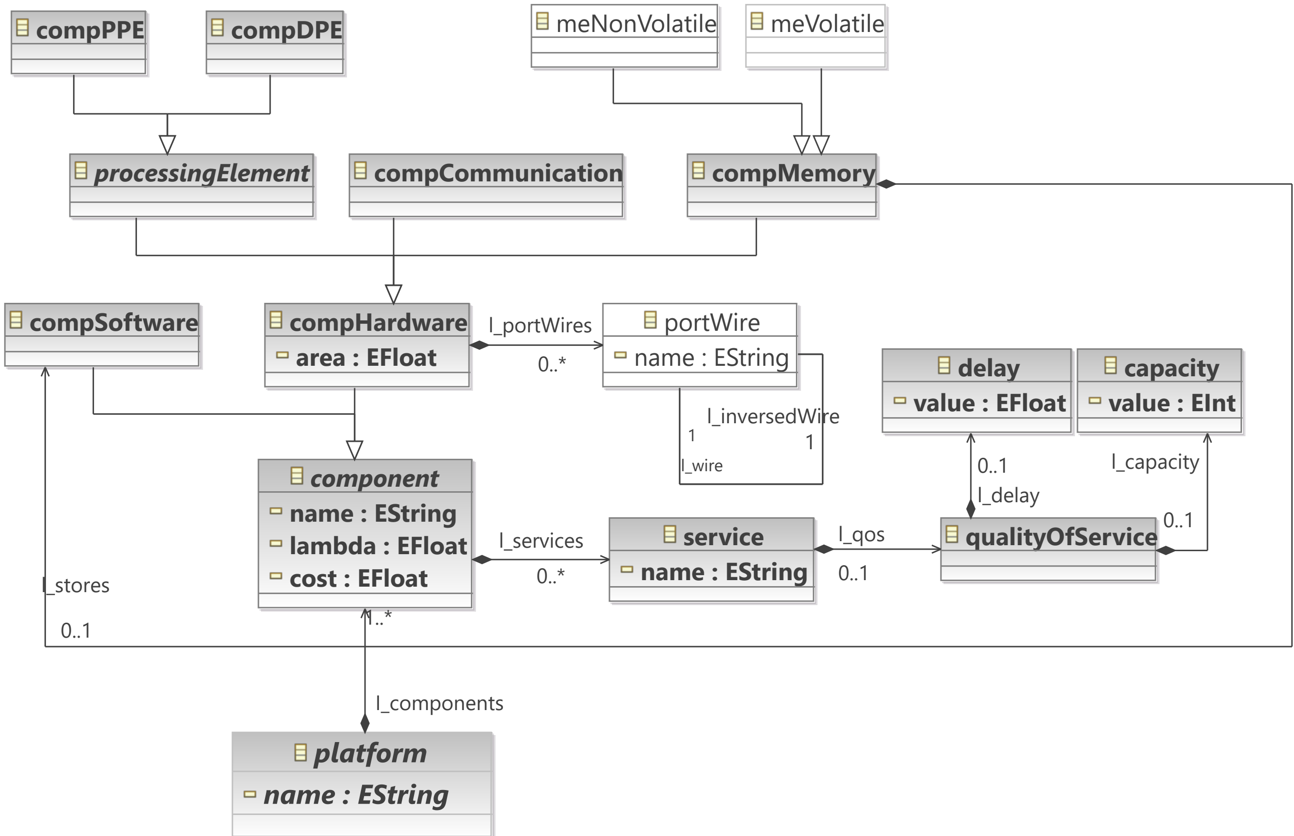


Figure 3.1: architectural part related to the component. Gray elements refer to the ModES platform meta-model [41]. White elements are defined in this thesis.

Table 3.1: quality of service and properties specializations.

Component type	Service	Metric (quality of service)	
		Delay	Capacity
compCommunication	data transfer	interconnection delay	N/P
compMemory	storage	read/write delay	memory size (KB, GB...)
compPPE	Computation (Instruction set)	mean inst. execution time	inst. set size (number)
compDPE	Computation (logic block (LB))	mean LB execution time	number of logic blocks

the different component types inherit the common features described in the component. A component can offer *services*.

For each service provided by a *compHardware*, *delay* and *capacity* are generic properties defined in the ModES meta-model *qualityOfService* (bottom-right elements in Figure 3.1). A *service* is characterized by *qualityOfService* (QoS) which carries some metrics that are useful for the system evaluation:

- a *compMemory* provides a storage *service* with a read/write *delay*; the *capacity* of a *compMemory* is defined by the memory size;
- a *compCommunication* (interconnection) provides a communication *service* with an interconnection delay;
- a GPP (in *compPPE*) provides a computation *service* by its instruction set with a mean execution time of an instruction;
- a dedicated processor (in *compDPE*) also provide a computation *service* through a logic block matrix with a mean execution *delay* of a logic block;
- a PE component can offer a *service* with a computing capacity (*capacity*) which corresponds to a number of operations it can execute;
- as presented in Chapter 1, the failure rates (λ_{PF} and λ_{TF}) are used to estimate the reliability of the component.

Table 3.1 represents a general description of the component in an MPSoC according to our meta-model. Each component owns a value of "cost". The "cost" term here is used as a constraint on the quantity of required components. In the most basic way, each component has a cost value of 1. It implies that the total cost of the system is the total number of components used. But in another term, the "cost" value can be the volume of a component if designers are interested in optimizing the volume of the whole system. In that context, the values of the elements are different and the optimal solution may differ from the above. Thus, the concept of "cost" here is used in a broad sense depending on the demand explored by designers such as volume, area (if 2D circuit), quantity, financial cost. But there

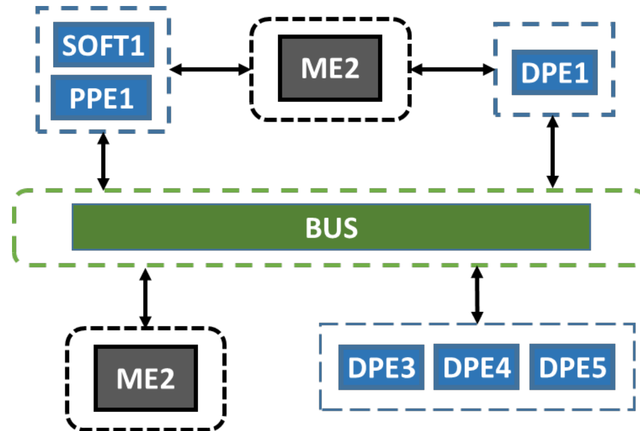


Figure 3.2: an example of an MPSoC platform.

is a common feature that the total cost of the whole system at the platform level is calculated as the sum of all costs of all components used.

As such, on a basic view, we have described elements (platform, component) of meta-model that allows building a relatively complete platform. However, if observing the components separately, the evaluation of components and platforms will be fragmented. Definition 3.1 is to connect these components and allow to create a complete meta-model.

Definition 3.1. A subsystem is composed of one type of hardware component and possibly several versions of software components.

The description of subsystems supports the redundancy modeling used in tolerance strategies. Subsystem model is used in many works about redundancy allocation for the fault tolerance and reliability evaluation [115].

For better understanding, we consider an example of an MPSoC platform in Figure 3.2. This platform is composed of 8 components: 2 ME components, 4 DPE components, 1 PPE component and, 1 COMM component. These components communicate with others through a link. As mentioned earlier, these links shape the topology of the platform. Each component can operate independently but the group of the three components ($DPE3$, $DPE4$, $DPE5$) creates a subsystem that executes a same function.

In the point of reliability view, if 3 PEs execute one function, the reliability of the result for the function is increased (TMR strategy). If the individual components are observed separately, it will make it difficult to evaluate the reliability for a function and therefore it is difficult to evaluate the reliability of a platform for an application. Meanwhile, using the subsystem concept still does not affect the unity of an entire platform. Thus, the subsystem is not only the intermediate level between the platform level and the component level but also a bridge between a platform model and an application model. Evaluating a platform without the companion of an application is not useful but, if the platform model and application model are intertwined too much, then the re-usability, the extensibility of the platform is greatly limited. The subsystem level both ensures the platform model's linkage with the application model, but also creates independence to the platform model, especially in terms of evaluation of the reliability.

In our work, a supplementary architectural level is added namely the *subsystem* level. Indeed evaluating whether requirements of a function are met may rely on the evaluation of several components

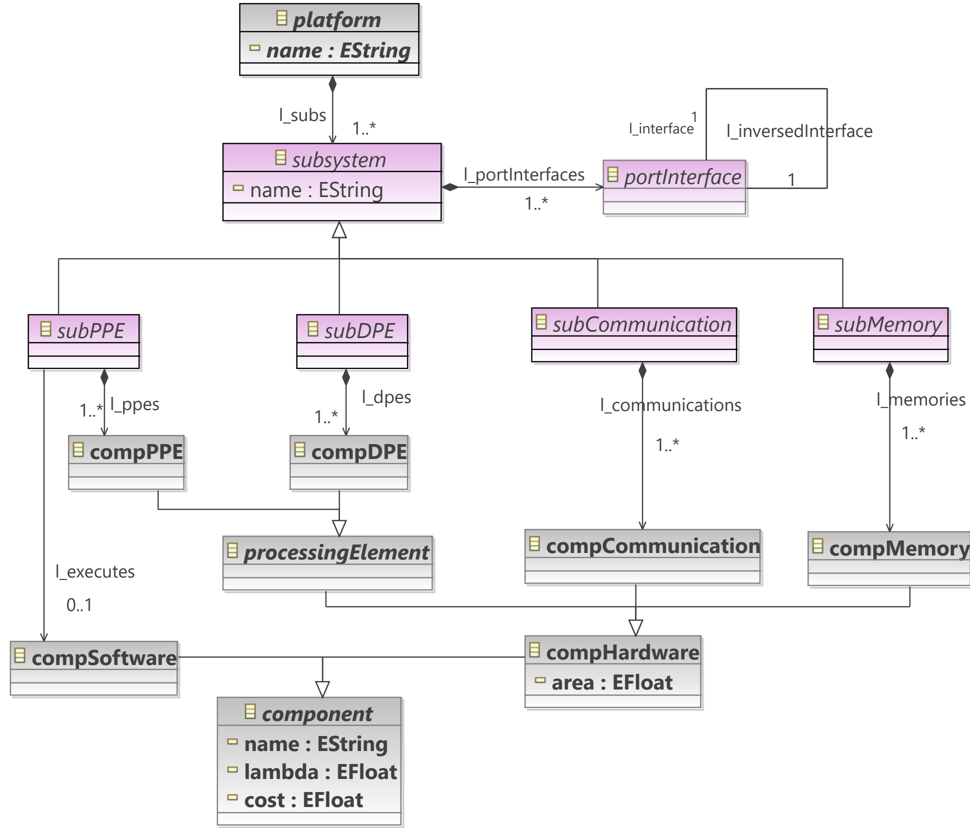


Figure 3.3: proposed meta-model of subsystem. Gray and bold elements refer to Figure 3.1. Violet and italic elements represents the subsystem level.

as a whole. For example, a software component always runs on a GPP component for a given function. To know whether the requirements of the function are met, we must evaluate both the software component and the PPE component in their common viewpoint. Furthermore, applying fault tolerance strategies like redundancy implies to evaluate the reliability of "group" of components.

The *subsystem* is defined (Definition 3.1, top-middle violet elements in Figure 3.3) in the scope of our platform meta-model as follow:

- a $platform$ may contain several different $subsystems$;
- a $subsystem$ may be one of the four types corresponding to components: $subPPE$, $subDPE$, $subCommunication$, $subMemory$;
- a $subsystem$ contains at least one component; all hardware components in a subsystem are of the same type;
- however, there is an exception when a $subPE$ executes 2 consecutive functions for which these two functions have data exchange, the data does not need to be stored on another $subMemory$ that can be stored on a local memory right inside that $subPE$; this is to reduce latency for the execution and is also consistent with the reality;
- $portInterface$ represents the bounding of a subsystem.

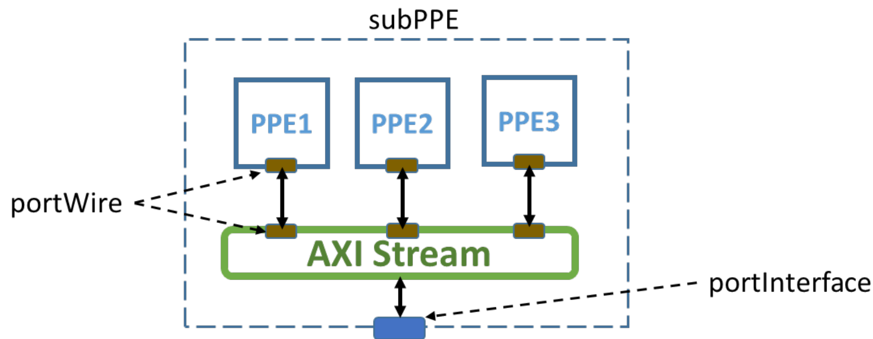


Figure 3.4: an example: a $compCommunication$ in a $subPPE$.

$portInterface$ reference defines connection wires between subsystems, which is required to calculate the connection delay between subsystems. Moreover, for a given function, $portInterface$ block allows to define the relationship between elements of a platform upon which the function relies. For example, when a $subPPE$ computes a function, we need to know in which $subMemory$ the data required by the execution is saved. Thus, components connect to each other within a subsystem via $portWire$, and subsystems connect together within a platform via $portInterface$. Herein, a problem is that it may be necessary to have a $compCommunication$ inside a subsystem to connect components in that subsystem as shown in Figure 3.4. However, for simplicity, we only consider a subsystem containing only one component type and then $portInterface$ and $portWire$ can represent $compCommunication$ in terms of delay and reliability.

Behaviors of a $compSoftware$ depends on its supporting hardware components. For this reason, the relation between a $compSoftware$ and a $compHardware$ needs to be defined. In the ModES, authors admit this but it is not defined on their platform meta-model. To evaluate the subsystem reliability and performances, we need to know all of the component relationships in a subsystem. Therefore, $subPPE$ is composed of $compPPE$ and $compSoftware$.

3.1.2 Fault tolerance in the meta-model

After having presented the architecture part of the meta-model, now we can introduce the tolerance strategy part ($faultTolerance$). Several fault tolerance strategies can be applied to a $subsystem$ (redundancies, re-execution, correction codes etc.) [116] as shown in Figure 3.5. The figure describes the fault-tolerance aspect of the meta-model.

A tolerance strategy performance is evaluated by computing the $estimation()$ method of a given subsystem. This method computes the reliability of the subsystem through a set of probability formulas taking into account each of the subsystem's component reliability. Of course, the $estimation()$ method is also composed of the cost and time evaluations of the considered strategies. Each strategy has its own typical parameters. For example, there are typical strategies presented in Chapter 1:

- the $errorCodeCorrection$ strategy supports the fault tolerance for a memory with redundant bits ($redundantBits$) (see Section 1.3.4);

- with the k-out-of-n strategy, a subsystem is functional when there are equal to or greater than k working components (n being the number of redundant components in the subsystem). There are two types of redundancy: spatial and temporal. Spatial redundancy is when several $components$

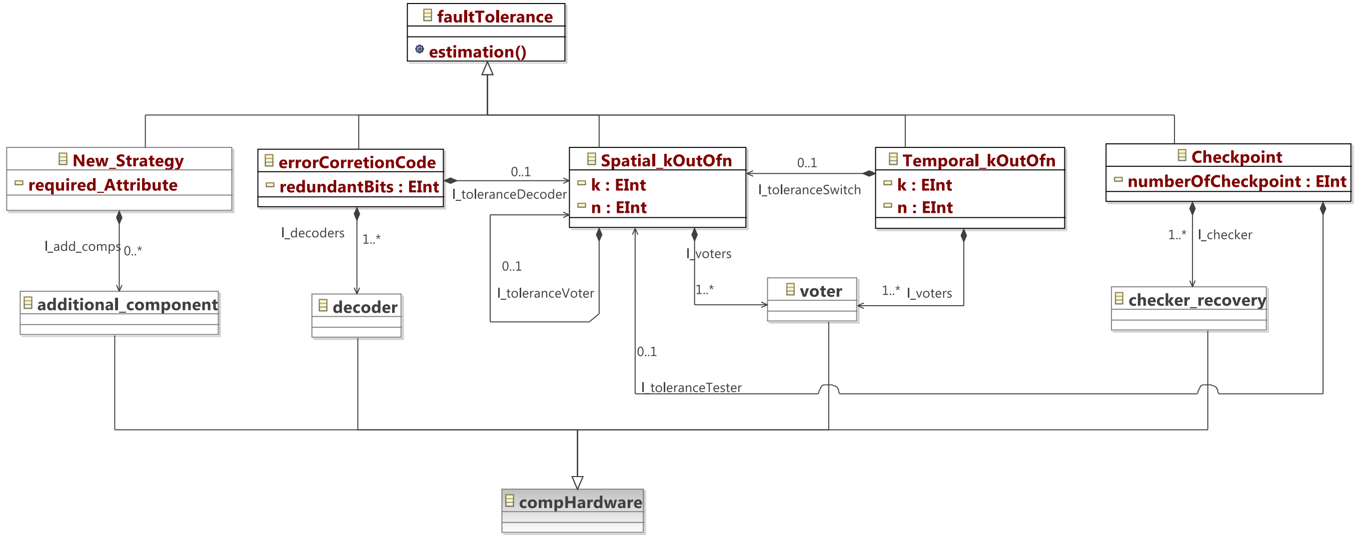


Figure 3.5: fault tolerance strategies on the proposed platform meta-model.

are used in the same *subsystem* (a typical representative is TMR). Temporal redundancy is when a function is re-executed several times in only one *component* in a *subsystem* (a typical representative is TReR) (see Section 1.3.1 and 1.3.2);

- the *Checkpoint* strategy saves the function's states by several check-points during the execution of that function on *subPPE* or *subDPE*, so that it can restart from that point in case of failure (see Section 1.3.3).

In the tolerance strategies, there are additional dedicated hardware components required by each specific strategy such as *voter*, *decoder*, *checker_recovery*. These components also inherit the features of *componentHardware*. These components can affect the reliability and cost of subsystems as well as the entire platform. Therefore, we can apply the redundancy tolerance strategy also for these components (*toleranceDecoder*, *toleranceVoter*, *toleranceTester*).

The use of any other strategy depends on the designers and therefore they can update the strategy into the meta-model. Each new strategy can be added through a class (eg. like *New_Strategy*). Also as the other strategies, inside the class, we can define required attributes of the new strategy. Of course, any strategy should be quantified through a method *estimation*(). The class *additional_component* represents the required specific hardware of the new strategy.

Thus, such strategies are ready to use and evaluate. The meta-model in Figure 3.5 can be considered as an available library of fault-tolerance strategies that designers can use for their systems.

Figure 3.6 describes the relationship between the fault tolerance and the elements of a platform. For easy viewing, Figure 3.6 is cut into two images including Figure 3.8 and Figure 3.7, respectively, the right and left parts of Figure 3.6.

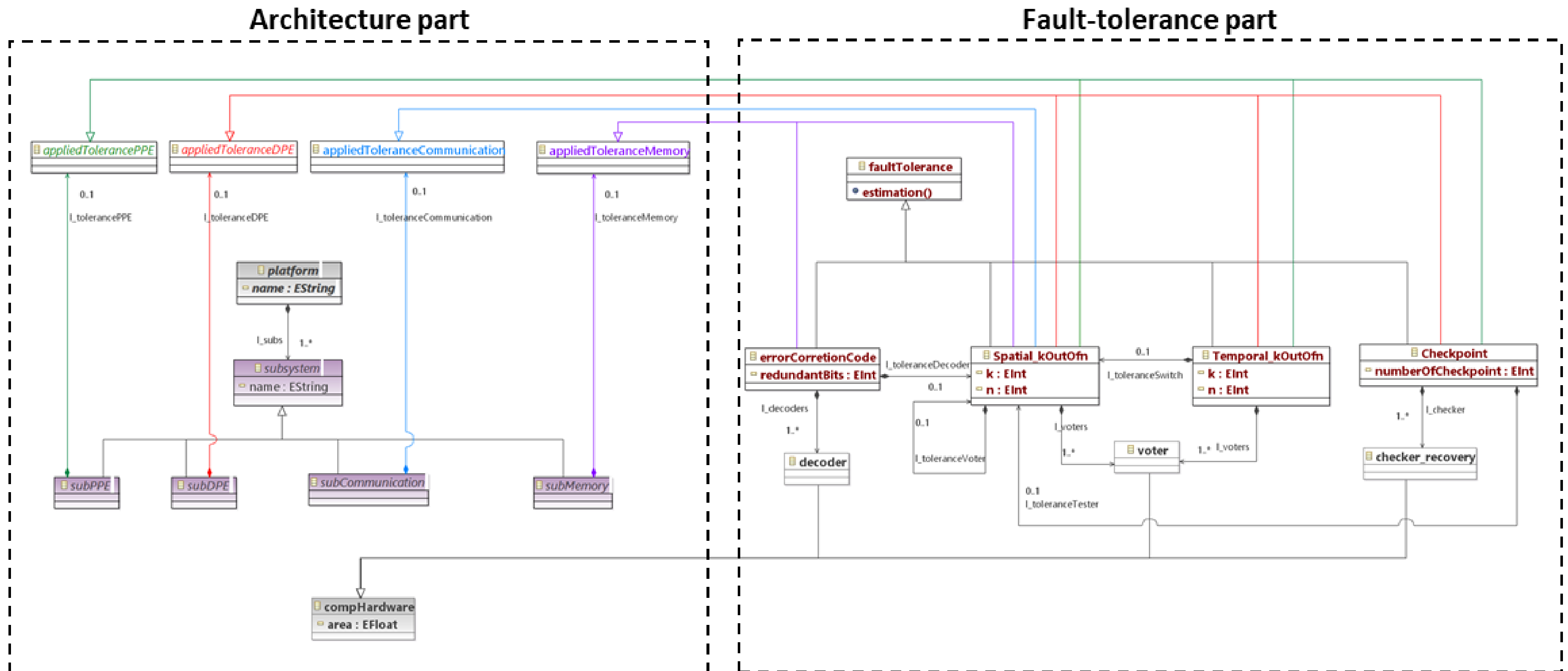


Figure 3.6: relation between the simplified architectural part and the fault tolerance part of the proposed platform meta-model. The left part (Architectural part) refers to Figure 3.8. The right part (Fault-tolerance part) refers to Figure 3.7.

The right part (Figure 3.7) represents the tolerance aspect of the meta-model that also is presented earlier in Figure 3.5.

The left part represents the architectural part shown in Figure 3.8. To recall, the architectural part always includes elements shown in Figures 3.1 and 3.3. However, to make better to follow, we obscure elements that do not have direct connection to the fault tolerant part. A DPE subsystem is composed of DPE components and a corresponding fault-tolerance strategy. It means that only one fault-tolerance strategy is chosen to apply on a subsystem and the strategy will affect all DPE components of the subsystem. The strategy must be selected from the library and must be compatible with the DPE subsystem. The class $appliedToleranceDPE$ represents a fault-tolerance strategy applied on a DPE subsystem. The $appliedToleranceDPE$ consequently can be the $Spatial_kOutOfn$, the $Temporal_kOutOfn$, or the $Checkpoiting$ (green class in Figure 3.6).

Similarly, the $appliedTolerancePPE$, $appliedToleranceCommunication$ and $appliedTolerance-Memory$ classes represent the tolerance strategy applied to a PPE subsystem, a COMM subsystem, and a ME subsystem respectively. Appropriate strategies corresponding to each subsystem type also are linked to the library. For example, a tolerance strategy applied to a memory subsystem ($appliedToleranceMemory$) can be $errorCodeCorrection$ or $kOutOfn$.

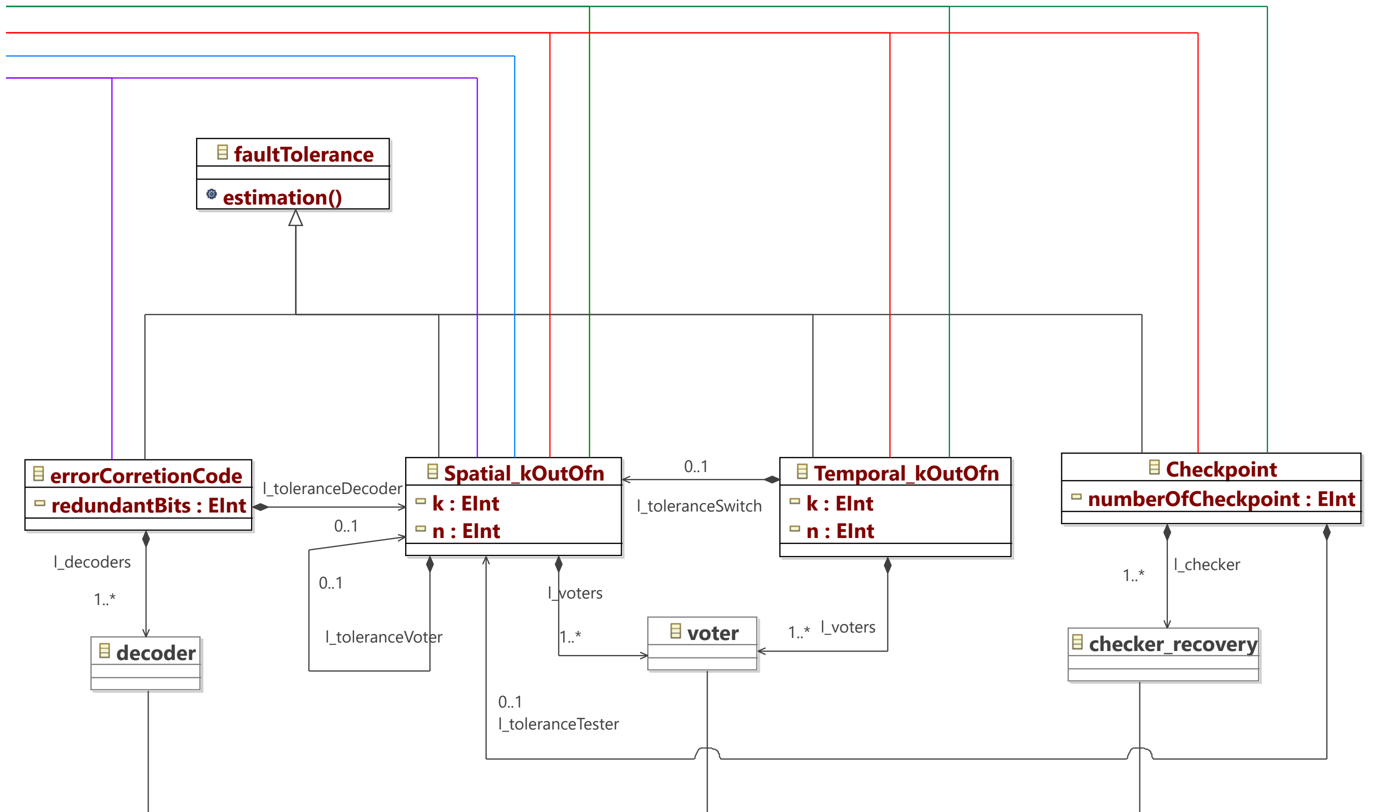


Figure 3.7: fault tolerance part of the proposed platform meta-model.

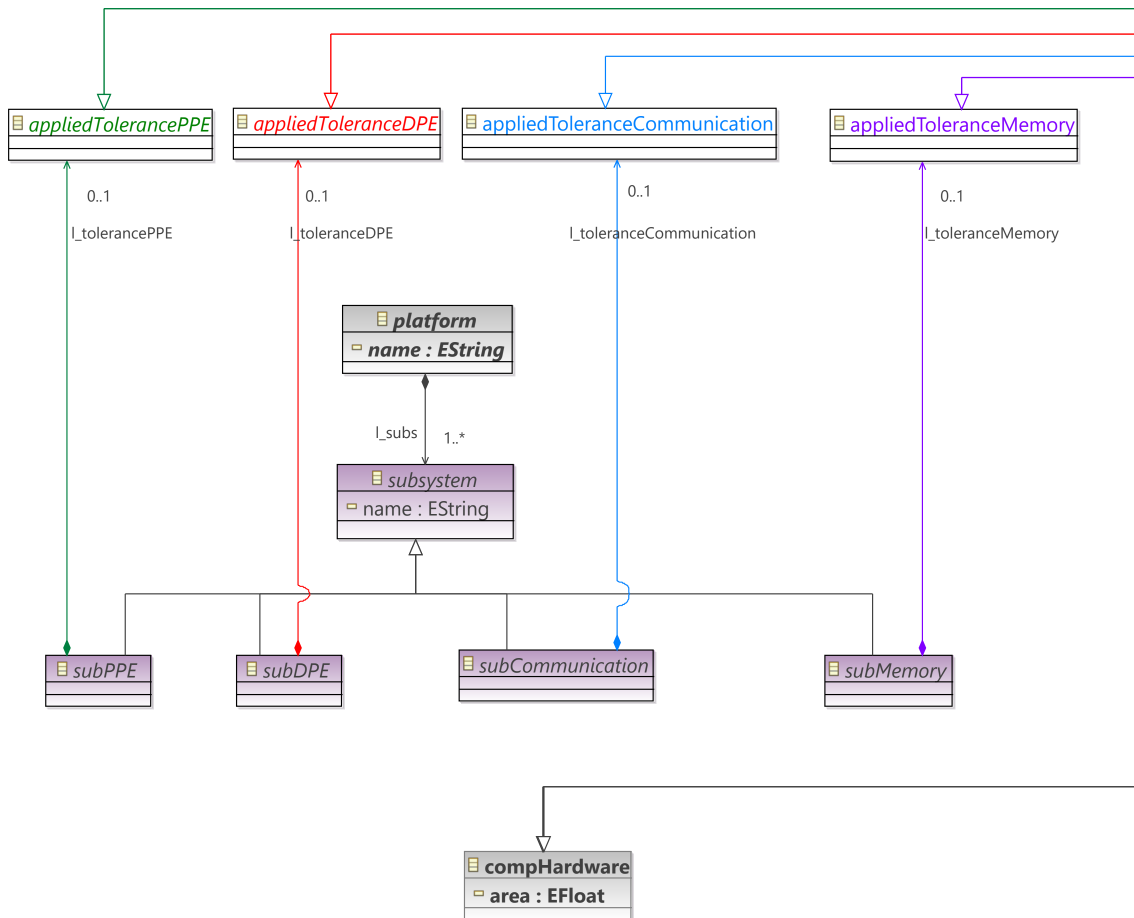


Figure 3.8: the simplified architectural part related to the fault tolerant. The $platform$, $subDPE$, $subPPE$, $subCommunication$, $subMemory$ and $compHardware$ elements refer to Figure 3.3.

The whole UML view of the meta-model is given in Figure 3.9. This figure refers to Figure 3.1, 3.3 and 3.8 for the architectural part, and Figure 3.5 for the fault-tolerance part. The meta-model that is presented in [117] allows designers to build a platform taking into account the fault tolerance problem .

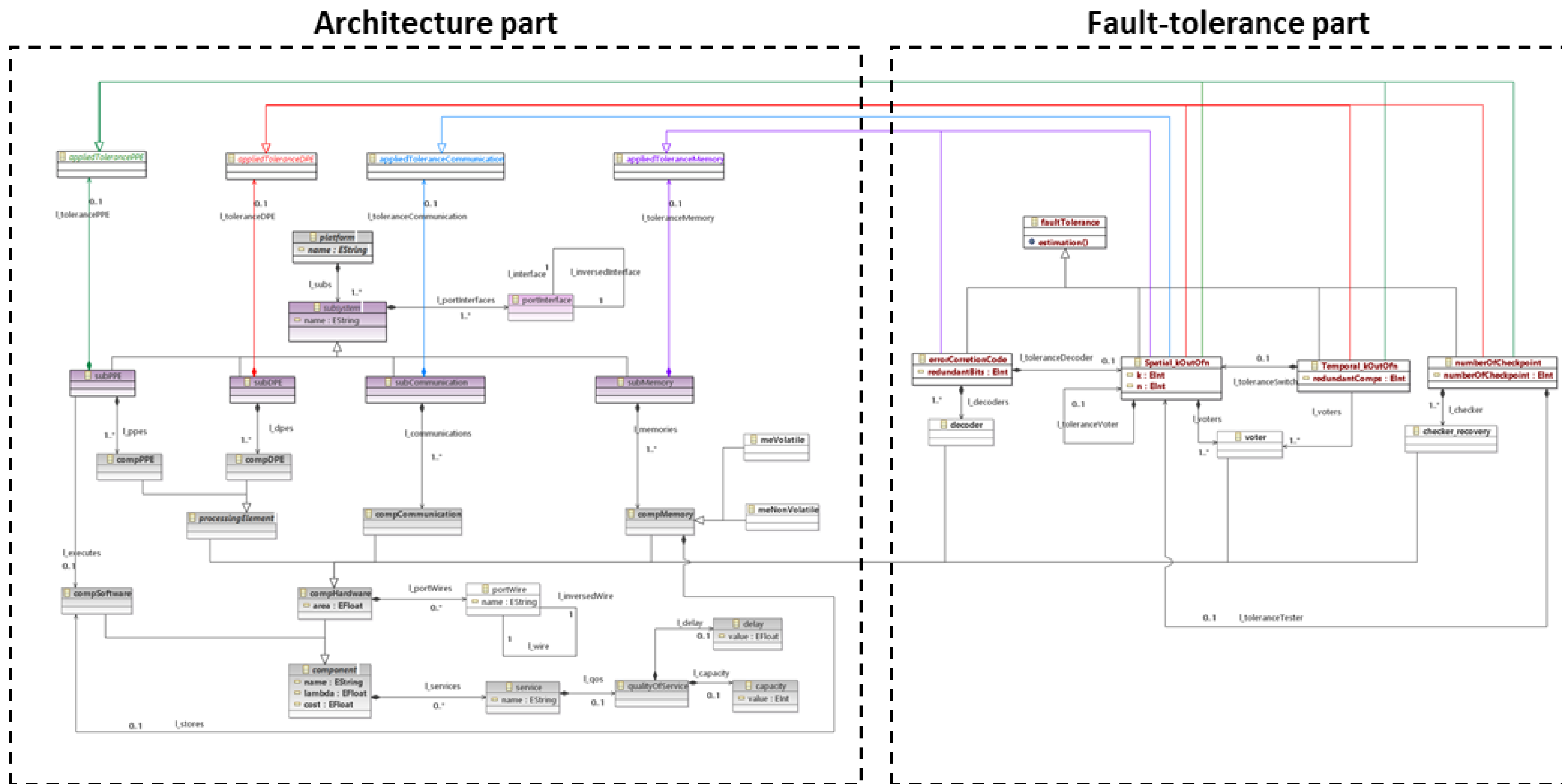


Figure 3.9: overall view of the platform meta-model. The left part (Architectural part) refers to Figure 3.1, 3.3 and 3.8. The right part (Fault-tolerance part) refers to Figure 3.5.

3.2 Supporting tool

Models built by our framework are based on the meta-model defined in the previous section. The tool is built on the Sirius environment with the workspace called PolarSys [118], which is an Eclipse project allowing the creation of graphical modeling workbench. We created a workbench which allows designers to describe their platform through a graphical user interface.

Figure 3.10 illustrates an example of such a platform "*Example*" composed of six subsystems which hold different component types: 3 PE subsystems (in blue), 2 memory subsystems (in white) and 1 communication subsystem (in green). The whole platform is placed in the center of the workspace with an underlined label (**EXAMPLE**), which is the name of the platform. The palette on the right (Figure 3.10) is composed of basis elements that allow the user to "drag and drop" to create the desired architecture. The subsystems connect together through ports and wires. Components and their parameters are declared inside corresponding subsystems. Herein the connection between the two subsystems is bi-directional, but the mono-directional connection is entirely possible at the designer's discretion. In the left side, there are four gray buttons. These buttons are designed to perform the DSE process automatically, details of this process will be described in the next Chapter. The function of each node and the overall relationship of this thesis are:

- VALIDATE: to execute the validation process on the platform to check if the input settings are correct. Note that before starting a DSE process, some parameters need to be set up from the designer (these parameters will be mentioned in the next Chapter). This button allows the evaluation if these input parameters are valid or not. If there is no error, the color of this button will turn to purple with "VALIDATED" status as in Figure 3.11. If not, there is a notification box of the problem/cause for the error. The button color is red with "INVALIDATED" status.

- GENERATE: to execute the generation process. Based on the platform model, this process will translate this model into programming language and include it in a DSE process. The programming language we use for the core of the DSE process is python. This process ends with a notification box and this button turns to purple with "GENERATED" status (Figure 3.11).

- RUN DSE: to execute the DSE process. The process finishes and then the button turns to purple with "DSE FINISHED" status (Figure 3.11). It means that an optimal solution has been found.

- SHOW: to show graphically the solution onto the workbench (Figure 3.11).

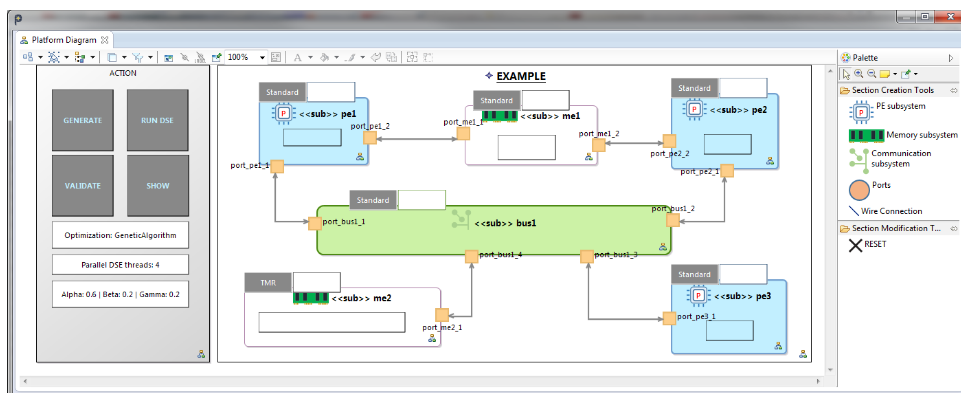


Figure 3.10: illustration of the workbench for the modeling of a 6-subsystem platform.

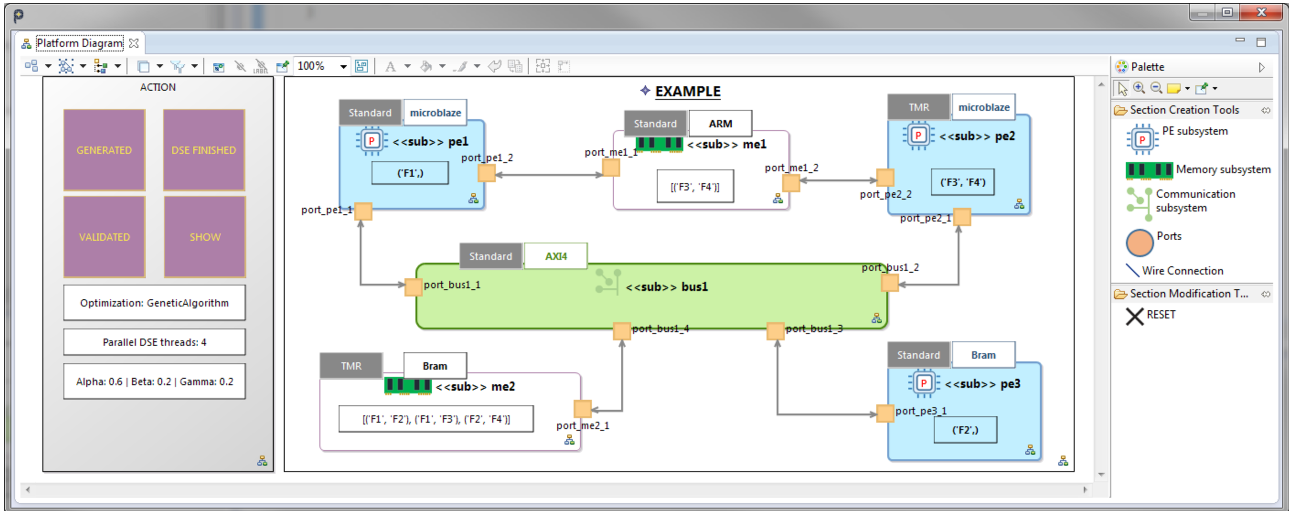


Figure 3.11: four big buttons in the left have changed from the gray to pink, meaning the DSE process is complete and the result has been found. The best solution is found shown in the workbench.

The white box under the four buttons indicates the search strategy used in the DSE process. In this tool, three search strategies can be used: Simulated Annealing, Genetic Algorithm, and Comprehensive Search. Details of these strategies will be presented in the next Chapter. Users can choose which search strategy option to apply to their DSE process before clicking the button VALIDATE. The search strategies can be executed in several parallel threads to increase the search speed (**Parallel DSE threads**). The number of threads depends on the capacity (core quantity) of the user's computer. The parameters **Alpha, Beta, Gamma** are used in the optimization process that will be explained in detail in the next Chapter.

When the button SHOW is clicked, the solution is updated to the graphical interface. The gray label on each subsystem icon indicates the used tolerance strategy. The white label on each subsystem icon indicates the component used in the subsystem. The subsystems **pe1**, **me2**, **bus1**, and **pe3** are **Standard**, which means that no tolerance strategy is applied onto these subsystems. The subsystems **pe2**, **me1** imply the **TMR** to reach the reliability requirement.

The name of functions of the considered application is indicated inside each PE subsystem. For example, the subsystem **pe1** executes a function named **F1**. The subsystem **pe2** executes two functions named **F3** and **F4**. The subsystem **pe3** executes a function named **F2**.

Data communications between functions on the application are mapped on the memory subsystems. The white box inside each memory subsystem indicates the data. For example, the data exchanged from **F3** to **F4**, denoted as $(F3, F4)$, is stored in the subsystem **me2**. $(F1, F2)$, $(F1, F3)$, and $(F2, F4)$ are stored in the subsystem **me1**.

The button RESET in the right palette allows resetting all from the beginning as shown in Figure 3.10 if users wants to perform a new DSE process.

By double-clicking into each subsystem icon on the Platform Diagram, the tool automatically generates and navigates into the corresponding Subsystem Diagram as shown in Figure 3.12. There are

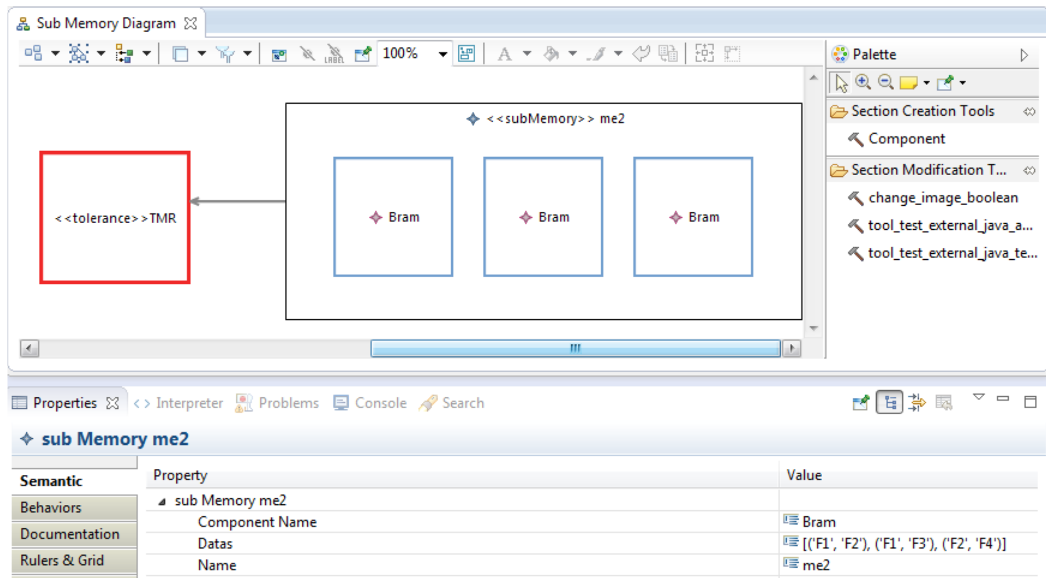


Figure 3.12: workbench inside a subsystem diagram; the centre is the Subsystem Diagram; the palette on the right is composed of the tools that allow to modify, customize the subsystem, the box in the bottom shows parameters of the Subsystem.

2 main boxes. The one with the title «subME» (or «subPE» or «subCOMM» of other types) describes the components inside the subsystem. The red one with the title «tolerance» describes which fault-tolerance strategy can be used for the subsystem.

Users can optionally customize the architecture within the subsystem according to their intents such as the number of components, the fault-tolerance strategy, or the input parameters of a component. The strategies available for selection are within the strategies defined in the meta-model. Similar to the Platform Diagram, the Subsystem Diagram also has a palette (in the right) that allows the creation of components. There are some input parameters of a components that needs to be defined such as *Name*, *Cost* to be used in the DSE process.

When the DSE process finishes, this tool automatically generates a statistics table as shown in Figure 3.13. The first column indicates the names of subsystems. Each line concerns a subsystem

new Table Platform

	Subsystem Tolerance	Subsystem cost	Platform reliability	Platform cost
sub Communication bus1	Standard	1.0		
sub Memory me1	Standard	1.0		
sub PEs pe1	Standard	1.0		
sub PEs pe2	TMR	4.0		
sub PEs pe3	Standard	1.0		
sub Memory me2	TMR	4.0		
platform example			0.6783925	12.0

Figure 3.13: table generated automatically in terms of fault tolerance and cost of the platform according to the best solution found in a DSE process.

of the platform. The last line of this table concerns the whole platform. The column "Subsystem Tolerance" indicates the fault tolerance strategy applied on the corresponding subsystems. The cost of each subsystem is displayed on the column: "Subsystem cost" (in terms of the number of used components). Finally, in the bottom line, the reliability and the cost overall of the platform are shown.

3.3 Summary

This chapter 3 presented the meta-model derived from ModES that allows building heterogeneous MPSoC platforms with fault-tolerance features. An intermediate level is created called "subsystem". Through the "subsystems", the fault tolerance parts are connected to the architectural part in the meta-model. The fault tolerance part is composed of fault-tolerance strategies. Finally, a supporting tool based on the proposed meta-model is introduced with its interface graphic workstation. We have built this tool on the Eclipse-based environment (Sirius) that allows to integrate the platform model into the DSE process. Moreover, this tool executes automatically the DSE process.

In that way, this chapter addresses the three following key points: 1) a meta-model is proposed to cover the deficiency of fault tolerance in literature; 2) the meta-model serves as a bridge between the different tools, between different programming languages, and different design stages, allowing designers to have a unified and coherent view of an MPSoC platform. 3) the meta-model, the fault model, and the tolerant strategies are the premises we use for the DSE process in the next chapter.

Chapter 4

Design space exploration

Abstract: In the previous chapter, we proposed a meta-model for heterogeneous MPSoC platform models, which is the core of our DSE framework. This chapter presents the details of the exploration process in the framework. Firstly, we figure out how to step into the design space; more clearly, what are the inputs of the process, how to generate a solution, and how to evaluate the solution. Secondly, an optimization process is presented to find the efficient solution according to reliability, cost, and execution time.

Contents

4.1 Design space **70**
4.1.1 Initialization 71
4.1.2 Mapping process 73
4.1.3 Solution evaluation 86
4.2 Optimization process **95**
4.2.1 Objective function 95
4.2.2 Search strategies 96
4.3 Summary **97**

4.1 Design space

This section provides the way to generate and explore a design solution. Figure 4.1 describes the design solution generator. The three main parts in this flow correspond to the subsections 4.1.1 (Initialization), 4.1.2 (Mapping process), and 4.1.3 (Solution evaluation process).

Initialization is the preparation step for a DSE process. In this step, an input application, a platform and available components are initialized with their parameters. The platform meta-model and the definitions we presented in the previous chapter are the basis of the architectural DSE process. From the input models, a mapping process is performed to find a mapping solution. Definition 4.1 gives necessary notions of a mapping process:

Definition 4.1. The **mapping** includes a set of rules, constraints to allocate resources and to map required functions and data on the resources.

Definition 4.1.1. A **rule** is a guide to define the relation between elements (application, platform) in an MPSoC system, that is strict and impossible to change in a specific DSE framework.

Definition 4.1.2. A **constraint** is a limitation for one or more features (quantity or quality) of an MPSoC system that can be changed or set by designers in a specific DSE framework.

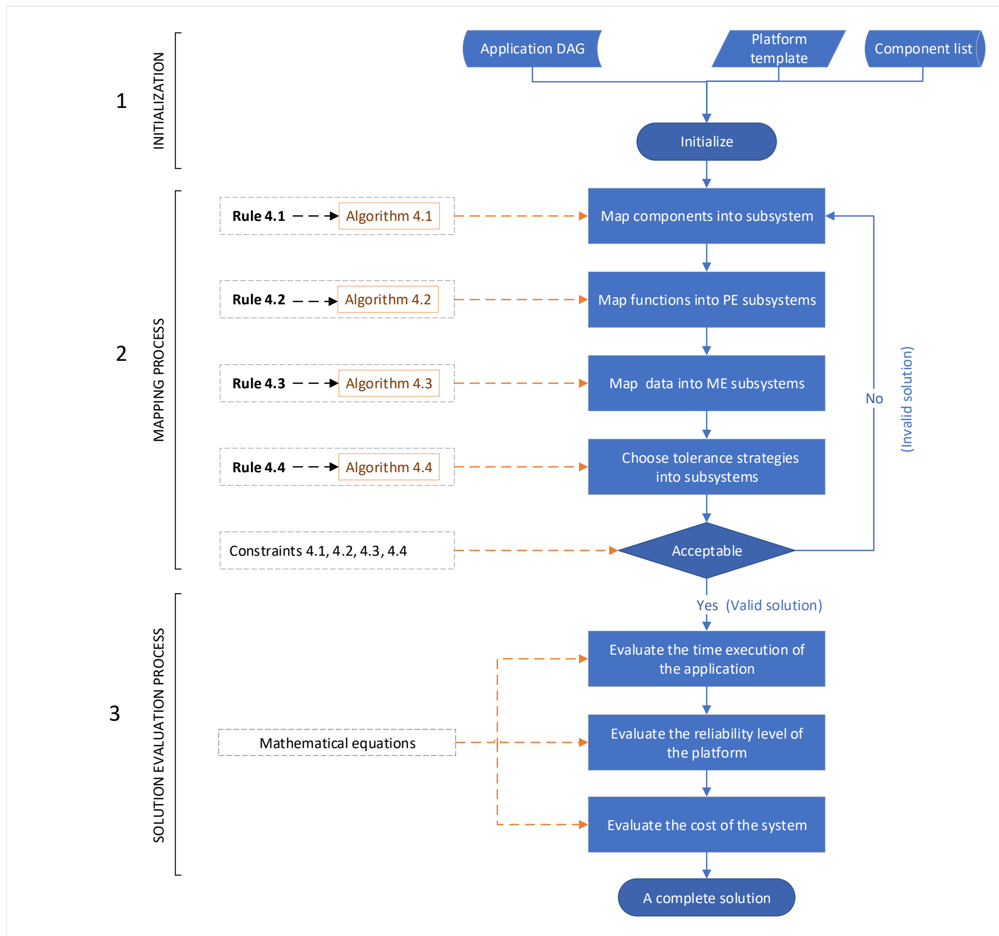


Figure 4.1: proposed design solution generator flow.

More specifically, without respect to constraints, a system may still be operated but does not produce the desired results or give wrong results. Besides, if rules are not respected then the design is considered as meaningless, cannot be operated.

As shown in Figure 4.1, in the second step (**Mapping process**), there are four rules (**Rule** 4.1 to 4.4). These rules are implemented as algorithms that allow creating a design in the most basic form without any constraints. Then, the design is verified under constraints (herein **Constraint** 4.1 to 4.4). If it satisfies the constraints (a valid solution), it is passed to the third step (**Solution evaluation process**); otherwise (an invalid solution), we need to go back to create another design. The rules and constraints are presented in the second Subsection.

Then, one solution would be to go through a set of assessments. The evaluation process (in Definition 4.2) allows designers to review the solution quantitatively and this is the basis for selecting the best solution in the next steps of the DSE.

Definition 4.2. The **solution evaluation** is composed of mathematical models, equations, and algorithms to quantify the performance of the considered solutions. The data used for the calculations are derived from the available input parameters.

In that way, this solution-generator process is carried out repeatedly helping the designer to reach a solution point in the design space.

4.1.1 Initialization

The models of application, platform, and components are used as input to this process. An **application** is modeled as a DAG as defined by Definition 1.1. As reviewed in Chapter 2, the DAG is used in a lot of studies due to its ability to model different application structures and application types. In addition, there are other variants based on the DAG core such as directed graphs, Annotated Directed Acyclic Graph (ADAG), data dependencies task graphs, etc. Thus, for reasons of a wider compatibility and ease of expansion, the DAG is selected for application modeling in this thesis. Listing 4.1 gives an **instance** of the model building for the application in Python. The object *Function* owns two attributes: *_name* and *_size*. The object *Data* is exchanged between functions on the edges. Designers can build an application with two methods *add_funcNode* (create a function) and *add_edge* (create the connection between functions).

```
1  class Function(object):
2    _name = "init"
3    _size = 0 #number of operation
4  class Data(object):
5    _name = "init"
6    _size = 0 #bytes
7  class App(object):
8    def __init__(self):
9      self.func_list = []
10     self.edges = collections.defaultdict(list)
11     self.data_list = {}
12   def add_funcNode(self, func):
13     self.func_list.append(func)
14   def add_edge(self, from_function, to_function, dataBlock):
15     self.edges[from_function].append(to_function)
16     self.data[(from_function, to_function), dataBlock._ID] = dataBlock #modeled by
       object Data
```

Listing 4.1: Application model building in Python code

If you only consider the mapping of functions of a given application to a fixed available platform, the solution space is really small. And so the discovery of new platform designs is also very limited.

For example, an application with 3 functions; and a platform with 3 DPE subsystems. One or more functions can be mapped on a DPE subsystem. The different order of the functions on the same sub-system will also give a different solution. As such, we have a maximum of 60 solutions ($C_3^1 \times P_3^3 + 3 \times C_3^1 \times C_2^1 \times P_2^2 + C_3^1 \times C_2^1 \times C_1^1$) where C_i^j is j-combinations from a given set of i elements and P_i^j is j-permutations from a given set of i elements. However, at the positions of the DPE subsystems, if we can choose either DPE or PPE, the number of solutions will be $60 \times 3^3 = 1620$. Thus, to expand the design space, an input platform is considered as a **template** defined in Definition 4.3.

Definition 4.3. A **platform template** is a platform model that designers consider but where the subsystems are empty. Each of these empty subsystems is only defined by its type and wait to be filled by components.

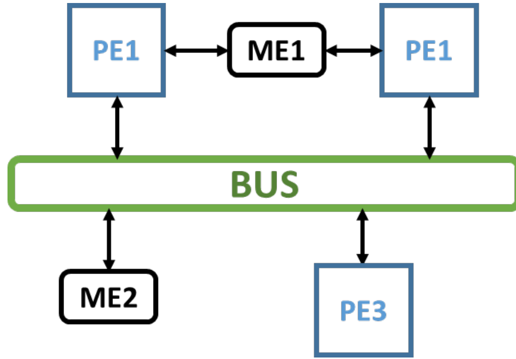
Definition 4.3 means a platform template includes ME subsystems, COMM subsystems, and PE subsystems. A PE subsystem can be specified as DPE or PPE depending on which component type is assigned to the PE subsystem. This also helps to generate more solutions than predetermining strictly PPE subsystems and DPE subsystems. A **component** is modeled according to the platform meta-model defined in the previous Chapter with all its pre-defined parameters.

Definition 4.4. All **components** that can be provided for use during design are on a list of components, called: component list **cL**. $cL = \{compE_1, compE_2, ...compE_k | k \in \mathbb{N}\}$.

Figure 4.2a presents an example of a platform template. There are 6 empty subsystems. We can select component elements from a component list **cL** (Figure 4.2b) to fill this template and create a finished platform. Each line in **cL** presents a component and its input parameters such as:

- *type*: type of component;
- $\lambda_{\mathrm{comp}_{PF}}$ and $\lambda_{\mathrm{comp}_{TF}}$): failure rates by permanent and transient faults;
- *capacity*: number of functions can be executed of a $compDPE$ or a $compPPE$, memory size of a $compMemory$;
- $delay/Speed$: speed factor for a $compDPE$ or a $compPPE$ (number of operations per second), delay of a $compMemory$ or a $compCommunication$ (ms);
- *cost*.

Listing 4.2 presents an instance of a model building for the platform template in Python. An object *Subsystem* is characterized by three attributes. 1) *_ID* is the identifier of a subsystem: 2) *_type* is the type of subsystem (PE, ME, $COMM$) and must be declared at the time of creation of the subsystem. and 3) *assigned_comp* is initialized as *None* (empty subsystem) but will be filled with a pre-defined object *Component* during the mapping process. A platform template owns two methods 1) *add_subNode* (add a new subsystem into the template) and 2) *add_connect* (create a new connection between subsystems). There is a small note here that the connection delay between two subsystems can be weighted by a variable *distance*.



a) a platform template.

Component list

ID	Type	λ_{PF}	λ_{TF}	Capacity	Delay/speed	Cost
dpeC1	DPE	150	185	1 function	speed_factor: 10^10 op/s	1
ppeC1	PPE	200	250	Multi functions	speed_factor: 10^9 op/s	1
dpeC2	DPE	100	125	1 function	Speed_factor: 10^8 op/s	1
busC1	COMM	100	180	N/A	Delay: 0,1 ms	1
meC1	ME	14	33	3KBytes	Delay: 0,5 ms	1
meC2	ME	30	92	6Kbytes	Delay: 0,5 ms	1
...				...	...	

b) a list of component.

Figure 4.2: a platform template and a component list prepared before a DSE process.

```
1  class Subsystem(object):
2    _ID = "init"
3    _type = 0     # 1 : PE
4                  # 2 : ME
5                  # 3 : COMM
6    assigned_comp = None # object component
7
8
9  class Platform_template(object):
10   def __init__(self):
11     self.sub_list = []
12     self.connections = collections.defaultdict(list)
13     self.distances = {}
14
15   def add_subNode(self, subsystem):
16     self.sub_list.append(subsystem)
17
18   def add_connect(self, _from_subID, _to_subID):
19     self.connections[_from_subID].append(_to_subID)
20     self.distances[(_from_subID, _to_subID)] = 0
```

Listing 4.2: platform-template model building in Python code

4.1.2 Mapping process

This subsection shows how a valid solution is built, which corresponds to the second step in Figure 4.1. Firstly, the problem is standardized under an ILP formulation. Secondly, a platform solution is established based on fixed rules. And finnally, a set of constraints is presented to prune the design space and to eliminate invalid solutions.

4.1.2.1 Integer Linear Programming formulation

From the definitions, the mapping problem can be transformed into an integer linear programming problem, they are formulated by the four types of ILP variables: A mapping solution is characterized by 4 points: component, function, data, fault tolerance strategy. Parameters and variables used in the ILP formulation are shown in Table 4.1.

Table 4.1: Description and notation of parameters and variables used in ILP formulation.

Name	**Type**	**Definition**
V		The set of function in the given application
n_{func}	Constant	The number of functions in the given application
F_j		The function j of the given application
$F_j.size$	Constant	The size of F_j (operations)
E		The set of edges in the application
e_{j_k}		The directed edge from F_j to F_k in the given application
D		The set of data in the application
n_{data}	Constant	The number of data blocks in the application
d_{j_k}		The data block from F_j to F_k in the given application
$d_{j_k}.size$		The size of d_{j_k} (bytes)
$subL$		The set of subsystems in the given platform
n_{sub}	Constant	The number of subsystem in the given platform
sub_i		The subsystem i in the given platform
$sub_i.type$		The type of sub_i DPE, PPE, PE, ME, COMM PE = {DPE, PPE}
V_{sub_i}		The set of functions mapped in the subsystem i
D_{sub_i}		The set of data mapped in the subsystem i
cL		The set of available components in library (component list)
n_{compE}	Constant	The number of elements in the component list
$compE_j$		The component element j in cL
$compE_j.type$		The type of $compE_j$ (DPE, PPE, ME, COMM)
$compE_j.speed_factor$	Constant	The speed factor of the PE component $compE_j$ (operations/s)
$compE_j.\lambda_{\text{comp}_{PF}}$	Constant	The failure rate of permanent faults of $compE_j$ (FIT)
$compE_j.\lambda_{\text{comp}_{PF}}$	Constant	The failure rate of transient faults of $compE_j$ (FIT)
$compE_j.capacity$	Constant	The capacity of $compE_j$
		For PE components, the maximum number of operations that the component can perform (operations)
		For ME components, the maximum size of the component (bytes)
$compE_j.delay$	Constant	The delay to access $compE_j$
		For ME components, the delay of the read/write access (seconds)
		For COMM components, the delay to transfer a data block (seconds)
$compE_j.quantity$	Constant	The maximum quantity of $compE_j$ is available for use
$cM_{sub_i}^{compE_j}$	Variable	$compE_j$ is allocated to sub_i
$fM_{F_j}^{sub_i}$	Variable	F_j is mapped to sub_i
$dM_{d_{j_k}}^{sub_i}$	Variable	d_{j_k} is mapped to sub_i
$tM_{sub_i}^{k}$	Variable	sub_i supports the fault-tolerance strategy k

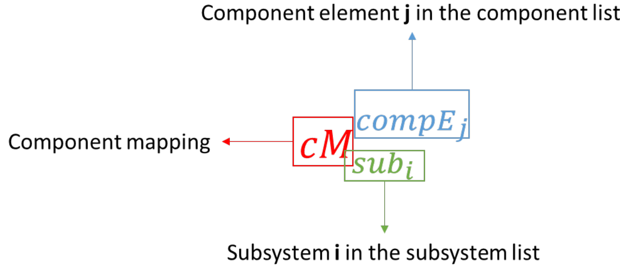


Figure 4.3: Description of $cM_{sub_i}^{compE_j}$.

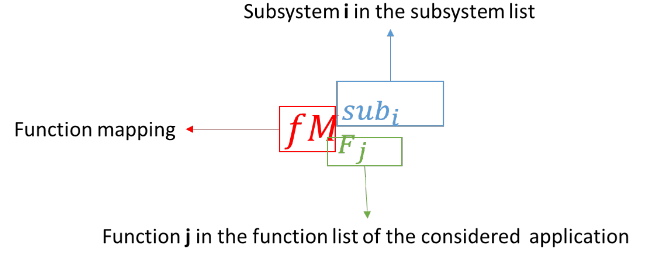


Figure 4.4: Description of $fM_{F_j}^{sub_i}$.

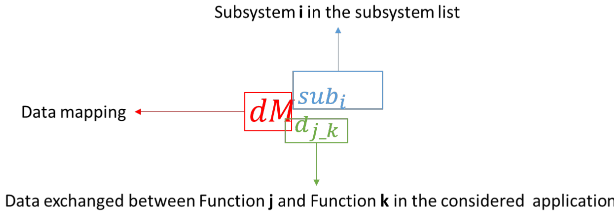


Figure 4.5: Description of $dM_{d_{j_k}}^{sub_i}$.

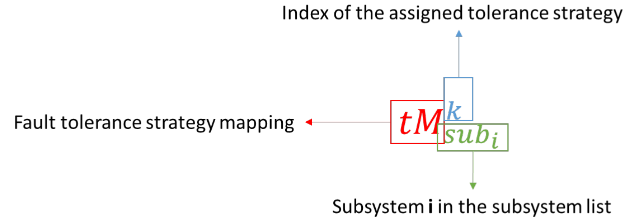


Figure 4.6: Description of $tM_{sub_i}^{k}$.

- $cM_{sub_i}^{compE_j}$ represents a component mapping in terms that a component element is chosen to be mapped on a subsystem, where: $compE_j \in cL$, $sub_i \in subL$, $i, j \in \mathbb{N}$. Figure 4.3 visually describes the notion of $cM_{sub_i}^{compE_j}$;
- $fM_{F_j}^{sub_i}$ represents a function mapping in terms that a PE subsystem handle a function, where: $sub_i \in subL$, $F_j \in V$, $i, j \in \mathbb{N}$. Figure 4.4 visually describes the notion of $fM_{F_j}^{sub_i}$;
- $dM_{d_{j_k}}^{sub_i}$ represents a data mapping in terms that a ME subsystem stores a data block, where: $sub_i \in subL$, $d_{j_k} \in D$, $i, j, k \in \mathbb{N}$ Figure 4.5 visually describes the notion of $dM_{d_{j_k}}^{sub_i}$;
- $tM_{sub_i}^{k}$ represents a tolerance mapping in terms that a fault tolerance strategy is chosen for a subsystem $\big(k \in \mathbb{N}\big)$. In the scope of this thesis, we consider: 0 - no tolerance, 1 - TMR, 2 - TReR. So, we have $k \in \{0, 1, 2\}$. Figure 4.6 visually describes the notion of $tM_{sub_i}^{k}$.

As we can see, each instance of an above variable type is created by combining the input elements of the **Initialization** step. For example, $fM_{F_2}^{sub_3}$ is the combination of the subsystem 3 with the function 2. Clearly, an instance of the complete mapping (x_{sol}) is represented by a set of 4 vectors as in Equation 4.1 where:

- the first vector represents the component mapping that is composed of n_{sub} variables;
- the second vector represents the function mapping that is composed of n_{func} variables;
- the third vector represents the data mapping that is composed of n_{data} variables;

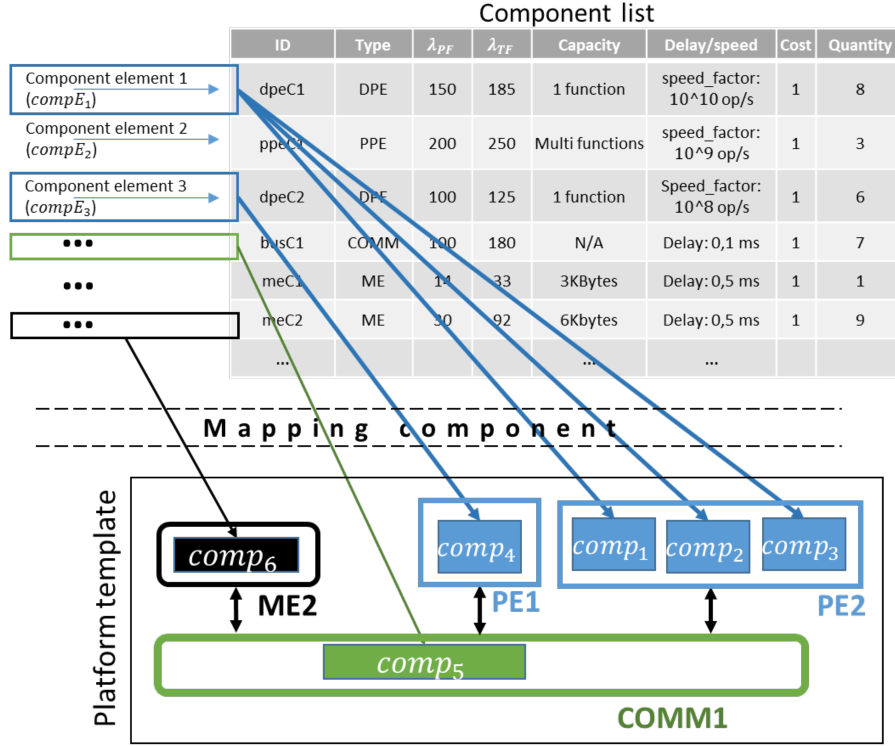


Figure 4.7: Illustration of component list, platform template, and component mapping.

- the fourth vector represents the tolerance strategy mapping that is composed of n_{sub} variables;
- $n_{\text{sub}}, n_{\text{func}}, n_{\text{data}}, n_{\text{sub}} \in \mathbb{N}$; $p_1, p_2, ...p' \in \{1, 2, ...n_{\text{compE}}\}$;
  $i_1, i_2, ...i' \in \{1, 2, ...n_{\text{sub}}\}$; $j_1, j_2, ...j' \in \{1, 2, ...n_{\text{sub}}\}$;
  $k_1, k_2, ...k' \in \{0, 1, 2\}$; $d_1, d_2, ...d_{n_{data}} \in D$.

Each variable in x_{sol} has value "1". Any other variable is created by the combination that not listed in x_{sol} has a value "0".

$$
\begin{aligned}
x_{\text{sol}} = \Big\{ & \{cM_{sub_1}^{compE_{p1}}, cM_{sub_2}^{compE_{p2}}, ...cM_{sub_{n_{\text{sub}}}}^{compE_{p'}}\}, \\
& \{fM_{F_1}^{sub_{i_1}}, fM_{F_2}^{sub_{i_2}}, ...fM_{F_{n_{\text{func}}}}^{sub_{i'}}\}, \\
& \{dM_{d_1}^{sub_{j_1}}, dM_{d_2}^{sub_{j_2}}, ...dM_{d_{n_{data}}}^{sub_{j'}}\}, \\
& \{tM_{sub_1}^{k_1}, tM_{sub_2}^{k_2}, ...tM_{sub_{n_{\text{sub}}}}^{k'}\}\Big\}
\end{aligned} \tag{4.1}
$$

Figure 4.7 shows the relation between a component, a component list and a platform. In the platform, a **component** is a real and unique entity, denoted as $comp_i$ - the component i in the platform. There may be many identical components in the platform. These components are selected from the component list (through the mapping process), in other words, $comp_i$ is an instance of $compE_j$. Each component in the platform has a reference from a component element in the component list. Each component element i in the list, denoted as $compE_i$, describes the properties of a specific components such as *ID*,

Type, *Failure rate*, etc. Thus, it must not have 2 identical *compE* in component list, as described in Equation 4.2.

$$compE_i = compE_j \ \forall compE_i, compE_i \in cL \text{ if and only if } i = j. \tag{4.2}$$

4.1.2.2 Rules

As presented in Definition 4.1, the rules allow to generate a mapping solution for the platform. Hence, these rules must be fully coherent with the proposed meta-model. In our DSE framework, they are strict.

The structure of each rule is presented in the following 3 points:

- content of the rule;
- description in ILP formulation;
- algorithm implementation.

Rule 4.1. *(Component) (1) A subsystem is only composed of one hardware-component element from the available component list which corresponds to the type of the subsystem. (2) All empty subsystems must be filled by component element of the component list. (3) A PPE component elements is always accompanied by a SOFT component elements.*

Rule 4.1 provides an instruction for filling empty subsystems within the platform template. The combination of different components in the component list can be created on different platforms. Equation 4.3 corresponds to the first sentence in Rule 4.1 ($compE_j \in cL$, $sub_i \in subL$, $i, j \in \mathbb{N}$):

$$cM_{sub_i}^{compE_j} = \begin{cases} 1, & compE_j \text{ is mapped on (selected for) } sub_i \\ & \text{and } compE_j.type \in sub_i.type \\ 0, & \text{otherwise} \end{cases} \tag{4.3}$$

Obviously, sub_i can be assigned exactly only one hardware component element of the component list, as defined in Equation 4.4:

$$\sum_{compE_j \in cL \wedge \text{ hardware}} cM_{sub_i}^{compE_j} = 1 \tag{4.4}$$

Synthesizing from Equation 4.3, 4.4 and Rule 4.1 is that there are two component elements on the same subsystem if and only if they are 1 PPE and 1 SOFT.

$$\sum_{compE_j \in cL} cM_{sub_i}^{compE_j} \leqslant 2 \tag{4.5}$$

with Equation 4.6,

$$\begin{aligned} &\Big(\sum_{compE_j \in cL} cM_{sub_i}^{compE_j} = 2 \\ \iff &\exists! \ compE_j, \ cM_{sub_i}^{compE_j} = 1 \wedge compE_j.type = PPE \\ &\text{and } \exists! \ compE_k, \ cM_{sub_i}^{compE_k} = 1 \wedge compE_k.type = SOFT \Big) \end{aligned} \tag{4.6}$$

Algorithm 4.1 deploys Rule 4.1. Firstly, we select a subsystem from the subsystem list of the given platform template (line 1-2). Then, a component element (*selected_compE*) is selected from the component list (**cL**) (line 3). If the selected component element is the same type of the selected subsystem, the component element is allocated to the subsystem. Otherwise, the component-element selection is repeated until the component type matches the subsystem type (line 4-6). If the *selected_compE* is PPE, a SOFT component element must be selected to accompany the PPE component (line 8-14). The whole process is repeated until all subsystems are filled by a component. *method.choice* is the method to select an element from a list/set. This method depends on the exploration strategy of designers such as random, sequential, hybrid etc. Finally, the *platform.template* is filled by components, and from now on is can be used as a *platform* (line 15).

Algorithm 4.1 map a component into a subsystem

Require: *platform_template*, *component_list*
Ensure: *platform_template.filled* = *TRUE* (is mapped components)

//Repeat this algorithm until all subsystems are filled by components

1: *subs_list* ← *platform_template.subsystem_list*;
2: *selected_sub* ← *method.choice*(*subs_list*);
//Select a subsystem in the subsystem list of the platform template
3: *selected_compE* ← *method.choice*(*component_list*); //Select a component element of the component list
4: **while** *selected_compE.type* **not in** *selected_sub.type* **do**
5:    *selected_compE* ← *method.choice*(*component_list*);
6: **end while**
7: *selected_sub.assigned_comp* ← *selected_compE*;
8: **if** *selected_compE* = "PPE" **then**
9:    *selected_softE* ← *method.choice*(*comp_list*);
   //Look for a SOFT component element from the component list to map onto the selected PPE component
10:    **while** *selected_softE.type* ≠ "SOFT" **do**
11:       *selected_softE* ← *method.choice*(*comp_list*);
12:    **end while**
13:    *selected_sub.assigned_comp.assigned_soft* ← *selected_softE*;
14: **end if**
15: *platform* ← *platform_template*

Rule 4.2. *(Function) (1) A function is mapped on only one PE subsystem. (2) But a PE subsystem can handle several functions. (3) In a full mapping, every functions must be mapped.*

Rule 4.2 provides an instruction to map the functions onto the executive subsystem (PE subsystems) and ensure that all functions are mapped. The first sentence in Rule 4.1 is formulated in ILP as Equation 4.7:

$$fM_{F_j}^{sub_i} = \begin{cases} 1, & F_j \text{ is mapped on } sub_i \wedge sub_i.type = PE \\ 0, & \text{otherwise} \end{cases} \tag{4.7}$$

Obviously, a function is only mapped to a maximum of 1 PE subsystem (this equation goes with the

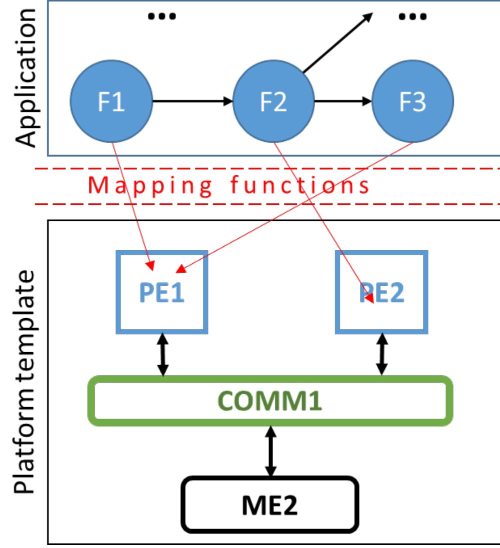


Figure 4.8: Application graph, platform, and function mapping.

second and the third sentences of Rule 4.2):

$$\sum_{i=1}^{|subL|} fM_{F_j}^{sub_i} = 1 \tag{4.8}$$

Algorithm 4.2 describes the mapping of a function to a PE subsystem. A function is selected from the list of functions in the given application (line 1-2). Then we select a PE subsystem from the platform (line 3-7). If the selected function is already mapped to the selected subsystem (line 15-16), the selected will be removed from the selected subsystem and wait to be mapped on another subsystem. If the selected function is not mapped to the selected subsystem (line 8), we will check whether this function has been mapped on any other subsystem (line 9). If yes, the function is removed from the current PE subsystem (line 10-12) and we map the selected function to the selected subsystem (line 14). As with Rule 4.1, *method.choice* is set up by the designer's DSE strategy. Finally, the whole process is repeated until all functions are mapped. The *remove* actions (line 11 and line 16) are to ensure that a function is not mapped on two different PE subsystems. Moreover, this also helps that the two solutions found consecutively are not the same if the generation process is a chain of inheritance. A chain of inheritance is a chain that starts with a solution, new solutions are derived from the original initialization and grows up as a tree. The *remove* actions maybe not necessary if *method.choice* ensures that each function is only checked only once time.

Figure 4.8 illustrates a mapping process. F_1 and F_3 are mapped onto the subsystem $PE1$ and F_2 is mapped onto $PE2$. Algorithm 4.2 allows to map all functions of the given application to subsystems on the platform.

The execution order of functions on the same PE depends on the order in which the functions are mapped onto that PE. For example, in Figure 4.8, F_1 is mapped onto $PE1$ before the mapping of F_3, thus, the execution order in $PE1$ is $F_1 \rightarrow F_3$. However, if F_3 is mapped before F_1, the order of execution is $F_3 \rightarrow F_1$. Of course, the second order could cause errors and is an invalid solution.

Algorithm 4.2 map a function into a PE subsystem

Require: $platform$, app_graph
Ensure: $app_graph.function_list.mapped = TRUE$ (all functions are mapped)
//Repeat this algorithm until all functions are mapped

```
 1: func_list ← app_graph.function_list; //Get the list of all functions
 2: selected_func ← method.choice(func_list); //Select a function in the application
 3: subs_list ← platform.subsystem_list;
 4: selected_sub ← method.choice(subs_list);
    //Select a PE subsystem in the subsystem list of the platform
 5: while selected_sub.type ≠ "PE" do
 6:    selected_sub ← method.choice(subs_list);
 7: end while
 8: if selected_func not in selected_sub.assigned_funcs then
 9:    for sub in subs_list do
10:       if selected_func in sub.assigned_funcs then
11:          sub.assigned_funcs.remove(selected_func);
             //Remove the selected function in the current PE subsystem
12:       end if
13:    end for
14:    selected_sub.assigned_funcs.add(selected_func);
15: else
16:    selected_sub.assigned_funcs.remove(selected_func);
17: end if
```

Algorithm 4.3 map a data block into a Memory subsystem

Require: $platform$, app_graph
Ensure: $app_graph.data_list.mapped = TRUE$ (all data blocks are mapped)

```
    //Repeat this algorithm until all data blocks are mapped
 1: da_list ← app_graph.data_list //Get the list of all functions
 2: for data in da_list do
 3:   if producer(data), consumer(data) are mapped on the same PE subsystem then
 4:     if producer(data), consumer(data) are executed sequentially and consecutively on the PE subsystem
        then
 5:       da_list.remove(data); //There is no need to consider the mapping of this data block
 6:     end if
 7:   end if
 8: end for
 9: selected_data ← method.choice(da_list); //Select a data block in the application
10: subs_list ← platform.subsystem_list;
11: selected_sub ← method.choice(subs_list);
    //Select a ME subsystem in the subsystem list of the platform
12: while selected_sub.type ≠ "ME" do
13:   selected_sub ← method.choice(subs_list);
14: end while
15: if selected_data not in selected_sub.assigned_data then
16:   for sub in subs_list do
17:     if selected_data in sub.assigned_data then
18:       sub.assigned_data.remove(selected_data); //Remove the selected data block from the cur-
          rent ME subsystem
19:     end if
20:   end for
21:   selected_sub.assigned_data.add(selected_data);
22: else
23:   selected_sub.assigned_data.remove(selected_data);
24: end if
```

Rule 4.3. *(Data) (1) A data block is stored on only one ME subsystem. (2) But an ME subsystem can store several data blocks. (3) A data block is not stored on any ME subsystem if it satisfies simultaneously the following conditions: 1/ two functions that exchange the data block are mapped on the same PE subsystem; 2/ the order to execute the two functions on the PE subsystem is the same as the execution order described on the application DAG; 3/ the two functions are executed consecutively on that PE subsystem. (4) If all of the above conditions are not satisfied, a data block must be stored on an ME subsystem.*

Rule 4.3 provides a guideline to allocate the data blocks onto the memory subsystem. Equation 4.9 formulates the first sentence of Rule 4.3:

$$dM_{d_{j_k}}^{sub_i} = \begin{cases} 1, & \text{the data block from } F_j \text{ to } F_k \text{ is stored in } sub_i \wedge sub_i.type = ME \\ 0, & \text{otherwise} \end{cases} \tag{4.9}$$

So, a data block is only stored on a maximum of 1 ME subsystem and it does not always need to be stored in 1 ME subsystem (based on all sentences of Rule 4.3), what gives Equation 4.10:

$$\sum_{i=1}^{|subL|} dM_{d_{j_k}}^{sub_i} \begin{cases} = 0, & fM_{F_j}^{sub_m} = fM_{F_k}^{sub_m} = 1, \text{and } F_k \text{ is executed immediately after } F_j \\ = 1, & otherwise. \end{cases} \tag{4.10}$$

Algorithm 4.3 describes a way to implement Rule 4.3. Firstly, a data block is selected among data blocks of the given application (line 1-2). Then, an ME subsystem is selected from the subsystem list of the platform (line 3). Next, a process of checking whether any blocks of data do not need to be mapped (from line 2 to line 8). $producer(data)$ and $consumer(data)$ are respectively the functions of which data block $data$ is the input and the output. The lines from 9 to 23 are similar to the function-mapping process, we remove the selected data block from its current ME subsystem. This is to ensure that a data block is not allocated to two different ME subsystems. The whole process is repeated until all data blocks are mapped to ME subsystems.

Rule 4.4. *(Tolerance strategy) (1) A subsystem can support a fault-tolerance strategy which is available in the library for its type. (2) For a specific architecture solution, a subsystem can apply only one tolerance strategy.*

Rule 4.4 provides an instruction to assign a fault-tolerance strategy to a subsystem. Equation 4.11 describes the first sentence of Rule 4.4, where $k \in \{0, 1, 2\}$.

$$tM_{sub_i}^{k} = \begin{cases} 1, & sub_i \text{ support the fault tolerance strategy } k \\ 0, & otherwise. \end{cases} \tag{4.11}$$

In the scope of this thesis, we consider the value of k: 0 – no tolerance, 1 – TMR, 2 – TReR. This needs to be updated if we add others fault-tolerance strategies. The strategy with the index "0" and "1" can be used for all type of subsystem, "2" is only used for PE subsystems. Each subsystem just applies only one fault-tolerance strategy, described in Equation 4.12:

$$\forall i < ||subL|| \ \sum_{k} (tM_{sub_i}^{k}) = 1 \tag{4.12}$$

Algorithm 4.4 describes an implementation of the rule. We select suitable strategies for each selected subsystem. *method.choice* (line 2) chooses a subsystem for the mapping. Then, if the subsystem is PE, it may be applied a fault-tolerance strategy (TMR or TReR) or no tolerance (line 3-4). Otherwise, the other type of subsystem my be applied TMR or no tolerance (line 6). The process stops when all subsytems are checked.

Algorithm 4.4 select a fault tolerance strategy for a subsystem

Require: $platform$

Ensure: $platform.tolerance = TRUE$ (is set fault tolerance)

    //Repeat this algorithm until all subsystems are checked
1: $subs_list \leftarrow platform.subsystem_list$;
    //Select a subsystem in the subsystem list of the platform
2: $selected_sub \leftarrow method.choice(subs_list)$;
3: **if** $selected_sub =$ "PE" **then**
4: $\quad selected_sub.tolerance_strategy \leftarrow method.choice(0, 1, 2)$
5: **else**
6: $\quad selected_sub.tolerance_strategy \leftarrow method.choice(0, 1)$
7: **end if**

4.1.2.3 Constraints

When we have generated a mapping solution, some constraints point out whether this solution is valid or not. Constraints are less mandatory than the rules and can be changed and replaced depending on the purposes of designers such as limitation of resource, technology, application time. These constraints can be applied during the mapping but we put them separately for better understanding because they depend on the purpose of designers so that our framework users just need to change the constraint here without changing the algorithms introduced above. In this thesis, we use the four following constraints (Constraint 4.1, 4.2, 4.3, 4.4) to identify a valid solution.

Constraint 4.1. *(Capacity) the total size of functions is mapped to a PE subsystem must not exceed the capability of the PE component element of that subsystem.*

Constraint 4.1 is expressed by Equation 4.13, where $sub_i \in subL$, $i, j, k \in \mathbb{N}$; $F_j.size$ is the size of F_j (operation); ($compE_k.capacity$) is the capacity of a PE component of $compE_k$, as following:

$$\sum_{F_j \in V} (fM^{sub_i}_{F_j} \times F_j.size) \leq \sum_{compE_k \in cL} (cM^{compE_k}_{sub_i} \times compE_k.capacity) \tag{4.13}$$

Constraint 4.2. *(Size) the total size of data blocks is mapped to an ME subsystem must not exceed the capacity of the ME component element of that subsystem.*

Constraint 4.2 is expressed by Equation 4.14, where $sub_i \in subL$, $i, j, k, q \in \mathbb{N}$; $d_{j_k}.size$ is the size of d_{j_k}; ($compE_q.capacity$) is the capacity of an ME component of $compE_q$, as following:

$$\sum_{d_{j_k} \in D} dM^{sub_i}_{d_{j_k}} \times d_{j_k}.size \leq \sum_{compE_q \in cL} (cM^{compE_q}_{sub_i} \times compE_q.capacity) \tag{4.14}$$

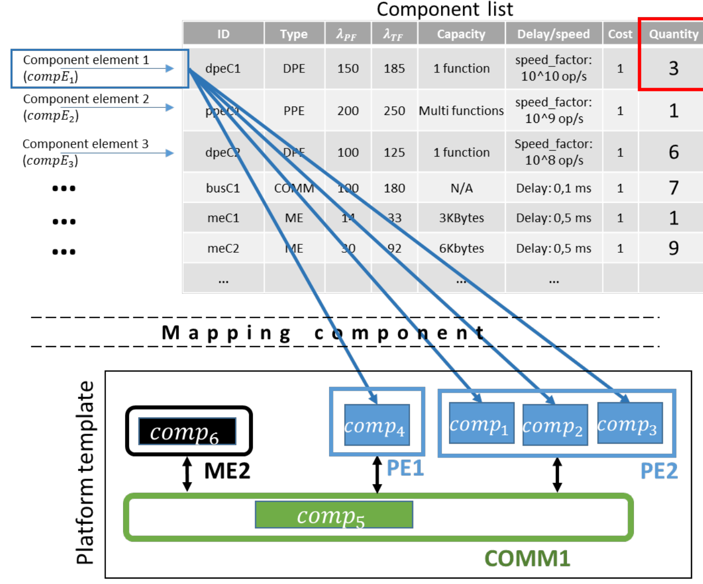


Figure 4.9: An example of the component mapping does not respect Constraint 4.3.

Constraint 4.3. *(Quantity) the number of components used on the entire platform must not exceed the pre-indicated "quantity" of the corresponding component element in the component list. (This is the resource limitation).*

Constraint 4.3 is expressed by Equation 4.15, where $(compE_j.quantity)$ is the maximum number of the type $compE_j$ indicated in the component list; $(\text{number_of_used_components}_{sub_i}^{compE_j})$ is the number of used components of the type $compE_j$ in the subsystem sub_i, with $sub_i \in subL$, $i, j \in \mathbb{N}$, as following:

$$\sum_{sub_i \in subL} (cM_{sub_i}^{compE_j}) \times (\text{number_of_used_components}_{sub_i}^{compE_j}) \leq compE_j.quantity \tag{4.15}$$

To be clearer, for example in Figure 4.9, $(compE_3.number) = 6$, designers can only use up to 6 components (of the type $compE_3$) on the entire platform. $(\text{number_of_used_components}_{sub_i}^{compE_j})$ depends on the fault tolerance strategy applied on the subsystem sub_i. Figure 4.9 shows an example that does not respect Constraint 4.3. The predefined *quantity* of the component element 1 is 3 (as defined in the red square) but 4 components of type $compE_1$ are used on the whole platform.

Constraint 4.4. *(Path) a PE subsystem can only send data to or receive data from an ME subsystem or another PE subsystem if and only if there is a direct link between them or a path via COMM subsystems.*

For example, when a function F_j is mapped on a subsystem sub_{i_1}, an input (or output) data of F_j is mapped on sub_{i_2} $(j \in \{1, 2, ...n_{\text{func}}\}$; $i_1, i_2 \in \{1, 2, ...n_{\text{sub}}\})$. It needs to have a valid path between sub_{i_1} and sub_{i_2} to allow the data transfer for F_j. To evaluate this, we consider each subsystem on the platform template as a node and the platform template is a graph of connected nodes. The Dijkstra's algorithm [119] is used to find the shortest path between 2 subsystems on a platform. The shortest path (called $Dijk_path(platform_template, sub_{i_1}, sub_{i_2})$) that is a list of nodes in order from sub_{i_1} to sub_{i_2}

does not include the start node (sub_{i_1}) and the end node (sub_{i_2}). Then, we apply the Constraint 4.4 to calculate the real distance between sub_i and sub_k: $real_distance(sub_{i_1}, sub_{i_2})$.

Algorithm 4.5 describes how to calculate the **real distance**. Firstly, we obtain the chain of nodes that is the shortest path between sub_{i_1} and sub_{i_2} by method $Dijk_path$ (line 1). If the number of nodes in the path is infinite (line 2), it means that there is no connection between the two subsystems. If there is no node in the path (line 4), it means that 2 subsystems are connected directly. So, the **real distance** between 2 subsystems is 0 (line 5). Otherwise, we consider each node in the path (line 6-14). If the path passes through a COMM subsystem (line 8), the distance is increased by 1 unit (line 9). If there is any node of other types (PE, ME) (line 10), the connection is blocked at that node and the distance is also considered infinite (line 11). Finally, we can obtain a value of $real_distance(sub_{i_1}, sub_{i_2})$.

So, we can see that $real_distance(sub_{i_1}, sub_{i_2})$ will always come with a $real_path(sub_{i_1}, sub_{i_2})$; where $real_path(sub_{i_1}, sub_{i_2})$ is a set containing a sequence of subsystems from sub_{i_1} to sub_{i_2}.

Thus, Constraint 4.4 is formulated in ILP by Equation 4.16, where $j, k \in \{1, 2, ...n_{\text{func}}\}$; $i_1, i_2 \in \{1, 2, ...n_{\text{sub}}\}$:

$$\left[\begin{array}{lcl} real_distance(sub_{i_1}, sub_{i_2}) \times dM_{d_{j_k}}^{sub_{i_1}} \times fM_{F_k}^{sub_{i_2}} & < & \infty \\ real_distance(sub_{i_1}, sub_{i_2}) \times fM_{F_j}^{sub_{i_1}} \times dM_{d_{j_k}}^{sub_{i_2}} & < & \infty \end{array}\right. \tag{4.16}$$

Algorithm 4.5 distance between 2 subsystems

Require: $platform_template$, sub_{i_1}, sub_{i_2}
Ensure: $real_distance(sub_{i_1}, sub_{i_2})$

```
1: path = Dijk_path(platform_template, sub_i1, sub_i2);
   //the shortest path between sub_i1 and sub_i2
   //list of nodes in order from sub_i1 to sub_i2 real_distance(sub_i1, sub_i2) = 0
2: if path.length = ∞ then
3:    real_distance(sub_i1, sub_i2) = ∞
      //no connection between 2 subsystems
4: else if path.length = 0 then
5:    real_distance(sub_i1, sub_i2) = 0
      //direct connection
6: else
7:    for sub in path do
8:       if sub.type = COMM then
9:          real_distance(sub_i1, sub_i2) += 1;
            //connection through a COMM subsystem
10:      else
11:         real_distance(sub_i1, sub_i2) = ∞
12:         break;
13:      end if
14:   end for
15: end if
```

Thus, we can now create a valid solution including component mapping, function mapping, data mapping, and fault tolerance mapping that satisfy the rules and constraints. With such a solution, our system can be implemented but it may not be the best solution.

4.1.3 Solution evaluation

This subsection details how to evaluate a complete solution, which corresponds to the third part in Figure 4.1. Three values are taken into account such as performance (execution time), fault tolerance (reliability), resource usage (cost). Corresponding to each type of evaluation, Definitions 4.5, 4.6 and 4.7 introduce the concepts of execution time, reliability and cost, respectively.

4.1.3.1 Execution time evaluation of an application

Definition 4.5. the execution time of an application mapped on a platform is the longest duration to finish the output functions and is counted from the earliest beginning time of the input functions.

The aim is to estimate the execution time of the given application. However, firstly, we need to estimate the execution time of each function. Then, we consider applying a scheduling strategy to the application.

Execution time of a function The execution time of a function τ_{F_j} is composed of the duration of data reading, data receiving, processing function, data sending and data writing as given by Equation 4.17 where: $F_j \in V$, $j \in \mathbb{N}$;

$\tau_{me_{\text{in}}}(F_j)$ is the delay to access a ME subsystem that stores an input data of F_j (called the input ME subsystem); $\sum \tau_{me_{\text{out}}}$ is the total time to access all ME subsystems that store the output data of F_j (called the output ME subsystems);

$\tau_{pe}(F_j)$ is the duration of F_j performed on its assigned PE subsystem; $\sum \tau_{me_{\text{in}}}(F_j)$ is the total time to access all ME subsystems that store the input data of F_j;

$\sum \tau_{path_{\text{in}}}(F_j)$ is the total time to transfer all data on the paths between all the input ME subsystems and the PE subsystem on which F_j is mapped;

$\sum \tau_{path_{\text{out}}}$ is the total time to transfer the data on the paths between the output ME subsystems and the mapped PE subsystem of the function j.

$$\tau_{F_j} = \sum \tau_{me_{\text{in}}} + \sum \tau_{path_{\text{in}}} + \tau_{pe} + \sum \tau_{path_{\text{out}}} + \sum \tau_{me_{\text{out}}} \tag{4.17}$$

The latency of each ME subsystem ($\tau_{me_{\text{in}}}(F_j)$) can be calculated by the equations 2 and 4 in Table 4.2. The latency is the product of the delay of writing / reading 1 byte and the size of a data block. For example, a memory has a read delay 0.84 us/byte, so, the latency to read a block of 10 bytes is $0.84 \times 10 = 8.4$ us.

The duration of a function on a PE subsystem ($\tau_{pe}(F_j)$) depends on the fault tolerance strategy applied to the subsystem. Thus, we can calculate the duration by the corresponding equations (1, 4 and 5) in Table 4.2.

If the connection between a PE subsystem and an ME subsystem is direct, the latency is zero. If a PE subsystem has to access an ME subsystem through COMM subsystems, the latency of a COMM subsystem can be calculated by the equations (3 and 4) in Table 4.2. The delay evaluation of a COMM subsystem is proportional to a factor, called δ_{sub_i}. This path-contention delay model is derived from the work in [4]. δ_{sub_i} of a COMM subsystem is the total number of links of PE and COMM subsystems that "really" share the same COMM subsystem. The term "really" means that δ_{sub_i} only counts links that really used that COMM sub_i to transmit data/signals when the platform executes the application. Besides, $real_path(sub_{i_1}, sub_{i_2})$ mentioned previously in Constraint 4.4 is the set of all COMM between two considered subsystems. Based on the equations in Table 4.2 and this set, we can obtain $\sum \tau_{path_{\text{in}}}(F_j)$ and $\sum \tau_{path_{\text{out}}}$.

Table 4.2: List of equations to calculate the execution time for each subsystem type depending fault tolerance strategy.

No	**Name**	**Type**	**Equation**
1	No tolerance	τ_{pe}	$\tau_{\text{noTol}} = (fM_{F_j}^{sub_i}) \times (cM_{sub_i}^{compE_k}) \times (F_j.size)/(compE_k.speed_factor)$
2		τ_{me}	$\tau_{\text{noTol}} = (cM_{sub_i}^{compE_j}) \times (compE_j.delay) \times (dM_{d_k}^{sub_i}) \times (size(d_k))$
3		τ_{comm}	$\tau_{\text{noTol}} = (cM_{sub_i}^{compE_j}) \times (compE_j.delay) \times \delta_{sub_i}$
4	TMR	all types	$\tau_{\text{TMR}} = \tau_{\text{noTol}} + \tau_{\text{voter}}$
5	TReR	τ_{pe}	$\tau_{\text{TReR}} = 3 \times \tau_{\text{noTol}} + \tau_{\text{voter}}$

Figure 4.10 illustrates an example case to calculate the execution time of F_2. F_1 is mapped onto PE_1; F_2 and F_3 are mapped onto PE_2. Because F_2 and F_3 are mapped onto the same PE subsystem, the data block d_{2_3} is not assigned into any ME subsystem. Thus, there is also no output data path for F_2. The data blocks d_{1_2} is mapped onto ME_2 (the input ME subsystem of F_2), this data block is transferred through the $COMM_1$. Note that $\sum \tau_{path_{\text{in}}} = delay(COMM_1) \times 2$ because there are two connections with the COMM subsystem (PE_1 and PE_2). Therefore, the execution time of F_2 is calculated as τ_{F_2} in Figure 4.10.

Thus, if $start_{F_j}$ represents the time at which the execution process of F_j begins, the time at which the function F_j is completed is given by $finish_{F_j}$, by Equation 4.18, where $j \in \{1, 2, ...n_{\text{func}}\}$:

$$finish_{F_j} = start_{F_j} + \tau_{F_j} \tag{4.18}$$

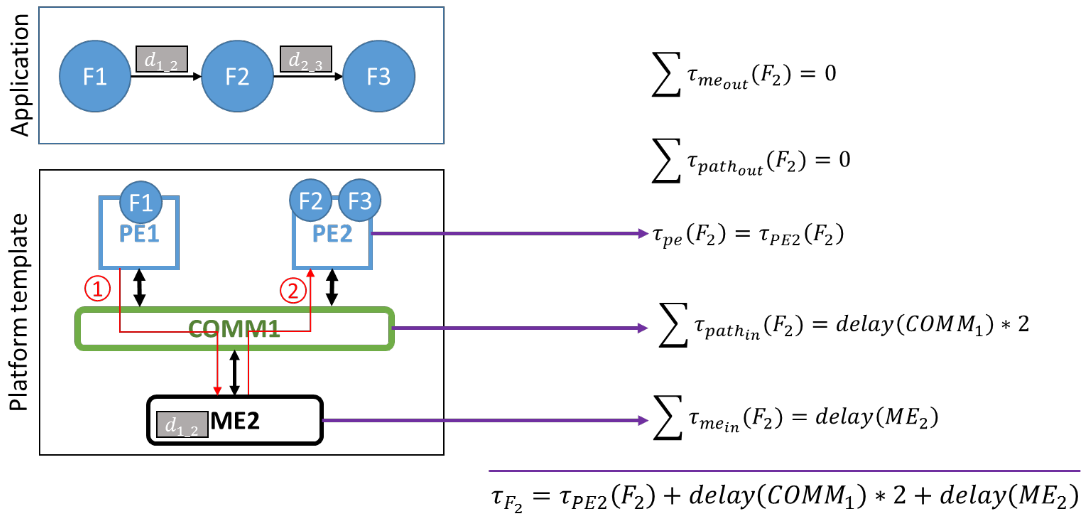


Figure 4.10: Connection between two functions.

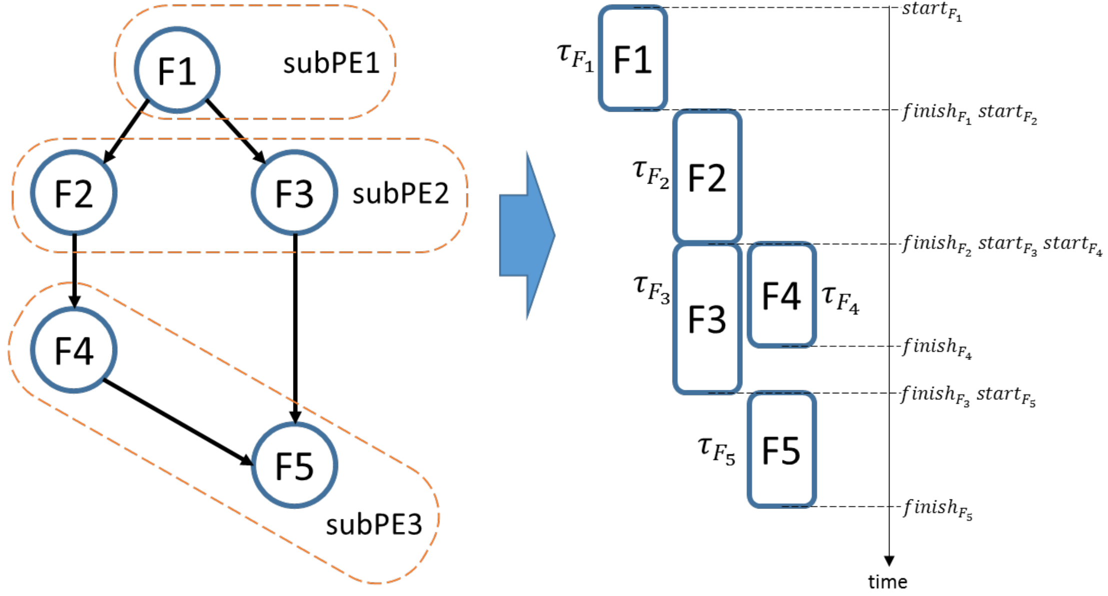


Figure 4.11: An example of function mapping and scheduling.

Scheduling When we have computed the execution time of each function, we use these results to estimate the execution time of an application. Scheduling is the process that determines the order in which each function is executed. Herein, the function scheduling for the MPSoC can be regarded as "*schedule order sequence; resource mapping sequence*" presented in [120]. Therefore, there are two **scheduling constraints** that we need to rely on to calculate the execution time of the application: 1) functions dependencies in the application DAG; 2) order of functions in the PE subsystem of a mapping solution.

For example, in Figure 4.11, with the first scheduling constraint, F_2 and F_3 can only start when F_1 finishes. F_5 can only start if both of F_4 and F_3 are ended. For the second scheduling constraint, we need a mapping solution to evaluate the sequential execution of the functions. A mapping example of the application as: $\{F_1\}$ is mapped on **subPE1**; $\{F_2, F_3\}$ is mapped on **subPE2**; $\{F_4, F_5\}$ is mapped on **subPE3**. In this mapping solution, $\{F_1, F_2, F_4, F_5\}$ and $\{F_1, F_3, F_5\}$ are executed in the order of the application DAG. Besides, F_2 and F_3 are executed in the order mapped in **subPE2**. It means that F_3 can only start when F_2 finishes. All functions have to comply with both of these rules in which the first has a higher priority than the second one. Therefore, the sequence of functions in the example solution is given by $F_1 \rightarrow F_2 \rightarrow \{F_3 \ // \ F_4\} \rightarrow F_5$.

Therefore, the scheduling can be generalized by the following two Constraints 4.5 and 4.6:

Constraint 4.5. *(schedule order sequence) for two functions on an edge of the application, the end function of the edge only begins when the begin function ends.*

Constraint 4.5 is expressed by Equation 4.19, where $j, k \in \{1, 2, ...n_{\text{func}}\}$:

$$e_{j_k} \implies start_{F_k} \geqslant finish_{F_j} \tag{4.19}$$

Constraint 4.6. *(resource mapping sequence) if two functions are mapped on the same PE subsystem, these two functions cannot be executed at the same time.*

Constraint 4.6 is expressed by Equation 4.20, where: $j_1, j_2 \in \{1, 2, ...n_{\text{func}}\}$, $i \in \{1, 2, ...n_{\text{sub}}\}$:

$$fM_{F_{j1}}^{sub_i} = 1, fM_{F_{j2}}^{sub_i} = 1 \implies start_{F_{j2}} \geqslant finish_{F_{j1}} \vee start_{F_{j1}} \geqslant finish_{F_{j2}} \tag{4.20}$$

Obviously, according to Definition 4.5 (about the execution time of an application), the execution time of a solution mapping (x_{sol}) is given by Equation 4.21 where:
$\forall j, k \in \{1, 2, ...n_{\text{func}}\}, finish_{system} = max(finish_{F_j} | \nexists e_{j_k})$;
$\forall j, k \in \{1, 2, ...n_{\text{func}}\}, start_{system} = min(start_{F_j} | \nexists e_{k_j})$.

$$T_{\text{sys}}(x_{\text{sol}}) = finish_{system} - start_{system} \tag{4.21}$$

For example, in the case of Figure 4.11, the execution time of the system for this mapping solution is express in Equation 4.22:

$$\begin{aligned} T_{sys}(x_{\text{sol}4.11}) &= finish_{F_5} - start_{F_1} \\ &= \tau_{F_1} + \tau_{F_2} + \tau_{F_3} + \tau_{F_5} \end{aligned} \tag{4.22}$$

4.1.3.2 Reliability evaluation

Definition 4.6. the reliability of an MPSoC system during an execution is the probability that no failure occurs on the result of the system for the whole duration.

If the subsystems are independent, the reliability of a platform is the product of the reliability of all subsystems. Therefore, we need to consider how to estimate the reliability of each subsystem with and without fault-tolerance strategies. The transient fault and the permanent fault are presented in Chapter 1. Herein we use the fault models to estimate the reliability of an MPSoC system. Hereafter, we present the estimation of the reliability of each type of subsystem (PE, ME and COMM) and their corresponding fault strategies.

PE subsystems Multi-functions can be mapped on one PE component. The transient fault can disappear after a PE component terminates a function and starts executing a new function. For example in Figure 4.12, at the component level, a transient fault occurred in the duration of F_2 then it disappears while executing F_3 because a new activities override and erase the previous transient fault. However, this does not happen with the permanent fault. Once a permanent fault occurs, a component is considered as unusable.

Figure 4.12: A transient fault appears and disappears on a PE component.

No tolerance: if a PE subsystem does not support fault tolerance mechanism, the reliability of the subsystem i containing the component k after m periods of the application is given by Equation 4.23 (the definition of a period is presented in Definition 1.1) which describes the probability that no permanent failure occurs during the operation duration and also no transient failure occurs on any function duration, where: $j \in \{1, 2, ...n_{\text{func}}\}$; $i \in \{1, 2, ...n_{\text{sub}}\}$; $k \in \{1, 2, ...n_{\text{compE}}\}$;

V_{sub_i} is the set of functions mapped on the subsystem i ($V_{sub_i} = \{F_j | fM_{F_j}^{sub_i} = 1, j \in \{1, 2, ...n_{\text{func}}\}\}$);

τ_{F_i} is the duration that the PE subsystem performs the function F_j, is calculated as τ_{noTol} - the equation 1 in Table 4.2;

$\tau_{\text{pf}_i} = \sum_{F_j \in V_{sub_i}} \tau_{\text{F}_j}$ is the total operation time of the PE subsystem in a period of the application (or in another word, the considered duration that the permanent failure may appear in a period of the application);

$R_{\text{PF}}(\tau_{\text{pf}_i}, compE_k)$ is the no-permanent-failure probability of the component k during τ_{pf_i} estimated from Equation 1.3;

$R_{\text{TF}_j}(\tau_{\text{F}_j}, compE_k)$ is the no-transient-failure probability of the component k during τ_{F_j} estimated from Equation 1.4.

$$
\begin{aligned}
&R_{\text{sub}_i(m)}(cM_{sub_i}^{compE_k} = 1; tM_{sub_i}^{0} = 1) \\
&= R_{\text{PF}}^{m}(\tau_{\text{pf}_i}, compE_k) \times \prod_{F_j \in V_{sub_i}} R_{\text{TF}_j}^{m}(\tau_{\text{F}_j}, compE_k)
\end{aligned}
\tag{4.23}
$$

For sake of clarity, we use $R_{\text{PF}}(\tau_{\text{pf}_i})$ instead of $R_{\text{PF}}(\tau_{\text{pf}_i}, compE_k)$, and $R_{\text{TF}_j}(\tau_{\text{F}_j})$ instead of $R_{\text{TF}_j}(\tau_{\text{F}_j}, compE_k)$ to shorten formulas with the same meaning.

Triple Modular Redundancy: In Figure 4.13, a PE subsystem is composed of three processors that execute a function in parallel. The result is correct if at least 2-out-of-3 processors have the same result. So, Equation 4.24 gives the reliability estimation of the TMR subsystem after the first period of the application, where $R_{\text{v}}^{m}(\tau_{\text{voter}})$ is the reliability of the voter during the operation duration τ_{voter}:

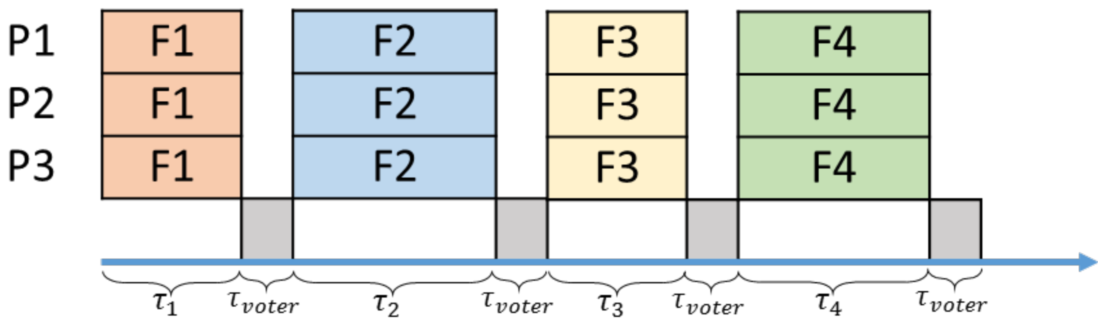


Figure 4.13: Multi functions are mapped on the same PE subsystem with TMR.

$$
\begin{aligned}
&R_{\mathrm{sub}_i(1)}(cM_{sub_i}^{compE_k} = 1; tM_{sub_i}^1 = 1)\\
&= \big(R_{\mathrm{PF}}(\tau_{\mathrm{pf}_i})\big)^3 \times \prod_{F_j \in V_{sub_i}} \Big((R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^3 + 3(1 - R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))(R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^2\Big) \times R_{\mathrm{v}}(\tau_{\mathrm{voter}})\\
&+ 3\big(R_{\mathrm{PF}}(\tau_{\mathrm{pf}_i})\big)^2\big(1 - R_{\mathrm{PF}}(\tau_{\mathrm{pf}_i})\big) \times \prod_{F_j \in V_{sub_i}} \Big((R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^3 + (1 - R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))(R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^2\Big) \times R_{\mathrm{v}}(\tau_{\mathrm{voter}})
\end{aligned} \tag{4.24}
$$

Note: As described in our meta-model, the voter can be considered as a PE component with the pre-defined parameters such as its voter reliability $R_{\mathrm{v}}(\tau_{\mathrm{voter}})$, its delay τ_{voter} and its cost c_{voter}.

Equation 4.24 is composed of two terms. The first one describes that no permanent failure in the whole execution time and, at most there is only one transiently faulty component in the execution of each function. The second one indicates that there is only one permanently faulty component in the whole execution and, if there is transient failure, the transient failure only appears on the permanently faulty component. In any case, the voter must have no failure because no strategy is considered on the voter for now. As the events are independent, the reliability after m periods of the application is given by Equation 4.25.

$$
\begin{aligned}
&R_{\mathrm{sub}_i(m)}(cM_{sub_i}^{compE_k} = 1; tM_{sub_i}^1 = 1)\\
&= \big(R_{\mathrm{PF}}^m(\tau_{\mathrm{pf}_i})\big)^3 \times \prod_{F_j \in V_{sub_i}} \Big((R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^3 + 3(1 - R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))(R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^2\Big)^m \times R_{\mathrm{v}}^m(\tau_{\mathrm{voter}})\\
&+ 3\big(R_{\mathrm{PF}}^m(\tau_{\mathrm{pf}_i})\big)^2\big(1 - R_{\mathrm{PF}}^m(\tau_{\mathrm{pf}_i})\big) \times \prod_{F_j \in V_{sub_i}} \Big((R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^3 + (1 - R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))(R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^2\Big)^m \times R_{\mathrm{v}}^m(\tau_{\mathrm{voter}})
\end{aligned} \tag{4.25}
$$

Elements in Equation 4.25 are clarified as following:

- $\big(R_{\mathrm{PF}}^m(\tau_{\mathrm{pf}_i})\big)^3$ implies that no permanent fault appears on the 3 components during m periods;
- $\Big((R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^3 + 3(1 - R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))(R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^2\Big)$ implies that al most only one transient fault appears in an execution of F_j on the 3 components. So, after m periods, the reliability is $\Big((R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^3 + 3(1 - R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))(R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^2\Big)^m$;
- $R_{\mathrm{v}}^m(\tau_{\mathrm{voter}})$ implies that no fault appears during m periods on the voter;
- $3\big(R_{\mathrm{PF}}^m(\tau_{\mathrm{pf}_i})\big)^2\big(1 - R_{\mathrm{PF}}^m(\tau_{\mathrm{pf}_i})\big)$ implies that at most only one permanent fault appear during m periods on the 3 components;
- $\Big((R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^3 + (1 - R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))(R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^2\Big)$ implies that a transient fault can only appear on the permanently faulty component, no fault appears on the 2 others. So, after m periods, the reliability is $\Big((R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^3 + (1 - R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))(R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^2\Big)^m$.

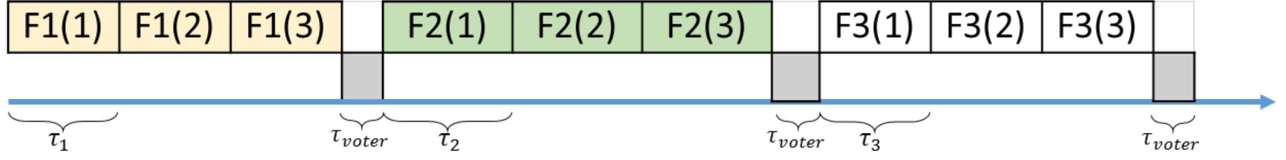


Figure 4.14: Multi functions are mapped on the same PE subsystem with TReR.

Triple Re-execution Redundancy: Figure 4.14 describes the TReR strategy. A PE subsystem that is composed of one processor. The processor performs any mapped function 3 times. The result is correct if at least 2-out-of-3 execution are correct. However, since permanent faults cannot be recovered and transient faults disappear after each execution, it should be noted that the reliability of the subsystem is the probability that there is no permanent fault and at most only one execution time fails by the transient fault. So, the reliability of the TReR subsystem is given by Equation 4.26 after m periods of the application.

$$\begin{aligned} &R_{\mathrm{sub}_i(m)}(cM_{sub_i}^{compE_k} = 1; tM_{sub_i}^2 = 1) \\ &= (R_{\mathrm{PF}}(3\tau_{\mathrm{pf}_i}))^m \times \prod_{F_j \in V_{sub_i}} \left((R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^3 + 3(1 - R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))(R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^2\right)^m \times R_{\mathrm{v}}^m(\tau_{\mathrm{voter}}) \end{aligned} \tag{4.26}$$

Elements in Equation 4.26 are clarified as following:

- $(R_{\mathrm{PF}}(3\tau_{\mathrm{pf}_i}))^m$ implies that there is no permanent fault in the whole m periods;
- $\left((R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^3 + 3(1 - R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))(R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^2\right)$ implies that during the triple times of the execution of F_j, there is at most only one execution time fails by the transient fault. So, $\left((R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^3 + 3(1 - R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))(R_{\mathrm{TF}_j}(\tau_{\mathrm{F}_j}))^2\right)^m$ implies that the execution of F_j is correct during m periods;
- $R_{\mathrm{v}}^m(\tau_{\mathrm{voter}})$ implies that no faults in the voter during m periods.

ME subsystems As mentioned earlier, we establish the equations for No-tolerance and TMR for ME subsystem.

No Tolerance: The reliability of a ME subsystem after m periods of the application is given by the Equation 4.27:

$$R_{\mathrm{sub}_i(m)}(cM_{sub_i}^{compE_k} = 1; tM_{sub_i}^0 = 1) = R_{\mathrm{PF}}^m(\tau_{\mathrm{pf}_i}) \times R_{\mathrm{TF}_j}^m(\tau_{\mathrm{F}_j}) \tag{4.27}$$

Triple Modular Redundancy for a ME subsystem after m periods of the application is given by the Equation 4.28, where:

τ_{pf} is the whole operation time of the considered ME subsystem in a period of the application; in other words, τ_{pf} is the considered duration that permanent faults can occur;

τ_{tf} is the duration that transient faults can appear on the considered ME subsystem.

$$
\begin{aligned}
&R_{\text{sub}_i(m)}(cM^{compE_k}_{sub_i} = 1; tM^{1}_{sub_i} = 1) \\
&= \big(R^m_{\text{PF}}(\tau_{\text{pf}})\big)^3 \times \Big((R_{\text{TF}}(\tau_{\text{tf}}))^3 + 3(1 - R_{\text{TF}}(\tau_{\text{tf}}))(R_{\text{TF}}(\tau_{\text{tf}}))^2\Big)^m \times R^m_{\text{v}}(\tau_{\text{voter}}) \\
&+ 3\big(R^m_{\text{PF}}(\tau_{\text{pf}})\big)^2\big(1 - R^m_{\text{PF}}(\tau_{\text{pf}})\big) \times \Big((R_{\text{TF}}(\tau_{\text{tf}}))^3 + (1 - R_{\text{TF}}(\tau_{\text{tf}}))(R_{\text{TF}}(\tau_{\text{tf}}))^2\Big)^m \times R^m_{\text{v}}(\tau_{\text{voter}})
\end{aligned}
\tag{4.28}
$$

It is assumed that the data stored in an ME subsystem (First In, First Out (FIFO) type) has to exist at least through one period of the application. After that, new data arrives and overrides completely the old data. At the component level, if there is a transient fault, it can disappear when a new period comes. Therefore, within the scope of one application execution, for ME subsystems, $\tau_{\text{tf}} = \tau_{\text{pf}} = \tau_{\text{op}}$, is the whole operation time of the considered ME subsystem in a period of the application. For a ME subsystem, all durations of memory accesses into the ME subsystem i in a period of the application are given Equation 4.29, where: D_{sub_i} is the set of all data mapped in the subsystem i ($D_{sub_i} = \{d_{j_k} | dM^{sub_i}_{d_{j_k}} = 1; d_{j_k} \in D; sub_i \text{ is } ME; j,k \in \mathbb{N}\}$); τ^{me}_{op} are obtained through the process of calculating the execution time of the application.

$$
\tau^{me}_{\text{op}}(sub_i) = \sum_{d_{j_k} \in D_{sub_i}} \big(\tau_{me_{\text{in}}}(F_j)\big) + \sum_{d_{j_k} \in D_{sub_i}} \big(\tau_{me_{\text{out}}}(F_k)\big) \tag{4.29}
$$

COMM subsystems Herein we establish the equations of the No-tolerance and the TMR for COMM subsystems.

No Tolerance: The reliability of a COMM subsystem after m periods of the application is given by the Equation 4.30:

$$
R_{\text{sub}_i(m)}(cM^{compE_k}_{sub_i} = 1; tM^{0}_{sub_i} = 1) = R^m_{\text{PF}}(\tau_{\text{pf}_i}) \times R^m_{\text{TF}_j}(\tau_{\text{F}_j}) \tag{4.30}
$$

Triple Modular Redundancy for a COMM subsystem after m periods of the application is given by the Equation 4.31, where: τ_{pf} is the whole operation time of the considered COMM subsystem in a period of the application; τ_{tf} is the duration that transient faults can appear on the considered COMM subsystem.

$$
\begin{aligned}
&R_{\text{sub}_i(m)}(cM^{compE_k}_{sub_i} = 1; tM^{1}_{sub_i} = 1) \\
&= \big(R^m_{\text{PF}}(\tau_{\text{pf}})\big)^3 \times \Big((R_{\text{TF}}(\tau_{\text{tf}}))^3 + 3(1 - R_{\text{TF}}(\tau_{\text{tf}}))(R_{\text{TF}}(\tau_{\text{tf}}))^2\Big)^m \times R^m_{\text{v}}(\tau_{\text{voter}}) \\
&+ 3\big(R^m_{\text{PF}}(\tau_{\text{pf}})\big)^2\big(1 - R^m_{\text{PF}}(\tau_{\text{pf}})\big) \times \Big((R_{\text{TF}}(\tau_{\text{tf}}))^3 + (1 - R_{\text{TF}}(\tau_{\text{tf}}))(R_{\text{TF}}(\tau_{\text{tf}}))^2\Big)^m \times R^m_{\text{v}}(\tau_{\text{voter}})
\end{aligned}
\tag{4.31}
$$

For COMM, when the new period begins, the transient failures in the previous periods are completely eradicated. It implies that at the component level, if there is a transient fault, it can disappear when a new period comes. Therefore, within the scope of one application execution, for COMM subsystems, such as for ME, $\tau_{\text{tf}} = \tau_{\text{pf}} = \tau_{\text{op}}$ is the whole operation time of the considered COMM subsystem in a period of the application. All duration of data transfer through the COMM subsystem in a period of the application is given by Equation 4.32, where τ^{comm}_{op} is obtained through the process of calculating the time to transfer the data between ME subsystems and PE subsystems, by Equation 4.32 where $j \in \{1, 2, ..., n_{\text{func}}\}$.

$$
\tau^{comm}_{\text{op}}(sub_i) = \sum(operation_time_of_sub_i) \tag{4.32}
$$

Platform Once the reliability of all subsystem calculated, the platform reliability of a solution (x_{sol}) after m periods of execution of the application is given by Equation 4.33, where $R_{\text{sub}_i(m)}$ is the individual reliability of all subsystems, as following:

$$R_{\text{platf(m)}}(x_{\text{sol}}) = \prod_{i=1}^{n_{\text{sub}}} R_{\text{sub}_i(m)} \tag{4.33}$$

4.1.3.3 Cost evaluation

Definition 4.7. The cost of a platform is the total cost of all the active components used in that platform.

The cost evaluation is intended to ensure the balance of hardware resources with goals of performance and reliability when designing a system. Each component has a cost value. When a component is mapped to a subsystem, depending on the fault-tolerance strategy applied to the subsystem, the cost of that subsystem is calculated according to the equations given in Table 4.3. Therefore, the platform cost of a mapping solution (x_{sol}) defined in Definition 4.7, is given by the Equation 4.34, where "sub$_i$ in use" means that the cost of a subsystem is only counted for the equation if and only if the subsystem is used (the PE subsystem is mapped functions, the ME subsystem is mapped data, the COMM subsystem actually has the data transmission).

$$C_{\text{platf}}(x_{\text{sol}}) = \sum_{i=1}^{n_{\text{sub}}} C_{\text{sub}_i} \text{ where: sub}_i \text{ in use} \tag{4.34}$$

In Table 4.3, the first row show the equations to calculate the cost of a *subPPE* when no tolerance is applied. The cost is the sum the cost of its *compPPE* and *compSoftware*. With the other types of subsystem (the second row), the cost (no tolerance) is the cost of its component. The cost of a *subPPE* applied TMR (third row) includes the cost of 3 *compPPE* and 1 *compSoftware* and 1 voter. With the other types of subsystem (the fourth row), the cost with TMR includes the cost of 3 components and 1 voter. The cost of a subsystem applied TReR equals the cost of the subsystem without tolerance plus the cost of a voter.

Table 4.3: List of equations used to calculate the cost of fault tolerance strategies of a subsystem C_{sub_i}.

	Name	**Type**	**Equation**
1	No tolerance	PPE	$C_{\text{noTol}} = (cM_{sub_i}^{compE_k}) \times (compE_k.cost) + compE_{soft}.cost$
2		DPE or ME or COMM	$C_{\text{noTol}} = (cM_{sub_i}^{compE_k}) \times (compE_k.cost)$
3	TMR	PPE	$C_{\text{TMR}} = 3 \times (cM_{sub_i}^{compE_k}) \times (compE_k.cost) + compE_{soft}.cost + c_{\text{voter}}$
4		DPE or ME or COMM	$C_{\text{TMR}} = 3 \times C_{\text{noTol}} + c_{\text{voter}}$
5	TReR	PE	$C_{\text{TReR}} = C_{\text{noTol}} + c_{\text{voter}}$

4.2 Optimization process

We can generate solutions in a design space. Now, there are two questions need to be answered:

- for two different solutions, which one is better?
- which solution is best in the design space and how to find that point?

For the first question, we need to define a method for comparing two solutions. In other words, we try to scalarize the set of objectives (time, reliability, cost) into a single objective. Then, to answer the second question, we need to integrate the whole process (from the mapping process to the optimum function) into a search strategy that allows finding the best solution.

4.2.1 Objective function

There are three performance criteria of a mapping solution: execution time, reliability level and cost. The objective is to *a)* maximize the reliability level (maximum is 1), *b)* to minimize the cost and *c)* to minimize the execution time. There are several methods to solve this problem [121]. Basically, with the optimization of a multi-objective problem, we try to set up formulas to compare the two solutions, or try to find the best set of solutions instead of just one solution. Using a method depends on the needs of designers. However, the availability of the criteria of each solution such as the proposition of this thesis makes it less difficult to apply the optimal methods. Within this manuscript, we choose the Weighted Metric Method [122]. This is an easily-implement for method, if it is ideal, it will give a single optimal solution. This also helps us easily verify our framework. This method uses a weighted distance metric of any solution from the ideal solution. It is thus necessary to know the minimum and the maximum objective values and then to define the ideal solution.

Thus, from the idea of the method, the objective function to evaluate simultaneously three sub-objective values of a mapping solution x_{sol}, is proposed as the Equation 4.35. α, β, γ in the Equation 4.35 are factors that can be set by designers to give more weight to the parameters they wish to favor where: $\alpha + \beta + \gamma = 1$, $\alpha, \beta, \gamma \geq 0$,; $R_{\text{platf(m)}}$, T_{sys} and C_{platf} are respectively the platform reliability, the application execution time and the cost of a mapping solution that are calculated from the Section 4.1.3.

$$func(x_{\text{sol}}) = \alpha \times (R_{\text{platf(m)}} - 0) + \beta \times \frac{T_{max} - T_{\text{sys}}}{T_{max}} + \gamma * \frac{C_{max} - C_{\text{platf}}}{C_{max}} \tag{4.35}$$

- $(R_{\text{platf(m)}} - 0)$ means that the minimum value of the reliability is 0, "the bigger the better".
- $(T_{max} - T_{\text{sys}})$ means "the bigger the better". T_{max} is the possible longest value of the execution time estimated from the worst case. The worst execution time T_{max} is given by the Equation 4.36.

$$T_{max} = \sum_{j}^{n_{\text{func}}} \overset{F_j}{\max}(\tau_{\text{TReR}}) + \sum_{j,COMM}^{n_{\text{sub}}} (2 \times n_{\text{data}} \times \max_{sub_j(cL)}(\tau_{\text{TMR}})) + 2 \times n_{\text{data}} \times \overset{n_{\text{sub}}}{\max_{sub_j^{me}(cL)}}(\tau_{\text{TMR}}) \tag{4.36}$$

- $(C_{max} - C_{\text{platf}})$ means "the bigger the better". Respectively, C_{max} is the possible biggest value of the cost estimated from the worst case. The most expensive cost C_{max} is estimated by the Equation 4.37.

$$C_{max} = \sum_{j}^{n_{\text{sub}}} \max_{sub_j(cL)} (C_{TMR}) \tag{4.37}$$

Undoubtedly, our objective problem is is expressed as Equation 4.38:

$$\begin{cases} \textbf{maximize} & func(x_{\text{sol}}) \text{ from Equation 4.35} \\ \textbf{subject to} & x_{\text{sol}} \text{ from the design-solution generator} \end{cases} \tag{4.38}$$

4.2.2 Search strategies

We can now distinguish 2 points in the design space to know which one is better. So, hereafter, we will use three very popular strategies in literature to find the best solution and indicate how to integrate these strategies into our DSE.

4.2.2.1 Comprehensive search (CS)

The first one, the "simplest" and most classic strategy of the optimization is the Comprehensive Search (CS). We will calculate from the first solution until the final solution, without missing any possible solutions in the design space.

The terms "simplest" implies that the strategy is easy to understand and to implement. However, this strategy may face time problems if the solution space is extremely large. Its advantage is only that the found solution (if possible) is definitely the best solution.

4.2.2.2 Simulated annealing (SA)

The second one is the SA [123]. This strategy is based on the simulation of a cooling process of metal, glass, or crystal. The process is that heating above its melting point, holding its temperature, and then cooling it very slowly until it solidifies into a perfect crystalline structure.

SA is composed of two processes: *1)* one for the *solution generation* and *2)* the other for the *solution acceptance*. It chooses a random move (a move = a *solution generation*) from the neighborhood. Then, if the move is evaluated better than its current position, the move is definitely accepted to become the current position. If the move is worse (i.e. lesser quality), it can be accepted based on a probability model. Choosing a move is the *solution acceptance*. The probability model can be as Boltzmann distribution, Cauchy distribution, Markov chain Monte Carlo, etc. Thereby, we can integrate SA into our DSE framework as:

- the *solution generation* is equivalent to our "mapping process".
- the *solution acceptance* is equivalent to our "solution evaluation" and our "objective function" (Equation 4.35).

There are many efforts that provide some open-source tools to simulate this SA process. We only need to integrate our modeling codes, evaluation functions into these tools. This greatly reduces development time and is easy to contribute to the community. For this work, we used an available library in Python: **simanneal**. According to the authors in this library, the temperature model is an exponential algorithm.

4.2.2.3 Genetic algorithm (GA)

The third possible approach is the GA [123]. Basically, GA is based on the neo-Darwinian paradigm for simulating the natural evolution of biological systems. GA starts with a set of *individuals* called a population. An *individual* in a population is represented by a *chromosome*, that competes and exchanges information with the others in the population. A population is usually initiated randomly, then the population is evolved by generating from some selection/reproduction procedure to produce the next generation. Normally, the population size is preserved throughout each generation. At each generation, the *fitness* of each individual is evaluated. Then, based on their fitness, individuals are probabilistically selected to mate and produce offspring. At this moment, the crossover and the mutation randomly occurs. An *individual* with a high *fitness* has a high probability of being selected, and therefore, its *chromosome* has many opportunities to be inherited in next generations. The process will be repeated over many generations until a termination criterion is met. The termination criterion can be set as a maximum number of generations, or the convergence of the gene-types of the individuals.

Thereby, we integrate the GA algorithm into our DSE framework as:

- an *individual* is a solution in our DSE framework.
- a *chromosome* is the string of a solution encoded in Equation 4.1.
- the fitness of each individual (solution) is evaluated by Equation 4.35

As with SA, we use an open-source library for GA in Python: sklearn-deap was developed based on the DEAP [124].

4.3 Summary

This chapter presented the entire fault-tolerance DSE part of our framework. The part is composed of 2 principal problems: fault-tolerance design space generation, and optimization.

We proposed some contributions for the fault-tolerance design space generation. The first one is the mapping process that consists of the component mapping, the function mapping, the data mapping, and the fault-tolerance strategy mapping. The second one is the set of evaluation algorithms that consists of the time evaluation, the reliability evaluation, and the cost evaluation.

The optimization process proposed an objective function and the integration of search strategies into our DSE process. The objective function allows the comparison of solutions and indicates which one is better. Then, three search strategies such as the Comprehensive Search, the Simulated Annealing, and the Genetic Algorithm are to find the best solution among the possible design space.

Chapter 5

Experimental evaluation

Abstract: The DSE framework is proposed in the previous chapters with all its tools, which allows modeling, mapping, evaluating, optimizing the design of reliable MPSoCs. Now, in this chapter, we applied this framework for some case-studies and to give some discussions on this and show the effectiveness of the approach.

Contents

5.1 Case-study description . . . 100
5.1.1 Sobel filter . . . 100
5.1.2 Harris detector . . . 102
5.2 DSE results . . . 104
5.2.1 Sobel filter . . . 104
5.2.2 Harris detector . . . 109
5.3 Summary . . . 117

5.1 Case-study description

In this section, we describe case-studies to test our DSE framework.

5.1.1 Sobel filter

On this case study, the application model of Case-study 1 is presented. Then, the platform template and the list of component.

5.1.1.1 Application

The Sobel filter is a simple and very popular application that is an edge-detection for image processing. Figure 5.1 and Table 5.1 describes the details of functions in the application. There are four main functions: **get Pixels**, **Gx**, **Gy** (measurements of the gradient component of each orientation), and **Abs** (gradient magnitude). The code implementation of the functions is described in Listing A.1, A.2, A.3, A.4 in Appendices.

These functions are executed on Microblaze processors (Zedboard) [125] to find the worst-case execution time (wcet) shown in Table 5.1. At each time, this application processes a 9x9-pixel frame. So, the sizes of the data exchanged between functions are described in the table.

5.1.1.2 Platform

The platform template is composed of 4 PE subsystems, 1 COMM subsystem, and 1 ME subsystem. Figure 5.2 describes the platform template used for this case study. To fill out the empty subsystems of the platform template, there are available components described in Table 5.2.

The failure rates of "Microblaze" are assumed following the worst-case parameters of the device Xilinx 7-serie FPGA [126]. A Microblaze has 1089 Look-Up Table (LUT)'s and 969 registers, LUTs are typically built as 32x2 bits, a register has 32-bit [127]. The transient failure rate is 75 FIT/Mb [128], so λ_{TF} of the Microblaze can be estimated as 7.6 FIT. Assuming that an unlimited number of functions can be mapped on one Microblaze (**Capacity**: multi-functions).

The software component is considered as having no permanent fault and has a transient failure rate as shown in the table.

It is noted that in the general model of an application, the size of a function is represented by its number of operations. Then, the execution time of a function on PE is calculated by the product of

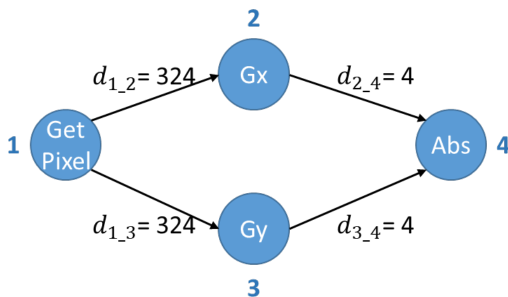


Figure 5.1: the DAG of the Sobel-filter application

Table 5.1: details of functions and data in the Sobel application.

Function	**wcet running on** Microblaze (μs)	**Data**	**size** (Bytes)
Get Pixels	85	d_{1_2}	324
Gx	1009	d_{1_3}	324
Gy	1009	d_{2_4}	4
Abs	86	d_{3_5}	4

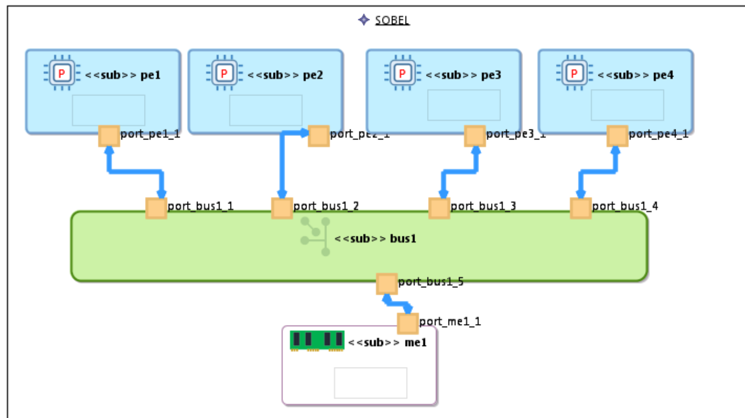


Figure 5.2: platform template for the case study-1 is composed of 6 subsystems: 4 *subPE*s (in blue), 1 *subCommunication* (in green), and 1 *subMemory* (in white)

Table 5.2: component parameter list in the case study-1.

ID	**Type**	λ_{PF} (FIT)	λ_{TF} (FIT)	**Capacity**		Cost (component unit)	Quantity
Microblaze	*PPE*	10	7.7	multi-functions	speed factor: predifined	1	10
softComp	*SOFT*	0	15			1	unlimited
BRAM	*ME*	10	72.7	up to 128 KBytes	delay: 0.8 μs/byte	1	5
AXI interconnection	*COMM*	10	1.18	NA	delay: 0.6 μs	1	3

the size of the function and the inverse of the speed factor of that PE. The overall purpose is only to calculate the execution time of the functions. If the designers have obtained already time value of the function execution as in this example, we can always use that input regardless of the number of operations.

With the size of 128KBytes, the transient-failure rates of a "BRAM" component can be estimated as 72.7 FIT (because the transient failure rate is 70 FIT/Mb for BRAM [128]). The transient-failure rates of "AXI" component (182 LUTs and 130 registers) is 1.18 FIT .

The delay of BRAM is measured by sending a byte from the processor to a BRAM without AXI bus interconnection. Thus, the delay of AXI bus is the difference between sending 1 byte from 1 processor to a BRAM with and without the AXI bus. The code implementation is presented in Appendix A.3.

Assuming that the whole system operates in the environment as the test High-temperature operating life [128], so the permanent failure rate of every components is 10 FIT.

Cost of each component is 1 unit. When setting the value of each component as such, we want to minimize the number overall of components used for the system.

The fault-tolerance strategies considered in this example are the TMR and the TReR. In the two strategies, the voter can be considered as a PE component. The voter component in Zynq is build on AXI protocol [129]. Therefore, for simplicity, the parameters of the voter are assumed with $R_{\text{v}}(\tau_{\text{voter}}) = 1$, τ_{voter} = delay of "AXI" bus, $c_{\text{voter}} = 1$ and no tolerance is applied on the voter.

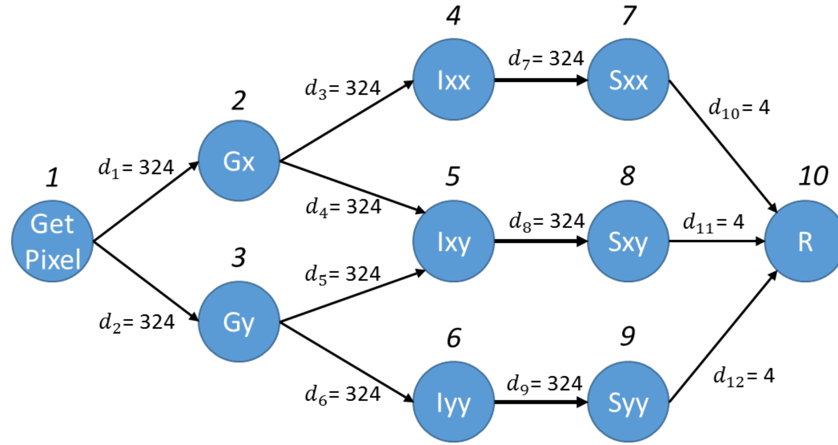


Figure 5.3: the DAG of the 10-function application in the case study-3.

Table 5.3: functions and data in the Harris application.

Function	**Detail**	**wcet running on** (μs)		**Data**	**size** (Bytes)
		Microblaze	ARM		
F_1	Get pixel	83	22	d_1	324
F_2	Gradient x	8685	302	d_2	324
F_3	Gradient y	8685	302	d_3	324
F_4	Product x	3300	123	d_4	324
F_5	Product x & y	685	45	d_5	324
F_6	Product y	3300	123	d_6	324
F_7	Sum x	470	21	d_7	324
F_8	Sum x & y	470	21	d_8	324
F_9	Sum y	470	21	d_9	324
F_{10}	Corner response	111	5	d_{10}	4
				d_{11}	4
				d_{12}	4

5.1.2 Harris detector

To increase complexity, Case-study 2 is composed of a 10-function application and a 15-subsystem platform template.

5.1.2.1 Application

The application model in Figure 5.3 is the Harris Corner Detector [130] that is used in the image processing to extracting corners of an image. The worst case execution time (wcet) of these functions running on Microblaze and ARM processors (Zedboard) is given in Table 5.3. Note that herein to be brief, we used the data formalism d_i instead of d_{i_j}. The position of each data is indicated in Figure 5.3. This application processes 9x9-pixel frames (36 bytes). So, the sizes of the data exchanged between functions are described in the table accordingly. The code implementation of the functions is described in Listing A.5 to A.11 in Appendices.

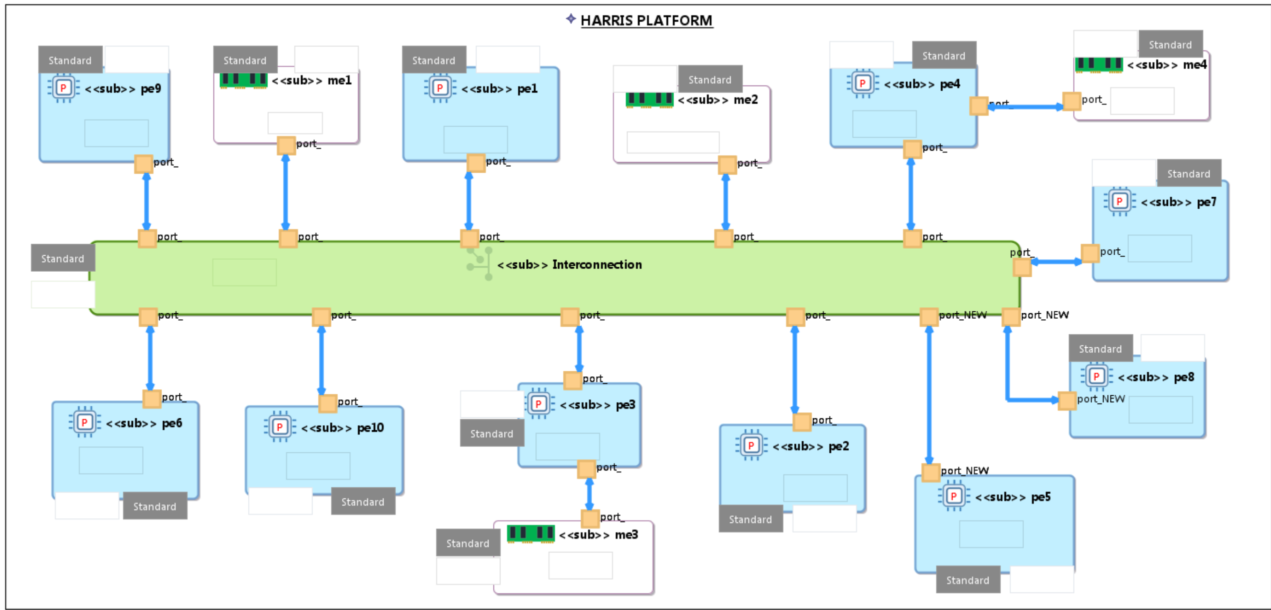


Figure 5.4: platform template in the case study-2 is composed of 15 subsystems.

Table 5.4: component parameter list in the case study-2.

ID	**Type**	λ_{PF} (FIT)	λ_{TF} (FIT)	**Capacity**		Cost (component unit)	Quantity
Microblaze	*PPE*	10	7.7	multi-functions	speed factor: predifined	1	10
ARM A9	*PPE*	1000	150	multi-functions	speed factor: predifined	1	2
softComp	*SOFT*	0	15			1	unlimited
BRAM	*ME*	10	76.8	up to 128 KBytes	delay: 0.8 μs/byte	1	5
AXI interconnection	*COMM*	10	1.18	NA	delay: 0.6 μs	1	3

5.1.2.2 Platform

The platform template and the component list are described respectively in Figure 5.4 and Table 5.4. Because the application has 10 functions, the platform template is composed of up to 10 PE subsystems (in blue). There are four ME subsystems (in white). The subsystems communicates through a COMM subsystem (in green).

In assuming that our platform is built on Xilinx Zed-board. Therefore, the parameters of components are defined in Table 5.4. Note that there are only maximally 2 ARM processors. The failure rates of ARM processor are declared from [131].

5.2 DSE results

This section describes the results of our DSE framework with the two previously described case-studies. Each case study is considered with 3 search strategies GA, SA, and CS. These DSE experiments are executed on a machine: Intel Core 4-core i7-5600U running at 2.6 GHz with 16 Gbytes RAM.

It should be noted that all three strategies run in Python. Each strategy is programmed in parallel to maximize the capabilities of the machine and reduce the DSE duration. This means that in each strategy, there are four independent jobs to run in parallel (by using the open-source library *multiprocessing* of Python). Each job is responsible for the mapping process and the solution evaluation. In each case-study, there are three main parts:

- **search strategies - parameter setup**: setting parameters of search strategies using in the DSE framework;
- **results**: show the configuration of found design solutions, the reliability, the execution time, and cost of these solutions;
- **discussion**: analysis of the results and remarks.

5.2.1 Sobel filter

Before going into the details of each search strategy, some common parameters in the objective function need to be established as in Table 5.5.

$T_{\max}$ and $C_{\max}$ are fixed in each case study and estimated by the inputs described in Subsection 5.1.1. The system moves a 9×9-pixel window of filter through every pixels. Therefore, in each period, the window stays at a pixel to process a 9×9-pixel frame. It is assumed that the system executes 100000 photos, in which each photo has a size of 1080×720 pixels. Therefore, the number of periods (m) is $10^5 \times 1080 \times 720 = 7776 \times 10^7$.

In this case study, the optimization process is implemented with each pair of 2 objectives: (reliability vs time), and (cost vs reliability). Assuming the importance of objectives is the same, the weighting factors are given as in the bottom line of Table 5.5.

5.2.1.1 Search strategies - parameter setup

In the search strategies, there are some parameters that need to be set up.

Genetic algorithm (GA) The GA parameters needed to be set before running are:

- *population_size* is the size of the population;
- *generation_number* is the number of generation. This number should be chosen so that the average fitness of the population converges;

Table 5.5: parameters in the objective function of Case-study 1.

Name	Definition	Value
m	number of periods	7776×10^7
$T_{\max}$	longest execution time	7952 μs
$C_{\max}$	most expensive cost	28
$\{\alpha, \beta, \gamma\}$	weighting factors	$\{0.5, 0.0, 0.5\}$; $\{0.5, 0.5, 0.0\}$

Table 5.6: GA parameters in Case-study 1.

Name	Value
population_size	150
generation_number	100
tournament_size	3
gene_mutation	0.03
gene_crossover	0.4

- *tournament_size* is the number of running tournaments among a few individuals to choose randomly from the population;
- *gene_mutation* is the probability of gene mutation in a chromosome;
- *gene_crossover* is the probability of gene swap between two chromosomes.

There is no "free lunch" so it is difficult to select good parameter sets from the beginning. Therefore, to start, the parameters are chosen as being commonly observed in the literature [132] for simple and small case-studies. Table 5.6 gives the parameters used in this case study.

Simulated Annealing (SA) The SA parameters also need to be set before executing, the parameters used for the SA strategy are provided in Table 5.7.

- T_{max} is the initial temperature that should be set to accept roughly 98% of the moves [133]. After 20 times of test runs, we observe that $T_{max} > 50$ allows achieving the desired acceptance;
- T_{min} is the stopping temperature. Normally, the simulation process stops when the temperature reaches a value as close as possible to zero;
- *num_steps* is the number of iterations in the whole cooling process. The bigger the value is, the slower the process is, and therefore, the probability to find the optimal global solution increased. *num_steps* is proportional to the size of the problem instance. The default number of steps recommended by the authors of **simanneal** is 50000; However, we tried with several values to find a good number of steps at $\{\alpha, \beta, \gamma\} = \{0.5, 0.5, 0.0\}$. At each value of *num_step*, we ran the SA search 10 times. The goal is to fins a solution with the highest value of the objective function in Equation 4.35. The **best solution** that is found on all these runs is considered as the **best global solution**. Figure 5.5 shows the impact of *num_steps* on the probability to find the best solution. The probability to find the best solution is represented by the number of times among 10 runs that the best solution appears. We can see that the greater the number of steps is, the higher the probability of finding the best solution is. So, from the figure, *num_steps* > 70000 helps to achieve a percentage of 100%.

Table 5.7: SA parameters in Case-study 1.

Name	**Value**
T_{max}	55
T_{min}	0.0001
num_steps	70000

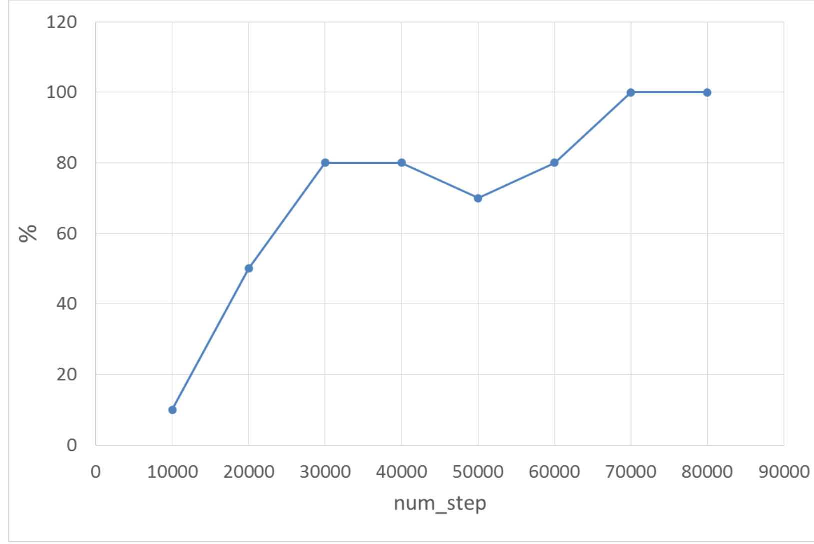


Figure 5.5: Impact of the number of steps on the probability to find the best solution.

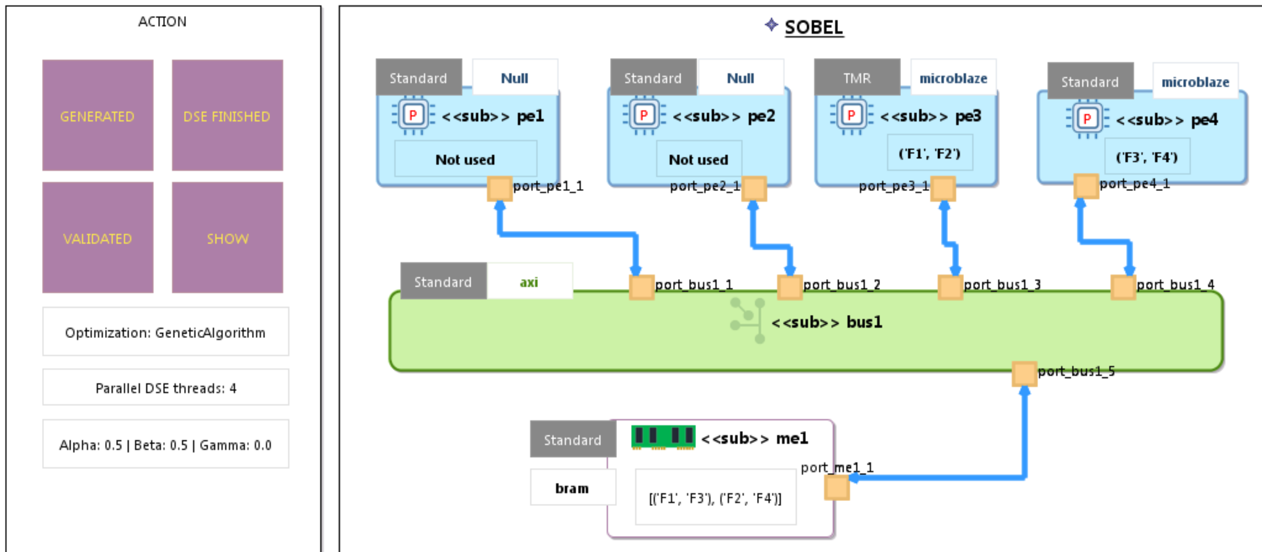
Comprehensive search (CS) This strategy goes through each one solution in the whole space and then perform an exhaustive search. No parameters need to be set. The total number of solutions to be explored is more than 1990656 solutions (1 configuration for the component mapping; 6144 permutation configurations for the function mapping; 1 configuration for the data mapping; $3^4 \times 2^2$ configurations for the fault-tolerance mapping). This value was given after CS ended its execution.

5.2.1.2 Results

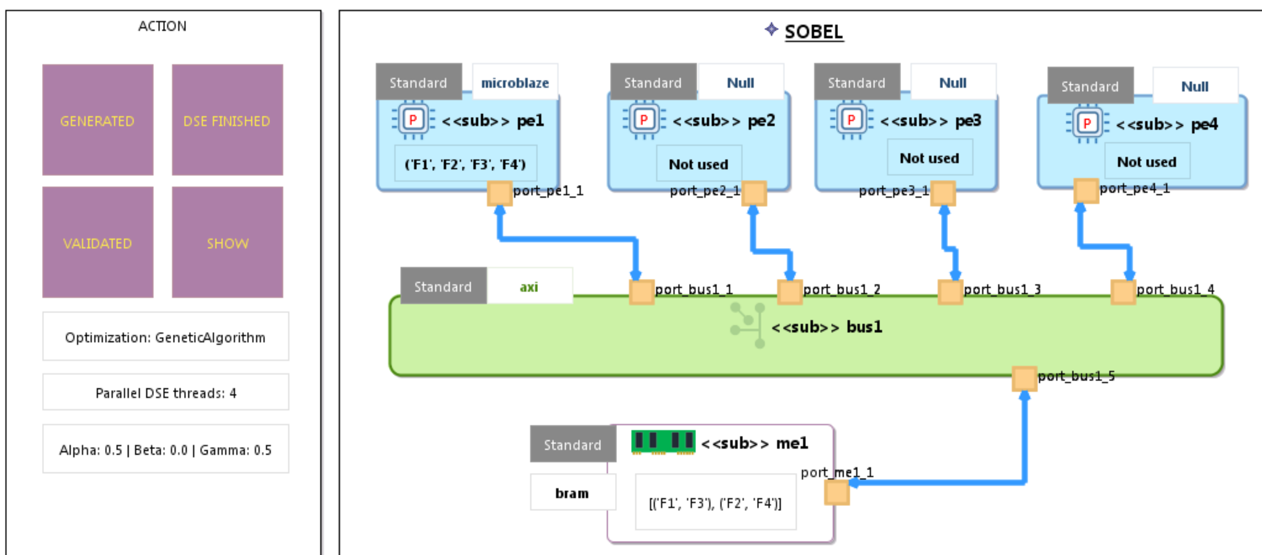
GA and SA are used for all instances of Case-study 1. At each instance of Case-study 1, we ran the GA (SA) search 10 times. The best solution that is found among all these runs is considered the best solution. All three search strategies give the same results.

Figure 5.6a and 5.6b describe the optimal solutions found in Case-study 1. As we presented in Chapter 3, the blue blocks are PE subsystems, the green blocks are COMM subsystems and the white blocks are ME subsystems. On these all blocks, the bordered gray node shows the fault-tolerance strategy applied in the corresponding subsystem and the bordered white nodes shows the component used in the corresponding subsystem. The blue block (PE subsystem) indicates which function is mapped on the PE subsystem or it shows that the subsystem is "not used". The white block (ME subsystem) indicates which data is stored on the ME subsystem. For example, in Figure 5.6a, ('F1', 'F3') represents d_{1_3} and ('F2', 'F4') represents d_{2_4}, both are stored on the ME subsystem $me1$.

Table 5.8 shows the execution time, the cost and the reliability level of these found solutions. The second column is composed of the sets of values of $\{\alpha, \beta, \gamma\}$. T_{sys} and C_{platf} are respectively the execution time and cost of the system. The third column show the reliability of the solution corresponding to each value of m (number of periods). It should be noted that since the unit of the failure rate is a failure per 10^9 hours, the value of the reliability here with exactly 9 decimal digits is acceptable. From the reliability, we estimate the number of failed period among m periods: $P_{\text{fail}} = m \times R_{\text{platf}}$.



(a)



(b)

Figure 5.6: result found and shown on the DSE tool by all three search strategies (a) with $\{\alpha = 0.5, \beta = 0.5, \gamma = 0.0\}$. (b) with $\{\alpha = 0.5, \beta = 0.0, \gamma = 0.5\}$.

Table 5.8: reliability, execution time and cost of found solutions in the case study-1.

No	$\{\alpha, \beta, \gamma\}$	R_{platf}	T_{sys}	C_{platf}
1	$\{0.5, 0.5, 0.0\}$	0.99726007	1737 μs	9.0
	Number of failed period (P_{fail1}) among m periods	213056956	N/A	N/A
2	$\{0.5, 0.0, 0.5\}$	0.99649173	2742 μs	4.0
	Number of failed period (P_{fail2}) among m periods	272803075	N/A	N/A
Improvement of No.1 compared to No.2		$\frac{P_{\text{fail2}}-P_{\text{fail1}}}{P_{\text{fail2}}}$	$\frac{T_{\text{sys2}}-T_{\text{sys1}}}{T_{\text{sys2}}}$	$\frac{C_{\text{platf2}}-C_{\text{platf1}}}{C_{\text{platf2}}}$
		21.9%	36.65%	−125%

Table 5.9: failure rate of the system.

Solution	λ_{platf}	Equivalent ASIL
No.1	$\sim 7.3 \times 10^{-8}$ failure per hour	ASIL B or C
No.2	$\sim 6.0 \times 10^{-7}$ failure per hour	ASIL A

Analysis and comments In the solution No.2, we do not see the d_{1_2} because the F_1 and F_2 are mapped on the same PE subsystem and they are executed successively. Therefore, d_{1_2} does not need to be mapped on any ME subsystem. This is similar for d_{3_4}.

Besides, with $\{0.5, 0.0, 0.5\}$ of $\{\alpha, \beta, \gamma\}$, all four functions are mapped on only one PE subsystem, called the solution No.2. This solution gives a favorable cost (number of used components) compared to the solution No.1 (125%). We try to compare the improvement of the solution No.1 above the solution No.2 in the bottom line of Table 5.8. The better timing is achieved with the solution No.1 (36.65%). Although the No.2 uses fewer subsystem numbers, its overall system reliability is still lower than that of the No.1. And, in all values of m, the number of faults of the solution No.1 is only about 21.9% of the fault number of No.2. As such, the correlation between these two solutions can prove partially the rationality in choosing the solution of our DSE framework. As we know, T_{sys} is considered as the execution time of the first period of the application. Thus, after m periods, the worst case execution time of the system can be estimated as in Equation 5.1, where: λ_{platf} and T respectively are the failure rate and the execution time of the system.

$$T \approx m \times T_{\text{sys}} \tag{5.1}$$

Then, assuming that the failure from faults arrival on the whole system follows a Poisson distribution, the reliability of a platform is given by Equation 5.2, where: λ_{platf} and T respectively are the failure rate and the execution time of the system.

$$R_{\text{platf}}(T) = e^{-\lambda_{\text{platf}}.T} \tag{5.2}$$

Therefore, from the value of m, T_{sys}, and R_{platf}, we can estimate the failure rate as the second column in Table 5.9. In comparison to the ASIL standard (mentioned in Chapter 1), we can recognize that the solution No.2 corresponds to Level A. Level A is the lowest level in the 4 safety level of ASIL. If designers need to achieve a higher level of safety, the weight of the system reliability needs to be greater. For example, we try $\{0.99, 0.0, 0.01\}$ for $\{\alpha, \beta, \gamma\}$. The best solution is shown in Table 5.10. As we can see, some subsystems now require a fault tolerance strategy. If using the same the method above to calculate the ASIL level, this solution is at the level D.

Table 5.10: found solutions in the case study-1 with $\{\alpha = 0.99,\ \beta = 0.0,\ \gamma = 0.01\}$.

Sub ID	**Mapped component**	**Mapped element**	**Tolerance**	R_{platf}	T_{sys}	C_{platf}
PE1	Microblaze & softComp	$F_1,\ F_2,\ F_3,\ F_4$	**TMR**	0.99999302	2790 μs	10
ME1	BRAM	$d_{1_3},\ d_{2_4}$	**TMR**			
COMM1	AXI		0			
PE2	Not used	Not used	Not used			
PE3	Not used	Not used	Not used			
PE4	Not used	Not used	Not used			

Table 5.11: three search strategies in the DSE process of Case-study 1.

Name	GA	SA	CS
Probability to find the optimal solution	100%	100%	100%
$\{\alpha = 0.5,\ \beta = 0.5,\ \gamma = 0.0\}$			
Average exploration duration Improvement compared to CS	99.8 seconds 108 times	1498.7 seconds 7 times	3 hours 1 times
$\{\alpha = 0.5,\ \beta = 0.0,\ \gamma = 0.5\}$			
Average exploration duration Improvement compared to CS	275 seconds 39 times	1538.0 seconds 7 times	3 hours 1 times
Total potential solutions	1990656		
Explored solutions	$\approx$ 15000	$\approx$ 70000	1990656
Explored valid solutions	Unknown	Unknown	1029024 (51.7%)

5.2.1.3 Discussion

Table 5.11 shows the index of three strategies integrated with the mapping and evaluation process. In this case study, all three strategies allow to find the optimal solution at a rate of 100%. If using the exhaustive exploration in design space, we need 3 hours to browse through all possible solutions. However, by allowing the integration of different search strategies, our DSE framework greatly reduces the search time.

With SA, the exploration duration is reduced to 7 times that of CS. In particular, the duration by GA is improved more than 39 times compared to CS when GA is used in our framework. In particular, with $\{0.5, 0.5, 0.0\}$, GA is faster than CS 10866 times. However, CS gives a solid assurance when exploring all 1990656 solutions (51.7% of them are valid solutions) while SA only explore up to about 3.5% (70000) of solutions and GA only 0.75% (15000) of them.

Compared to other methods in the literature, our DSE framework allows us to open up a broader solution space. As in the literature (Chapter 2, Subsection 2.2.3), the existed studies are only focused on the fault tolerance of PE components. As such, in Case-study 1, the number of potential solutions is only 497664, that only accounts for 25% of the number of potential solutions compared to our framework. It implies that our DSE framework not only helps to spot more potential solutions but also makes the assessment more comprehensive and the solution found better.

5.2.2 Harris detector

The common parameters in the objective function need to be established as in Table 5.12. In assuming that the considered application processes 100000 photos, in which each photo sizes is 1080×720. Thus, the number of periods of this system (m) is $10^5 \times 1080 \times 720 = 7776 \times 10^7$. We try the optimization

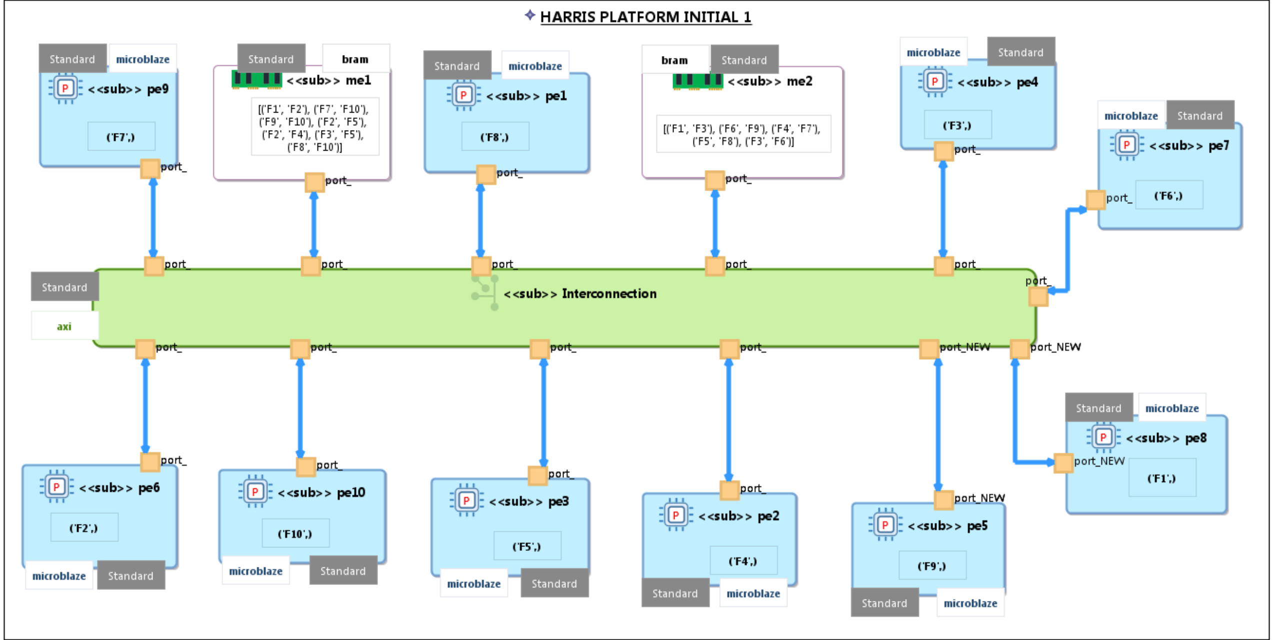


Figure 5.7: initial solution in which each PE subsystem only performs one function, "INITIAL 1", (R_{platf} = 0.93206171, T_{sys} = 14.91 ms, C_{platf} = 23.0).

with 2 different sets of weighting factors. With the objective to improve the reliability, the value of α is always greatest.

Table 5.12: parameters in the objective function of Case-study 2.

Name	Def	Value
m	number of periods	7776×10^7
$T_{\max}$	longest execution time	85.48 ms
$C_{\max}$	most expensive cost	70
$\{\alpha, \beta, \gamma\}$	weighting factors	$\{1.0, 0.0, 0.0\}$ $\{0.5, 0.25, 0.25\}$

Initial solution without using the DSE process

An initial solution is defined as a solution created through the designer's knowledge without using any DSE tool. Depending on the level of different knowledge of the designer, the effectiveness of the initialization solution may vary. In this case, to create an initialization solution without using the DSE process, it's assumed that a designer thinks in a very "naive" way:

- in the first initial state, each PE subsystem only performs one function (as Figure 5.7);
- in the second initial state, all functions are mapped into a single processor (as Figure 5.8).

Reliability, execution time and cost of the initial solutions are calculated according to our method and shown in the caption of each corresponding figure. In the case-study 1, we don't consider an initial solution because that is a simple case. If with the "naive" way, designers can still create an optimal-like

Figure 5.8: initial solution in which all functions are mapped into a single processor, "INITIAL 2", (R_{platf} = 0.90795063, T_{sys} = 30.64 ms, C_{platf} = 4.0).

solution without using our DSE process. Therefore, considering an initial solution and comparing with the optimal results in the previous case-study does not make much sense.

5.2.2.1 Search strategies - parameter setup

In this case-study, the number of solutions is very large:

- 2^{10} configurations for the component mapping (each $subPE$ has 2 possibilities to choose $compPE$);
- $> 10^{10}$ configurations for the function mapping (each function has 10 possibilities to be mapped in a $subPE$);
- 4^{12} configurations for the data mapping (each data has 4 possibilities to be mapped into a $subMemory$);
- $3^{10} \times 2^5$ configurations for the tolerance strategy (each $subPE$ has 3 possibilities to choose a fault-tolerance strategy and each $submemory$ or $subCommunication$ has 2 possibilities to choose a fault-tolerance strategy);
- so, there are more than 3×10^{26} potential solutions in the design space for this case study.

With such a very large design space, finding solutions by CS is very time consuming, so in this case, we only used GA and SA. Large design space has a large number of potential good solutions but at the same time the number of invalid solutions is also great. So, pruning the design space is necessary to reduce exploration time. It's easy to see that we cannot change anything in the application model or the list of components, the only thing that can be changed is the platform template. The initialization

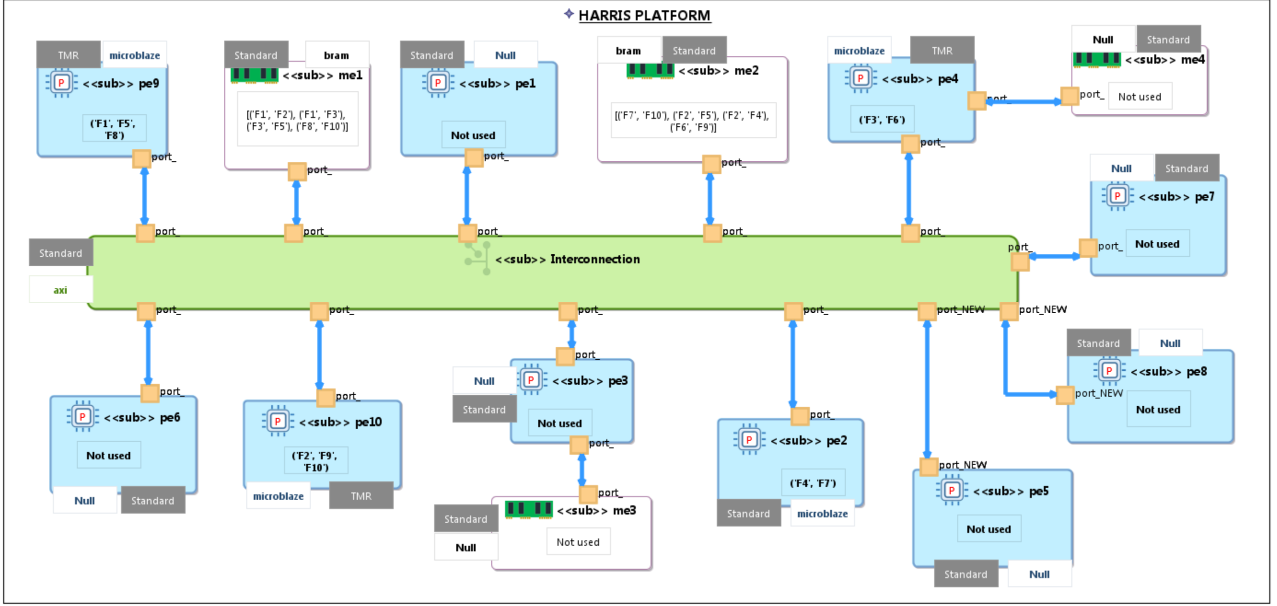


Figure 5.9: good platform found with $\{\alpha = 1.0, \beta = 0.0, \gamma = 0.0\}$ by SA, ($R_{platf} = 0.97406249$, $T_{sys} = 14.33$ ms, $C_{platf} = 20.0$).

template may not be a good template with a lot of unnecessary subsystems. Therefore, we can prune the design space through finding a good template first. We will combine these two search strategies (GA and SA) to explore the design space of the case study. Thus, we try to explore the design space according to the following:

- use SA to find the good platform template from the initial platform template;
- remove all unused subsystems;
- use GA to find the optimal solution with the good platform template.

SA is chosen for the first step because it generally gives a "good" solution (not the "best"). SA is specially suitable for problems where finding an approximate global optimum is more important than finding a precise optimum in finite time [134]. As such, with SA we can find a solution with a "good" template that is close to the "best" template without spending too much time. Then, based on the found template, GA is used to look for the optimal solution. The advantage of GA is to find good quality solutions from a population of points. Therefore, it has the ability to avoid being trapped in local optimal solution better than SA which searches from a single point [134]. So, we choose GA for this second step.

Find the good platform template with SA In this step, the DSE process runs 50 times with SA with the same SA parameters as Case-study 1. Among 50 found results, the result with the best value of the objective function (Equation 4.35) is as to make the platform template.

Figure 5.9 show the solution found by SA with $\{\alpha, \beta, \gamma\} = \{1.0, 0.0, 0.0\}$. We can see that the solution has only four PE subsystems and two ME subsystems. Based on it, Figure 5.11 presents the template used in the DSE process with $\{1.0, 0.0, 0.0\}$ in the next step.

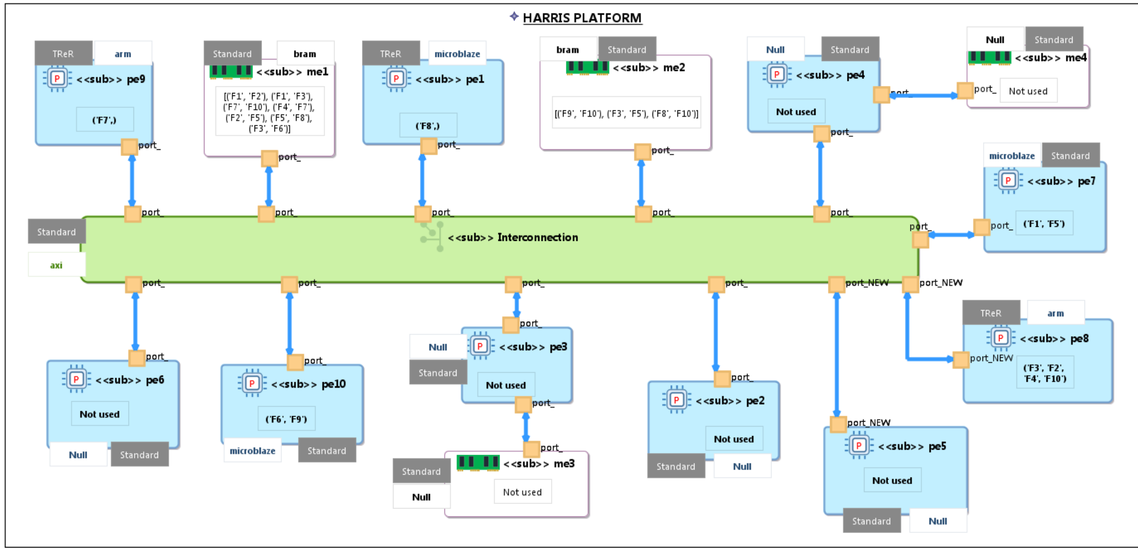


Figure 5.10: good platform found with$\{\alpha = 0.5,\ \beta = 0.25,\ \gamma = 0.25\}$ by SA.
(R_{platf} = 0.90381600, T_{sys} = 8.2 ms, C_{platf} = 16.0).

Figure 5.10 show the solution found by SA with $\{\alpha, \beta, \gamma\} = \{0.5, 0.25, 0.25\}$. We can see that the solution has only five PE subsystems and two ME subsystems. Based on it, Figure 5.12 presents the template used in the DSE process with {0.5, 0.25, 0.25} in the next step.

Find the optimal solution with GA The DSE process runs 10 times with GA with the same GA parameters in Case-study 1 (Table 5.6). The best result among 10 found results is considered as the best solution. The platform templates that are used in this step are presented in Figure 5.11 and 5.12 comes from the previous step.

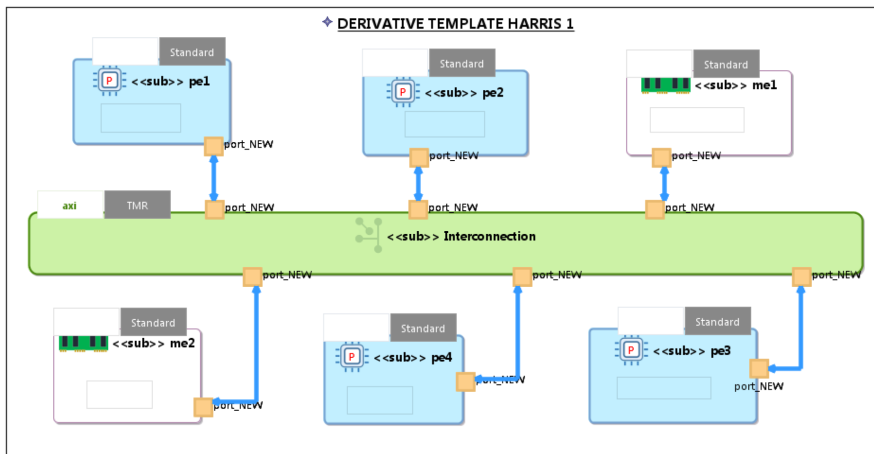


Figure 5.11: platform template used in the DSE process with $\{\alpha = 1.0,\ \beta = 0.0,\ \gamma = 0.0\}$ by GA.

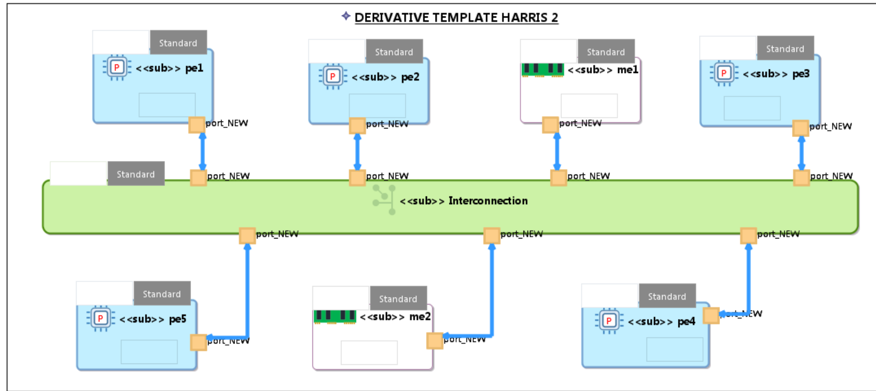


Figure 5.12: platform template used in the DSE process with $\{\alpha = 0.5,\ \beta = 0.25,\ \gamma = 0.25\}$ by GA.

5.2.2.2 Results

Herein, we consider results found with the two sets of values of $\{\alpha,\ \beta,\ \gamma\}$.

The first set is $\{\alpha = 1.0,\ \beta = 0.0,\ \gamma = 0.0\}$

Figure 5.13 shows the best solution found by GA with $\{\alpha = 1.0,\ \beta = 0.0,\ \gamma = 0.0\}$. We can see that all subsystems support TMR. Because the weights of the execution time and the cost are zero, it implies that there is no restriction on the cost or time of implementation, the DSE process seeks only the most reliable solution. Therefore, applying TMR on the entire platform is reasonable. In addition, the ARM processor has a high processing speed but is not used because it can be use only 2 times so TMR cannot be applied on the ARM processor. Thus, there are no PE subsystems that use ARM in this case. Only three PE subsystems are used to execute the functions and only one ME subsystem is used to store the data. Microblaze processors are used in the PE subsystems. The time to find this

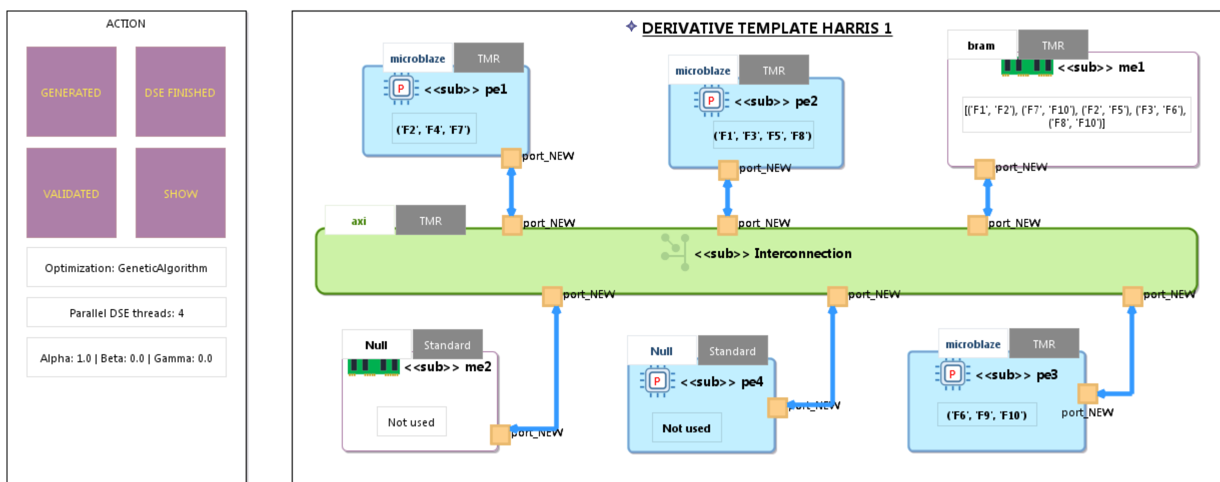


Figure 5.13: result found and shown on the DSE tool with $\{\alpha = 1.0,\ \beta = 0.0,\ \gamma = 0.0\}$, $(R_{platf} = 0.99964876,\ T_{sys} = 1.79\text{ ms},\ C_{platf} = 23.0)$. Exploration time: 11148 s.

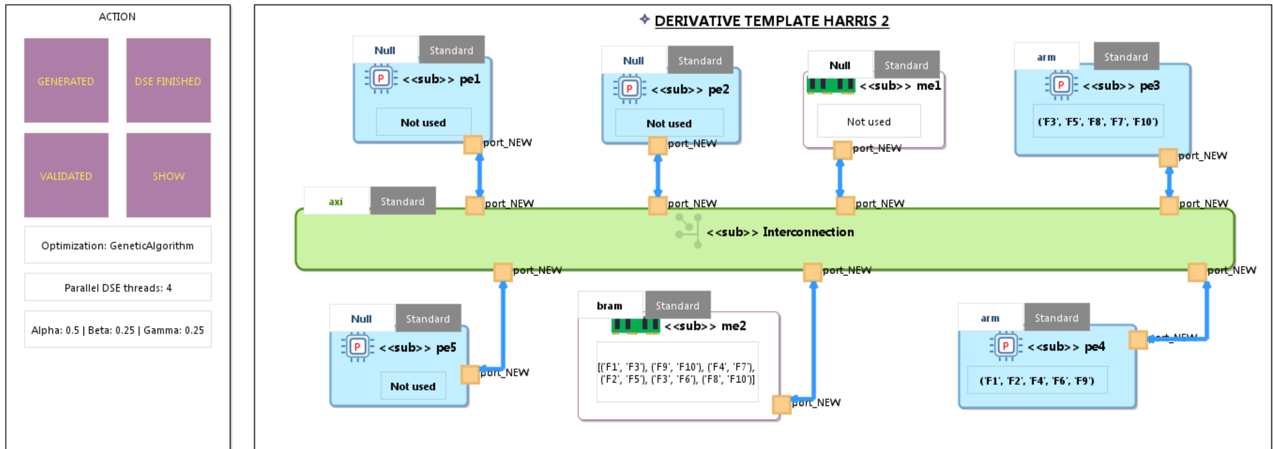


Figure 5.14: result found and shown on the DSE tool with $\{\alpha = 0.5,\ \beta = 0.25,\ \gamma = 0.25\}$, ($R_{platf} = 0.95287649$, $T_{sys} = 13.66$ ms, $C_{platf} = 6.0$). Exploration time: 2 days.

solution is 11148 seconds ($\approx$ 3 hours). Two rightmost columns shows the reliability, execution time and cost of the two "naive" initial solutions. The reliability, execution time and cost of the solution found by DSE process are compared with the initial solutions in Table 5.13.

The second set is $\{\alpha = 0.5,\ \beta = 0.25,\ \gamma = 0.25\}$

Figure 5.14 shows the best solution found by GA with $\{\alpha = 0.5,\ \beta = 0.25,\ \gamma = 0.25\}$. We can see that no fault-tolerance strategy is applied in the platform . Only two PE subsystems with ARM processors are used to execute the functions and only one ME subsystem is used to store the data. The time to find this optimal solution is 2 days. The reliability, execution time and cost of the solution found by DSE process are compared with the initial solutions in Table 5.14.

5.2.2.3 Discussion

From the two result table (Table 5.13 and 5.14), we can see the effectiveness of our DSE framework.

With $\{\alpha = 1.0,\ \beta = 0.0,\ \gamma = 0.0\}$ (Table 5.13), the optimization process finds the solution with the highest reliability. Table 5.15 shows the improvement of the optimal solution compared to the initial solutions. Considering the reliability of the initial solutions is 100%, the reliability of the optimal solution improves 7% compared to INITIAL 1 and 10% compared to INITIAL 2. The INITIAL 1

Table 5.13: reliability, execution time and cost of solutions found by DSE process with $\{\alpha = 1.0,\ \beta = 0.0,\ \gamma = 0.0\}$.

Name	**Solution found by DSE**	**without DSE**	
		INITIAL 1	**INITIAL 2**
Reliability	0.99964876	0.93206171	0.90795063
Execution time	13.66 ms	14.91 ms	30.64
Cost	23.0	23.0	4.0
Objective value	0.99964876	0.93206171	0.90795063
Exploration duration	3 hours	N/A	N/A

Table 5.14: reliability, execution time and cost of solutions found by DSE process with $\{\alpha = 0.5, \beta = 0.25, \gamma = 0.25\}$.

Name	**Solution found by DSE**	**without DSE**	
		INITIAL 1	**INITIAL 2**
Reliability	0.95287649	0.93206171	0.90795063
Execution time	1.79 ms	14.91 ms	30.64 ms
Cost	6.0	23.0	4.0
Objective value	0.93059472	0.76661198	0.83650077
Exploration duration	2 days	N/A	N/A

Table 5.15: improvement of the solution found by DSE process compared to the "naive" initial solutions with $\{\alpha = 1.0, \beta = 0.0, \gamma = 0.0\}$.

Name	**INITIAL 1**	**Improvement percentage solution found by DSE**	**INITIAL 2**	**Improvement percentage solution found by DSE**
Reliability	0%	7.25%	0%	10.09%
Execution time	0%	8.38%	0%	55.41%
Cost	0%	0%	0%	−475%
Objective value	0%	7.25%	0%	10.09%

uses too much PE subsystems so its reliability is lower than INITIAL 2. Both initial solutions have a lowest level of reliability than the optimal solution due to the absence of fault-tolerance strategies. Similarly, in terms of execution time, the found solution improves 8% compared to INITIAL 1 and 55% compared to INITIAL 2. INITIAL 2 has an advantage in terms of cost but in this case, there is no cost and time constraint, so the found solution is still a better solution.

With $\{\alpha = 0.5, \beta = 0.25, \gamma = 0.25\}$ (Table 5.14), the research process find solutions that have a constraint between all three quantities: reliability, execution time, and cost. Table 5.16 shows the improvement of of the optimal solution compared to the initial solutions. The reliability of the solution improves 2.23% compared to INITIAL 1 and 4.94% compared to INITIAL 2. The improvement of the reliability of the found solution in this case is not as good as the previous case because it has constraints on cost and execution time. That is why no fault-tolerance strategy is used in the solution for this case (as shown on Figure 5.14). In terms of execution time, the best solution improves up-to 87% compared to INITIAL 1 and 94% compared to INITIAL 2. In terms of cost, the solution improves 21% compared to INITIAL 1 but its cost is 50% higher than INITIAL 2. Finally, the improvement of the objective function of the optimal solution is better than both initial solutions. Therefore, it can be concluded that our DSE framework really allows finding good solutions even in very complex cases and very large design spaces (more than 3×10^{26}). We can see that TReR does not appear in the

Table 5.16: improvement of the solution found by DSE process compared to the initial solutions with $\{\alpha = 0.5, \beta = 0.25, \gamma = 0.25\}$.

Name	**INITIAL 1**	**Improvement percentage of solution found by DSE**	**INITIAL 2**	**Improvement percentage of solution found by DSE**
Reliability	0%	2.23%	100%	4.94%
Execution time	0%	87.99%	100%	94.15%
Cost	0%	73%	100%	−50%
Objective value	0%	21.39%	100%	11.24%

final results because the execution time increases very high meanwhile the reliability level increases very small. Moreover, if we don't consider the time constraint, the improvement of this fault-tolerance strategy is also negligible compared to TMR in terms of the objective function with the considered $\{\alpha, \beta, \gamma\}$ sets.

5.3 Summary

In this chapter, two case-studies were investigated. The experimental evaluation of our DSE framework has been discussed. It shows the ability to integrate different search strategies into the framework. Furthermore, it also shows the exploration time efficiency in using the framework compared to the exhaustive search. In particular, a deeper design space with more potential solutions than those already available in the literature.

Chapter 6

Conclusions and perspectives

Abstract: This chapter summarizes all the contributions presented in this manuscript and proposes some perspectives to develop this work in the future.

Contents

6.1 Conclusions 120
6.2 Perspectives 121

6.1 Conclusions

In the context of the rising computing need of embedded applications, increasing the complexity of platforms makes the design space larger and contains many potential solutions. Moreover, demand for fault tolerance has also emerged as an urgent problem in particularly important for systems such as cars, airplanes, space, and nuclear operations. Indeed, designing such systems is still a challenge due to the complexity of heterogeneous multiprocessor systems-on-chip architecture which combines parallelism, heterogeneous programming and fault-tolerance integration. Because of the limit about time, programming difficulties, and cost, the growing complexity of those systems is also a problem for designing as the number of design solutions explodes and can no longer be evaluated manually. Therefore, a new set of tools is needed at an higher-level of abstraction in order to abstract the lower-level design complexity, explore, and evaluate automatically design solutions.

The evaluation of a design is strongly dependent on function/task mapping, data mapping, and the choice of hardware/software components. In addition, to enhance fault tolerance, the designs need to incorporate additional fault-tolerance strategies such as spatial redundancy or temporal redundancy. At this time, evaluating a design also depends on the choice of the fault-tolerant strategy.

In this manuscript, we have presented a framework to make possible the design of Heterogeneous Multiprocessor Systems-on-Chip supporting fault-tolerance. It provides designers with a design flow that allows to choose a solution integrating fault tolerance strategies in the design based on their system knowledge. Several contributions were presented to achieve this goal.

First, a platform meta-model is proposed to provide a unified definition as a bridge between the different tools, between different programming languages, and different design stages. Based on it, designers can specify a generic architecture description to cope with the lack of the underlying platform model for MPSoC systems. Especially, this meta-model integrates fault-tolerance aspects with an intermediate level called "subsystem". Through the "subsystems", the fault tolerance parts are connected to the architectural part in the meta-model. This meta-model is like a backbone for the design-tool flow. The multiple levels of the meta-model allow designers to involve in the design by expressing the design constraints according to their levels of expertise.

Second, a mapping process allowing the generation of a design solution has been proposed. The process focuses on four main points: *1)* component mapping, *2)* function mapping, *3)* data mapping, and *4)* fault-tolerance strategy mapping. Especially, the fault tolerance strategies are the main means to increase system reliability. To build a valid design space, every solution must respect rules and constraints. The problem is standardized under the ILP formulations, that is convenient for expressing in mathematical form the problem and also easier for programming of the algorithms.

Third, a set of mathematical equations is established to evaluate the performance criteria of a solution as execution time, cost, and reliability. The evaluation is based on the configuration of a solution obtained from the mapping process.

Fourth, these previous points creates an automated and scalable DSE framework that is composed of a design space generation, an evaluation and an optimization phases. The optimization integrates available methods such as Simulated Annealing, or Genetic Algorithm. The framework is integrated into a Java tool. This tool, with a graphical user interface allows modeling, creates an MPSoC platform and executes the exploration process.

Finally, two case-studies are introduced. Experimental results showed that the DSE framework provides an effective exploration of large design space and results close or equal to comprehensive approach. Moreover, the reliability of the found solutions are better than the solutions built without the framework.

6.2 Perspectives

From the review from Chapte 2, we observe that some steps still need to be considered. So to extend further our work, we propose the followings:

- Extend the performance evaluation method with other criteria such as energy, area, temperature, etc. Our DSE framework considers 3 aspects such as execution time, reliability and cost. However, applying fault-tolerance strategies can affect other criteria. For example, energy and temperature are two of the biggest concerns of the design space exploration studies as we pointed out in Chapter 2. Reliability level is a mathematical function of temperature and so is energy. Therefore, in a process of maximizing the reliability level, energy and temperature of a platform may be affected. Thus, the more aspects that are considered, the more comprehensive the evaluation of solutions is. And then, the objective function needs to be changed to match the designer's concerns;

- Implement optimal fault-tolerance results in a real working environment with real fault-injection scenarios to measure their real reliability. Because of limitations on hardware resources and time limits, this has not been completed within the scope of this thesis. The process of operating in a real environment can provide actual data of the system. Doing so, we could compare the results of this framework with real cases and verify the reliability level of components.

- Apply the presented DSE framework to larger case-study systems. On one hand, we could obtain more information about the capacity, applicability, scalability of the DSE framework. On the other hand, we could improve and update the framework for platforms, applications, and systems including more subsystems, more communication structure and more requirements than the case-studies invested;

- Input data. Parameters of used components greatly influence the quality of an exploration process. Our work uses data from the catalogs of manufacturers like Xilinx. In initial steps of design, this data is appropriate and our DSE process can give designers a view of possible solutions of a platform. However, in order to make better and more accurate designs, input data needs to be invested more carefully. If an input is good, an output may be good;

- Extend the list of fault-tolerance strategies. Our work has only considered two fault-tolerance strategies, TMR and TReR. This is a limitation in our work because many other fault-tolerance strategies are available in the literature and have also been mentioned in our proposed meta-model. However, to integrate more strategies, performance model of newly proposed strategies needs to be set up. Building these performance models requires a further study. The more fault-tolerance strategies are added, the more flexible the framework is and so it meets more and more needs of users. Based on our propsed meta-model, with the constraints and rules we have built, the roadmap for integrating other strategies is possible

- Consider the compromise between time, cost and reliability. The relationship between these three parameters is shown in the results of this manuscript, but it is necessary to look beyond their influence on each other. This allows to build the target function more effectively and therefore the solution given is more accurate. In addition, we can see that the selection of $\{\alpha, \beta, \gamma\}$ is also the reason why TReR is less likely to appear in optimal solutions. This can be explained more clearly if we have further research on the relationship of cost, reliability and execution time;

- Add more optimization-searching methods. Our framework supports for arbitrary integration of different optimization-searching methods. Since there is no perfect method, integrating many different strategies into this framework both help to increase the capacity of the framework and allows designers to choose the search engine they desire;

- Add the configuration of inputs (application and component list) into the graphical user interface of the modeling tool described in Section 3.2. This would be helpful to set the input parameters in the modeling tool;

Bibliography

[1] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: survey of current and emerging trends," in *Proceedings of the 50th Annual Design Automation Conference*, p. 1, ACM, 2013.

[2] M. Hübner and J. Becker, *Multiprocessor system-on-chip: hardware design and tool integration.* Springer Science & Business Media, 2010.

[3] M. Bakhouya, M. Daneshtalab, M. Palesi, and H. Ghasemzadeh, "Many-core system-on-chip: architectures and applications," *Microprocessors and microsystems*, vol. 43, pp. 1–3, 2016.

[4] Z. J. Jia, A. Núñez, T. Bautista, and A. D. Pimentel, "A two-phase design space exploration strategy for system-level real-time application mapping onto mpsoc," *Microprocessors and Microsystems*, vol. 38, no. 1, pp. 9–21, 2014.

[5] P. Kapur, H. Pham, A. Gupta, and P. Jha, *Software reliability assessment with OR applications.* Springer, 2011.

[6] N. E. M. F. T. AIRBUS, Eurocopter, "Reliability methodology for electronic systems," *FIDES guide 2009 Edition A*, September 2010.

[7] U. SPECS, "Military handbook reliability prediction of electronic equipment," *MIL-HDBK-217F*, 2 December 1991.

[8] D. J. Smith and K. G. Simpson, *Safety critical systems handbook: a straight forward guide to functional safety, IEC 61508 (2010 Edition) and related standards, including process IEC 61511 and machinery IEC 62061 and ISO 13849.* Elsevier, 2010.

[9] ISO, "International standard iso 26262," *Geneva*, 2011.

[10] E. Dubrova, *Fault-tolerant design.* Springer, 2013.

[11] T. T. Nguyen, M. Thevenin, A. Mouraud, G. Corre, O. Pasquier, and S. Pillement, "High-level reliability evaluation of reconfiguration-based fault tolerance techniques," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 202–205, IEEE, 2018.

[12] E. Gouno and L. Guérineau, "Failure rate estimation in a dynamic environment," *Economic Quality Control*, vol. 30, no. 1, pp. 1–8, 2015.

[13] S. Scharoba, M. Schölzel, T. Koal, and H. T. Vierhaus, "On reliability estimation for combined transient and permanent fault handling," in *Electronic Conference (BEC), 2014 14th Biennial Baltic*, pp. 73–76, IEEE, 2014.

[14] A. Vallero, A. Savino, S. Tselonis, N. Foutris, M. Kaliorakis, G. Politano, D. Gizopoulos, and S. Di Carlo, "Bayesian network early reliability evaluation analysis for both permanent and transient faults," in *On-Line Testing Symposium (IOLTS), 2015 IEEE 21st International*, pp. 7–12, IEEE, 2015.

[15] H. Aliee, L. Chen, M. Ebrahimi, M. Glaß, F. Khosravi, and M. B. Tahoori, "Towards cross-layer reliability analysis of transient and permanent faults," *arXiv preprint arXiv:1405.2914*, 2014.

[16] N. Foutris, M. Kaliorakis, S. Tselonis, and D. Gizopoulos, "Versatile architecture-level fault injection framework for reliability evaluation: A first report," in *On-Line Testing Symposium (IOLTS), 2014 IEEE 20th International*, pp. 140–145, IEEE, 2014.

[17] "White paper t04007be-3 2009.4, failure mechanism of semiconductor devices.," Accessed: 2019-02-26. https://industrial.panasonic.com/content/data/SC/PDF/ww_aboutus_reliability_t04007be-3.pdf.

[18] D.-C. Hsu, M.-T. Wang, J. Y.-m. Lee, and P.-C. Juan, "Electrical characteristics and reliability properties of metal-oxide-semiconductor field-effect transistors with zr o 2 gate dielectric," *Journal of applied physics*, vol. 101, no. 9, p. 094105, 2007.

[19] P. Suri and P. Raheja, "A study on weibull distribution for estimating the reliability," *International Journal Of Engineering And Computer Science*, vol. 4, no. 07, 2015.

[20] A. K. Verma, A. Srividya, and D. R. Karanki, *Reliability and safety engineering*, vol. 43. Springer, 2010.

[21] A. Das, A. Kumar, and B. Veeravalli, "Aging-aware hardware-software task partitioning for reliable reconfigurable multiprocessor systems," in *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*, pp. 1–10, IEEE, 2013.

[22] V. Gherman, S. Evain, F. Auzanneau, and Y. Bonhomme, "Programmable extended sec-ded codes for memory errors," in *VLSI Test Symposium (VTS), 2011 IEEE 29th*, pp. 140–145, IEEE, 2011.

[23] M. Barbero, F. Jouault, and J. Bézivin, "Model driven management of complex systems: Implementing the macroscope's vision," in *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, pp. 277–286, IEEE, 2008.

[24] G. E. Box, "Robustness in the strategy of scientific model building," in *Robustness in statistics*, pp. 201–236, Elsevier, 1979.

[25] I. Ivanov, J. Bézivin, and M. Aksit, "Technological spaces: An initial appraisal," 2002.

[26] J. Bézivin, "Model driven engineering: An emerging technical space," in *International Summer School on Generative and Transformational Techniques in Software Engineering*, pp. 36–64, Springer, 2005.

[27] A. R. Da Silva, "Model-driven engineering: A survey supported by the unified conceptual model," *Computer Languages, Systems & Structures*, vol. 43, pp. 139–155, 2015.

[28] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *2007 Future of Software Engineering*, pp. 37–54, IEEE Computer Society, 2007.

[29] M. F. d. S. Oliveira, "Model driven engineering methodology for design space exploration of embedded systems," 2013.

[30] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Elsevier, 2010.

[31] C. Atkinson and T. Kuhne, "Model-driven development: a metamodeling foundation," *IEEE software*, vol. 20, no. 5, pp. 36–41, 2003.

[32] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen, *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.

[33] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *Computer*, vol. 36, no. 4, pp. 45–52, 2003.

[34] R. B. Atitallah, L. Bonde, S. Niar, S. Meftali, and J.-L. Dekeyser, "Multilevel mpsoc performance evaluation using mde approach," in *System-on-Chip, 2006. International Symposium on*, pp. 1–4, IEEE, 2006.

[35] O. Labbani, J.-L. Dekeyser, P. Boulet, and É. Rutten, "Introducing control in the gaspard2 data-parallel metamodel: Synchronous approach," in *International Workshop MARTES: Modeling and Analysis of Real-Time and Embedded Systems*, 2005.

[36] F. Herrera, H. Posadas, P. Penil, E. Villar, F. Ferrero, R. Valencia, and G. Palermo, "The complex methodology for uml/marte modeling and design space exploration of embedded systems," *Journal of Systems Architecture*, vol. 60, no. 1, pp. 55–78, 2014.

[37] "Scope website," Accessed: 2018-12-17. http://www.teisa.unican.es/gim/en/scope.

[38] J. Huang, S. Barner, A. Raabe, C. Buckl, and A. Knoll, "A framework for reliability-aware embedded system design on multiprocessor platforms," *Microprocessors and Microsystems*, vol. 38, no. 6, pp. 539–551, 2014.

[39] D. Doering, "A model driven engineering methodology for embedded system designs-hipao2," in *Industrial Informatics (INDIN), 2014 12th IEEE International Conference on*, pp. 787–790, IEEE, 2014.

[40] W. Ahmad, B. M. Yildiz, A. Rensink, and M. I. A. Stoelinga, "A model-driven framework for hardware-software co-design of dataflow applications (extended version)," 2016.

[41] M. F. da Silva Oliveira, *Model-driven engineering methodolgy for design space exploration of embedded systems*. PhD thesis, University of Paderborn, 2014.

[42] E. Carvalho, C. Marcon, N. Calazans, and F. Moraes, "Evaluation of static and dynamic task mapping algorithms in noc-based mpsocs," in *System-on-Chip, 2009. SOC 2009. International Symposium on*, pp. 087–090, IEEE, 2009.

[43] P. K. Sahu and S. Chattopadhyay, "A survey on application mapping strategies for network-on-chip design," *Journal of Systems Architecture*, vol. 59, no. 1, pp. 60–76, 2013.

[44] Y.-J. Chen, C.-L. Yang, and Y.-S. Chang, "An architectural co-synthesis algorithm for energy-aware network-on-chip design," *Journal of Systems Architecture*, vol. 55, no. 5-6, pp. 299–309, 2009.

[45] S. Cao, Z. Salcic, Z. Li, S. Wei, and Y. Ding, "Temperature-aware multi-application mapping on network-on-chip based many-core systems," *Microprocessors and Microsystems*, vol. 46, pp. 149–160, 2016.

[46] D. Li and J. Wu, "Energy-efficient contention-aware application mapping and scheduling on noc-based mpsocs," *Journal of Parallel and Distributed Computing*, vol. 96, pp. 1–11, 2016.

[47] J. González-Domínguez, G. L. Taboada, B. B. Fraguela, M. J. Martín, and J. Tourino, "Automatic mapping of parallel applications on multicore architectures using the servet benchmark suite," *Computers & Electrical Engineering*, vol. 38, no. 2, pp. 258–269, 2012.

[48] X. An, A. Gamatié, and E. Rutten, "High-level design space exploration for adaptive applications on multiprocessor systems-on-chip," *Journal of Systems Architecture*, vol. 61, no. 3-4, pp. 172–184, 2015.

[49] J. Huang, C. Buckl, A. Raabe, and A. Knoll, "Energy-aware task allocation for network-on-chip based heterogeneous multiprocessor systems," in *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pp. 447–454, IEEE, 2011.

[50] H. Elmiligi, F. Gebali, and M. W. El-Kharashi, "Power-aware mapping for 3d-noc designs using genetic algorithms," *Procedia Computer Science*, vol. 34, pp. 538–543, 2014.

[51] K. Pang, V. Fresse, S. Yao, and O. A. De Lima Jr, "Task mapping and mesh topology exploration for an fpga-based network on chip," *Microprocessors and Microsystems*, vol. 39, no. 3, pp. 189–199, 2015.

[52] A. Mosayyebzadeh, M. M. Amiraski, and S. Hessabi, "Thermal and power aware task mapping on 3d network on chip," *Computers & Electrical Engineering*, vol. 51, pp. 157–167, 2016.

[53] P. Mehrvarzy, M. Modarressi, and H. Sarbazi-Azad, "Power-and performance-efficient cluster-based network-on-chip with reconfigurable topology," *Microprocessors and Microsystems*, vol. 46, pp. 122–135, 2016.

[54] B. Ouni, I. Mhedbi, C. Trabelsi, R. B. Atitallah, and C. Belleudy, "Multi-level energy/power-aware design methodology for mpsoc," *Journal of Parallel and Distributed Computing*, vol. 100, pp. 203–215, 2017.

[55] L. Jozwiak, M. Lindwer, R. Corvino, P. Meloni, L. Micconi, J. Madsen, E. Diken, D. Gangadharan, R. Jordans, S. Pomata, *et al.*, "Asam: Automatic architecture synthesis and application mapping," *Microprocessors and Microsystems*, vol. 37, no. 8, pp. 1002–1019, 2013.

[56] M. Arjomand, S. H. Amiri, and H. Sarbazi-Azad, "Efficient genetic based topological mapping using analytical models for on-chip networks," *Journal of Computer and System Sciences*, vol. 79, no. 4, pp. 492–513, 2013.

[57] A. Bonfietti, M. Lombardi, M. Milano, and L. Benini, "Maximum-throughput mapping of sdfgs on multi-core soc platforms," *Journal of Parallel and Distributed Computing*, vol. 73, no. 10, pp. 1337–1350, 2013.

[58] A. Cilardo, D. Socci, and N. Mazzocca, "Asp-based optimized mapping in a simulink-to-mpsoc design flow," *Journal of Systems Architecture*, vol. 60, no. 1, pp. 108–118, 2014.

[59] M. N. S. M. Sayuti and L. S. Indrusiak, "A constructive task mapping algorithm for hard real-time embedded nocs," in *Systems, Process and Control (ICSPC), 2015 IEEE Conference on*, pp. 123–128, IEEE, 2015.

[60] R. Brillu, *Efficient design and programming of Multiple Processors System on Chip architectures*. PhD thesis, 2014.

[61] A. Aravindhan, S. Salini, and G. Lakshminarayanan, "Cluster based application mapping strategy for 2d noc," *Procedia Technology*, vol. 25, pp. 505–512, 2016.

[62] A. K. Singh, W. Jigang, A. Prakash, and T. Srikanthan, "Mapping algorithms for noc-based heterogeneous mpsoc platforms," in *Digital System Design, Architectures, Methods and Tools, 2009. DSD'09. 12th Euromicro Conference on*, pp. 133–140, IEEE, 2009.

[63] A. K. Singh, T. Srikanthan, A. Kumar, and W. Jigang, "Communication-aware heuristics for run-time task mapping on noc-based mpsoc platforms," *Journal of Systems Architecture*, vol. 56, no. 7, pp. 242–255, 2010.

[64] A. J. Page, T. M. Keane, and T. J. Naughton, "Multi-heuristic dynamic task allocation using genetic algorithms in a heterogeneous distributed system," *Journal of parallel and distributed computing*, vol. 70, no. 7, pp. 758–766, 2010.

[65] G. Castilhos, M. Mandelli, L. Ost, and F. G. Moraes, "Hierarchical energy monitoring for task mapping in many-core systems," *Journal of Systems Architecture*, vol. 63, pp. 80–92, 2016.

[66] G. Mariani, G. Palermo, V. Zaccaria, and C. Silvano, "Design-space exploration and runtime resource management for multicores," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, p. 20, 2013.

[67] G. Mariani, G. Palermo, C. Silvano, and V. Zaccaria, "A design space exploration methodology supporting run-time resource management for multi-processor systems-on-chip," in *Application Specific Processors, 2009. SASP'09. IEEE 7th Symposium on*, pp. 21–28, IEEE, 2009.

[68] B. Khodabandeloo, A. Khonsari, F. Gholamian, M. H. Hajiesmaili, A. Mahabadi, and H. Noori, "Scenario-based quasi-static task mapping and scheduling for temperature-efficient mpsoc design under process variation," *Microprocessors and Microsystems*, vol. 38, no. 5, pp. 399–414, 2014.

[69] T. Maqsood, S. Ali, S. U. Malik, and S. A. Madani, "Dynamic task mapping for network-on-chip based systems," *Journal of Systems Architecture*, vol. 61, no. 7, pp. 293–306, 2015.

[70] X. Zhou and Z. Zhu, "A dynamic task mapping algorithm for sdnoc," *Microelectronics Journal*, vol. 63, pp. 58–65, 2017.

[71] C.-H. Huang, C.-Y. Wang, and P.-A. Hsiung, "Elastic superposition task mapping for noc-based reconfigurable systems," *Microprocessors and Microsystems*, vol. 51, pp. 297–312, 2017.

[72] N. Chatterjee, S. Paul, P. Mukherjee, and S. Chattopadhyay, "Deadline and energy aware dynamic task mapping and scheduling for network-on-chip based multi-core platform," *Journal of Systems Architecture*, vol. 74, pp. 61–77, 2017.

[73] M. F. Reza, D. Zhao, H. Wu, and M. Bayoumi, "Hotspot-aware task-resource co-allocation for heterogeneous many-core networks-on-chip," *Computers & Electrical Engineering*, vol. 68, pp. 581–602, 2018.

[74] W. Quan and A. D. Pimentel, "Scenario-based run-time adaptive mpsoc systems," *Journal of Systems Architecture*, vol. 62, pp. 12–23, 2016.

[75] L. Huang, F. Yuan, and Q. Xu, "Lifetime reliability-aware task allocation and scheduling for mpsoc platforms," in *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.*, pp. 51–56, IEEE, 2009.

[76] C. Lee, H. Kim, H.-w. Park, S. Kim, H. Oh, and S. Ha, "A task remapping technique for reliable multi-core embedded systems," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 307–316, ACM, 2010.

[77] R. A. Shafik, B. M. Al-Hashimi, and J. S. Reeve, "System-level design optimization of reliable and low power multiprocessor system-on-chip," *Microelectronics Reliability*, vol. 52, no. 8, pp. 1735–1748, 2012.

[78] S. Aminzadeh and A. Ejlali, "A comparative study of system-level energy management methods for fault-tolerant hard real-time systems," *IEEE transactions on computers*, vol. 60, no. 9, pp. 1288–1299, 2011.

[79] C. Bolchini and A. Miele, "Reliability-driven system-level synthesis for mixed-critical embedded systems," *IEEE Transactions on Computers*, vol. 62, no. 12, pp. 2489–2502, 2013.

[80] S.-H. Kang, H. Yang, S. Kim, I. Bacivarov, S. Ha, and L. Thiele, "Reliability-aware mapping optimization of multi-core systems with mixed-criticality," in *Proceedings of the conference on Design, Automation & Test in Europe*, p. 327, European Design and Automation Association, 2014.

[81] S.-h. Kang, H. Yang, S. Kim, I. Bacivarov, S. Ha, and L. Thiele, "Static mapping of mixed-critical applications for fault-tolerant mpsocs," in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2014.

[82] H. Gall, "Functional safety iec 61508/iec 61511 the impact to certification and the user," in *2008 IEEE/ACS International Conference on Computer Systems and Applications*, pp. 1027–1031, IEEE, 2008.

[83] J. Huang, *Towards an Integrated Framework for Reliability-Aware Embedded System Design on Multiprocessor System-on-Chips.* PhD thesis, Technical University Munich, 2014.

[84] A. Das, A. Kumar, and B. Veeravalli, "Reliability-driven task mapping for lifetime extension of networks-on-chip based multiprocessor systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 689–694, EDA Consortium, 2013.

[85] A. Das, A. Kumar, and B. Veeravalli, "Communication and migration energy aware task mapping for reliable multiprocessor systems," *Future Generation Computer Systems*, vol. 30, pp. 216–228, 2014.

[86] N. Chatterjee, S. Paul, and S. Chattopadhyay, "Task mapping and scheduling for network-on-chip based multi-core platform with transient faults," *Journal of Systems Architecture*, 2018.

[87] L. Huang and Q. Xu, "Energy-efficient task allocation and scheduling for multi-mode mpsocs under lifetime reliability constraint," in *Proceedings of the conference on Design, automation and test in Europe*, pp. 1584–1589, European Design and Automation Association, 2010.

[88] O. Derin, D. Kabakci, and L. Fiorin, "Online task remapping strategies for fault-tolerant network-on-chip multiprocessors," in *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, pp. 129–136, ACM, 2011.

[89] B. H. Meyer, A. S. Hartman, and D. E. Thomas, "Cost-effective slack allocation for lifetime improvement in noc-based mpsocs," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pp. 1596–1601, IEEE, 2010.

[90] A. S. Hartman, D. E. Thomas, and B. H. Meyer, "A case for lifetime-aware task mapping in embedded chip multiprocessors," in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pp. 145–154, IEEE, 2010.

[91] C.-L. Chou and R. Marculescu, "Farm: Fault-aware resource management in noc-based multiprocessor platforms," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pp. 1–6, IEEE, 2011.

[92] P. Axer, M. Sebastian, and R. Ernst, "Reliability analysis for mpsocs with mixed-critical, hard real-time constraints," in *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 149–158, ACM, 2011.

[93] C. Ababei, H. S. Kia, O. P. Yadav, and J. Hu, "Energy and reliability oriented mapping for regular networks-on-chip," in *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, pp. 121–128, ACM, 2011.

[94] I. Ukhov, M. Bao, P. Eles, and Z. Peng, "Steady-state dynamic temperature analysis and reliability optimization for embedded multiprocessor systems," in *Proceedings of the 49th Annual Design Automation Conference*, pp. 197–204, ACM, 2012.

[95] S. Tosun, "Energy-and reliability-aware task scheduling onto heterogeneous mpsoc architectures," *The Journal of Supercomputing*, vol. 62, no. 1, pp. 265–289, 2012.

[96] H. R. Faragardi, R. Shojaee, and N. Yazdani, "Reliability-aware task allocation in distributed computing systems using hybrid simulated annealing and tabu search," in *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pp. 1088–1095, IEEE, 2012.

[97] F. Khalili and H. R. Zarandi, "A fault-tolerant low-energy multi-application mapping onto noc-based multiprocessors," in *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*, pp. 421–428, IEEE, 2012.

[98] F. Khalili and H. R. Zarandi, "A reliability-aware multi-application mapping technique in networks-on-chip," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pp. 478–485, IEEE, 2013.

[99] F. Bolanos, F. Rivera, J. E. Aedo, and N. Bagherzadeh, "From uml specifications to mapping and scheduling of tasks into a noc, with reliability considerations," *Journal of Systems Architecture*, vol. 59, no. 7, pp. 429–440, 2013.

[100] A. Mahabadi, S. Zahedi, and A. Khonsari, "Reliable energy-aware application mapping and voltage–frequency island partitioning for gals-based noc," *Journal of Computer and System Sciences*, vol. 79, no. 4, pp. 457–474, 2013.

[101] A. Das, A. Kumar, and B. Veeravalli, "Energy-aware communication and remapping of tasks for reliable multimedia multiprocessor systems," in *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pp. 564–571, IEEE, 2012.

[102] A. Das, A. Kumar, B. Veeravalli, C. Bolchini, and A. Miele, "Combined dvfs and mapping exploration for lifetime and soft-error susceptibility improvement in mpsocs," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pp. 1–6, IEEE, 2014.

[103] A. Das, A. Kumar, and B. Veeravalli, "Reliability and energy-aware mapping and scheduling of multimedia applications on multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 869–884, 2016.

[104] A. Das, A. Kumar, and B. Veeravalli, "Temperature aware energy-reliability trade-offs for mapping of throughput-constrained applications on multimedia mpsocs," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pp. 1–6, IEEE, 2014.

[105] L. Zhang, K. Li, Y. Xu, J. Mei, F. Zhang, and K. Li, "Maximizing reliability with energy conservation for parallel task scheduling in a heterogeneous cluster," *Information Sciences*, vol. 319, pp. 113–131, 2015.

[106] L. Zhang, K. Li, K. Li, and Y. Xu, "Joint optimization of energy efficiency and system reliability for precedence constrained tasks in heterogeneous systems," *International Journal of Electrical Power & Energy Systems*, vol. 78, pp. 499–512, 2016.

[107] X. Xiao, G. Xie, C. Xu, C. Fan, R. Li, and K. Li, "Maximizing reliability of energy constrained parallel applications on heterogeneous distributed systems," *Journal of Computational Science*, 2017.

[108] A. Namazi, M. Abdollahi, S. Safari, and S. Mohammadi, "A majority-based reliability-aware task mapping in high-performance homogenous noc architectures," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 17, no. 1, p. 28, 2018.

[109] A. Namazi, M. Abdollahi, S. Safari, and S. Mohammadi, "Lorap: Low-overhead power and reliability-aware task mapping based on instruction footprint for real-time applications," in *Digital System Design (DSD), 2017 Euromicro Conference on*, pp. 364–367, IEEE, 2017.

[110] N. K. R. Becchu, V. M. Harishchandra, and N. K. Y. Balachandra, "System level fault-tolerance core mapping and fpga-based verification of noc," *Microelectronics Journal*, vol. 70, pp. 16–26, 2017.

[111] B. N. K. Reddy, M. Vasantha, and Y. N. Kumar, "An energy-efficient fault-aware core mapping in mesh-based network on chip systems," *Journal of Network and Computer Applications*, 2017.

[112] N. K. R. Beechu, V. M. Harishchandra, and N. K. Y. Balachandra, "High-performance and energy-efficient fault-tolerance core mapping in noc," *Sustainable Computing: Informatics and Systems*, vol. 16, pp. 1–10, 2017.

[113] N. Chatterjee, P. Mukherjee, and S. Chattopadhyay, "Reliability-aware application mapping onto mesh based network-on-chip," *Integration*, vol. 62, pp. 92 – 113, 2018.

[114] W. Gao, Z. Qian, and P. Zhou, "Reliability-and performance-driven mapping for regular 3d nocs using a novel latency model and simulated allocation," *Integration*, 2018.

[115] R. Soltani, "Reliability optimization of binary state non-repairable systems: A state of the art survey," *International Journal of Industrial Engineering Computations*, vol. 5, no. 3, pp. 339–364, 2014.

[116] H. Mushtaq, Z. Al-Ars, and K. Bertels, "Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems," in *Design and Test Workshop (IDT), 2011 IEEE 6th International*, pp. 12–17, IEEE, 2011.

[117] T. T. Nguyen, A. Mouraud, M. Thevenin, G. Corre, O. Pasquier, and S. Pillement, "Model-driven reliability evaluation for mpsoc design," in *Design and Architectures for Signal and Image Processing (DASIP), 2017 Conference on*, pp. 1–6, IEEE, 2017.

[118] V. Viyović, M. Maksimović, and B. Perisić, "Sirius: A rapid development of dsm graphical editor," in *Intelligent Engineering Systems (INES), 2014 18th International Conference on*, pp. 233–238, IEEE, 2014.

[119] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[120] J. Oh and C. Wu, "Genetic-algorithm-based real-time task scheduling with multiple goals," *Journal of systems and software*, vol. 71, no. 3, pp. 245–258, 2004.

[121] R. T. Marler and J. S. Arora, "Survey of multi-objective optimization methods for engineering," *Structural and multidisciplinary optimization*, vol. 26, no. 6, pp. 369–395, 2004.

[122] I. Giagkiozis and P. J. Fleming, "Methods for multi-objective optimization: An analysis," *Information Sciences*, vol. 293, pp. 338–350, 2015.

[123] K.-L. Du and M. Swamy, *Search and Optimization by Metaheuristics*. Springer, 2016.

[124] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, jul 2012.

[125] e. m. AVNET, "Zedboard (zynq evaluation and development) hardware user's guide," *Version 2.2*, 27-Jan-2014.

[126] X. DS180, "7 series fpgas data sheet: Overview," vol. V2.6, 2018.

[127] X. UG984, "Microblaze processor reference guide," *v2014.1*, 2014.

[128] X. UG116, "Device reliability report," *First Half*, 2018.

[129] Xilinx, "Microblaze triple modular redundancy (tmr) subsystem," *Vivado Design Suite PG268 v1.0*, November 14, 2018.

[130] K. G. Derpanis, "The harris corner detector," *York University*, 2004.

[131] H. Mike and S. Neil, "Designing embedded systems for high reliability with the 66ak2gx dsp + arm processor," *Texas Instruments*, May 2016.

[132] A. E. Eiben and S. K. Smit, *Evolutionary Algorithm Parameters and Methods to Tune Them*, pp. 15–36. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[133] W. Ben-Ameur, "Computing the initial temperature of simulated annealing," *Computational Optimization and Applications*, vol. 29, no. 3, pp. 369–385, 2004.

[134] E.-G. Talbi, *Metaheuristics: from design to implementation*, vol. 74. John Wiley & Sons, 2009.

Publications

1 Nguyen, T. T., Mouraud, A., Thevenin, M., Corre, G., Pasquier, O., & Pillement, S. (2017, September). Model-driven reliability evaluation for MPSoC design. In Design and Architectures for Signal and Image Processing (DASIP), 2017 Conference on (pp. 1-6). IEEE.

2 Poster - Nguyen, T. T., Thevenin, M., Mouraud, A., Corre, G., Pasquier, O., & Pillement, S. (2018, May). High-level reliability evaluation of reconfiguration-based fault tolerance techniques. Reconfigurable Architectures Workshop (RAW), 2018 Workshop. IEEE.

3 Poster - 12th Colloque2017 de GDR SoC/SiPa Bordeaux, FRANCE.

4 Poster - Ecole d'hiver Francophone sur les Technologies de Conception des Systemes embarques Heterogenes FETCH 2018.

Appendices

Appendix A

Code implementation

The code in this section is implemented in C language and run on Zedboard (Zynq Evaluation & Development Board) ??. The tool flow that is used is composed of Vivado HLx 2016.3 and Vivado SDK 2016.3.

A.1 Sobel filter code implementation

```
/* in_data is the data matrix stored in the DDRAM */
/* [y, x] is the coordinates of a pixel on the image */
float sobel_GP(float** indata, int x, int y){
  int i, j;  /* Loop variable */
  float get_pixel[9][9];
  for (i = -4; i < 5; i++) {
    for (j = -4; j < 5; j++) {
      if((x < 4) || (y < 4) || (x > x_size1 - 4 ) || (y > y_size1 - 4)){
        /* if pixel with the coordinates [y+i][x+j] does not exist, its value is set to
     0 by default */
        get_pixel[i+1][j+1] = 0.0;
      }
      else{
        get_pixel[i+1][j+1] = indata[y+i][x+j];
      }
    }
  }
  return get_pixel;
}
```

Listing A.1: "Get pixel" function

```
float sobel_GX(float *p_block9x9)
{
  /* Definition of Sobel filter in X direction */
  float weight[9][9] = {{ 1,     6,     14,     14 ,     0,    -14 ,  -14 ,    -6 ,    -1},
      {8 ,    48 ,   112,    112,      0 , -112,   -112,    -48,     -8},
      {28,    168,    392,    392,      0,  -392,   -392,  -168,    -28},
      {56,    336,    784,    784,      0,  -784,   -784,  -336,    -56},
      {70,    420,    980,    980 ,     0 , -980 ,  -980,  -420,    -70},
      {56 ,   336,    784 ,   784 ,     0 , -784 ,  -784,  -336 ,   -56},
      {28 ,   168 ,   392 ,   392,      0 , -392,   -392,  -168 ,   -28},
      { 8 ,    48 ,   112 ,   112  ,    0 , -112,   -112 ,   -48  ,   -8},
```

```
12       { 1   ,    6 ,    14   ,  14 ,     0 ,  -14   , -14 ,    -6   ,  -1 }};
13   float value_gx = 0;
14   int i, j;  /* Loop variable */
15   /* Linear transformation */
16   for (i = -4; i < 5; i++) {
17     for (j = -4; j < 5; j++) {
18       //Calculate GX
19       value_gx += *(p_block9x9 + (i+4)*8 + j + 4) * weight[i + 4][j + 4] ;
20     }
21   }
22   return value;
23 }
```

Listing A.2: "Gradient in horizontal axis" function

```
1 float sobel_GY(float *p_block9x9)
2 {
3   /* Definition of Sobel filter in Y direction */
4   float weight[9][9] = {{ 1,      6,     14,     14 ,     0,    -14 ,  -14 ,    -6 ,    -1},
5       {8 ,    48 ,  112,   112,      0 , -112,  -112,   -48,     -8},
6       {28,   168,   392,   392,      0,  -392,  -392,  -168,    -28},
7       {56,   336,   784,   784,      0,  -784,  -784,  -336,    -56},
8       {70,   420,   980,   980 ,     0 , -980 , -980,  -420,    -70},
9       {56 ,  336,   784 ,  784 ,     0 , -784 , -784,  -336 ,   -56},
10      {28 ,  168 ,  392 ,  392,      0 , -392,  -392,  -168 ,   -28},
11      { 8 ,   48 ,  112 ,  112   ,   0 , -112,  -112 ,  -48   ,  -8},
12      { 1   ,    6 ,    14   ,  14 ,     0 ,  -14   , -14 ,    -6   ,  -1 }};
13   float value_gy = 0;
14   int i, j;  /* Loop variable */
15   /* Linear transformtion */
16   for (i = -4; i < 5; i++) {
17     for (j = -4; j < 5; j++) {
18       //Calculate GY
19       value_gy += *(p_block9x9 + (i+4)*8 + j + 4) * weight[j + 4][i + 4] ;
20     }
21   }
22   return value_gy;
23 }
```

Listing A.3: "Gradient in vertical axis" function

```
1 int sobel_abs(float valueX, float valueY)
2 {
3   float value_abs = 0;
4   value_abs = sqrt(pow(valueX, 2) + pow(valueY, 2));
5   return value_abs;
6 }
```

Listing A.4: "Gradient magnitude" function

A.2 Harris conner dectector code implementation

```
1 /* in_data is the data matrix stored in the DDRAM */
2 /* [y, x] is the coordinates of a pixel on the image */
3 float harris_GP(float** indata, int x, int y){
4   int i, j;  /* Loop variable */
5   float get_pixel[9][9];
```

```
 6    for (i = -4; i < 5; i++) {
 7      for (j = -4; j < 5; j++) {
 8        if((x < 4) || (y < 4) || (x > x_size1 - 4 ) || (y > y_size1 - 4)){
 9          /* if pixel with the coordinates [y+i][x+j] does not exist, its value is set to
       0 by default */
10          get_pixel[i+1][j+1] = 0.0;
11        }
12        else{
13          get_pixel[i+1][j+1] = indata[y+i][x+j];
14        }
15      }
16    }
17    return get_pixel;
18  }
```

Listing A.5: "Get pixel" function

```
 1  float** cal_GX(float** p_block9x9)
 2  {
 3    /* Definition of filter in horizontal direction */
 4    float weight[9][9] = {{ 1,    6,    14,    14 ,    0,   -14 ,  -14 ,   -6 ,   -1},
 5        {8 ,   48 ,  112,   112,     0 , -112,  -112,   -48,    -8},
 6        {28,   168,   392,   392,     0,  -392,  -392,  -168,   -28},
 7        {56,   336,   784,   784,     0,  -784,  -784,  -336,   -56},
 8        {70,   420,   980,   980 ,    0 , -980 , -980,  -420,   -70},
 9        {56 ,  336,   784 ,  784 ,    0 , -784 , -784,  -336 ,  -56},
10        {28 ,  168 ,  392 ,  392,     0 , -392,  -392,  -168 ,  -28},
11        { 8 ,   48 ,  112 ,  112  ,   0 , -112,  -112 ,  -48  ,  -8},
12        { 1   ,   6 ,   14  ,  14 ,    0 ,  -14  , -14 ,   -6  ,  -1 }};
13    float* values = calloc(9*9, sizeof(float));
14    float** Gx = malloc(9*sizeof(float*));
15    int i, j, k;  /* Loop variable */
16    for (i=0; i<9; ++i)
17    {
18      Gx[i] = values + i*9;
19    }
20    float sum = 0;
21    /* linear transformation */
22    for (i = 0; i < 9; i++) {
23      for (j = 0; j < 9; j++) {
24        for (k = 0; k < 9; k++) {
25          sum = sum + p_block9x9[i][k]*weight[k][j];
26        }
27        Gx[i][j] = sum;
28        sum = 0;
29      }
30    }
31    return Gx;
32  }
```

Listing A.6: "Gradient in horizontal axis" - Gx function

```
 1  float** cal_GY(float** p_block9x9)
 2  {
 3    /* Definition of filter in vertical direction */
 4    float weight[9][9] = {{ 1,    6,    14,    14 ,    0,   -14 ,  -14 ,   -6 ,   -1},
 5        {8 ,   48 ,  112,   112,     0 , -112,  -112,   -48,    -8},
```

```
6       {28,   168,   392,   392,     0,  -392,  -392,  -168,   -28},
7       {56,   336,   784,   784,     0,  -784,  -784,  -336,   -56},
8       {70,   420,   980,   980 ,    0 , -980 , -980,  -420,   -70},
9       {56 ,  336,   784 ,  784 ,    0 , -784 , -784,  -336 ,  -56},
10      {28 ,  168 ,  392 ,  392,     0 , -392,  -392,  -168 ,  -28},
11      { 8 ,   48 ,  112 ,  112  ,   0 , -112,  -112 ,  -48  ,  -8},
12      { 1   ,   6 ,   14   ,  14 ,    0 ,  -14  , -14 ,   -6  ,  -1 }};
13    float* values = calloc(9*9, sizeof(float));
14    float** Gy = malloc(9*sizeof(float*));
15    int i, j, k;  /* Loop variable */
16    for (i=0; i<9; ++i)
17    {
18      Gy[i] = values + i*9;
19    }
20    float sum = 0;
21    /*linear transformation */
22    for (i = 0; i < 9; i++) {
23      for (j = 0; j < 9; j++) {
24        for (k = 0; k < 9; k++) {
25          sum = sum + p_block9x9[i][k]*weight[j][k];
26        }
27        Gy[i][j] = sum;
28        sum = 0;
29      }
30    }
31    return Gy;
32  }
```

Listing A.7: "Gradient in vertical axis" - Gy function

```
1  float** cal_IxI(float** p_block9x9)
2  {
3    float* values = calloc(9*9, sizeof(float));
4    float** IxI = malloc(9*sizeof(float*));
5    int i, j;  /* Loop variable */
6    for (i=0; i<9; ++i)
7    {
8      IxI[i] = values + i*9;
9    }
10   /* Generation of image2 after linear transformtion */
11   for (i = 0; i < 9; i++) {
12     for (j = 0; j < 9; j++) {
13       //Calculate I x I
14       IxI[i][j] = pow(p_block9x9[i][j], 2) ;
15     }
16   }
17   return IxI;
18 }
```

Listing A.8: "Matrix product X vs X" - Ixx and Iyy functions

```
1  float** cal_IxJ(float** p1_block9x9, float** p2_block9x9)
2  {
3    float* values = calloc(9*9, sizeof(float));
4    float** IxJ = malloc(9*sizeof(float*));
5    int i, j;  /* Loop variable */
6    for (i=0; i<9; ++i)
```

```
7   {
8     IxJ[i] = values + i*9;
9   }
10  for (i = 0; i < 9; i++) {
11    for (j = 0; j < 9; j++) {
12      //Calculate I x J
13      IxJ[i][j] = p1_block9x9[i][j] * p2_block9x9[i][j];
14    }
15  }
16  return IxJ;
17 }
```

Listing A.9: "Matrix product X vs Y" - Ixy function

```
1 float cal_Sum(float** p_block9x9)
2 {
3   float value;
4   int i, j;  /* Loop variable */
5   for (i = 0; i < 9; i++) {
6     for (j = 0; j < 9; j++) {
7       //Calculate sum of all elements of matrix
8       value += p_block9x9[i][j];
9     }
10  }
11  return value;
12 }
```

Listing A.10: "Matrix sum" - Sxx, Sxy and Syy functions

```
1 float cal_response(float* xx, float* xy, float* yy)
2 {
3   float r = 0;
4   float det = (*xx)*(*yy) - pow(*xy, 2);
5   float trace = (*xx) + (*yy);
6   r = det - 0.04*(pow(trace, 2));
7   return r;
8 }
```

Listing A.11: "Harris response" - R function

A.3 Latency measurement on Microblaze and ARM

```
1 #include "xtime_l.h"
2
3 int main()
4 {
5   XTime start, stop;
6   int clk_number = 0;
7
8   XTime_GetTime(&start); // starting time
9
10  /* code to measure the execution time is placed here */
11
12  XTime_GetTime(&stop); // end count
13  clk_number =  stop - start; // number of clock
14
15  return 0;
```

```
}
```

Listing A.12: Code implementation for the execution-time measurement on ARM processor

```
#include "xtmrctr.h"

int main()
{
  XTmrCtr *timer;
  u32 start, stop;
  int TIMER_COUNTER = 0;
  int clk_number = 0;
  int init_status;

  init_status = XTmrCtr_Initialize(&timer, XPAR_AXI_TIMER_0_BASEADDR);
  XTmrCtr_SetResetValue(&timer, TIMER_COUNTER, 0);
  XTmrCtr_Start(&timer, TIMER_COUNTER);
  start = XTmrCtr_GetValue(&timer, TIMER_COUNTER);  // starting time

  /* code to measure the execution time is placed here */

  stop = XTmrCtr_GetValue(&timer, TIMER_COUNTER); // end count
  clk_number = stop - start;  // number of clock

  return 0;
}
```

Listing A.13: Code implementation for the execution-time measurement on Microblaze

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE NANTES

Titre : Exploration architecturale pour la tolérance aux fautes

Mots clés : Ingénierie dirigée par les modèles, exploration de l'espace de conception, tolérance aux fautes, MPSoC

Résumé : La fiabilité devient une caractéristique très importante du processus de conception d'un système embarqué. Par conséquent, l'élaboration de stratégies de tolérance aux fautes fait également partie des priorités lors des premières phases de conception des systèmes embarqués. Cette thèse vise à établir un cadre permettant de trouver la meilleure solution de plate-forme pour une application donnée dans des systèmes multiprocesseurs hétérogènes. La solution trouvée doit être intégrée à la tolérance aux fautes. Un nouveau méta-modèle de plate-forme intégrant la tolérance aux fautes est présenté qui joue le rôle d'infrastructure pour construire des modèles. Les modèles sont ensuite entrés dans un processus d'exploration de l'espace de conception. Dans la spécification utilisateur, les dimensions explorées incluent le choix du composant, le mapping des tâches, le mapping des données et le choix de la stratégie de tolérance aux fautes. Une nouvelle solution est générée et évaluée en matière de temps d'exécution, de coût et de niveau de fiabilité. Ensuite, un processus d'optimisation explore la meilleure solution parmi les espaces de conception. Un nouvel outil avec une interface utilisateur graphique permet de modéliser et d'exécuter le processus d'exploration. Il simplifie le processus en interagissant avec l'utilisateur via l'interface graphique et en automatisant le processus d'exploration de l'espace de conception. L'évaluation de la plate-forme MPSoC hétérogène sous l'impact de fautes transitoires et permanentes est une partie très importante de l'exploration pour aider des concepteurs à choisir la stratégie de tolérance aux fautes appropriée en ce qui concerne un compromis avec les exigences de l'application. Enfin, des études de cas sont investies. Les résultats expérimentaux ont montré que le cadre DSE fournit une exploration efficace de grands espaces de conception.

Title : Model-driven architecture exploration for fault tolerance improvement

Keywords : Model Driven Engineering, Design Space Exploration, Fault Tolerance, MPSoC

Abstract: Reliability becomes a very important feature in the design process of an embedded system. Therefore, the development of fault tolerance strategies is also among the priorities in the early design phases of embedded systems. This thesis aims to establish a framework that allows finding the best platform solution for a given application in heterogeneous Multi-Processor System-on-Chip (MPSoC) systems. The found solution must be integrated the fault tolerance.

A new platform meta-model integrated the fault tolerance is presented that roles an infrastructure to build models. The models are then inputs to a Design Space Exploration process. From the user specification, explored dimensions include hardware choice, task mapping, data mapping, and fault-tolerance-strategy choice. A new solution is generated and evaluated in terms of execution time, cost and, reliability level. Then, an optimization process will explore the best solution among the design space. A new tool with a graphical user interface allows to model and run the DSE process. It simplifies the process by interacting with the user through the graphical interface and automating the process of exploring design space. Evaluation of heterogeneous MPSoC platform under the impact of transient and permanent faults is a very important part of the DSE to help designers choose the appropriate strategy fault tolerance regarding a compromise with the requirements of the application. Finally, case-studies are invested. Experimental results showed that the DSE framework provides an effective exploration of large design space.