

Variability-intensive applications over highly-configurable platforms : Early feasibility and optimality analysis

Sami Lazreg

▶ **To cite this version:**

Sami Lazreg. Variability-intensive applications over highly-configurable platforms : Early feasibility and optimality analysis. Embedded Systems. Université Côte d'Azur, 2020. English. NNT : 2020COAZ4070 . tel-03197885

HAL Id: tel-03197885
https://theses.hal.science/tel-03197885

Submitted on 14 Apr 2021

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

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

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Applications Variables sur Plateformes Configurables : Analyse anticipée de faisabilité et d'optimalité

Sami Lazreg

Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis (I3S)

Présentée en vue de l'obtention du grade de docteur en Informatique de l'Université Côte d'Azur.

Dirigée par : Philippe Collet, Professeur des Universités, Université Côte d'Azur

Co-encadrée par : Sébastien Mosser, Professeur, Université du Quebec à Montreal

Soutenue le : 04/12/2020

Devant le jury, composé de :

- Laurence Duchien, Professeure des Universités, Université de Lille
- Olivier Barais, Professeur des Universités, Université de Rennes
- Mireille Blay-Fornarino, Professeur des Université, Université Côte d'Azur
- Houari Sarhaoui, Professeur, Université de Montreal
- Olivier Bantiche-Kamensky, Ingénieur de Recherche, Renault Software Labs

Visteon

Applications Variables sur Plateformes Configurables : Analyse anticipée de faisabilité et d'optimalité

Jury :

Directeur de thèse et Co-encadrant :

- Philippe Collet, Professeur des Universités, Université Côte d'Azur
- Sébastien Mosser, Professeur, Université du Quebec à Montréal

Rapporteurs :

- Laurence Duchien, Professeur des Universités, Université de Lille
- Olivier Barais, Professeur des Universités, Université de Rennes

Examinateurs :

- Mireille Blay-Fornarino, Professeur des Universités, Université Côte d'Azur
- Houari Sarhaoui, Professeur, Université de Montreal

Invité :

- Olivier Bantiche-Kamensky, Ingénieur de Recherche, Renault Software Labs

Abstract

Software-intensive embedded systems, such as automotive systems, are increasingly built from highly-variable applications targeting evermore configurable hardware platforms. Moreover, there are often various ways to implement a given application on a specific platform. This threefold variability leads to an immense number of system design alternatives. The notorious problem is to establish, at early stages of development, which designs fulfill and optimize functional and non-functional requirements. Traditional system design frameworks capture system requirements and specifications to derive and evaluate every design automatically. However, they use enumeration-based techniques and offer poor scalability at both modelling and analysis stages. On the other hand, variability modelling approaches exploit commonalities between different but related products to efficiently evaluate the whole product line. Yet, given system specifications, they lack to automatically derive the design space while only specific facets of the problem are evaluated in isolation. We propose a model-driven framework that combines and extends both approaches. It captures requirements and specifications in the form of variable data-flows and configurable hardware platforms, with non-functional constraints and a cost function. An original mapping algorithm then derives the design space automatically and generates it in the form of a variability-aware model of computation, which encodes every system design. Novel verification algorithms then pinpoint suitable designs efficiently. The benefits of our approach are evaluated through a real-world case study from the automotive industry.

Keywords: Model-Based embedded-system design, Feasibility analysis, Optimization, Model of computation, Behavioral Product Line, Quantitative properties, Variability-aware model-checking

Résumé

Les systèmes embarqués sont implémentés à partir d'applications variables ciblant des plateformes matérielles configurables. De plus, une application peut être implémentée de plusieurs façons sur une plateforme. Cette triple variabilité engendre un nombre astronomique de conceptions système alternatives. Le problème crucial est alors d'établir, au plus tôt, quelles sont les conceptions système qui satisfont et optimisent les exigences fonctionnelles et non-fonctionnelles. Généralement, les approches de conception de systèmes capturent les exigences et spécifications pour automatiquement dériver et évaluer toutes les alternatives. Cependant, ces outils sont énumératifs, ce qui peut les rendre inapplicables à grande échelle. D'un autre côté, les approches de modélisation de la variabilité exploitent les points communs entre les différents produits pour évaluer efficacement toute la ligne de produits. Pourtant, ces approches ne traitent que certaines parties du problème et ne fournissent aucun moyen de dériver l'espace de conception automatiquement. Nous proposons une approche qui combine et étend ces deux méthodes. Après avoir capturé les exigences et spécifications sous la forme d'un flot de donnés variables, d'une plateforme matérielle configurable, d'une fonction de cout et de contraintes non-fonctionnelles, nous dérivons un espace de conception encodé par une ligne de produit comportementale. Finalement, un algorithme de vérification permet d'identifier efficacement les conceptions de système les plus adaptées. Les avantages de notre approche sont évalués à travers un cas d'étude industriel automobile.

Mots-clés : Conception de système embarqué, Analyse de faisabilité, Optimisation, Modèle de calculabilité, Ligne de produits comportementale, Propriétés quantitatives, Vérification de modèle variable

Remerciements

Si je devais, dans cette section, mentionner explicitement toutes les personnes qui m'ont aidé à développer cette thèse de doctorat, cette section serait plus longue que le manuscrit lui-même. Ma famille, mes amis, mon équipe de recherche, mon jury de thèse, mes collaborateurs industriels, etc. Ils m'ont aidé, bien plus qu'ils ne le pensent et je les remercie tous du fond du coeur. Mais voici les personnes qui ont profondement marqué le developpment de cette thèse.

Emmanuel Roncoroni, qui fût une figure internationalement connu dans le développement de tableau de bords automobiles, mais aussi un mentor, m'a proposé un sujet de recherche passionnant, issu d'une simple question "quelles plateformes matérielles seraient les plus à même à répondre aux exigences fonctionnelles et non fonctionnelles des clients?" (qui étaient alors des concessionnaires automobiles, clients de Visteon Electronics). Sa vision n'a jamais été de remplacer les ingénieurs. Au contraire, son but était de les assister au mieux face a la compléxité grandissante des systèmes qu'ils devaient produire. Il fût convaincu que des recherches théoriques et appliquées étaient nécessaires, mais que seul une personne consciente des problèmes industriels et scientifiques pouvait mener à bien.

Philippe Collet a été mon directeur de thèse. Mais avant cela, ce fut un professeur exemplaire, il fût également mon directeur d'apprentissage. Je n'ai nul doute à dire qu'il fût essentiel à l'élaboration de cette thèse. J'ai trouvé une personne à l'écoute, mais aussi critique, ce qui m'a permis de fournir tous les efforts requis à l'élaboration d'une thèse de doctorat. Je n'oublierai jamais cette complicité, cette confiance et ce soutien sans failles, et ce, en toutes circonstances. Je ne peux recommander meilleur directeur de thèse.

Sébastien Mosser fût également un de mes professeurs et un de mes encadrants de thèse sur qui je pus compter à chaque moment. Je n'oublierais pas ses commentaires ardus mais constructifs, tout au long de cette aventure. Que ce soit durant mes études secondaires ou durant mes premières recherches, ses avis m'ont toujours permis d'améliorer mes traveaux.

Maxime Cordy est un des pionniers en termes de conception et vérification de modèles comportementaux de systèmes à fortes variabilités. J'eus la chance et le plaisir de le rencontrer dans un moment charnière. Nos premières discussions furent extrêmement intéressantes et enrichissantes. J'eus trouvé un frère intellectuel qui comprenait l'étendue de la difficulté mais aussi l'enjeu et la portée de nos recherches communes. Il n'y a qu'un pas entre la théorie et la pratique. Ce pas est, certes, osé.

Je n'aurais de cesses, de méditer, chaque conseil de ces personnes exceptionnelles, d'une intégrité hors norme et d'une grande humilité, qui furent une source de motivation indispensable. Je serais éternellement reconnaissant envers ces personnes sans qui l'élaboration de cette thèse de doctorat aurait été impossible pour ma part. Pour finir, je n'oublierai jamais ce but, dont cette thèse est issue, celui d'assister au mieux l'ingénieur dans des choix de conceptions hautement cornéliens. Ni, celui d'être mesuré, voir sceptique quant à toute solution proposée. Et finalement, le besoin de recherche fondamentale comme outil pour celui qui aura le courage de l'appliquer dans des problèmes industriels concrets. L'apogée du scientifique n'est pas de montrer qu'il a raison, mais d'essayer, de toutes ses forces, de prouver qu'il a tort, sans y arriver.

I dedicate this thesis to Emmanuel Roncoroni,

Contents

1 Introduction **9**
- 1.1 Context . . . 9
- 1.2 Problem . . . 10
- 1.3 Industrial Practices . . . 11
- 1.4 Challenges . . . 11
- 1.5 Contributions . . . 13
- 1.6 Outline . . . 14

I Background **15**

2 Motivations **16**
- 2.1 Industrial Case Study . . . 16
  - 2.1.1 Running example . . . 17
  - 2.1.2 Suitable Designs . . . 23
  - 2.1.3 Discussion . . . 25
- 2.2 Detailed Challenges . . . 26

3 State of the Art **29**
- 3.1 Model Based Design of Embedded Systems . . . 29
  - 3.1.1 Y-Chart Pattern Overview . . . 30
  - 3.1.2 Modeling and Mapping System Specifications . . . 31
  - 3.1.3 Assessing the System Design Alternatives . . . 31
  - 3.1.4 Model-Checking Pitfall . . . 33
- 3.2 Software Product Line Engineering . . . 34
  - 3.2.1 Product Line Applications to embedded systems . . . 34
  - 3.2.2 Behavioral Product Line . . . 35
- 3.3 Discussion . . . 35
  - 3.3.1 Challenge 1 : Capturing Variable System Specifications . . . 35
  - 3.3.2 Challenge 2 : Deriving Automatically the Design Space . . . 36
  - 3.3.3 Challenge 3 : Evaluating Efficiently the Multifaceted Problem . . . 37
  - 3.3.4 Conclusion . . . 37

II Contributions **38**

4 Framework Overview **39**
- 4.1 Modeling and Mapping System Specifications . . . 40
- 4.2 Assessing the System Design Space . . . 41

5 Modeling and Mapping System Specifications **42**
5.1 Applications as Variable Data-Flows . . . 42
5.2 Platforms as Variable Resource Graphs . . . 44
5.3 Variability-Aware Mapping Strategy . . . 47
5.4 Conclusion . . . 50

6 Assessing the System Design Alternatives **51**
6.1 Design Space as a Behavioral Product Line . . . 51
6.1.1 Background . . . 51
6.1.2 Featured Transition Systems . . . 55
6.2 Variability-Aware Validation Process . . . 62
6.2.1 Background . . . 62
6.2.2 Model Checking Lots of System Designs . . . 64
6.2.3 Conclusion . . . 75

7 Toward a Complete Framework **76**
7.1 Modeling Non Functional Concerns . . . 76
7.1.1 System Specifications with Non Functional Properties . . . 77
7.1.2 Non Functional Requirements . . . 78
7.2 Assessing Non Functional Concerns . . . 79
7.2.1 Design Space as a Behavioral Product Line with Quantitative Properties . . . 80
7.2.2 Variability-aware Validation . . . 83
7.3 Conclusion . . . 90

III Validation **93**

8 Framework Implementation **94**
8.1 Implementation Overview . . . 94
8.2 System Specifications and Mapping Models . . . 95
8.2.1 Applications as Variable Data-Flows . . . 95
8.2.2 Platforms as Variable Resource Graphs . . . 97
8.2.3 Non-Functional Requirements . . . 98
8.2.4 Variability-Aware Mapping Process . . . 98
8.3 Assessing the System Design Space . . . 99
8.3.1 Design Space as a Behavioral Product Line . . . 99
8.3.2 Variability-Aware Validation Process . . . 103

9 Industrial Evaluation **105**
9.1 Case Study . . . 105
9.2 Preliminary Experiment . . . 105
9.3 Qualitative Experiment . . . 107
9.4 Scalability Experiment . . . 108
9.4.1 Product-Based Versus Family-Based Verification . . . 108
9.4.2 Optimal Design in ProVeLines and UPPAAL . . . 110
9.5 Threats to Validity . . . 111
9.5.1 Internal validity . . . 111
9.5.2 External validity . . . 112
9.5.3 Conclusion Validity . . . 112

10 Conclusion and Perspectives **113**
10.1 Review of the Challenges 113
10.2 Final Discussion 115
10.3 Perspectives 116
10.3.1 Integration to Automotive Industry 116
10.3.2 Applications to Other Systems 117
10.3.3 Variability-Aware Statistical Model-Checking 118
10.3.4 Abstractions of Behavioral Product Line 118
10.3.5 Multi-Level Design Space Exploration 119

List of Figures

2.1 Our instrument cluster system case study 16
2.2 System specifications 18
2.3 System Design Implementations 18
2.4 Alternative data-flow variants 20
2.5 Alternative platform configurations 21
2.6 Suitable Designs 24
2.7 Pareto front with some of the optimum designs. 25
2.8 More realistic size Instrument cluster specifications 28

3.1 Y-Chart Design Space Exploration Pattern 30

4.1 Framework Models and Processes 40

5.1 Variability-Intensive Application Functional Specification 44
5.2 Highly-Configurable Platform Functional Specification 46
5.3 Application Mapping Steps 50

6.1 Automaton capturing System Variant of Design B 53
6.2 Automaton capturing System Variant of Design D 54
6.3 System featured automaton 58
6.4 System Design Space Variability 59
6.5 Execution Traces of System Variant of Design B 65
6.6 An execution trace producible by the System Variant of Design D . . . 66
6.7 An FTS execution trace producible by system variant B 67
6.8 An FTS execution trace producible by system variant D 68
6.9 The shortest FTS invalid execution trace producible by a huge amount of system variants 69
6.10 Depth First Search of Reachables$((D1||D2||RAM)_{FA})$ function. 73

7.1 An excerpt of the PFM corresponding to the case study. 80
7.2 Platform Featured Weighted Automaton. 82
7.3 The execution of Design C 85
7.4 Run-time consumption of Design C 86
7.5 Depth First Search of Optima(UC) function. 91

8.1 Framework Models and Processes 94
8.2 Meta-models implementations 96

List of Algorithms

1 $M(app, plt)$ 49

2 Reachables(fa) 72

3 optima(fwa, pfm, ζ) 89

Listings

8.1 Running Example Application 95
8.2 Running Example Platform 97
8.3 Non-Functional Requirements 98
8.4 Mapping Algorithm 98
8.5 Generate Running Example Formal Models from Design Space 99
8.6 Part of Running Example Design Space variability in TVL 100
8.7 Part of Running Example Design Space behavior in fPromela 101
8.8 Part of Running Exemple ProVeLines output 103
8.9 Part of Valid Design Space Variability in TVL 104

List of Tables

9.1 Result for Preliminary Experiment using ProVeLines Family-Based model checker. 106
9.2 Results for Product-Based Versus Family-Based Verifications. 109
9.3 Results of ProVeLines-CORA and UPPAAL-CORA for designs optimizations. 110

Chapter 1

Introduction

1.1 Context

Scale and complexity of software-intensive systems such as cyber-physical and large scale embedded systems have reached historic levels. While the Gartner Group expects more than twenty billion of connected objects [Eddy, 2015], automotive systems are developed with several millions of lines of code driving hundreds of electronic hardware [Charette, 2009]. In many of these systems, requirements engineering and design activities are of utmost importance in industry to reduce a wide range of risks at early stages of development. However, these development steps are tightly intertwined and involve complex multi-criteria design decision-making over various concerns.

As an example, let us consider an infotainment service in an automotive system. Specifying such service entails defining functional and non-functional (*a.k.a.* quality) requirements. Functional requirements would specify what and how the *Human-Machine Interface* (HMI) content should be displayed into the car display. This embedded HMI-rendering is constrained by the hardware platform at the functional level, such as not exceeding the available memory or not misusing processing pipelines. While typical examples of non-functional requirements are constraints on manufacturing cost, execution time, or even rendering quality. The notoriously difficult problem is to establish, at early stage of development, whether such requirements are feasible and what is the best system design to implement it with more confidence. That requires either to prototype or to have massive knowledge of a lot, if not all, of the possible design alternatives. This is unrealistic in a context of high-level competition where companies must deliver better solutions from more complex requirements and do so timely [Broy et al., 2011].

Basically, a data-flow oriented automotive embedded-system is developed in order to fulfill these requirements. The system consists of *i)* a data-flow processing *application* (*i.e.*, a data-flow graph capturing what and how the HMI content should be process) driving and feeding a *ii)* resource-limited hardware *platform* (*i.e.*, heterogeneous hardware components like non-programmable processors and data storage units) to provide efficient and high-quality graphics rendering at the lowest cost. This separation of concerns between the applications "what do we have to do" and the platforms "what can we do" is not specific to automotive embedded system. Internet of Things, grid computing, and even production plans also have to consider platform limitations to meet and optimize requirements.

1.2 Problem

There are various ways to implement a given application on a specific platform. Known as the famous application *mapping and scheduling* problem [Sigdel et al., 2009], *e.g.*, choose a specific processor to process a given task or select a specific memory unit to store a given data. Therefore, in addition to scheduling problem and mapping variability, other extensive variation points are growing at the requirements and specifications levels. Since platforms and applications are more and more developed as a product line to target multiple ranges of systems[1]. There are two additional main sources of extensive variability. First, at the application level, multiple data-flow variants can be engineered from requirements, differing in, *e.g.*, the size of the flowing data chunks, the ordering of the processing tasks, or the choice between alternative, functionally-equivalent tasks. Second, there exists a diversity of configurable hardware platforms that can differ, *e.g.*, in memory capacities and processing pipelines. This threefold variability is typical in embedded systems [Pretschner et al., 2007].

The number of system variants (*i.e.*, a specific mapping of a specific application variant to a specific platform variant) growths exponentially in terms of design variation point from multiple sources (*e.g.*, requirements, application, platform, mapping, software, hardware, protocol). Yet, each system variant may exhibit multiple executions due to the different scheduling opportunities (*e.g.*, tasks can be executed onto resources in different orders). Unfortunately, it often leads to a large number of variants from a million (10^6) to a billiard 10^{15}, while every single variant could exhibit up to a trillion (10^{12}) different possible executions for a large scale embedded systems [Sigdel et al., 2009].

A *system design alternative* (or *design* for short) is then composed of a system variant (*i.e.*, design structure) and a scheduling execution (*i.e.*, design behavior). Both elements are mandatory in order to implement the design in latter stages of development. Among these design alternatives, not all are able to realize the functional requirements due to hardware functional limitations. The same holds for the non-functional requirements due to limited hardware capacities. Given these numbers, a systematic consideration of all design alternatives is unfeasible for the system engineers, whereas the high level of competition in industry puts high pressure on them to deliver optimal solutions and do so timely [Broy et al., 2011]. Efficient automation to assist engineers, therefore, appear as a necessity.

Examples of questions the engineers need to answer are:

- *Which designs can properly render the specified HMI to the screen?*
- *Which feasible designs can be manufactured with a budget of 100$ or less?*
- *Which feasible designs can render the HMI in less than 16 ms?*
- *Which feasible designs, with a high-definition rendering quality and a manufacturing cost lower than 50$, exhibit the fastest execution time?*
- *Which feasible designs reach the best trade-off between rendering quality, manufacturing cost and execution time?*

[1] Contrary to application-specific hardware platform where the hardware is synthesized for a specific application logic.

1.3 Industrial Practices

In industry, quick and approximated prototyping methods are still largely adopted to answer to those questions. This methodology has the advantage of getting a system running in front of customers. However, while a successful implementation assesses the functional and non-functional feasibility of a particular system design, optimality cannot be demonstrated. Worst still, finding functional or performance problems at late system development stages could lead to major risks (*e.g.*, significant delay, economical cost, project failure). Besides, having a good picture of promising designs with such methods requires to find and implement a lot of, if not all, designs. This means dealing manually with variability concerns, which is not humanly possible in our class of problems. Nevertheless, even if proactive system experts find and implement nearly optimal designs thanks to decades of experience in system engineering, no one can formally prove their design choices to customers or third-party teams.

To reduce these gaps, a plethora of model-based system design methods are used in industry as well. Instead of prototyping designs, a model-based design framework captures and reasons formally on all designs through model abstractions (*i.e.*, system specifications, design space and model of computations). This approach reduces major risks of system prototyping methods. Making possible relevant design choices at early stages of development. Furthermore, prototyping only promising designs is thus enough to answer to those questions with more confidence. The key factors of this approach are the quality of models abstractions (*i.e.*, relevance, accuracy, expressivity) and quality of reasoning (*i.e.*, analysis time, correctness).

Analytical methods give quick results but often turn out largely suboptimal, if not completely wrong. On the contrary, low-level simulators provided by platform suppliers are highly accurate, but analyzing all system variants requires implementing fine-grained simulations for all of them, which is unrealistic. System-level design frameworks[2] seem appropriate in terms of model abstraction and reasoning quality, but lack of capturing to three variability levels. Where the number of variants could increase exponentially according to the amount of design variations, the modeling and assessing time of every variant could lead to scalability issues. Making thus a significant obstacle for any industry adoption. In the end, current practice is deemed very unsatisfactory. Our industry partner made these observations, Visteon Electronics, an international leader in automotive systems, and are also corroborated by surveys such as [Broy, 2006].

1.4 Challenges

The most appropriate frameworks used in industry lack to capture to three levels of variability and present scalability issues. Nevertheless, answering those questions requires not only a way to deal efficiently with three levels (*i.e.*, application, mapping, platform) variability-induced combinatorial explosion but also a method to reason simultaneously and efficiently about various facets of the design engineering problem emerging from different types of concerns: feasibility/satisfiability *and* optimality; functional *and* non-functional requirements; and different types of aspects: behavioural *and* structural aspects of the system. Concretely, thus requires solving all

[2]System level-design is a trade-off between low-level electronic system and high-level analytical methodologies to maximize the results accuracy while minimizing the analysis time.

combinations of concerns on aspects (*multifaceted problems*) efficiently. The different facets can be classified by concern as follow:

- *The design satisfies the functional requirements*, that is, checking both the FC requirements that depend on the structure of the design (system variant); *can be implemented?* and checking FC requirements that also depend on its behaviour (system execution); *is the HMI properly rendered?*.
- *The design satisfies the non-functional requirements*, that is, checking both the structure: *can be manufactured within this budget?* and also the behaviour of the design; *is the HMI rendered at a minimum of 30 frames per second?*
- *The design optimizes quality attributes*. This facet requires considering both structural and behavioral quality attributes simultaneously. Optimizing only those that depend on the structure; *is the cheapest?* or only those that depend on the behavior; *is the fastest?* can lead to suboptimal solutions.[3].

System Design Engineering approaches used in industry efficiently capture system requirements and specifications (*i.e.*, application, platform) with *domain specific languages*. An *application mapping* algorithm then derives the resulting design space. Finally, the design space is transformed into *models of computations* in order to evaluate all facets of the problem. While some specific techniques attempt to handle and manage either platform variability [Sima and Bertels, 2009, Sigdel et al., 2009, Palermo et al., 2009] or application variability [Schor et al., 2012, Van Stralen and Pimentel, 2010, Wildermann et al., 2011a, Palermo et al., 2008], none of these approaches allow engineers to capture variability present in both application and platform specifications. This entails modeling and assessing each variant iteratively, which could lead to major scalability issues.

On the other hand, *Product Line Engineering* approaches capture behaviors of variable systems through *behavioral product line* formal models. Such formalisms exploit commonalities between different but related variants to efficiently assess the whole set of products in a single run. However, these approaches only assess specific facets of the problem in isolation[4]. Moreover, given application and platform specifications, these frameworks are not capable of automatically map these specifications in order to derive the resulting design space. This would imply to manually derive the design space[5], which is tedious, time-consuming, and error-prone.

To solve all facets of the problem efficiently, we determine three challenges to be tackled in our context :

1. Capturing functional and non-functional variable requirements and specifications that can vary at both the application and platform levels; applications being represented by alternative data flows supporting concurrent behaviour, while platforms are described as configurable hardware components;

[3]This facet falls into a multi-objective optimization problem as the cheapest system that exhibits the fastest execution may not be the fastest system that exhibits the cheapest cost.

[4]Structural and behavioral aspects, as well as functional and non-functional concerns, are mainly addressed in isolation by state of the art frameworks.

[5]More precisely, the various ways of mapping every application variant on every platform configuration in order to derive the design space.

2. Deriving, from the application and platform models, the resulting design space (*i.e.*, all possible design alternatives) capturing all the structural, behavioural, functional and non-functional properties and variations of each design alternative;

3. Evaluating simultaneously and efficiently all facets of the problem; the functional feasibility, the non-functional satisfiability, and optimality at both structural and behavioral aspects of each design alternative;

1.5 Contributions

In this thesis, we propose an original approach that combines and extends *embedded system design engineering* and *product line engineering* domains to provide the first tooled framework able to solve, formally, the three challenges described in the above paragraphs. Finally giving the means to make appropriate design decisions at early stages of development.

The proposed framework is model-driven and *i)* captures requirements, variable data-flow and platform specifications with domain-specific languages extended with variability concerns (in a Y-Chart form), *ii)* uses a novel variability-aware mapping algorithm to map variable data-flow application onto a description of the targeted configurable hardware platform, so to derive a variability-intensive embedded system design space in a behavioral product line form in order to, iii) reuses and extends automated reasoning techniques from both research domains (*i.e.* variability-aware model checking and cost-optimal reachability analysis) to explore and assess efficiently the functional feasibility, non-functional statisfiability and optimality at both structural and behavioural aspects of the whole design space in a single run.

1. A *modeling method*, extending application and platform model with variability concerns. Formal models extensions have been proposed; applications model extends data-flows supporting concurrent behaviour with possible flow variations and alternative values for properties (*e.g.*, data size, rendering quality) while platform model extends components-based systems with optional component and configurable properties (*e.g.*, memory capacity, clock frequency).

2. A *mapping algorithm* to derive from the application and platform models the resulting design space by mapping application elements into platform resources while capturing all the structural, behavioural, functional and non-functional properties and variations of each design alternative;

3. A *reasoning tool chain* unifying state-of-the-art techniques on product line reasoning with novel variability-aware model-checking algorithms capable of evaluating efficiently and simultaneously some or all problem facets of the whole design space at once;

4. A *qualitative evaluation* of the approach based on a lightweight real mid-end instrument cluster system from our industry partner. Functional and non-functional requirements were properly captured and assessed by our framework. Optimal system designs were correctly identified. Even on small data-flow and platform models, the optimal models were non-trivial to devise for industrial experts, showing the practical relevance of the proposed approach.

5. A *quantitative evaluation* assesses the scalability of our approach. It gives us confidence that our framework could be applied to the majority of systems (low-end and mid-end instrument cluster) developed by our industrial partner and similar systems developed elsewhere in industry. This evaluation shows possible new research opportunities to improve traditional system design engineering by considering system design spaces as behavioral product lines.

1.6 Outline

The remainder of the thesis is organized as follows.

- Part I provides details on our industrial case study, the expected outcomes of our framework with identified challenges (see Chap. 2). It then discusses state of the art (see Chap. 3).
- Part II proposes en end-to-end framework. The framework is firstly elaborated and applied to functional assessment to validate the approach (see Chap. 4, 5 and 6). The next Chapter (see Chap. 7) proposes a non-functional extension that allows to solve the entire *multifaceted* problem.
- Part III describes our implementation (see Chap. 8), qualitative and quantitative industrial evaluations (see Chap. 9), with identified threats to validity.
- Finally, the last chapter gives final remarks and presents the main perspectives of this thesis (see Chap. 10).

The research work done during this PhD has led to the following peer-reviewed publications.

Lazreg, Sami and Collet, Philippe and Mosser, Sébastien "Assessing the functional feasibility of variability-intensive data flow-oriented systems" Best Paper Award in the *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*[6]. This paper proposes a new approach to assess, in early design phases, the functional feasibility of embedded system design alternatives. Rather than enumerating and iteratively assessing all designs, the proposed framework reasons on behavioral product line to assess the whole design space in a single variability-aware model-checking run (see Chap. 4, 5 and 6). The experiments (see Sec. 9.2) show that this approach exploits behavioral commonalities between system designs to speed-up remarkably the verification process.

Lazreg, Sami and Cordy, Maxime and Collet, Philippe and Heymans, Patrick and Mosser, Sébastien "Multifaceted automated analyses for variability-intensive embedded systems" *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*[7]. This paper proposes to extend our framework in order to capture and assess efficiently the non functional satisfiability and optimality of the whole design space. (see Chap. 7). The family-based experiments 9.4 characterize the efficiency our variability-aware system design approach compared to a traditional *product-by-product* ones.

[6] `https://hal.archives-ouvertes.fr/hal-01660057/document`
[7] `https://hal.archives-ouvertes.fr/hal-02061251/document`

Part I

Background

Chapter 2

Motivations

2.1 Industrial Case Study

In this research, we collaborate with Visteon Electronics, a leading company developing solutions for the automotive industry such as instrument clusters, infotainment and connected vehicles. We introduce a pedagogical excerpt from a mid-end instrument cluster system developed by Visteon. We will use a pedagogical sample as a running example throughout this thesis to further illustrate and justify our approach. An *instrument cluster* is a speedometer and other instrumentation which, unlike traditional analog gauges, appear on an electronic visual display (see Fig. 2.1). By applying graphical processing effects on information related to the vehicle (e.g., 2D/3D gauges, 3D view of the car), an instrument cluster improves the driver experience.

The case study we present is an important module of a mid-end instrument cluster developed without using any model-based design methods during the 2015-2016 period. Our industrial collaborator proposed this particular module as it was surprisingly difficult to develop, while the complexity of application (number of task and data, etc.) and platform (number of processor and memories) was standard regarding other related modules. Thanks to decades of experience, system engineers finally delivered a system that satisfies every functional and non-functional requirement. However, as usual, such complex development requires multiple iterations and step back. Moreover, the performance and quality generally of the prototypes were differing from engineers'



Figure 2.1: Our instrument cluster system case study

expectations. Still, while the final implementation was exhibiting suitable performance and quality, none of the engineers could assert formally that the final implementation was optimal for the customer expectations.

2.1.1 Running example

To develop such systems, customers (*i.e.*, automakers) specify their functional requirements, "what and how they want the system does", usually illustrated by a desktop demonstration of the HMI. Non-functional requirements, such as rendering quality, budget, and responsiveness, are also defined. The role of a company such as Visteon is to implement the customer HMI in a data-flow-oriented automotive embedded system that i) reach functional requirements by rendering the requested HMI correctly and ii) reach the non-functional requirements by satisfying the customer quality and budget expectations.

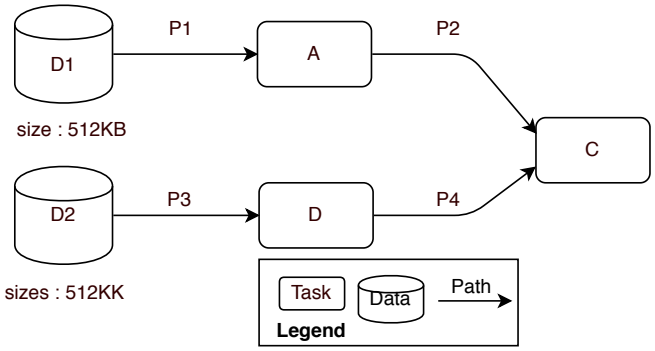
The data-flow-oriented embedded system is a given data-flow application (*i.e.*, engineered from the HMI specified by the customer) driving and feeding a specific hardware platform (*i.e.*, assembled by Visteon and third-party hardware manufacturers). The data-flow-oriented embedded system thus consists of two specifications, *i.e.*, application and platform, and two implementation matters, *i.e.*, mapping and execution of these specifications (engineered by system engineers). Finding the most suitable mapping and execution of an application variant on a platform configuration determines the project's success or failure. We now illustrate these elements:

- The data-flow processing *application*: the processing flow that fulfill the HMI's functional requirements is captured by a data-flow oriented application supporting concurrent behaviour. Fig. 2.3(a) illustrates a data-flow specification with functional properties (*e.g.*, image size, task), non functional properties (*e.g.*, image and task rendering quality) and quality attributes (*e.g.*, overall quality). Images are processed by graphical tasks[1]. Image `D1` is processed by tasks `A` and image `D2` is processed by task `D`. The image produced by `A`, and the image produced by `D`, are then processed by task `C`, which delivers the final result onto the display.

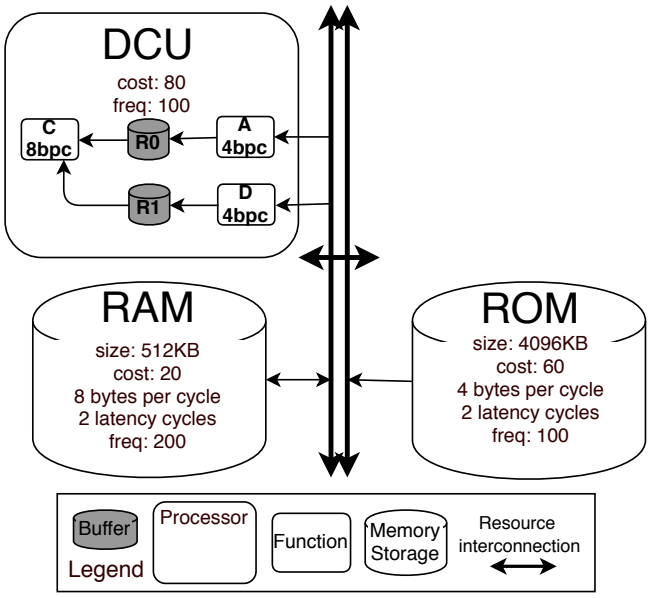
- A resource-constrained hardware *platform*: The hardware platform, Fig. 2.3(b), is described by a set of interconnected hardware components with functional properties (*e.g.*, memory capacity, processor functions), non-functional properties (*e.g.*, memory and processor bandwidth, cost, frequency) and quality attributes (*e.g.*, overall cost). Image processing functions `A, C, D` are provided by a non-programmable Display Controller Unit (`DCU`). Within the DCU, there is a processing pipeline composed of three functions `A`, `D` and `C`, which finally render the result onto the vehicle display. A processing pipeline is composed of several hardware-implemented processing functions that may differ in processing bandwidth (*i.e.*, different byte per cycle performance). Directed edges denote the processing flows followed by data that transit through the processing pipeline of processors using internal FIFO-buffers, while functions may be applied or not[2].

[1] Other data parameters than image sizes, compression or scale factors, such as transformation matrix, masks etc. can be ignored as they do not interfere with the finding of the suitable designs.

[2] Bus systems, memory controllers, memory banks and cache memories can be ignored as they do not interfere with the finding of the suitable designs.

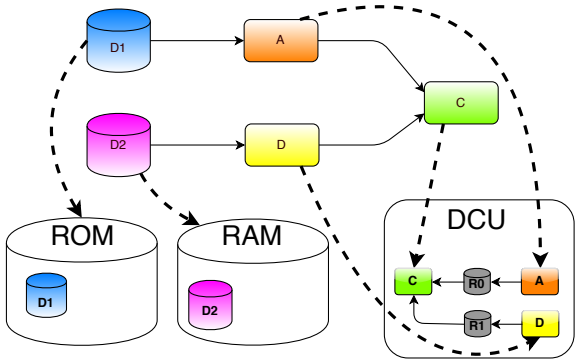


(a) The system dataflow specification

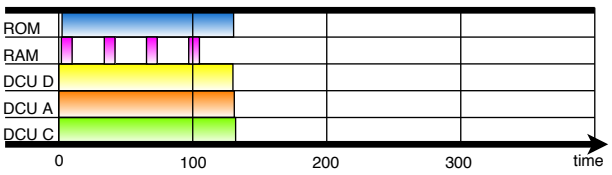


(b) The system platform specification

Figure 2.2: System specifications



(a) The system mapping variant implementation



(b) The system run-time execution implementation

Figure 2.3: System Design Implementations

- A *mapping* of the application on the platform: (Fig. 2.3(a)), illustrates a mapping (*i.e.*, software implementation) of the application onto the platform. Images D1 and D2 are, respectively, stored onto ROM and RAM. The data-flow processing application, composed of tasks A, D and C, is implemented using the DCU hardware pipeline. There are various ways of mapping this application on this platform as the images D1 and D2 can be stored on RAM or ROM. This mapping variant (Fig. 2.3(a)) is one of the 4 possible mappings (so called mapping space).

- A run-time *execution* of the mapping: Fig. 2.3(b) sketches a system execution[3]. Images are fetched from the memories and processed by the hardware functions at different processing bandwidths (*i.e.*, RAM bandwidth is higher than ROM one, byte per cycle processing capabilities of C hardware function of the DCU is twice faster than A and D ones). Bandwidth and latency of the hardware components and their interactions determine the overall execution time. A system variant may have multiple executions (execution space) according to scheduling opportunities of the application elements (task and images) over the platform resources (processor and memories).

Variability-intensive applications

In the instrument cluster domain, multiple dataflow variants can achieve the functional requirements (cf. Fig. 2.4). In our example, image D1 can be processed by functionally-equivalent tasks A or B, while D2 has three different possible resolutions. The task and image resolution impacts the HMI rendering quality. In our case, performing B instead of A improves the rendering quality significantly. Also, as the resolution of D2 increases, the overall quality raises as well. Such variability is mainly due to the fact that instrument clusters are more and more developed as product lines.

Fig. 2.4 specifies six alternative data-flows (application space), *i.e.*, application variants. The highest rendering quality is achieved by the application variant that processes the image D1 by the task B while the data D2 has the highest resolution (*i.e.*, 1024KB). On the contrary, the variant where D2 is at the lowest resolution and D1 is processed by the task A has the lowest rendering quality. The image resolution impacts the overall system quality but also the total amount of bytes of data to process, which can influence the rendering time.

Highly-configurable platforms

At the platform level (Fig. 2.5), hardware providers generally offer adapted platform product lines for each range of infotainment services. Thus, multiple platform configurations (platform space) are possible at hardware manufacturing time. in our example, some configurations propose an additional RAM storage and/or a graphical processing unit GPU to increase functional capacities and processing bandwidth of the platform (*e.g.*, GPU provides two parallel processing pipelines). The first is composed of A and B processing functions. The second is focusing on high-speed processing of task D[4].

GPU and RAM are optional as some basic applications only store input images in ROM and use DCU to process and render them directly to the display. Yet, they improve the

[3]Memory accesses optimization such as prefetching and burst modes are not detailed as it is a low-level system concern. Engineers can determine such details in late development stages.

[4]The D function of the GPU is implemented with more transistors and is thus 4 times faster than the D function of the DCU

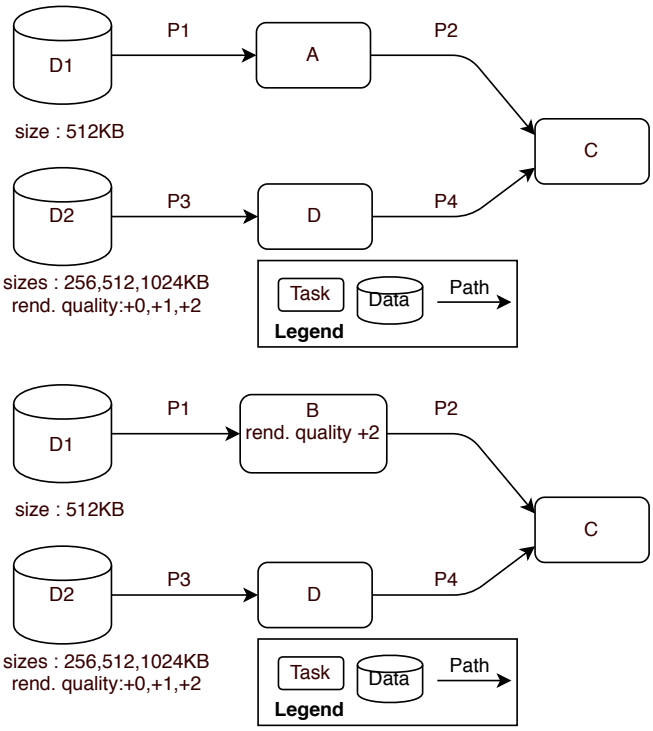


Figure 2.4: Alternative data-flow variants

platform processing and storage capacity for a higher manufacturing cost. Note that there is a presence condition between `RAM` and `GPU` as `RAM` acts as a dedicated cache memory for `GPU`. Whereas `RAM` can be used without `GPU`, *e.g.*, to store more data or larger images. In our example, `RAM` comes in three alternative capacities (at different costs). On the other hand, `RAM`, `GPU` and `ROM` have alternative frequencies acting as a scale factor for data processing/transfer bandwidth for processors and memories witch could influence the execution time.

As a result, this configurable hardware platform is a product line of 19 electronic targets. Each electronic target is a platform configuration of heterogeneous hardware components like non-programmable processors and data storage units with a determined cost, functionality, and capacity. The cheapest platform configurations, limited to 4096KB of storage, are those providing only the mandatory hardware components (`ROM` and `DCU`). Configurations with a `GPU` and the highest-capacity `RAM` are 2.42 times more expensive, but can store up to 6144KB. Also, processing task `D` is up to 8 times faster using `GPU` rather than `DCU`. To the same extent, the transfer bandwidth of `RAM` is much higher than `ROM` (4 times higher).

Variability-Intensive Embedded System Design Space

A system variant result from a specific mapping variant that implements a specific application variant onto a specific platform variant. This threefold variability we observed in the instrument cluster is typical of embedded systems [Pretschner et al., 2007]. The variant space grows exponentially in terms of design variations, and two different system variants thus present design variations at (one or many of) these three variability dimensions. Besides, a system variant may exhibit multiple possible executions (execution space) that can differ from scheduling of the application elements over the platform resources (*e.g.*, tasks can be executed onto resources in different

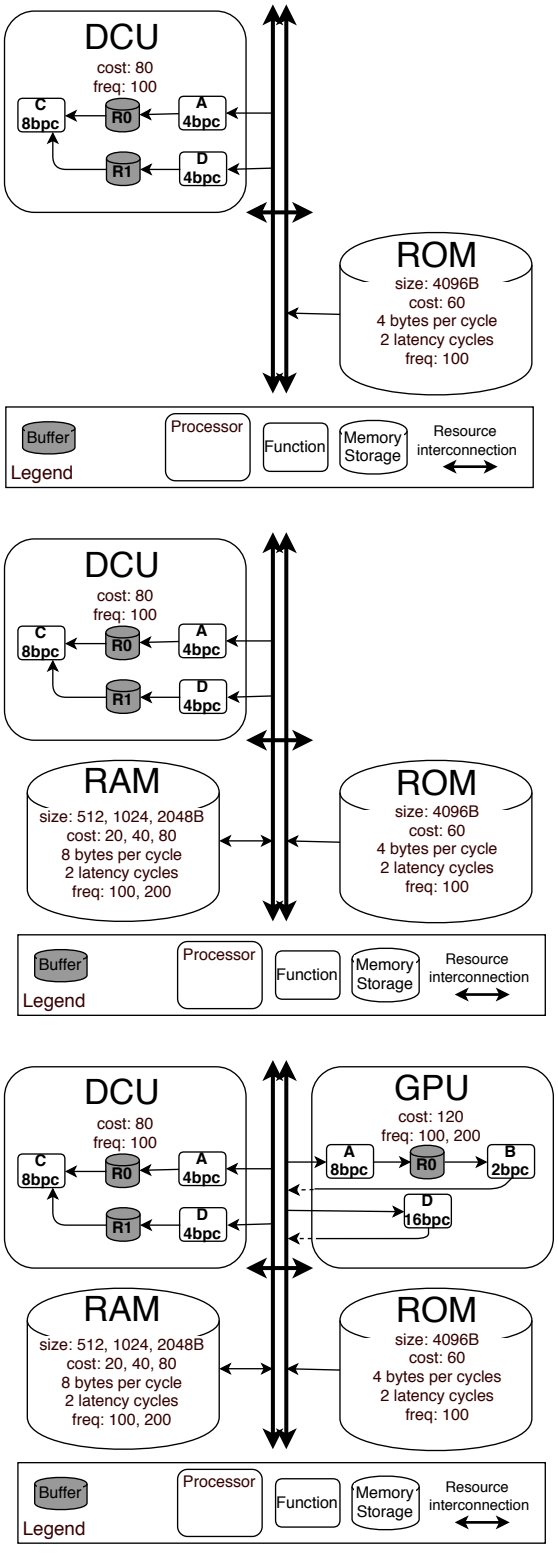


Figure 2.5: Alternative platform configurations

orders). A *system design alternative* (or *design* for short) is composed of a system variant (design structure) with a particular system execution (design behavior).

The resulting *Variability-Intensive Embedded System Design Space* is the set of every possible system designs. Among these design alternatives, not all fulfill the functional requirements due to hardware limitations on the functional level. The same holds for the non-functional requirements as the cost of the platform configuration, the quality of the given data-flow application variant, or the run-time execution of the system may not reach the expectations due to limited hardware capacities.

Functional requirements

To guarantee that a design is capable of rendering the HMI without any problems, the design should first exhibit a consistent structure (a compatible triplet of application, mapping, and platform variants). "Any task and data of the data-flow variant are mapped to, respectively, processors and memories of the platform configuration." *i.e.*, which tasks must be processed, which tasks exchange data with others, which memory storage is accessible by processors, and how tasks (resp. data) can be mapped onto processors (resp. memory storage).

The design structure (system variant) could also be inconsistent as it exhibits a platform variant with the GPU without the RAM (violating thus the platform consistency) or an application variant with both tasks `A` and `B` (application consistency violated). Also, if the mapping variant exhibits the task `B` mapped on GPU without a platform variant with `GPU`, or a data mapped into a non selected RAM (violating thus mapping to platform consistency) as well as a data mapped on an unreachable memory for the processor that implements the task, *i.e.*, processor where the task is mapped (which violates the mapping consistency).

Secondly, the design should also exhibit an error-free execution as a behavior. Graphical processors (*e.g.*, *GPU* and *DCU*) have limited hardware functions, and memories (*e.g.*, `RAM` and `ROM`) have limited storage capacities. The execution of the tasks must terminate without *bugs* (*e.g.*, such as deadlock) or violation of platform capabilities (*i.e.*, misusing hardware pipelines or communication paths between components, exceeding the available memory). This *termination* property not only depends on the structure of the design but mostly on its *behaviour*, as particular scheduling of tasks and data transfers may cause deadlocks or memory overflow. Some other traditional behavioral properties, such as safety or liveness checking, may also be required.

Non Functional requirements

In addition to *functional* (FC) Requirements, the design must also meet *Non-Functional* (NF) requirements. In our example, these commonly include a maximal manufacturing cost, a minimal rendering quality, and responsiveness (*i.e.* time to render graphics on the visual display from which frame per second is determined), called NF constraints. Not all designs can meet the whole set of NF constraints. A higher-quality data-flow variant may increase the total amount of bytes the platform should store and process. A platform configuration with more memory storage and processing performances generally comes at a higher manufacturing cost. Finally, the rendering time depends on the application variant workload/platform configuration capacities association but also their mapping and scheduling.

Manufacturing cost and rendering quality are quality attributes depending only on the *design structure*. Execution time, however, depends on both structure (*e.g.*,

data size, processor frequency) but emerges from the *design behavior* execution (*i.e.*, scheduling of tasks, processing bandwidths, and memory access operations). However, as market competition forces engineers to deliver the best system to each specific customer. Among many designs, they must find those offering the *best* trade-off between the quality attributes (*i.e.*, multi-objective optimization NF requirements).

2.1.2 Suitable Designs

Given such requirements and specifications, system engineers need to find some suitable designs that fulfill functional requirements but also meet (and even optimize) multiple quality attributes (manufacturing cost, rendering quality, execution time, etc.). This may involve making complex design choices at both application, platform, mapping, and scheduling levels. In the instrument cluster industry, general questions engineers have to answer are:

1. *Which designs can properly render the customer HMI to the vehicle display?*: The first step is usually to identify designs that meet at least the customer functional requirements.

2. *Which feasible designs, with an execution time less than 30.0 ms, have the cheapest manufacturing cost*: Responsiveness is important, and to make economies, part of the quality can be neglected.

3. *Which feasible designs, with an execution time less than 30.0 ms, expose the highest rendering quality?*: To propose an instrument cluster with high-grade quality, the focus can be on rendering quality.

4. *Which feasible designs, with a rendering quality score of, at least*[5]*, 1 (low-quality grade) and a manufacturing cost lower than 20.0$, exhibit the fastest execution time?*[6]: The focus is on responsiveness, but minimal quality and maximal budget are also defined.

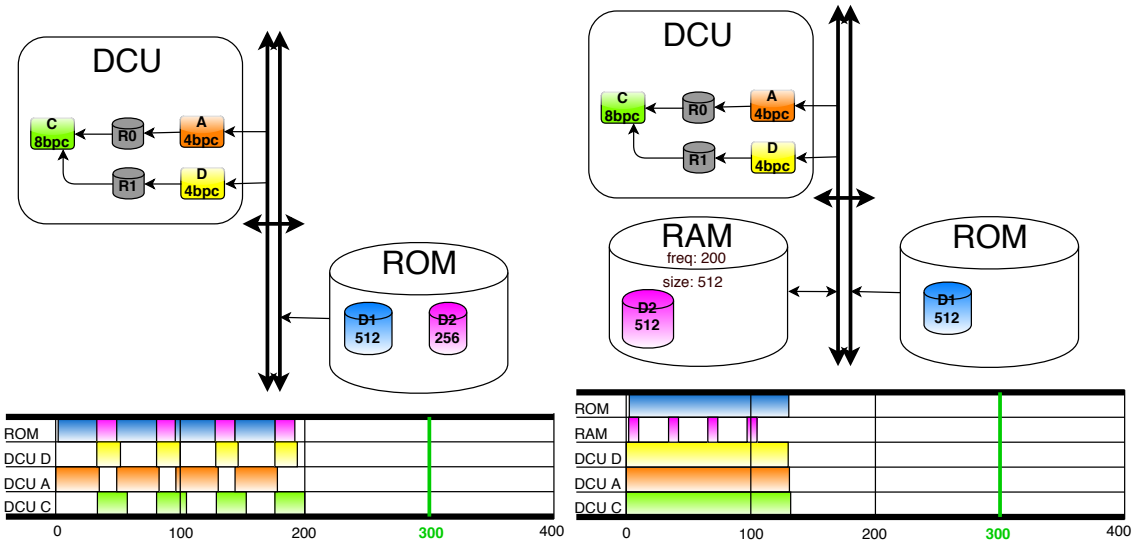
5. *Which feasible designs reach the best trade-off between rendering quality, manufacturing cost and execution time?*: While no clear expectations are defined, one can arbitrary select design by trade-off on multiple quality attributes.

To answer those questions, let us consider some suitable designs on our example (see Fig. 2.6). These designs present different design choices having complex influences on their functional feasibility, non-functional satisfiability, or even their optimality *w.r.t.* non-functional requirements. The designs presented fulfill the functional requirements except for the design (d), which exceed memory capacity. Design (d) is not feasible in practice; trying to prototype it is a waste of time and effort.

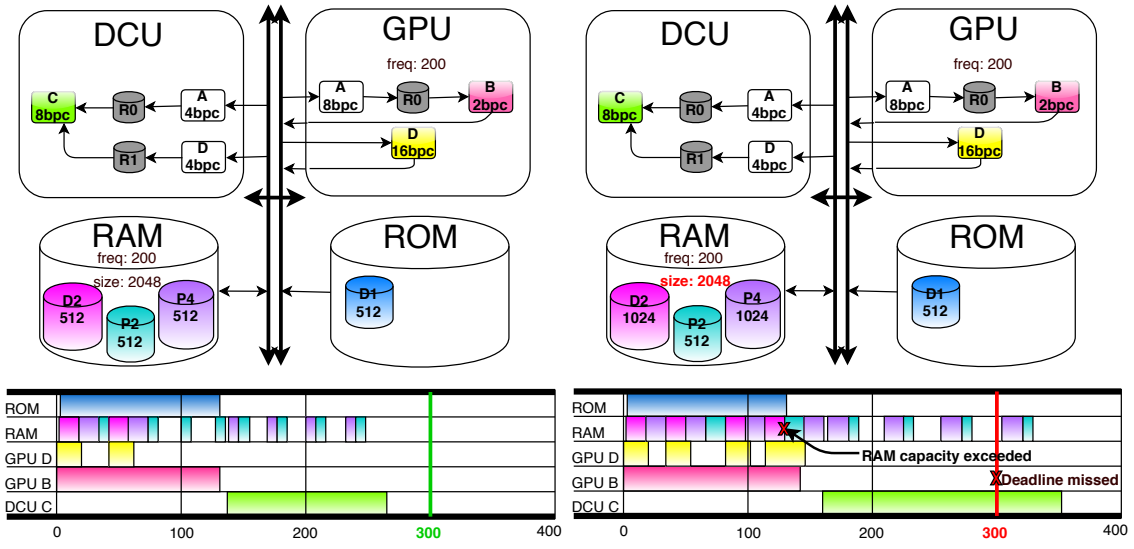
- According to its platform variant, the design (a) is one of the cheapest (*i.e.*, design choices to not select `RAM` nor `GPU` reduce the manufacturing cost drastically). Even if the `ROM` is the bottleneck, its execution time meets the non-functional requirements. However, due to its application variant, this design exhibits an extremely low rendering quality.

[5]The quality is generally informal and determined through lower or higher resolution, and more or less advanced graphics technologies increasing the visual quality of the HMI (*e.g.*, advanced filtering and anti-aliasing, lighting and shading, compression format

[6]In addition to run-time, other metrics such as processing or memory bandwidths consumption profiles can also be part of NF constraints.



(a) Cost 14.0$, Quality 0, Exec. Time 20.0ms **(b)** Cost 18.0$, Quality 1, Exec. Time 13.0ms



(c) Cost 34.0$, Quality 3, Exec. Time 26.0ms **(d)** Cost 34.0$, Quality 4, Exec. Time 35.0ms

Figure 2.6: Suitable Designs

- Design (b) presents a small size `RAM` on which `D2` (medium quality selected) is stored. This allows for fetching images in parallel, removing, thus, memory bottleneck. In addition, this provides a faster and higher rendering quality execution at an extra cost. Interestingly, design choices that reduce `D2` quality or memory frequency (*i.e.*, from 200 to 100Mhz) do not impact the execution time (*i.e.*,`RAM` memory bandwidth is far from being a run-time bottleneck).

- Design (c) exposes the highest cost, but also a high rendering quality as the engineers select a data-flow variant with `B` task and medium `D2` quality. However, this leads to an execution time dangerously close to non-functional constraints, thus presenting a risk to system responsiveness.

- According to its application variant, design (d) presents the highest rendering quality as it implements application variant with `B` processing and the highest `D2` resolution. Unfortunately, it seems that the NF constraint on execution time will definitely be violated. Moreover, `RAM` maximal storage capacity is also exceeded as it needs at least 2560KB.

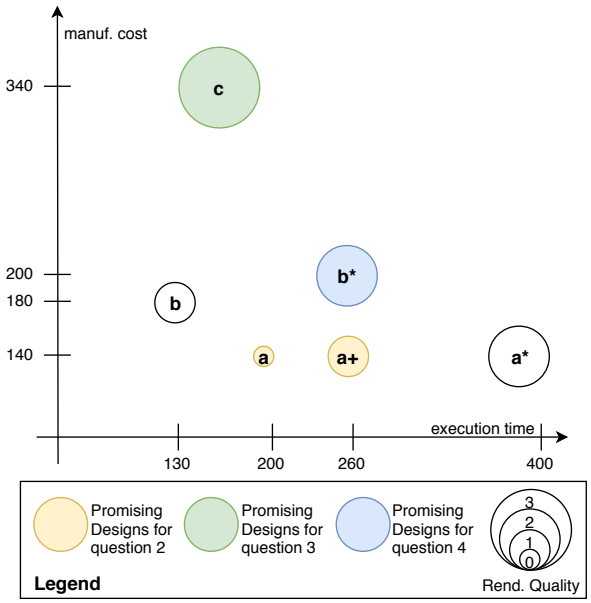


Figure 2.7: Pareto front with some of the optimum designs.

Fig. 2.7 shows a Pareto front [Hu et al., 2013] with some of the optimum designs. While some designs are clearly suitable solutions to engineering questions, others can be proposed as alternatives. For example, regarding question 2. Fig. 2.7 presents two risky solutions, a low quality design (a) and a close to deadline design (a+). One could advise the customer to select the design (b) to further developments as quality and run-time performance increase, respectively, up to 100% and 35% while the cost increase of 40%.

2.1.3 Discussion

The instrument cluster from which we extract the running example has been designed and implemented on a real instrument cluster by Visteon engineers. Thanks to decades in embedded system engineering, they could make - not without difficulties - suitable

design choices to finally get a prototype that meets and optimize requirements. If the platform had been still in development, the engineers would not have the possibility to prototype several designs to get a suitable solution.

Surprisingly, the size and complexity of the data-flows variants and platform configurations were medium to low, according to engineers. The extra difficulty may have emerged from the variability present in these specifications and their possible mapping and scheduling opportunities. This instrument cluster development was relatively recent (2016) and tended to corroborate that variability at both system specifications and implementation levels are constantly growing [Brugali and Hochgeschwender, 2017, Passos et al., 2016] in various software and embedded system industries.

Given 6 application variants and 19 platform configurations, the running example shows up 1139 feasible system designs can be prototyped, from which 939 meet the requirements while all the executions of the designs can be encoded in a state machine with 690 000 different states. In contrast, the majority of mid-end instrument clusters (systems specifications illustrated in Fig. 2.8), the number of application variants, and platform configurations could reach several hundreds of alternatives. A state machine of at least 59 000 000 states is needed to encode all the execution traces of the 34 560 designs (see Chap. 9).

Besides the variant spaces, the complexity of a mid-end instrument cluster is also much higher. In our example, data-flow variants have (on average), 2 source images, 3 tasks, 4 data-paths, 2218KB of data to process, and platform configurations have of 5 hardware functions, and 4850KB storage capacity provide by 2 storage memories. Whereas majority of instrument clusters have data-flow variants with 13 source images, 16 tasks processing 6656KB of data through 25 paths and platform configurations have a storage capacity of 20MB provided by 4 storage memories and 24 hardware functions.

2.2 Detailed Challenges

Finding suitable designs w.r.t requirements requires not only a way to deal efficiently with a variability-induced combinatorial explosion at the three levels (*i.e.*, application, mapping, platform), but also a way to reason simultaneously and efficiently about various facets of the problem emerging from different types of concerns: feasibility/satisfiability *and* optimality; functional *and* non-functional requirements; and different types of aspects: behavioural *and* structural aspects of the system design. Concretely, this requires efficiently solving all combinations of concerns on aspects. The different facets can be classified by concern as follows:

- *Satisfying the functional requirements (**FC**)*, that is, checking the structure of the design (*i.e.*, system variant); is it consistent in order to be implemented? (facet **FC-S**) and checking its behaviour (*i.e.*, execution); is the HMI rendering terminate without any error? (facet **FC-B**).

- *Satisfying the non-functional requirements (**NF**)*, that is, checking those that depend only on the structure: can it be implemented within this budget? (facet **NF-S**) and those that also depend on the behaviour; is the design behavior renders the HMI at a minimum of 30 frames per second? (facet **NF-B**).

- *Optimizing multiple quality attributes* (facet **NFO**). This facet requires considering all quality attributes simultaneously. Optimizing the structure; which ones

are the cheapest? (facet **NFO-S**) and then the behaviour; which ones are the fastest? (facet **NFO-B**) or vice-versa may lead to sub-optimal solutions.

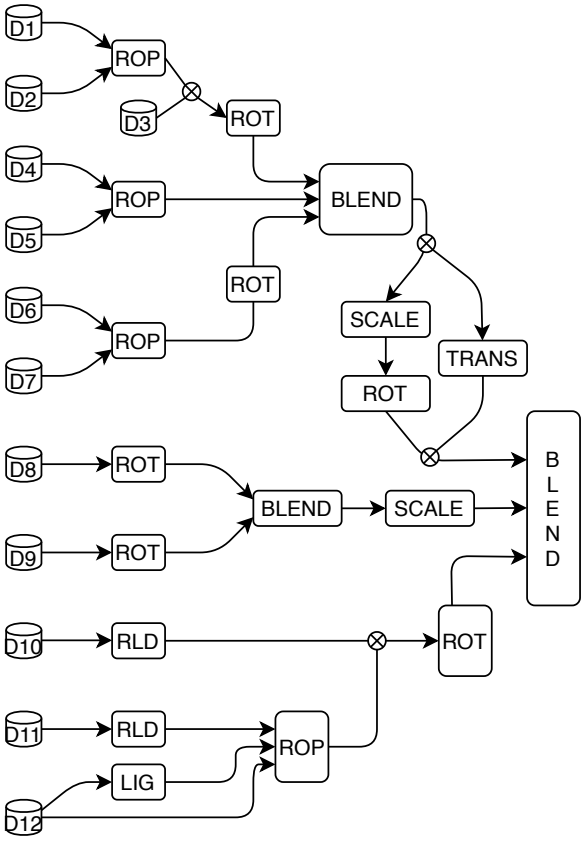
Reasoning on all the problem facets requires to analyze both design structure and behaviour simultaneously in order to check its functional feasibility, non-functional satisfiability, and optimality. As previously discussed, even for system experts, these activities are extremely tedious, time-consuming, and error-prone. High system variability at the three introduced levels (application, platform, and mapping) but also high complexity in system executions and scheduling prevents any enumeration-based exhaustive feasibility checking, let alone exhaustive reasoning/optimization on *quality attributes* (*e.g.*, cost, rendering quality, run-time). Given the complexity and variability of the vast majority of instrument cluster systems, systematic consideration of all design alternatives is unfeasible for system engineers. To proactively assist system engineers in finding suitable designs faster and with more confidence, efficient methods giving the means to make appropriate design choices at early stages of development appear as a necessity.

In this thesis, we propose a model-based system design framework. Instead of manually finding suitable designs, we propose a model-driven framework that captures, manages, and reasons on designs through model abstractions (*i.e.*, variable system specifications, mappings, and resulting variability-intensive design space). This approach reduces system prototyping gaps and makes possible design decisions at early development stage through model abstractions and simulations. In this approach, we determine three challenges to be tackled by this framework:

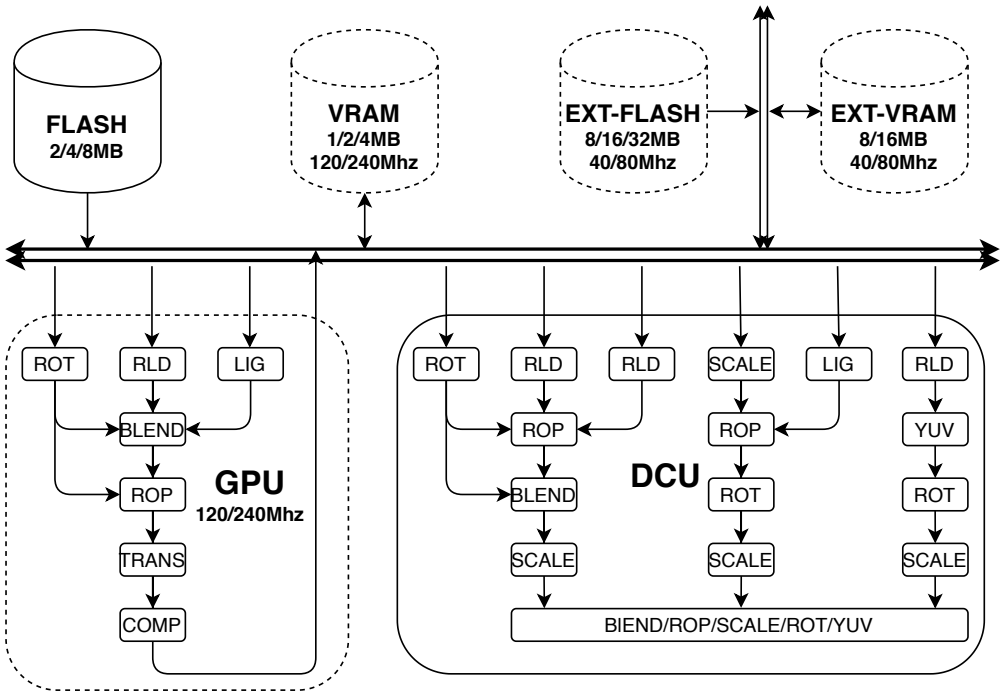
1. **Challenge 1**: Capturing functional and non-functional high-level variable requirements and specifications of embedded systems that can vary at both application and platform levels. The modeling method and languages should allow engineers to model many data-flow variants and platform configurations efficiently and at the adapted level of details. Imposing them to manually model all data-flow variant and platform configurations could be a threat to applicability in industry.

2. **Challenge 2**: Deriving automatically, from the application and platform models, the possible ways of mapping and scheduling application variants onto platform configurations (all consistent triplets of application, mapping and platform variants) and their behaviors. The resulting design space model has to take into account all variations from the system specifications (*i.e.*, application and platform) to system implementations levels (*i.e.*, mapping and scheduling) while capturing all the structural, behavioural, functional and non-functional aspects of each design alternative efficiently.

3. **Challenge 3**: Evaluating simultaneously and efficiently all facets of the problem **FC-S, FC-B, NF-S, NF-B, NFO**; the functional feasibility, the non-functional satisfiability, and optimality at both structural and behavioural aspects of each design alternative. The evaluation method should scale to industrial systems. Knowing that the design space grows exponentially with the number of design variation points and scheduling opportunities, exhaustive and enumeration-based methods may lack of scalability.



(a) Application specification



(b) Platform specification

Figure 2.8: More realistic size Instrument cluster specifications

Chapter 3

State of the Art

3.1 Model Based Design of Embedded Systems

"An embedded system is an engineering artefact involving computation that is subject to physical constraints" [Henzinger and Sifakis, 2007, Henzinger and Sifakis, 2006, Wymore, 2018]. These constraints emerge from limited hardware capabilities on which the embedded system is built. It is thus not possible to ignore hardware at both functional and non-functional level when designing embedded systems. Hardware abstraction that is generally present in software engineering cannot be reused in embedded system development. On the contrary, hardware and software artifacts are generally mixed (*i.e.*, hardware/software co-design [Edwards et al., 1997, De Michell and Gupta, 1997]) in order to meet the requirements that require to optimize the quality and performance of such systems while reducing their cost. Consequently, a holistic approach, combining requirement engineering [Macaulay, 2012], computation [Hopcroft et al., 2001] control theory [Abdelzaher et al., 2008], software [Van Vliet et al., 2008] and hardware design [Wescott, 2011] is often necessary.

Instead of directly prototyping designs, a model-based design framework captures and reasons formally on designs through model abstractions in order to generate system implementations [1] that fulfill the requirements [Edwards et al., 1997, De Michell and Gupta, 1997]. Nowadays, the complexity of system requirements and specifications leads to an immense number of system design alternatives. Such design spaces may be intractable to engineers. Strategies to support engineers to find the most suitable designs appear as a necessity.

We study model based-design of embedded systems as it proposes to identify by *design space exploration* the most suitable system designs from system specification models. A plethora of model-based design frameworks exists [2] [Densmore and Passerone, 2006, Sangiovanni-Vincentelli, 2007]. Some are software-centric, aiming to find suitable software design for a particular hardware architecture. On the other hand, hardware-centric frameworks help to design for a specific application domain. Finally, system-centric frameworks focus on the suitable mapping of an application onto hardware (and software) components. Our class of problems falls into a system-centric design paradigm.

Besides, a framework is built to model a particular kind of systems from control-dominant or dataflow-dominant systems (*e.g.*, train/plane controller, scheduling plan, IoT, data-flow-oriented or distributed system). Alternatively, some frameworks assess

[1] Process from which specifications are transformed to system is called *system synthesis*.

[2] Several hundreds.

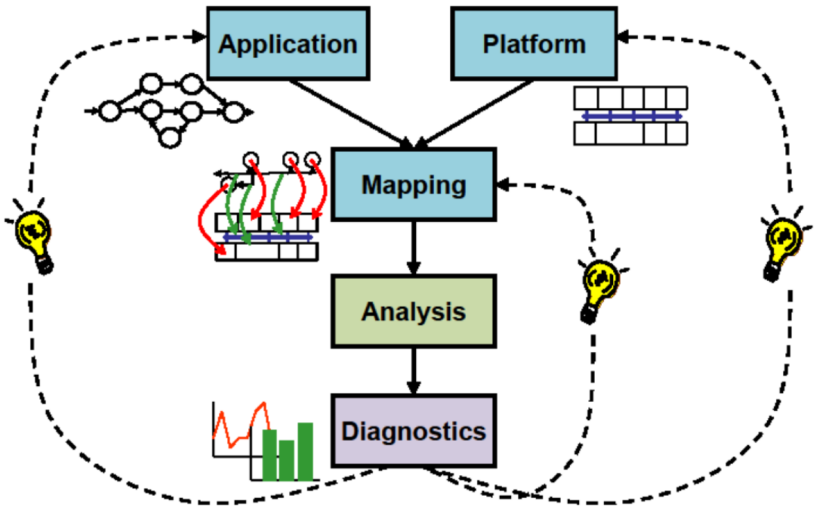


Figure 3.1: Y-Chart Design Space Exploration Pattern

a particular type of requirements or properties (*e.g.*, execution time, quality, energy consumption, bandwidth/throughput, temperature, reliability, security, robustness, quality assurance). The key factors of a framework are the quality of input models abstractions (*i.e.*, modelling time and expressivity) and quality of the evaluation method (*i.e.*, analysis time and correctness [3]). These factors are often intertwined and domain-dependent, which increases the variety of frameworks.

3.1.1 Y-Chart Pattern Overview

The Y-Chart pattern (see Fig.3.1) is a system-centric model-driven pattern. It offers a clear separation of concerns between application, platform, mapping, and analysis concerns based on models and models-transformations. This pattern consists of three stages. In the first stage, *a.k.a.* modelling stage, the system requirements and specifications are modelled by engineers. Platform specification represents the reusable architecture of hardware and software components. On the other side, the application specification only describes, independently to any specific platform or programming language, the functional [4] logic needed to fulfil the functional requirements.

Secondly, given these system specifications and non-functional requirements, the framework will map [5] the application onto the platform in order to derive a system design. The third stage is the analysis stage. The designs are generated into an analysis model such as analytical, computation, or simulation to evaluate the system design space. There is a profusion of methods to explore the design space [Gries, 2004, Singh et al., 2013, Singh et al., 2017] such as operational research, formal verification or simulation techniques. Random sampling, Monte-Carlo, Tabu, or best first search can optimize the design space exploration. When design space is huge, approximate exploration techniques such as genetic algorithm, simulated annealing, hill climbing can be used. At the end, even if impressing *exact* and *approximate* design space exploration techniques have been proposed, scalability is still an open issue [Gries, 2004, Singh et al., 2013, Singh et al., 2017].

[3] Also called implementation gap, to refer to the difference of performance between the model of the design and its physical implementation.

[4] Also called business logic.

[5] Words such as bind, deploy or implement are used in other domains.

3.1.2 Modeling and Mapping System Specifications

The expressiveness, semantics and level of details of the specifications models directly impact the correctness of the analysis results comparing to the *real* physical system. For example, to reach cycle-accurate accuracy with a close-to-zero implementation gap (*i.e.*, accuracy of $\tilde{1}00\%$ between the physical and modelled systems), C/C++/SystemC code and VHDL/VERILOG hardware logic are generally used by engineers to model, respectively, the application and platform. However, comparably to rapid prototyping, the time needed to model these *low-level* specifications and evaluate the entire design space at a cycle-accurate level is long and can take several months [6]. On the other hand, one can use generic models or domain specification languages (*e.g.*, UML Marte [Herrera et al., 2014], ADL [Grun et al., 1998]) to model application and platform. Such high-level formalism sacrifice accuracy to reduce the time needed to model and assess the design space.

Given application and platform specifications, a mapping strategy will automatically derive each system design for latter evaluation stage. Depending on the kind of applications and platforms, the adapted mapping strategy may differ. For example, given a data-flow or control-flow oriented application with static or dynamic workload and a homogeneous or heterogeneous platform, the mapping strategy may differ. Furthermore, non-functional requirements we focus on (*e.g.*, execution time, quality, cost, energy consumption, bandwidth/throughput, temperature, reliability) may also impact the mapping strategy. In the end, taxonomies have been proposed to classify hundreds of mapping strategies [Singh et al., 2013, Singh et al., 2017]. Following these taxonomies, our class of problem fall into design time heterogeneous *multi-application mapping on multi-platform*.

3.1.3 Assessing the System Design Alternatives

The evaluation method will assess and explore the resulting design space to find suitable designs *w.r.t.* functional and non-functional requirements. There are three main classes of evaluation methods; Analytical, computational and simulation-based analysis. The first class is based on declarative mathematical models. Relying on *Integer Linear Programming* (ILP and MILP) and *Constraint Satisfaction Problem* (CSP) resolution. "Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it." [Freuder, 1997]. At the contrary, *Model of Computations* [Lee and Sangiovanni-Vincentelli, 1998] (MoC) relying on formal methods [Sgroi et al., 2000] aims to model and verify computations of the system. Rather than "states the problem", the user "model the system" with a computational[7] formalism. Each system process (*e.g.*, task, memory, processor), their possible computations and interactions are represented by a set (network) of concurrent transition systems. Lastly, methods that aim to capture and mimic the behaviour of the physical system fall in the simulators class.

[6] This level of detail can be used to certify software, hardware or system design before any further development.

[7] Generally based on transition system with an execution semantics.

Design Space as a Constraint Satisfaction Problem

The root of analytical methods, such as CSP [Rossi et al., 2006] and ILP [Schrijver, 1998] are mathematical optimization and logic. Methods to maximize/minimize function objectives and find variable values for which the mathematical formula is satisfiable have been extensively developed since the 1950s. Modern and famous tools are CPLEX [8] and Gurobi [9]. Analytical formalism seems adapted to constraint satisfaction and constraint optimization problems.

Therefore, analytical models lack to capture concurrent system behaviour. Consequently, guarantee that the system satisfies temporal property [Pnueli, 1977, Alur and Henzinger, 1994, Bouyer et al., 2007](*e.g.*, such as deadlock-free, safeness, liveness) is limited. As an example, designing the cheapest railway controller that consume as less as possible energy is essential. Yet, it is fundamental to guarantee that the railway controller will always close the barrier (and have the battery to do so) while a train is crossing to avoid potential collision between trains and other vehicles.

Design Space as a Set of Model of Computations

On the other hand, system design frameworks that use model of computations [Lee and Sangiovanni-Vincentelli, 1998] allow to formally assess concurrent behavior in order to guarantee various property [Pnueli, 1977, Alur and Henzinger, 1994] and performance [Alur et al., 2001, Behrmann et al., 2001, Larsen et al., 2001, Behrmann et al., 2005, Zhang et al., 2017] trough famous model checkers such as *UPPAAL* and *SPIN*[10]. The train controller example can be designed and validated through *UPPAAL* [Behrmann et al., 2006] using *Weighted Computation Tree Logic* [Brihaye et al., 2004] (WCTL) to find optimal and guaranteed controller behavior.

While the analytical paradigm aims to find feasible and optimal solutions of a *mathematical* search space, *Model of Computation* MoC will simulate the system behavior in order to explore the *computational* state space. The fundamental MoCs are Automata 1943, Petri Nets 1962, Kahn Process Network 1974, Process calculi 1973, lambda calculus 1930. With the advent of software and system engineering, formal methods (*e.g.*, model checking, theorem proving, abstract interpretation) to verify and guarantee behavioural properties of critical systems (*e.g.*, military, nuclear plant, train controller, avionics, aerospace, ATM) have been developed. However, contrary to analytical methods, each system design alternative is encoded in a separate system model. Consequently, assessing the whole design space implies to evaluate a massive set of MoCs, which raise a scalability issue.

Multi-level Design Space Exploration

The three classes of evaluations have strength and weaknesses. Modern frameworks generally combine and mix evaluation methods in order to improve the design space exploration method [Pimentel et al., 2006, Bakshi et al., 2001] (*i.e.*, *multi-level* design space exploration). The exploration generally starts with analytical models to quickly prune *obviously-bad* design space regions. MoC can then be used to guarantee/optimize design behaviour formally. Finally, promising designs are simulated to mimic the "real system" and have a precise idea of the design's quality and performance. Yet, as the

[8] https://www.ibm.com/analytics/cplex-optimizer

[9] https://www.gurobi.com/

[10] Limited to functional assessment.

verification of the design's behaviour relies on MoC, each design is assessed iteratively. In contrast, analytical methods lack to verify concurrent systems. In the end, the tree class of evaluation lack of expressivity or scalability.

3.1.4 Model-Checking Pitfall

Model-checking analyzes state machine (also called transition system) and the state reachability relation (*i.e.*, which executions are reaching which states?). When a wrong system execution is detected, for example, a train controller execution that does not close the gate while the train is crossing the way, an execution *trac*e that violates the property (*i.e.*, a counter-example reaching a *bad* state) is generated as a diagnostic. However, each design alternative is modelled and assessed separately. Moreover, design space exploration methods that rely on MoC are usually facing some scalability issues[11].

State Space Explosion Problem

Given a *i)* state-based behaviour of a system and *ii)* a temporal properties formula that the system should respect. The model checking algorithm will automatically explore the system behaviour's state space to find events over executions that respect or violate required properties. Unfortunately, they suffer from one big and fundamental limitation, the state space explosion problem.

Nowadays, any industrial-scale embedded system has potentially a massive number of states. Often, the size of a state-space tends to grow exponentially in the number of its variables and processes. A variable of n bits could produce, at most, a state-space of 2^n states (*i.e.*, every different value lead to a different system state). With m variables of n bits, the state space could reach the impressive size of $2^{n \times m}$ different states. Moreover, interactions between concurrent components can produce a tremendous amount of states (*i.e.*, also known as processes interleaving). This interleaving also contains the scheduling aspect over process executions, which results in an enormous state space size $2^{p_1 \times p_2 \times \ldots \times p_n}$.

Variant Space Explosion Problem

Assessing every possible execution of every system variant seems inefficient. Yet, system design frameworks still assess their design space in iterative or enumeration-based manners. Advanced optimization techniques such as symbolic model checking [Biere et al., 1999], partial order reduction [Peled, 1998], state abstraction [Clarke et al., 1994] reduce the time needed to assess the state space of a single system. However, none of these optimizations aims to exploit structural nor behavioural commonalities between different but related design alternatives.

Every system state space is assessed independently, even if they exhibit common states or executions. Furthermore, as the number of system variant could growth exponentially according to design variation points at both application, mapping or platform levels, assessing the whole state space of every variant could be seen as another state space explosion factor [12].

[11] Heuristics are generally used to speed up the exploration process.

[12] Let F be the number of design variation points, 2^F the maximum number of variants, and S the average number of states by system variant. In the worst case, the number of states to explore to assess every variant is determined by $2^F \times S$

3.2 Software Product Line Engineering

A *Software Product Line* (SPL) is traditionally defined as "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission. Such features are developed from a common set of core assets in a prescribed way" [Clements, 2001]. Commonality and differences can be expressed in terms of features. Features intuitively characterize pieces of functionality in a software system. In general, not all feature combinations are considered valid products (*e.g.*, some features might be incompatible, mandatory, optional). To capture the set of valid products, feature diagrams can be used [Krzysztof and Eisenecker, 2000, Batory, 2005, Schobbens et al., 2007]. By defining hierarchy and constraints between feature, a feature model specifies the set of valid products *product line*.

In the end, the goal is to achieve mass customization [Pohl et al., 2005], *i.e.*, "the large-scale production of goods tailored to individual customers' needs" [Davis, 1989]. Rather than developing several similar systems (called products), SPL offer to develop a single configurable system from which similar system products could be generated *w.r.t.* their variations. We firstly study SPL applications to embedded system design and developments to pinpoints the main results and limitations. Secondly, we present several behavioral product line formalisms and evaluation methods. A behavioral product line aims to capture and assess also the behaviors of the whole product line.

3.2.1 Product Line Applications to embedded systems

There exist numerous applications of SPLE paradigm over various product lines of embedded system [Polzer et al., 2009, Botterweck et al., 2010, Liebig et al., 2009, Fischer et al., 2014, Brink et al., 2014, Streitferdt et al., 2005, Noir et al., 2016, Khalilov et al., 2016, Ross et al., 2017]. Rather than derives software-intensive systems, these applications generate consistent embedded system configurations. Systems are described as a set of configurable blocks that could be functional (*e.g.*, Simulink [13], Scade [14]), or hardware components (*e.g.*, VHDL, Sensors). All these approaches show the usefulness to capture all consistent embedded system configurations in a variability model to automate the implementation and verification of system configurations.

Some approaches [Siegmund et al., 2012, Murashkin et al., 2013, Zulkoski et al., 2014, Khalilov et al., 2016] extends product line paradigm with non functional attributes. A feature can have a structural cost (*e.g.*, manufacturing cost, reliability, quality). Only variants that satisfy or even optimize cost are derived for implementations or further analysis. Some attempts tried to capture also behavioural cost such as execution time, bandwidth and energy consumption as feature attributes. However, these costs emerge from behavioural interactions [Siegmund et al., 2015]. Feature interactions models tried to capture such performance influence but seem limited. Statistical learning [Valov et al., 2017, Guo et al., 2017, Zhang et al., 2015, Siegmund et al., 2015, Jamshidi et al., 2017a, Jamshidi et al., 2017b, Kolesnikov et al., 2019, kal,] can infer performance influence of highly configurable embedded systems. Prototyping or simulating a subset (*sample*) of system configurations is necessary to train the algorithm to predict the performance of the rest of the products.

[13] https://fr.mathworks.com/products/simulink.html

[14] https://www.ansys.com/products/embedded-software/ansys-scade-suite

3.2.2 Behavioral Product Line

A behavioural product line formalism captures not only the structure (*i.e.*, feature configurations) of every product but also their behaviours. Interestingly, the behavioural commonalities between different system variant are explicitly captured in a single transition-based system. Using this information, each state is checked once for every product that could produce it, leading to a significant speed up in verification time. Rather than assessing each product iteratively, variability-aware model checking techniques can assess efficiently every product's behaviour in a single run.

For instance, *Featured Transition System* [Classen et al., 2010a] (FTS) formalism is a transition-based model of computation that relies on states and transitions constrained by features to encode and analyzes through a variability-aware model checking algorithm both variant and state spaces of the entire product line efficiently. Extensions to validate a product line of real-time systems, featured timed automata [Cordy et al., 2012] have been proposed to capture and validate the whole real-time system product line efficiently. Thus, schedulability analysis [Boudjadar et al., 2013, Jaghoori et al., 2008], in terms of timing and deadline satisfaction, could be applied to real-time product line [Kim et al., 2016, Sabouri et al., 2012].

Other model of computations have been extended to capture and manage variability concerns such as modal transition system [ter Beek et al., 2016], modal I/O automata [Larsen et al., 2007], configurable parametric timed automata [Luthmann et al., 2017], product-line-CCS gruler2008modeling, featured Petri-nets [Muschevici et al., 2010]. Each formalism has a different syntax and semantics. Modal transition systems or modal I/O automata use optional "may" transitions enriched with feature constraints. Similarly, product-line-CSS is a process algebra with alternative choice operators between two process to model variability.

However, the behaviours are not directly linked to product (*i.e.*, features) which can be a major limitation. Given a particular product, deriving the possible behaviours could be unclear. By comparison, FTS formalism uses a feature model to capture the different products and a featured-automaton to capture their associated behaviours. There is no optional "may" transitions in FTS. Either a product could take the transition or not.

3.3 Discussion

Model-based design of embedded systems allows to derive and evaluate the whole design space of an embedded system from their system specifications. The Y-Chart approach offers a clear separation of concerns between specification modelling, mapping and analysis stage. Moreover, it is a domain-independent model-driven pattern that aims to limit coupling between modelling, mapping and analysis stages. Therefore, this flexible model-driven pattern allows to easily extends specification models or evaluation methods.

3.3.1 Challenge 1: Capturing Variable System Specifications

Besides the various level of details and formalisms in model-based system design engineering [Gries, 2004, Singh et al., 2013, Singh et al., 2017] no traditional approaches reach our **Challenge 1** as they do not capture our different variability dimensions of both application and platform specifications. They capture variability in application

(application space) side with *multi-modes* running [Negrean et al., 2013] or scenario awareness [Gheorghita et al., 2008, Palermo et al., 2008, Van Stralen and Pimentel, 2010, Schor et al., 2012]. Such variability is a run-time and dynamic one. Thus, applications could run in different modes or adapt to different scenarios. A *multi-variant* [Graf et al., 2013] or *variability-intensive* application, is a single application that has many possible possible realizations. The application variant is usually static and determined at design time[15]. However, some variability concerns such as alternative data-sizes and qualities are too fine-grained to be captured by a *multi-variant* application model.

Some methods allow to capture the platform variability (platform space) as a reconfigurable and flexible hardware architecture composed of FPGA or reconfigurable processor [Sigdel et al., 2009, Sima and Bertels, 2009, Wildermann et al., 2011a] This can even be done dynamically at run-time. Yet, these methods do not aim to capture a platform product line. Other approaches capture generic platforms where components can be optional and even have a variable multiplicity [Wildermann et al., 2011b, Graf et al., 2015]. However, fined grained configurable properties such as alternative memory capacities, hardware frequencies, and costs cannot be captured. Consequently, no input models capture a variable application efficiently and configurable platform specifications. In the end, the *Challenge 1* is not reached as none of the variability-aware approaches capture both concurrent behaviors and fined-grained variability aspects.

3.3.2 Challenge 2: Deriving Automatically the Design Space

In model-based system design engineering, a key aspect is to give the means to derive the system design space from higher-level system specifications (**Challenge 2**). Imposing the engineers to infer and model each possible system design manually is a major threat to any industry adoption. In our case, there are no specification models that capture our variable specifications. Therefore, no mapping strategy is directly reusable.

However, original approaches [Graf et al., 2013, Graf et al., 2014, Attarzadeh-Niaki and Sander, 2017] use boolean and multi-valued variables to encode both the variability of the application, the variability of the platform and the different mapping opportunities of application elements over platform resources. Mapping opportunities between application and platform are modeled with a decision tree. More precisely, the possible mappings of the *multi-variant* application and the *generic* platform are modeled in a CSP problem formulation. Some mapping constraints are added to prune incompatible mappings, application variant or platform configuration. Still, the mappings are not automatically inferred. Moreover, CSP formalism lacks to capture the concurrent behavioral aspect of a design space that allows checking temporal property. Consequently, the **Challenge 2**, is not reached as well.

In product-line engineering, the focus is to automatically model and manage the possible configurations to find the most suitable one. These frameworks do not capture embedded system specifications nor derive and explore the design space to find suitable designs. Deriving all possible mappings to model resulting system designs is problematic [Bokhari, 1981, Singh et al., 2013], as the activity would be tedious, time-consuming and error-prone. Consequently, no SPL approaches reach the **Challenge 2**.

[15] The variability is fixed before run time execution

3.3.3 Challenge 3: Evaluating Efficiently the Multifaceted Problem

The existing design space exploration methods do not efficiently address our multifaceted problem. Hence they do not reach the **Challenge 3** which is *Evaluating Efficiently the Multifaceted Problem*. Analytical methods lack to capture concurrent behaviour [Saraswat et al., 1991, Saraswat et al., 1994]. Consequently, assessing functional or non-functional behavioral properties (facets **FC-S**, **NF-S**) are limited. On the other hand, *Model of Computations* (MoCs) can capture and assess each system variant's behavioural properties. However, scalability may be an issue [Lee and Sangiovanni-Vincentelli, 1998, Aho, 2012].

In 2010, [Classen et al., 2010a] combine model checking with software product line [Classen et al., 2011b]. Resulting in a *Featured Transition System* (FTS) formalism, a new model of computation that capture the behaviour of the entire product line (*i.e.*, all behaviours of all products). However, only theoretical works exist to capture various behavioural resource cost [Fahrenberg and Legay, 2017a, Bauer et al., 2013]. Therefore, optimal resource scheduling can only be done *product-by-product*.

3.3.4 Conclusion

From decades of research in embedded system design, a plethora of model-based design frameworks have been developed and successfully applied to various cases. Given high-level application and platform specifications, such frameworks will automatically infer mappings from these two specifications to analyze the different design alternatives.

Model-driven *Y-Chart* pattern is applied to improve the usability and flexibility of the framework. However, application and platform input models lack to capture variable properties at both application and platform levels. Moreover, analysis methods are either limited to only structural or behavioral facets while *all-in-one* techniques are inefficient. Consequently, no framework allows system engineers to efficiently capture variability-intensive applications and highly-configurable platforms or analyze all facets of the resulting system design alternatives.

From combined researches in model checking and software product lines, *Featured Transition System* (FTS), a new model of computation has been developed. This formalism encodes and analyzes both structure and behavior of the entire product line efficiently. Interestingly, common states are checked once for multiple products. Yet, this formalism is limited to analyze the functional facets. Nevertheless, manually encode every possible design space decisions directly into an FTS is not feasible in practice. It would require to manually derive system design alternatives from variable specifications, which is too tedious and error-prone.

Part II

Contributions

Chapter 4

Framework Overview

In this chapter, we propose the first model-based design framework that combines Y-Chart pattern from embedded system design engineering and *Featured Transition System* (FTS) from product line engineering to capture and assess the functional feasibility, and only the functional feasibility, by capturing and assessing variable properties at both application and platform levels. The next chapter will extend the proposed framework to non-functional feasibility, non-functional satisfiability, and optimality by extending our Y-Chart models and FTS formalism with non-functional concerns. Extending and closing the gap between these two engineering domains, system design and product line, our novel framework aims at meeting the three defined challenges, each challenge being reached by extending a stage of the traditional Y-Chart pattern:

1. **Modeling stage:** to support variability-intensive application and highly-configurable platform specifications (Functional part of **Challenge 1**), we will extend application and platform models with variability concerns.

2. **Mapping stage:** to automatically infer an adapted system design space from these specifications (**Challenge 2**), we will propose a variability-aware mapping algorithm that maps variable application elements into configurable platform resources.

3. **Reasoning stage:** to reason functionally (facets **FC-S** and **FC-B**) on the adapted design space efficiently (Functional part of **Challenge 3**), we reuse the FTS behavioral product line formalism as a model of computation, to encode and evaluate efficiently functional requirements of the whole design space at once, determining each consistent variant (facets **FC-S**) with an execution that terminate properly (facets **FC-B**).

Given high-level variable application and configurable platform inputs that notably captures system specifications, the framework will automatically infer the design space by mapping each data-flow variant on each platform configuration, generate a FTS based behavioral product line from the system design space model, *i.e.*, representing all system designs (see Fig. 4.1) so that they can be directly checked in one run by a variability-aware model checker to validate the inferred system designs. As shown on Fig. 4.1, our framework mainly consists of two parts (*i.e.*, front-end and back-end) organized in three processes (*i.e.*, modeling, mapping and analysis stages) relying on four models. We give here an overview while the following sections will detail, illustrate and formalize these elements to show how the framework resolves our class of problem.

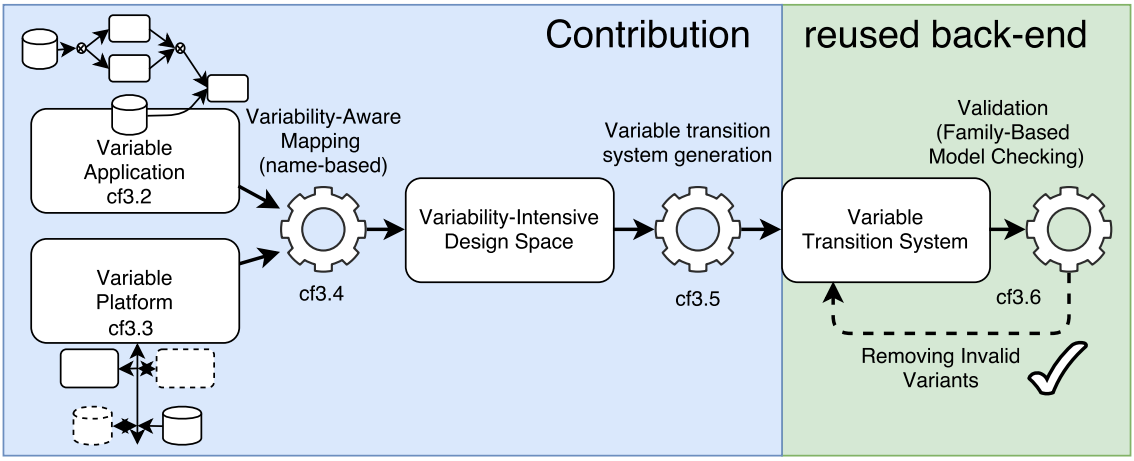


Figure 4.1: Framework Models and Processes

4.1 Modeling and Mapping System Specifications

- *Applications as Variable Data-Flows*: a functional expert is in charge of capturing the functional requirements of the embedded system through an extended concurrent data-flow specification (see Sec. 5.1). This model contains the classic structure and behavior of the data-flow (data, task, data-path, *etc.*), but also captures the variability in both structural properties (*e.g.*, data size) and behavioral properties (*e.g.*, alternative flows).

- *Platforms as Variable Resource Graphs*: on this side, a platform expert is in charge of expressing the platform specification as a templated concurrent component-based system (see Sec. 5.2). This model contains a set of components connected with each other. Similarly to the application one, the platform model also captures the variability of the defined components.

- *Variability-aware mapping*: The mapping algorithm (see Sec. 5.3) consumes the application and platform input models previously defined. It finds for each task data and data-path all the possible mappings on appropriate platform processors and memories; in order to generate the *Variability-Intensive Design Space*, *i.e.*, representing system designs. As the design space contains all the mapping of every application element on every platform component, the algorithm also generates constraints to prunes incompatible element mapping combinations *w.r.t.* structural, behavioral and variability constraints.

- *Variability Intensive System Design Space*: The Variability Intensive System Design Space captures all possible mappings of all application variants on all platform configurations. However, instead of having explicitly each system design as a different candidate, the variability intensive system design space has a single system design with explicit variation points that captures implicitly all system design variant and their design commonalities. Thus, each system design with a specific structure and behavior can be derived as an application variant, a platform configuration and possible mapping implementation between them.

4.2 Assessing the System Design Space

- *Design space as a behavioral product line*: From the system design space model, a *Behavioral Product Line* representing all system designs is generated (see Sec. 6.1). This product line is represented as a network of featured automata. A network of featured automata captures structure, behavior and variability of system elements (*e.g.*, task, processor, storage), so that we can reuse and adapt automated reasoning techniques to efficiently analyze automaton executions to identify valid and invalid system designs.

- *Variability-Aware Validation Process*: The validation process relies on variability-aware model-checking to assess structural and behavioral functional feasibility of the system variants represented by the behavioral product line (see Sec. 6.2). The network of featured automata, which represents the structure and behavior of the possible design alternatives, is checked against temporal property such as end state reachability to ensure that data-flows are correctly scheduled and executed onto the platform automaton. As a result, the validation solves and extracts all valid pairs of application variants and platform configuration (*i.e.*, respecting all structural, behavioral and variability constraints) in a single run. Additionally, we can provide the execution traces that satisfy the requirements. Such a trace shows not only the system variant able to execute it but also the execution they exhibit (*i.e.*, how the application tasks are executed and scheduled onto the platform), thereby helping the upcoming engineering of the designs.

The following Chapters will detail the different part of the framework. In the Chapter 5 we detail how our framework framework reach the functional part of both **Challenges 1 and 2** by provide *i)* input models that capture system specifications and *i)* a mapping algorithm than consumes these models to derive the system design space. In the Chapter 6, we detail how our resulting design space is transformed in a featured transition system in order to reuse a variability-aware model checking algorithm that assesses the functional requirements efficiently at both structural and behavioral aspects. Therefore reaching the functional part of the **Challenge 3** (*i.e.*, **FC-S** and **FC-B**). The Chapter 7 integrates the non functional concerns at both modelling (see Sec. 7.1) and assessing (see Sec. 7.2) stages in order to reach all the three challenges entirely.

Chapter 5

Modeling and Mapping System Specifications

To support variability-intensive application and highly-configurable platform specifications (**Challenge 1**), we propose adapted application and platform formal models extended with variability concerns. A single variability-intensive application model captures every application variants. Whereas a single highly-configurable platform model describes every platform configurations. Such models allow to capture system specifications that can vary at both application and platform levels.

To automatically infer an adapted system design space from these specifications (**Challenge 2**), we propose a variability-aware mapping algorithm that maps formally variable application elements into configurable platform resources. The resulting design space model has to take into account all variations from the system specifications (*i.e.*, application and platform) to system implementations levels (*i.e.*, mapping and scheduling) while capturing all the structural, behavioural and variability aspects.

5.1 Applications as Variable Data-Flows

Application models commonly capture the functional requirements defining "what do we want to do". More precisely, they capture the task to execute. Such models rely on various formalisms such as work-flow, data-flow, process network, Petri-nets, *etc.* In our case, Multiple data-flow variants specify the HMI content to be displayed into the car display. Each variant can differ in, *e.g.*, the size of the flowing data chunks, the ordering of the operation tasks, or the choice between alternative, functionally-equivalent, tasks (see Fig. 2.4).

However, none of the application models captures alternative task flows. Yet, alternative tasks and work-flows can be modeled by the *multi-variant model* [Graf et al., 2013]. Nevertheless, this model lack to capture some fine-grained variability concerns such as alternative data-sizes and qualities. Furthermore, this model is work-flow oriented. Meaning that the data path and data flows are not properly defined.

On the other hand, traditional data-flow graphs models such as [Kavi et al., 1986] capture every structural and behavioral aspect of our class of application but not the variability one. Modeling the different application variants (see Fig. 2.4) requires modelling each variant separately, which may lead to scalability issues in industry.

In order to capture the variability in our class of application, we propose to extend data-flow formalism that notably captures structure, behavior (data, task, data-path, *etc.*) of the embedded system through an extended concurrent data-flow model with

variability concerns. In the following, we formalize how each variability dimension, such as (data size, alternative flows, , *etc.*), is captured.

Definition 1 *A variable data-flow graph is a tuple :* $VDG = (N, Path, E, \Psi_{size}, \chi_d)$*:*

- $N = T \cup D$ *is a set of nodes (*T *are tasks and* D *are data),*
- $Path$ *is a set of communication paths by which data flows between producers and consumers (*i.e.*, tasks and datums),*
- $E \subseteq N \times Path \times T$ *is the set of flow processing between producers and consumers through available paths. Producers can be both input data or tasks while consumers are only tasks.*

$$I(t \in T) : \{p \in Path | (x, p, t) \in E\} \quad I(p \in Path) : \{prod \in N | (prod, p, x) \in E\}$$
$$O(n \in N) : \{p \in Path | (n, p, x) \in E\} \quad O(p \in Path) : \{t \in T | (x, p, t) \in E\}$$

$$|I(p)| \begin{cases} > 1, & \text{if } p \text{ has alternative producers} \\ = 1, & \text{if } p \text{ has a single producer} \end{cases}$$

$$|O(p)| \begin{cases} > 1, & \text{if } p \text{ has alternative consumers} \\ = 1, & \text{if } p \text{ has a single consumer} \end{cases}$$

$$|I(p)| + |O(p)| \begin{cases} > 2, & \text{if } p \text{ has alternative flows} \\ = 2, & \text{if } p \text{ has not flow variability} \end{cases}$$

According to the flow variability, $\theta_{alt}(n) \subseteq N$ *function returns the alternative nodes of node* n

$$\Theta_{alt}(n) : \{n' \in N | n' \neq n, \forall i \in I(n), n' \in O(i) \vee \forall o \in O(n), n' \in I(o)\}$$

Nodes n *are alternatives with* $\Theta_{alt}(n)$*. That are, alternative consumers of* n *input paths and alternatives producers of* n *output paths. Then* n *is in the variable nodes* $n \in Theta_{var}$ *if* $\Theta_{alt}(n) \neq \emptyset$*.*

- Ψ_{size} *is the datum size property,*
- $\chi_d : D \to (\Psi_{size} \to \nu)$ *defines the set of values* ν *that the size property* Ψ_{size} *of a data* d *can take.*

$$|\chi_n(d)(\Psi_{size})| \begin{cases} > 1, & \text{if } \Psi_{size} \text{ property of the data } d \text{ is variable} \\ = 1, & \text{if } \Psi_{size} \text{ property of the data } d \text{ is not variable} \end{cases}$$

In our vision, the functional expert captures efficiently the different data-flow variants in a single variable application model. Fig. 5.1 illustrates the variable data-flow-oriented application that the functional expert should define. This application model relies on the formal model previously defined. A single variable data-flow model represents multiple data-flow variants. To implicitly represent all these variants in a single model, we follow the same approach as in variable work-flows from [Graf et al., 2013], adding data-paths and allowing them to have multiple inputs and output tasks connected.

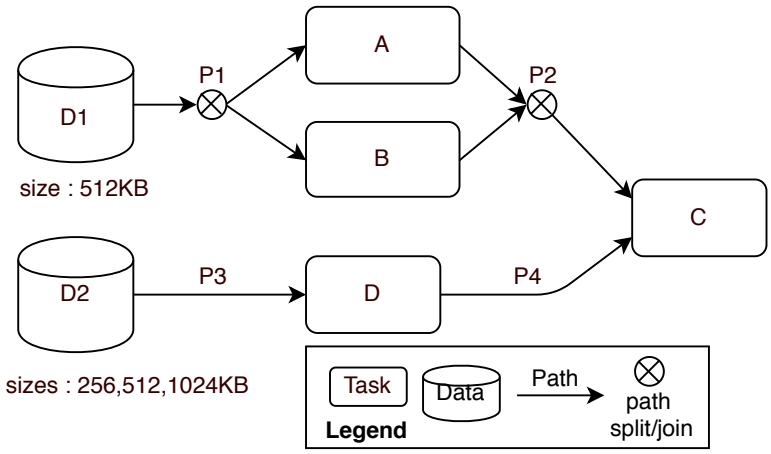


Figure 5.1: Variability-Intensive Application Functional Specification

Contrary to traditional data-flow models, a data-path can be connected to multiple alternatives to capture flow variability. If a path is connected to multiple input tasks $|I(p)| > 1$, alternative tasks can be then, at design time, selected to consume the data that will transit by this path. Similarly, $|O(p)| > 1$ means that an alternative task can produce the result that will transit by the path. But, If $|I(p)| = 1 \wedge |O(p)| = 1$, the data-path is connected to only one input and output task. In this case, the data-path has no flow variability.

As data can have alternative sizes, we also allow property to have a the set of alternative values. Each datum has at least one size $|\chi_d(d)(\Psi_{size})| = 1$. But if $|\chi_d(d)(\Psi_{size})| > 1$ the size of d is variable. Finally, in our case study the functional specification of the variable application $VDG_{uc} = (N_{uc}, Path_{uc}, E_{uc}, \Psi_{size}, \chi_{d_{uc}})$ is then formalized as:

- $N_{uc} \subseteq T_{uc} = \{a, b, c\} \cup D_{uc} = \{d_1, d_2\}, Path_{uc} = \{p_1, p_2, p_3\}$,
- $E_{uc} = \{(d_1, p_1, a), (a, p_2, c), (d_1, p_1, b), (b, p_2, c), (d_2, p_3, c)\}$,
- $I(a) = I(b) = p_1, I(c) = \{p_2, p_3\}$, $O(a) = O(b) = p_2, O(c) = \emptyset, O(d_1) = p_1, O(d_2) = p_3$,
- $I(p_1) = d_1, I(p_2) = \{a, b\}, I(p_3) = d_2$, $O(p_1) = \{a, b\}, O(p_2) = c, O(p_3) = c$,
- $\chi_{n_{uc}}(d_1)(\Psi_{size}) = 512, \chi_{n_{uc}}(d_2)(\Psi_{size}) = \{512, 1024\}$.

5.2 Platforms as Variable Resource Graphs

Platform models describe a set of hardware components such as memory, multi-pass processors, streaming processor, read-only memory, read-write memory, write-only memory, first-in-first-out buffers (FIFO) *etc.* Transition systems such as automata or Petri-nets usually capture components behavior. They interact with each other by communication port, link, connection, channel, *etc.* By providing algorithms to implement and execute particular tasks, store and move data *etc.* such component system captures the whole platform capabilities and define "what can we do" using this platform.

Hardware providers generally offer a platform product line for each range of applications where components can be optional, exclusive, *etc.* and have configurable

properties. For example, memory can have alternative storage capacities. Thus, multiple platform configurations are possible at hardware manufacturing time. In industry, a huge set a platform configurations could result from this variability.

Some methods allow to capture the platform variability as a reconfigurable and flexible hardware architectures [Sigdel et al., 2009, Sima and Bertels, 2009, Wildermann et al., 2011a]. FPGA or reconfigurable processor focuses on optimizing the integrated circuits (silicon area) regarding the instruction streams. This can even be done dynamically at run-time. Yet, these methods do not aim to capture a platform product line. In our context, each possible manufactured hardware configuration is a different hardware product.

Other approaches capture generic platforms. Components can be optional and even have a variable multiplicity [Wildermann et al., 2011b, Graf et al., 2015]. Yet, configurable properties such as alternative memory capacities, hardware frequencies and costs cannot be captured. Moreover, these approaches do not capture concurrent hardware behavior.

In order to capture a platform that can have optional resource components, variability constraints on components (*e.g.*, dependency, incompatibility) and variability properties of the component (*e.g.*, memory storage size), we propose a formal architecture model defined as follows.

Definition 2 *A variable resource graph is a tuple* $VRG = (R, C_s, \Theta, \Psi_{cap}, \chi_r)$ *where :*

- $R = P \cup S$ *is a set of resources where* P *is a set of processors and* S *is a set of storage memories,*
- $M = S \cup B$ *is a set of memories where* S *is a set of storage memories and* B *is a set of buffered (*e.g.*, first-in-first-out) memories,*
- P *is a set of processors* $p = (F, B, C_b \subseteq (F \times B) \times (B \times F))$ *where :*
  - F *of possible hardware fixed functions,*
  - B *is a set of processor internal first-in-first-out buffers*
  - $C_b \subseteq (F \times B) \cup (B \times F)$ *the connections between the different functions and buffers representing the possible processing pipelines of the processor.*
- $C \subseteq (P \times S) \cup (S \times P)$ *is a set of connections between processors and memories, where:*
  - $(f, s) \in C$ *means that the hardware function* f *can write onto storage memory* s*,*
  - $(s, f) \in C$ *means that the hardware function* f *can read from storage memory* s*;*

  The set of, input memories to a, processor $I(p)$*, processor function* $I(f)$*, and output memories from a, processor* $O(p)$*, processor function* $O(f)$ *are denoted by:*
  - $I(p \in P) : \{m \in M | (m, p) \in C_s\}, I(f \in F) : \{m \in M | p = (F, B, C_b) \in P, (m, f) \in C_b \vee m \in I(p)\}$,

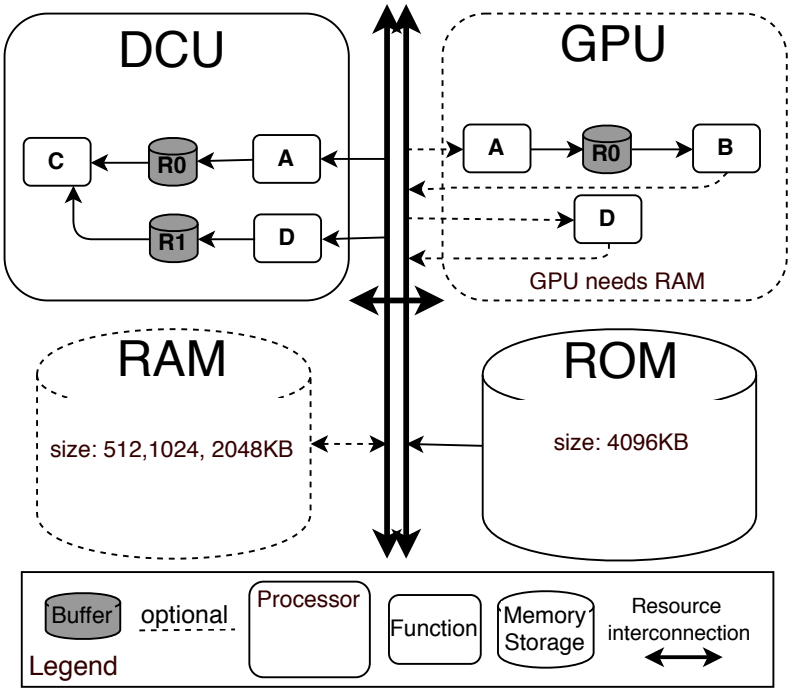


Figure 5.2: Highly-Configurable Platform Functional Specification

  - $O(p \in P) : \{m \in M | (p, m) \in C_s\}, O(f \in F) : \{m \in M | p = (F, B, C_b) \in P, (f, m) \in C_b \vee m \in O(p)\}$.

- $\Theta \subseteq 2^R$ *defines which resources may/must be present together;*
  - $\Theta_{Opt} : \{r \in R | \exists R' \in \Theta \wedge r \notin R'\}$
  - $\Theta_{Req}(r) : \{r' \in Opt | \forall R' \in \Theta \implies (r \in R' \implies r' \in R')\}$
  - $\Theta_{Exc}(r) : \{r' \in Opt | \forall R' \in \Theta \implies (r \in R' \implies r' \notin R')\}$
- Ψ_{cap} *is the storage capacity property,*
- $\chi_r : R \to (\Psi_{cap} \to \nu)$ *defines the set of values* ν *of the storage capacity property* Ψ_{cap} *that a memory storage* s *can take.*

At this level, a platform expert is envisioned to be in charge of modeling the platform product line provided by the hardware manufacturer (see Fig. 2.5) as a variable resource graph model. This model can be seen as a templated concurrent hardware component-based system extended with variability concerns. The Fig. 5.2 illustrates the variable resource graph model that the platform expert should define in order to capture the platform product line.

We capture implicitly the platform variability by introducing several functions, θ manages the optionality of a resource component. If $\theta(r) = \perp$ the resource is mandatory, otherwise the resource is optional, $\phi_{requires}$ and $\phi_{excludes}$ manages constrained relations of dependency and incompatibility between resource components. $\phi_{requires}(r) = r_0$ means that if r is implemented then r_0 must be implemented too. r depends on r_0. On the contrary $\phi_{excludes}(r) = r_0$ means that r and r_0 cannot be implemented on the same platform variant. r and r_0 are alternatives.

As memory storage can have alternative capacities, we introduce the function ξ which returns the set of alternative capacities $C = \xi(s)$, each memory storage has at least one size and if $|\xi(s)| > 1$ the capacity of s is variable.

In our case study the functional specification of the configurable platform VG_{uc} is then formalized as:

- $(Proc_{uc} = \{DCU, GPU\}, S_{uc} = \{RAM, ROM\},$
  $C_{s_{uc}} = \{(RAM, DCU), (ROM, DCU), (RAM, GPU), (ROM, GPU), (GPU, RAM)\})$
  where,
- $DCU = (F_{dcu} = \{a_{dcu}, c_{dcu}\}, B_{dcu} = r0_{dcu}, C_{b_{dcu}} = \{(a_{dcu}, r0_{dcu}), (r0_{dcu}, c_{dcu})\}),$
  $GPU = (F_{gpu} = \{a_{gpu}, b_{gpu}\}, B_{gpu} = r0_{gpu}, C_{b_{gpu}}\{(a_{gpu}, r0_{gpu}), (r0_{gpu}, b_{gpu})\})$, with,
- $I(GPU, a) = \{ROM, RAM\}, O(GPU, a) = \{r0_{gpu}, RAM\},$
  $I(DCU, c) = \{r0_{dcu}, ROM, RAM\}, O(DCU, c) = \emptyset.$
- $GPU, RAM \in \theta_{Opt}, DCU, ROM \in \theta_{Mand}, \theta_{Req}(GPU) = RAM, \theta_{Req}(RAM) = \emptyset$
- $\chi_{r_{uc}}(ROM)(\Psi_{cap}) = 4096, \chi_{r_{uc}}(RAM)(\Psi_{cap}) = \{512, 1024, 2048\},$

5.3 Variability-Aware Mapping Strategy

The mapping algorithm takes as inputs the variable data-flow and configurable platform models, previously defined by the domain experts, and generates a mapping model. Such models aim to capture the possible mappings of the application elements over the platform resources. Each mapping variant could raise consistency issues. For instance, a task has been mapped on a non-selectable processor. Worst, a set of alternatives task have been mapped as thought the platform should process all of them. To ensure mapping consistency from variable specifications, extra constraints are needed [Graf et al., 2013, Graf et al., 2014, Attarzadeh-Niaki and Sander, 2017].

We propose a mapping strategy to not only capture the consistent mappings of a single data-flow into a single platform as traditional frameworks do [Balarin, 1997, Eker et al., 2003, Bakshi et al., 2001, Davare et al., 2007, Basten et al., 2010, Pimentel et al., 2006], but to capture the consistent mappings of every data-flow variants onto every platform configurations. First, we define the mapping model and the required consistency constraints. Second, we propose a mapping algorithm that derives the design space from the variable data-flow and configurable platform models previously defined by the domain experts. We finally illustrate our mapping strategy in our case study to demonstrate its validity.

Definition 3 *A variability-aware data-flow-oriented mapping* $VM = (Map, Cst)$ *where:*

- $Map \subseteq (Tm \subseteq T \times F) \cup (Dm \subseteq D \times S) \cup (Pm \subseteq P \times M)$ *is the set of application element mappings over platform resources where:*
  - Tm *is the set of possible mappings of tasks on processors* $\forall (t, F) \in Tm$, t *can be mapped on processor function* $f \in F$ *because* f *can implement* t*, and the inputs of* t *can mapped on memories that can be reached by* f*,*
  - Dm *is the set of mappings of datum on storage memories,*
  - Pm *is the set of paths mapping on memories by which data can be produced,*

- *$Cst \subseteq (Cc \subseteq (N \times R \times P) \times R) \cup (Pc \subseteq (N \times R \times P) \times R)$ is the set of node mappings to path mappings constraints, where:*
  - *Cc is the set of consistency constraints that limit path mappings according to the consumer mappings. Map the node n on the resource r restrict the allowed mappings of the input paths of n, noted $i(n) \subseteq Path$ to reachable memories of r, noted $i(r) \subseteq M \subseteq R$,*
  - *Cp is the path mapping constraints according to the producer mappings. Similarly to Cc, Cp constraints the possible mappings of the output-paths depending on node mappings.*

The Variability-Aware Mapping function $M = VDG \times VRG \rightarrow VM$ sorts the data-flow topologically and finds appropriate mapping for each data, task and data-paths of the data-flow using resources of the resource graph. Each valid mapping should respect some mapping to mapping *constraints. These constraints ensure that all the mapped application over the platform resources are consistent.*

(1) All tasks are mapped to, a least, one processor function:

$$\forall t \in T, \exists (t, F) \in Tm, |F| > 0$$

(2) All data are mapped to, at least, one memory storage:

$$\forall d \in D, \exists (d, S) \in Dm, |S| > 0$$

(3) All data-paths are mapped to at least one appropriate memory. For data-path starting by a datum, the memory on which the datum is mapped has to be reachable by a processor function on which the task consuming the datum is mapped.

$$\forall e = (n, p, t) \in E, \exists (p, M) \in Pm, \exists (t, F) \in Tm, m \in M \wedge f \in F \wedge m \in I(f)$$

For data-path between tasks, the memory on which the output is mapped has to be reachable by the processor function on which the consumer is mapped.

$$\forall e = (t \in T, p, t') \in E, \exists (p, M) \in Pm, \\ \exists (t, F_t) \in Tm, \exists (t', F'_t) \in Tm, m \in O(f) \wedge m \in I(f')$$

These *mapping to mapping* constraints restrict the mapping opportunities to ensure that any possible mapping respects variable resource graph topology.

For every task mapping on a processor, the input and output data-paths must be mapped to reachable memory by the processor. Otherwise, the processor may not be able to fetch or store the required data. Beside *mapping to mapping constraints* there are two additional classes of consistency constraints that restrict the allowed mappings regarding application and platform variabilities. Application to mapping constraints and mapping to platform constraints. The first is concerning the alternative application flows, while the second is depending on optional platform resources. Obviously, if a resource is absent, the mappings using it are not possible anymore. Similarly, only elements of the application flow will require to be mapped.

The Algorithm 1 implements the variability-aware Mapping function. Basically, each input datum can be mapped on a storage memory (Line 1). Paths by which the datum can be consumed can be then mapped on storage memories (Line 2). Mapping opportunities are then recorded in the following form, if the data d is mapped on

Input: $app = (N,\ Path,\ E,\ \Psi_{size},\ \chi_d)$
$plt = (R,\ C_s,\ \Theta,\ \Psi_{cap},\ \chi_r)$
Output: $vm = (Tm, Dm, Pm, Mc, Mp)$

1 $Dm \leftarrow \bigcup_{d \in D} (d, S)$;
2 $Pm \leftarrow \bigcup_{d \in D, p_{out} \in O(d)} (p_{out}, S)$;
3 $Mp \leftarrow \bigcup_{d \in D, p_{out} \in O(d), s \in S} ((d, s, p_{out}), s)$;
4 $Mc \leftarrow (\emptyset)$;
5 **foreach** $t \in T$ **do**
6 $\quad Fcts \leftarrow \{f \in F | f \models t \wedge \forall p_{in} \in I(t), I(f) \cap \{M_{p_{in}} | (p_{in}, M_{p_{in}}) \in Pm\} \neq \emptyset\}$;
7 $\quad Tm \leftarrow Tm \cup (t, Fcts)$;
8 $\quad Pm \leftarrow Pm \bigcup_{p_{out} \in O(t)} (p_{out}, \bigcup_{f \in Fcts} O(f))$;
9 $\quad Mc \leftarrow Mc \bigcup_{f \in Fcts, p_{in} \in I(t)} ((t, f, p_{in}), I(f) \cap \{M_{p_{in}} | (p_{in}, M_{p_{in}}) \in Pm\})$;
10 $\quad Mp \leftarrow Mp \bigcup_{f \in Fcts, p_{out} \in O(t)} ((t, f, p_{out}), O(f))$;
11 **end**
12 **return** vm

Algorithm 1: $M(app, plt)$

storage s then the output data paths $\{p_0, p_1, ..., p_n\} \subseteq O(d)$ must be mapped on s too (Line 3). Next, for all the tasks, the hardware processing functions capable of implementing the task (Line 6) and capable of grabbing all the inputs (*i.e.*, all the input paths are, at least, mapped on one reachable memory) are identified (Line 7) and inserted (Line 8). For this consistency reason, mapping dependencies on the reachable inputs mappings of every task mappings are recorded (Line 9). Similarly, the mapping of the output paths (containing processing results) can only be mapped on every memory reachable by any of these hardware functions (Line 10). These mapping opportunities are recorded (Line 11) in the form, if the task t are mapped on the function f then output data paths $O(t)$ must be mapped on reachable (and writable) memories by f(*i.e.*, $O(f)$).

The mapping model of the case study VM_{uc} is then illustrated in Fig. 5.3 and formalized as:

- $(Tm_{uc} = \{(A, \{DCU_A, GPU_A\}), (B, GPU_B), (D, \{DCU_D, GPU_D\}), (C, C_{DCU})\}$,
- $Dm_{uc} = \{(D_1, \{ROM, RAM\}), (D_2, \{ROM, RAM\})\}$,
- $Pm_{uc} = \{(P_1, \{ROM, RAM\}), (P_2, \{DCU_{R0}, GPU_{R0}, RAM\}),$
  $(P_3, \{RAM, ROM\})\}), (P_4, \{DCU_{R1}, RAM\})\})$
- $Mc_{uc} = \{((A, DCU_A, P_1), \{ROM, RAM\}), ((A, GPU_A, P_1), \{ROM, RAM\}),$
  $((B, GPU_B, P_1), \{ROM, RAM\}),$
  $((D, DCU_D, P_3), \{ROM, RAM\}), ((D, GPU_D, P3), \{ROM, RAM\}),$
  $((C, DCU_C, P_2), \{DCU_{R0}, RAM\}), ((C, DCU_C, P_4), \{DCU_{R1}, RAM\})\}$
- $Mp_{uc} = \{((D_1, ROM, P_1), ROM), ((D_1, RAM, P_1), RAM),$
  $((D_2, ROM, P_3), ROM), ((D_2, RAM, P_3), RAM),$
  $((A, DCU_A, P_2), \{DCU_{R0}\}), ((A, GPU_A, P_2), \{GPU_{R0}, RAM\}),$

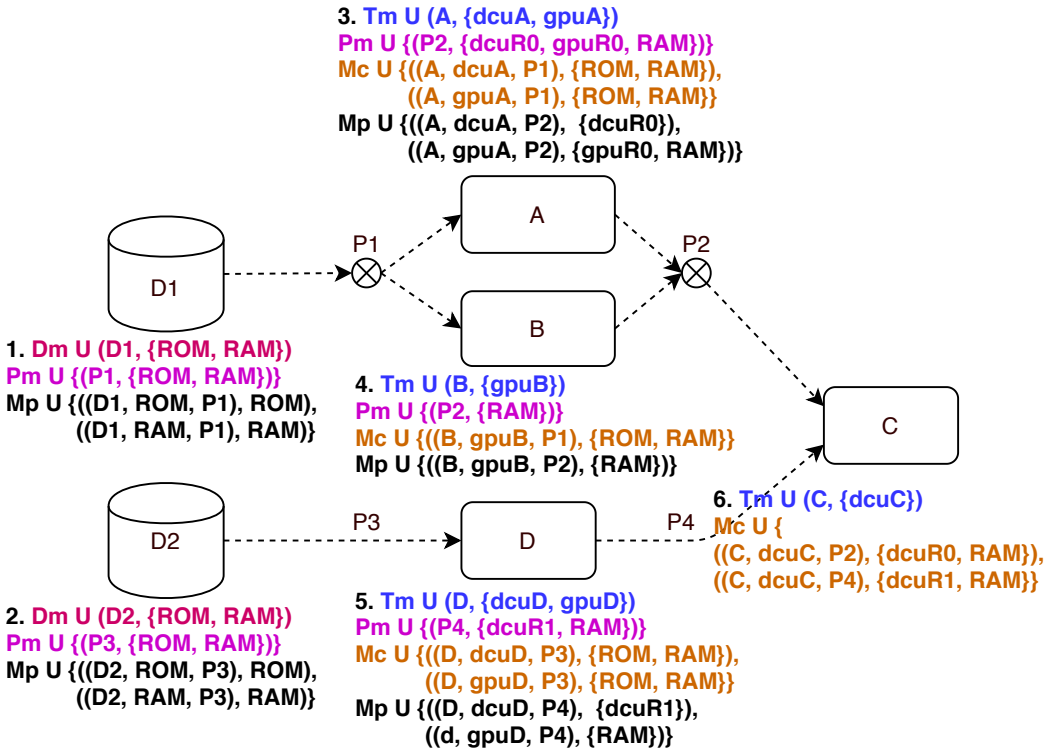


Figure 5.3: Application Mapping Steps

$((B, GPU_B, P_2), \{RAM\}),$
$((D, DCU_D, P_4), \{DCU_{R1}\}), ((D, GPU_D, P_4), \{RAM\})\}$

Finally, The design space representing all system designs, called *variability-intensive embedded system design space* is then composed of a data-flow, platform and mapping:

$$VDS = (VDG \times VRG \times VM)$$

5.4 Conclusion

In this chapter, we proposed input models to capture variable application (see Sec. 5.1) and configurable platform (see Sec. 5.2) system specifications. We extended traditional data-flow and platform component-based system with variability concerns. Therefore, the proposed models are expressive enough to capture structural, behavioral and variability aspects. As we discussed in Sec. 3.3.1, traditional models do not capture variability aspect while some others do not capture concurrent behaviors.

To derive the system design space automatically from the variable specifications, we developed a mapping algorithm (see Sec. 5.3). Our mapping model is represented by a decision tree with consistency constraints. Such structure share similarities with approaches that manually model their mapping space in a CSP formulation (see Sec. 3.3.2).

In the next Chapter 6, we detail how our design space is transformed in a *Featured Transition System* (FTS) in order to assess efficiently functional requirements at both structural and behavioral aspects.

Chapter 6

Assessing the System Design Alternatives

To evaluate simultaneously and efficiently the functional feasibility of our system design space (functional part of **Challenge 3**), we transform our design space model formally, previously defined, into the well known *Featured Transition System* (FTS). Firstly, we show that this behavioral product line formalism, composed by a state transition system and a feature diagram, is suitable for capturing both behavioral and structural aspects of our design space. Secondly, we illustrate how state-of-the-art variability-aware model checking techniques can be applied to assess both structural (**FC-S**) and behavioral (**FC-B**) functional requirements of the whole design space at once.

6.1 Design Space as a Behavioral Product Line

6.1.1 Background

To assess their system design space, traditional frameworks usually transform iteratively, for verification purpose, each design into analytical, simulation or computational models. Automata theory and model-checking techniques have been widely used to assess behavior of real-time and concurrent embedded systems [Bengtsson et al., 1996,Clarke et al., 1999]. Basically, the possible executions of a system (*i.e.*, its behavior) are captured through a state machine. Model-checking algorithms are then used to explore the state space efficiently (*i.e.*, the different states of the system) in order to validate or invalidate system behavior against functional requirements expressed as temporal logic properties [Pnueli, 1977] such as the absence of deadlock, end-state reachability, liveness, safety, *etc.*

The traditional method for our class of problem is to model each system variant's behaviour as two communicating automata. One capturing the behavior of the application and the other capturing the platform behavior. While the application automaton represents task and data behaviors, the platform one exhibits the processor and memories behavior. The application automaton have the responsibility to map and schedule tasks and transfer data using the platform automaton. The application and platform automata are usually networks of automata as they require to capture concurrent and communicating systems. Such systems are consisting of several interacting automata. The automaton capturing the whole state space of the system is then defined as their parallel composition (see Sec. 6.2). In the end, this state-based

model is a relatively high-level abstraction of the possible system executions[1]. Finally, end-state reachability property is checked [Dalsgaard et al., 2010] to ensure that the application could execute on the platform without error.

Definition 4 *An automaton is a tuple* $A = (S, s_0, S_f, Act, Trans, Ap, L)$ *such that:*

- S *is a finite set of states,* $s_0 \in S, S_f \subseteq S$*, are, respectively, a set of initial and final states,*
- Act *is a set of actions that are possibly composed of variable expressions and affectations, guards in a boolean expression form and rendez-vous communications through channels,*
- $Trans \subseteq S \times Act \times S$ *are state transitions,* $(s, \alpha, s') \in Trans$ *can be noted* $s \xrightarrow{\alpha} s'$
- Ap *is a set of atomic proposition and* $L : S \rightarrow Ap$ *labels states with atomic propositions such as* $L(s) \subseteq \mathcal{P}(Ap)$*.*

Figure 6.1 and 6.2 illustrate how, respectively, the behavior of two suitable system variants B_Ψ and D_Ψ[2]. Design B and D of our case study (previously introduced in Section 2.1.2) can be, respectively, captured into network of automata $B_{\Psi A}$ and $D_{\Psi A}$. The application automaton is then composed of input datum and task automata, while the platform automaton is composed of storage and processor automata. For example, application automaton of $B_{\Psi A}$ (see Fig. 6.1) exposes 2 input datum automata, capturing respectively the behavior of $D1$ and $D2$. Automata TA, TD and TC capture tasks behavior with one initial, final and possibly multiple progress states. Datum size are captured through state variables. Paths become *handshaked rendez-vous* communication channels [Bengtsson et al., 1996] used by task and input datum to send datum output completion signals before reaching their final state.

On the platform side, automata RAM and ROM capture the behavior of hardware storage while DCU captures the processor. Trough *handshaked rendez-vous* channels, graphical functions of processors, and fetch store mechanism are based on command (as a real hardware will do). By sending commands (*i.e.*, message through communication channels), application automaton is driving and feeding the platform one. Basically, datum and task automata process will interact with storage and processor automata process in order to allocate memory on storage or program processors pipeline. In addition to initial and progress states, storage and processor automata contain an error state reached if the storage capacity (*e.g.*, $RAM.free$) is exceeded or the processor pipeline (*e.g.*, $DCU.checkCmdBuffers()$[3]) is misused during an execution (see Sec. 6.2).

At the mapping level. Images $D1$ is stored onto ROM and $D2$ on RAM. At the run time execution level the images will send a *malloc*! command through channel of, respectively, ROM and RAM automata. The data-flow processing, composed of tasks A, D and C, is implemented using the DCU hardware pipeline as the tasks automata will send graphical task such as $DCUa$!, $DCUb$!, $DCUd$!, $DCUc$!, *etc.* commands through channel of DCU automaton. Exploring all executions of the system

[1] software/hardware executions of a system design

[2] We denote Ψ the structural aspect of the design, its system variant, while the behavioral one, its execution, will be denoted π see Sec. 6.2

[3] This function is highly dependent to the hardware internals and will not be detailed.

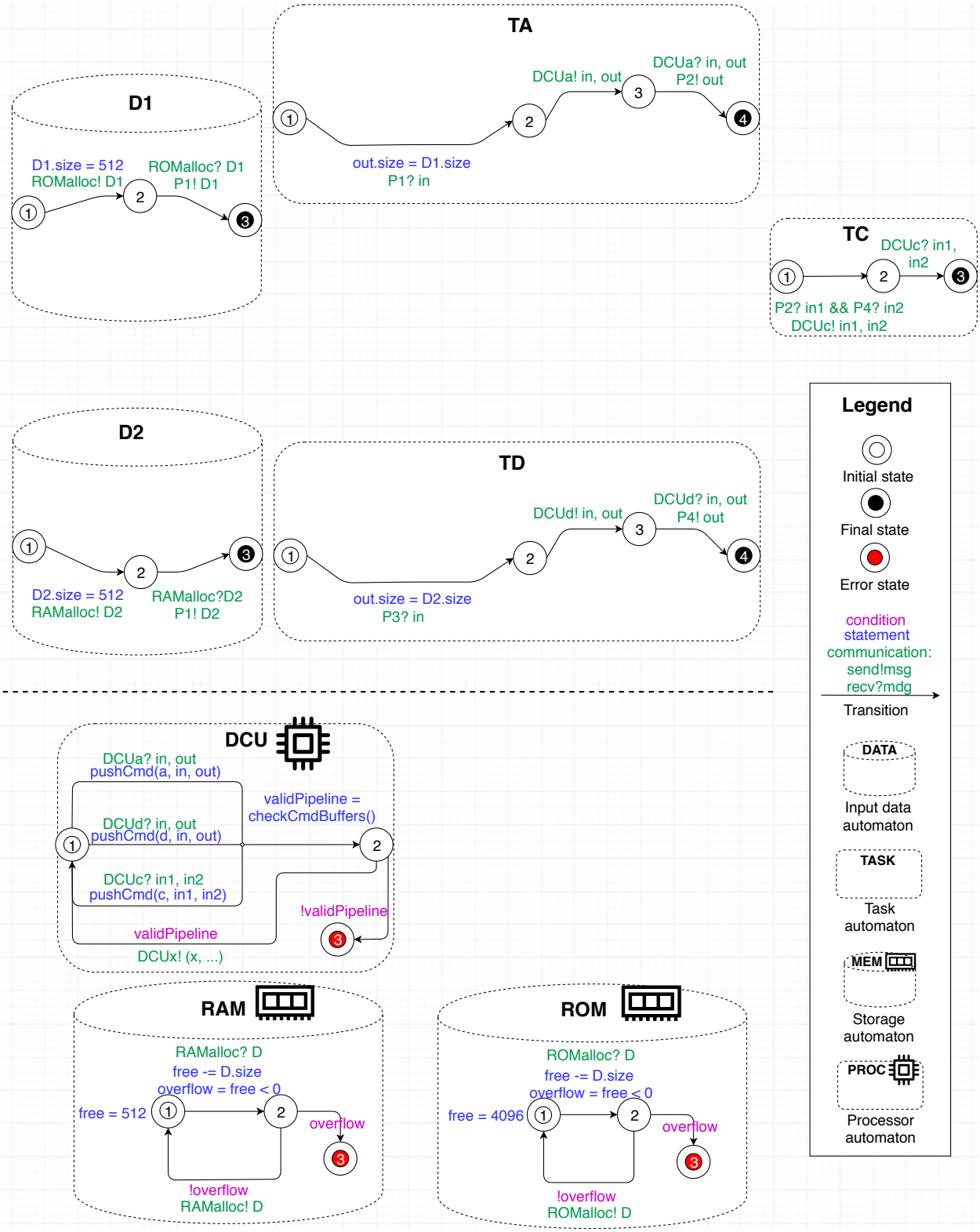


Figure 6.1: Automaton capturing System Variant of Design B

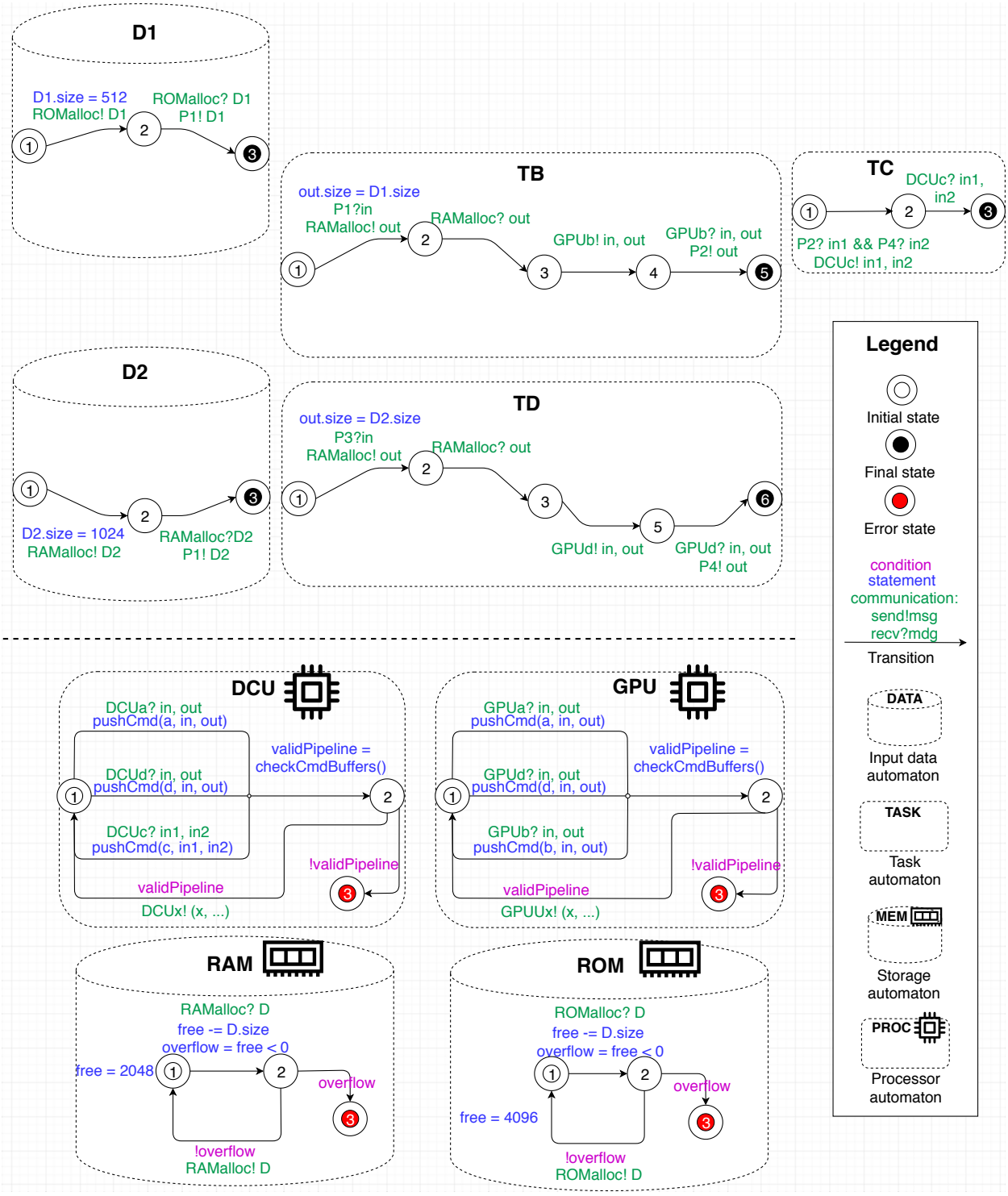


Figure 6.2: Automaton capturing System Variant of Design D

automaton, also called covering the state space, will determine if the system variant exhibit a behavior that satisfies the functional requirements (see Sec. 6.2).

Between the two state machines capturing, respectively, behavior of B_Ψ (see Fig. 6.1) and D_Ψ (see Fig. 6.2), we can notice differences, but also commonalities. First, as the system variant D is using graphical task B rather than A to process datum $D1$, a TB automaton is an alternative to TA automaton. The size of $D2$ also varies between the two system variants. Secondly, as variant D contains a GPU processor, a processor automaton GPU is present. RAM capacity also varies between the two system variants. Thirdly, contrary to variant B, the task TD is mapped on GPU rather than DCU. Consequently, at automaton level, TD automaton will send the processing command to the GPU automaton rather than DCU one. Moreover, while outputs of DCU processing are directly sent to the screen, using GPU requires to store outputs on RAM. Buffers for outputs are allocated at task automata level ($RAMalloc!out$ at TB and TD).

Even at automata level, we clearly observe the several variability concerns of our already introduced case study. The presence or absence of automata (*e.g.*, task, memory, processor, and datum), a value difference of a state variable (*e.g.*, input data size and memory storage capacity), a difference of channel communications (*e.g.*, automaton TD sending command to automaton GPU rather than DCU). Unfortunately, automata-based or simulation approaches are not meant to manage any variability in application and platform specifications or mapping. Consequently, that require to assess behavior of each system variant iteratively.

6.1.2 Featured Transition Systems

The featured transition system formalism has been developed from combined research in automata-based model checking and software product lines to analyze the behaviors of a set of related system variants called a behavioral product line. As discussed in Section 6.1, this formalism relies on states and transitions constrained by features to encode and analyzes through a variability-aware model checking algorithm both variant and state spaces of the entire product line efficiently. Capturing behavior and structure of every design alternative as, respectively, a variant space (using a featured model) and a variability-aware state space (using a featured automaton). Consequently, this formalism can capture both structural and behavioral aspects of every variability concern mentioned above in an efficient manner. Thus, the network of featured automata (see Fig. 6.3) combined with the feature model (see Fig. 6.4) can be used to assess behavior of (see Sec. 6.2) not a single system but every system variants of the case study.

Definition 5 *A featured automaton is a tuple* $FA = (S, s_0, S_f, Act, Trans, Ap, L, d, \gamma)$ *such that:*

- *where* $(S, s_0, S_f, Act, Trans, Ap, L) \in A$ *is an automaton,*
- $d = (F, \Phi_f)$ *is a feature model that captures the set of valid products.* F *is the set of features,* Φ_f *represents relations and constraints on features.* $DE \subseteq F \times F$ *are the set of parent/children relations between features and (*e.g., $f \implies f'$ *or* $f \wedge \neg f'$*) are cross cutting constraints between features. The feature model semantics* $[\![d]\!]_{FD} \subseteq \mathcal{P}(\mathcal{P}(F))$ *is the set of feature combinations representing valid product structures such as* $\forall p \in [\![d]\!]_{FD}, p \models \Phi_f$*,*

- $\lambda : trans \rightarrow \mathbb{B}(F)$ *is a total function that labels transitions with feature expressions and where* $[\![\gamma(trans)]\!]_{FD} \subseteq [\![d]\!]_{FD}$ *is the set of products that satisfy the expression.*

Feature expressions work similarly to presence conditions [Czarnecki and Antkiewicz, 2005]. The automaton of a particular system design is obtained by removing all transitions whose feature expression is not satisfied. This operation is called projection. *The projection of an* $fa \in FA$ *to a product*[4] $p \in [\![d]\!]_{FD}$*, noted* $fa_{|p}$*, is the automaton* $a = (S, s_0, S_f, Act, Trans', Ap, L)$ *with:*

$$Trans' = \{t \in Trans | p \in [\![\gamma(t)]\!]_{FD}\}$$

The automaton of system variants of, respectively, design B and D, noted $B_{\Psi A}$ and $D_{\Psi A}$ can be obtained by the projections:

$$B_{\Psi} = D2_SIZE_512 \& TA \& \neg TB \& D1_ON_ROM \& D2_ON_RAM \& TA_ON_DCU \& P2_ON_DCUr0 \& TD_ON_DCU \& P4_ON_DCUr1 \& RAM \& RAM_SIZE_512 \& \neg GPU$$

$$D_{\Psi} = D1_ON_ROM \& D2_ON_RAM \& D2_SIZE_1024 \& TB \& \neg TA \& P2_ON_RAM \& TD_ON_GPU \& P4_ON_RAM \& RAM \& RAM_SIZE_2048 \& GPU$$

$$UC_{|B_{\Psi}} \equiv B_{\Psi A}$$
$$UC_{|D_{\Psi}} \equiv D_{\Psi A}$$

The Role of the Feature Model

Rather than capturing every system variants in an inefficient enumeration-based manner. We focus on capturing system variations in order to derive any system variant. Design variations at application, platform or mapping concerns are captured by feature-oriented variation points. A design decision is then captured by making a feature selection. A functionally valid system variant is thus a product (or valid feature configurations) of this product line. The threefold variability dimensions are captured by a tri-dimensional (*i.e.*, $ApplicationVariability$, $MappingVariability$, $PlatformVariability$) feature model. Each sub-diagram is capturing a variability space (*i.e.*, application space, mapping space, platform space) from which one can derive every variant.

The $ApplicationVariability$ feature model captures every variability concerns of the application. Alternative size of the datum $D2$ is captured by the $D2_size$ XOR features group. To capture output and input variability of, respectively, $P1$ and $P2$, (*i.e.*, $P1$ can feed task A or B and $P2$ is feeded by Task A or B), we add $P1_To$ and $P2_From$ XOR features group. Task A and B are made optional features as they are exclusive. On the platform side, $PlatformVariability$ feature model captures the platform variability straightforwardly. Finally, every possible mapping choice of every application element over platform resources (*i.e.*, Datum mapped onto memories, task over processors) is captured as XOR features group.

[4]The term product has been introduced in the original definition, in our case, a product is a system design.

Application and platform consistency constraints are, respectively, added into $ApplicationVariability$ and $PlatformVariability$ feature models. The first one ensures that only valid flow variants are captured by, and thus can be derived from, the $ApplicationVariability$ feature model, while the second constraints the platform space $PlatformVariability$ to the manufacturable platform product line. Moreover, we add the three classes of mapping consistency constraints (*i.e.*, mapping to mapping, application to mapping and mapping to platform) into features constraints through the $MappingVariability$ feature model. Taking into account by the SAT-solver (see Sec. 6.2), these various constraints will guarantee that every design will exhibit a consistent system variant that satisfies the **FC-S** requirement *i.e.*, a functionally and valid (compatible) triplet of application, mapping and platform variants. Let the $SystemDesignVariability$ feature model be the tri-dimensional feature model composition thus capturing only valid system variants $p \in [\![d]\!]_{FD}$. The feature selections that do not represent a system variant that satisfies **FC-S** requirements are then denoted by $\mathcal{P}(F) \setminus [\![d]\!]_{FD}$.

The Role of the Featured Automaton

While the featured model captures the structural aspect of the system design space, the network of featured automata captures every possible execution of every system variants. To do so, the behavior of each variable application elements such as a task or a datum are respectively captured in a featured automaton interacting with other ones. Similarly, the behavior of each variable platform resources such as storage or processor is respectively captured in an interacting featured automaton. Hence, rather than capturing executions of a single storage configuration, we capture every execution of every storage configuration in a *featured state space*. To rely system variant with their state space, features transitions are constraining the product allowed to take that transition by requiring features.

By setting the $free$ variable accordingly, the RAM_SIZE_512 feature selection will allow a memory behavior that can only store 512KB of data (more will lead to error state) while the RAM_SIZE_2048 could exhibit a behavior where 2048KB of data have been successfully store (without any problem). If the feature TA is selected rather than TB, the datum completion signal of $D1$ will be received by TA. In addition, if the feature TA_On_GPUa is selected, the task TA will program and executed on the GPU rather than DCU. However, if the GPU feature is not selected, the featured state space will not include GPU behaviors.

Taking multiple featured transitions require feature selections representing particular design decisions. Because not every feature selections, noted $\mathcal{P}(F)$ represents a valid system variant. The feature model is of fundamental importance as it capture only the functionally valid system variants (that satisfy **FC-S** requirement) $[\![d]\!]_{FD} \subseteq \mathcal{P}(F)$. Using the feature model, only featured executions that belong to at least one valid system variant (product) will be explored. For example, executions where the path $P2$ is mapped on a DCU internal FIFO buffers (registers) whereas task TA is mapped on GPU (rather than DCU) do not belong to a valid variant because $]\!]P2_On_DCUr0 \& TA_On_GPU[\![_{FD} \cap [\![d]\!]_{FD} \subseteq = \emptyset$. However, a variant may only exhibit executions that do not correctly render the HMI. On the other hand, a system variant (that satisfies **FC-S** requirement) that exhibits an execution that renders the HMI without any error (satisfying thus **FC-B**) result in a functionally valid system design.

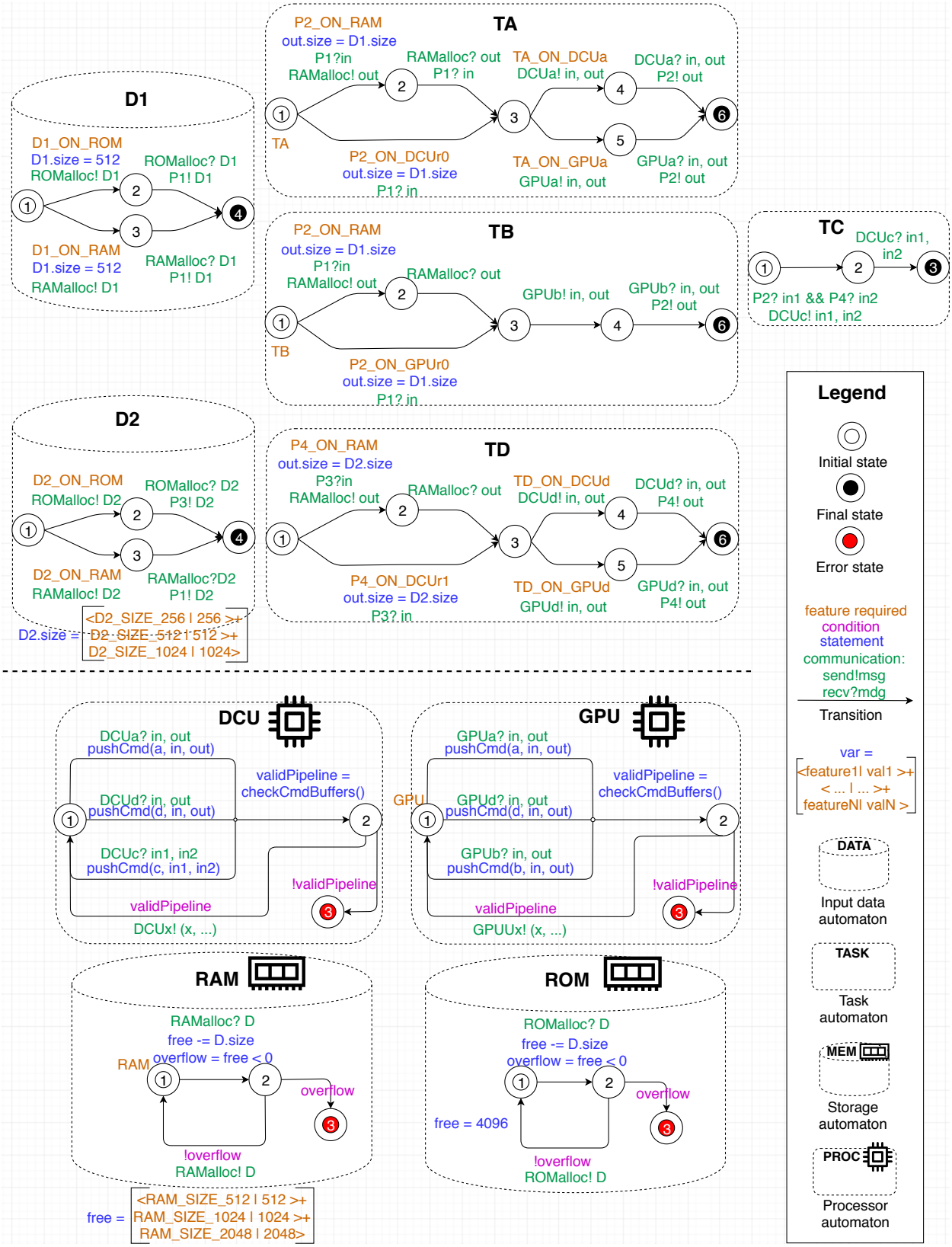
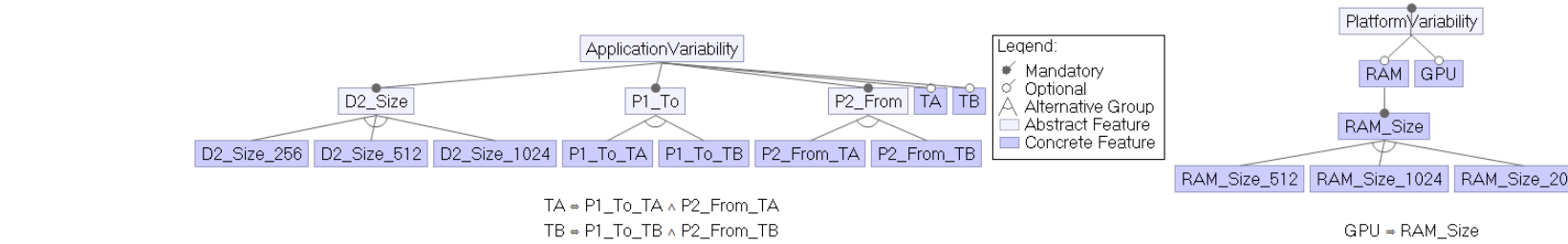
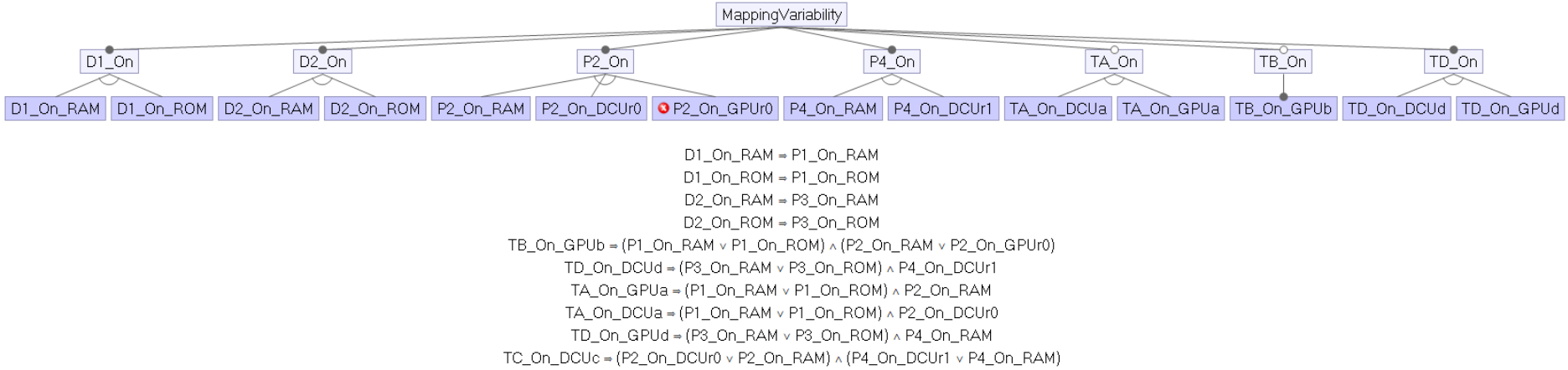


Figure 6.3: System featured automaton



(a) Application Variability

(b) Platform Variability



(c) Mapping Variability

Figure 6.4: System Design Space Variability

Deriving Automatically the Network of Featured Automata

Through a feature model, we show that we can capture the structural aspect of our design space (*i.e.*, the system variant space). Using a network of featured automata, we are able to capture the behavioral aspect of our design space (*i.e.*, the systems execution spaces). Making the removal of invalid system variants straightforward requires constraining the feature model not to consider those variants.

However, it is too tedious, error prone, and time consuming for engineers to use directly these formal models (network of featured automata (see Fig. 6.3) and the feature model (see Fig. 6.4) from specifications and mapping models[5]. Similarly to traditional frameworks that usually transform iteratively, each system variant behavior into analytical, simulation or computational model for behavioral verification purpose. Our framework also proposes functions to automatically derive formal models (featured automaton and feature model) from our previously defined variable dataflow *VDG*, variable resource graph *VRG*, and variable mapping models *VM*. But rather than transforming and verifying each system variant behavior iteratively, we transform every variant behavior into one behavioral formal model (*i.e.*, Featured Automaton).

Definition 6 *The function that transforms specifications* $app \in VDG, plt \in VRG, m \in VM$ *models into a network of featured automata* $nfa \subseteq FA^n$ *is defined by* $gen_{FA} : VDG \times VRG \times VM$ *where:*

1) *The function transforms each datum* d *in an featured automaton* fa*. Datum variable sizes are transformed into a COR feature group. While the first transitions of the Datum automaton set the size and allocate the datum on a storage, the seconds wait for storage acknowledgment and send the datum over the output paths.*

$$\begin{aligned}
&\forall d \in D, \exists fa \in nfa, |\chi_n(d)(\Psi_{size})| > 1 \implies \exists \mathcal{F}_{xor}(d, \chi_n(d)(\Psi_{size})) \subseteq DE \wedge \\
&\quad d \in N_{var} \implies \exists \mathcal{F}(d) \in N_{opt}, \exists (d, M) \in Dm, \forall m \in M, \forall size \in \chi_n(d)(\Psi_{size}), \\
&\quad \exists \{ s_0 \xrightarrow{\mathcal{C}(m)!?\mathcal{D}(d)} s, s \xrightarrow{\bigcup_{o \in O(d)} \mathcal{C}(o)!\mathcal{D}(d)} s_f \} \subseteq Trans \wedge L(s_f) = d.end \wedge \\
&\quad \gamma(s_0 \to s) = \mathcal{F}(d, m) \wedge \mathcal{F}(size) \wedge \begin{cases} d \in N_{var} & \mathcal{F}(d) \\ d \notin N_{var} & \top \end{cases}
\end{aligned}$$

2) *Each task* t *is transformed in featured automaton* fa*, the first transitions are gathering the inputs and allocating the outputs. The second transitions program the processors in order to accomplish the task. After receiving processor acknowledgment, the resulting outputs are sent over the output paths. Optionality of a datum is captured by an optional feature.*

$$\begin{aligned}
&\forall t \in T, \exists fa \in nfa, t \in N_{var} \implies \exists \mathcal{F}(t) \in N_{opt}, \exists (t, P) \in Tm, \forall p \in P, \\
&\quad \exists s_0 \xrightarrow{\bigcup_{i \in I(t)} \mathcal{C}(i)?\mathcal{D}(i) \bigcup_{o \in O(t), (o,M) \in Pm, m \in M} \mathcal{C}(m)!?\mathcal{D}(o)} s \in Trans \wedge \\
&\quad \gamma(s_0 \to s) = \bigwedge_{o \in O(t), (o,M) \in Pm, m \in M} \mathcal{F}(o, m) \wedge \begin{cases} t \in N_{var} & \mathcal{F}(t) \\ t \notin N_{var} & \top \end{cases} \wedge \\
&\quad \exists \{ s \xrightarrow{\mathcal{C}(p)!?\{\bigcup_{i \in I(t)} \mathcal{D}(i) \bigcup_{o \in O(t)} \mathcal{D}(o)\}} s', \wedge s' \xrightarrow{\bigcup_{o \in O(t)} \mathcal{C}(o)!\mathcal{D}(o)} s_f \} \subseteq Trans \\
&\quad \wedge L(s_f) = t.end \wedge \gamma(s \to s') = \mathcal{F}(t, p)
\end{aligned}$$

[5] An other major difficulty is to infer the valid mapping of the variable application over the configurable platform in order to model the feature model.

While the case study would require few days, more complex use cases would require several weeks.

3) *Feature constraint over data-path and task variability are added to capture only valid flows variants. Each variable data-path is captured in a XOR feature group of the $ApplicationVariability$ feature model (see. Fig. 6.4) In addition to, node/data-path consistency constraints are formalized into the feature model as features constraints.*

$$\forall p \in Path, |O(p)| > 1 \implies \exists \mathcal{F}_{xor}(p, O(p)) \subseteq DE \wedge \forall n \in O(p), \exists (\mathcal{F}(p,n) \iff \mathcal{F}(n)) \in \Phi$$
$$\forall p \in Path, |I(p)| > 1 \implies \exists \mathcal{F}_{xor}(I(p), p) \subseteq DE \wedge \forall n \in I(p), \exists (\mathcal{F}(n,p) \iff \mathcal{F}(n)) \in \Phi$$

4) *it creates for each storage s an featured automaton fa that represents basic memory behavior. The first transitions set the storage capacity. Alternative sizes are captured in a xor feature group of the feature model d. cons and cap are respectively the consumed size and the maximal capacity of the storage. Through channels, one can then allocate memory, and if there is not enough memory, an error is raised.*

$$\forall s \in S, \exists fa \in nfa, |\chi_r(s)(\Psi_{cap})| > 1 \implies \exists \mathcal{F}_{xor}(s, \chi_r(s)(\Psi_{cap})) \subseteq DE \wedge$$
$$s \in \Theta_{opt} \implies \exists \mathcal{F}(s) \in N_{opt}, \forall cap \in \chi_r(s)(\Psi_{cap}),$$
$$\exists \{ s_0 \xrightarrow{\mathcal{C}(s)?\mathcal{D}(d)} s, s \xrightarrow{free \geq 0, \mathcal{C}(s)!\mathcal{D}(d)} s_0, \exists s \xrightarrow{free < 0} s_{err} \} \subseteq Trans \wedge$$
$$L(s_{err}) = s.error \wedge \gamma(s_0 \to s) = \mathcal{F}(cap) \wedge \begin{cases} s \in \Theta_{opt} & \mathcal{F}(s) \\ s \notin \Theta_{opt} & \top \end{cases}$$

5) *It creates for each processor p an featured automaton fa that models basic graphic processor pipeline behavior. When a processor function is executed, the input and output are checked against an internal command buffer to verify that the hardware pipeline, and if it is misused, an error is raised.*

$$\forall p \in P, \exists fa \in nfa, p \in \Theta_{opt} \implies \exists \mathcal{F}(p) \in N_{opt}, \forall f \in F,$$
$$\exists \{ s_0 \xrightarrow{\mathcal{C}(p,f)?\mathcal{D}(d_0,\ldots,d_n)} s, s \xrightarrow{CmdBuf \models \top, \mathcal{C}(p,f)!(d_0,\ldots,d_n)} s_0, s \xrightarrow{CmdBuf \not\models \top} s_{err} \} \subseteq Trans \wedge$$
$$L(s_{err}) = p.error \wedge \gamma(s_0 \to s) = \begin{cases} p \in \Theta_{opt} & \mathcal{F}(p) \\ p \notin \Theta_{opt} & \top \end{cases}$$

6) *It creates feature constraints into the $PlatformVariability$ feature model to capture dependency and incompatibility constraints on resource components of the platform.*

$$\forall r \in R, \Theta_{req}(r) \neq \emptyset, (\mathcal{F}(r) \implies \bigwedge_{r' \in \Theta_{req}(r)} \mathcal{F}(r')) \in \Phi$$
$$\forall r \in R, \Theta_{exc}(r) \neq \emptyset, (\mathcal{F}(r) \implies \neg(\bigvee_{r' \in \Theta_{exc}(r)} r')) \in \Phi$$

7) *It transforms task, data, path mappings and their associated consistency constraints into features and features constraints the $MappingVariability$ feature model. The three classes of mapping constraints are transformed into feature constraints. Mappings to mappings constraints guarantee that task mapping will imply suitable input and output path mapping.*

$$\forall (n, R) \in Tm \cup Dm, \forall r \in R,$$
$$\exists (\mathcal{F}(n,r) \implies \bigwedge_{i \in I(n)} (\bigvee_{((n,r,i),M_i) \in Mc, m_i \in M_i} \mathcal{F}(i, m_i)) \bigwedge_{o \in O(n)} (\bigvee_{((n,r,o),M_o) \in Mp, m_o \in M_o} \mathcal{F}(o, m_o))) \in \Phi$$

*Application to mapping constraints ensures that each selected task will be mapped and vice versa (for alternative tasks). A mapping of optional task feature (*i.e.*, that represents alternative task) will imply to select both task and its mapping feature.*

$$\forall e \in N_{var}, \exists (e, R) \in Tm \cup Pm \implies (\bigvee_{r \in R} \mathcal{F}(e,r) \implies \mathcal{F}(e)) \in \Phi.$$

Mappings to platform constraints ensure that if a mapping requires an optional platform resources, this resource will be selected.

$$\forall r \in \Theta_{opt}, \exists (e, R) \in Tm \cup Pm \implies (\mathcal{F}(e, r) \implies \mathcal{F}(r)) \in \Phi$$

6.2 Variability-Aware Validation Process

6.2.1 Background

To assess the behavior of a system, a model-checking algorithm will assess its possible execution traces against a temporal formula. While an execution trace (or path) is a list of states and transitions that track a possible execution of the system, the temporal logic formula is a set of temporal properties that system behavior should respect. *End state reachability* property has been extensively used to find a system execution path that meets the functional requirements such as completing every tasks or protocol without errors **FC-B**.

Definition 7 *An execution trace π of automaton $a = (S, s_0, S_f, Act, Trans, Ap, L)$ is a list $(s_1, \alpha_1, s_2, \alpha_2, s_3, ..., s_n, \alpha_n, s_{n+1})$ also noted $s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} s_3 ... s_n \xrightarrow{\alpha_n} s_{n+1}$ and $\{(s_1, \alpha_1, s_2)\}, ..., (s_n, \alpha_n, s_{n+1})\} \in Trans$ and where $\pi \in paths(a, s_1)$. We define the semantics[6] of automaton such as:*

$$[\![a]\!]_A = paths(a, s_0)$$

A Computation Tree Logic (CTL) formula ϕ is an expression:

$$\phi : 1 | a \in Ap | \phi_1 \wedge \phi_2 | \neg\phi | E\Diamond\phi | E\Box\phi | A\Diamond\phi | A\Box\phi$$

CTL formulas are interpreted over an automaton and a state (a, s) such as:

$$\begin{aligned}
&a, s \models 1 \\
&a, s \models a \in Ap && \iff a \in L(s) \\
&a, s \models \neg\phi && \iff a, s \not\models \phi \\
&a, s \models \phi_1 \wedge \phi_2 && \iff (a, s \models \phi_1) \wedge (a, s \models \phi_2) \\
&a, s \models E\Diamond\phi && \iff \exists \pi \in paths(a, s), \exists i \geq 0, a, head(\pi_i) \models \phi
\end{aligned}$$

Where $head(\pi)$ denotes the first state of π and π_i the tail of the path starting at the ith state such as $\pi_0 = \pi$. $a, s_0 \models \phi$ notation will be reduced to $a \models \phi$.

Regarding the system variant B in Figure 6.1, the CTL end state reachability formula to assess that, at least, one execution is capable to render the entire content that the customer want to be displayed through HMI-rendering without any problems (**FC-B**). could be the following:

$$\begin{aligned}
\phi_B = E\Diamond(&D1.end \wedge D2.end \wedge TA.end \wedge TD.end \wedge \\
&TC.end \wedge \neg DCU.error \wedge \neg RAM.error \wedge \neg ROM.error)
\end{aligned}$$

[6]Semantics of automaton can also be defined by logic properties based equivalence [Cordy, 2014] on paths $[\![a]\!]_A = \{L(s_0), ..., L(s_n) | \exists s_0 \rightarrow ... \rightarrow s_n \in paths(a, s_0)\}$. But as our traces will be also semantically determined by quantitative properties, the traces based semantics seems more adapted [Classen, 2011]

Similarly, regarding the system variant D in Figure 6.2, the CTL formula could be:

$$\phi_D = E\Diamond(D1.end \wedge D2.end \wedge TB.end \wedge TD.end \wedge TC.end \wedge \\ \neg DCU.error \wedge \neg GPU.error \wedge \neg RAM.error \wedge \neg ROM.error)$$

These formulae can be expressed in natural language such as "I assume that there exists a path where every application element has terminated and where no hardware resource has raised an error." In other words, it represents a design that satisfies the CTL formula and thus meets the behavioral functional requirements (**FC-B**. Therefore, if implemented on an automotive system, the design will render the HMI correctly and without errors.

As explained previously, given a *i)* state-based behavior of a system and *ii)* a temporal formula that the system should respect. The model checking algorithm automatically explores the system behaviour's state space to find events over executions that respect or violate required temporal properties. As observed in our case study, embedded systems exhibit a concurrent behavior and usually consist of several interacting automata.

Basically, to program and execute the HMI-rendering, application automata send appropriate commands to hardware resources automata. To execute these commands, hardware resources automata also interact with each other. While each application element or platform resource automata captures the behavior of a system part, the parallel composition of such featured automata results in a network of featured automata capturing the whole system's behaviour.

Definition 8 *Given two automata a_1 and a_2 their parallel composition, synchronized over the set of shared communication actions $Act_1 \cap Act2$, written $a_1||a_2$, is the automaton $(S_1 \times S_2, Act_1 \cup Act_2, Trans, I_1 \times I_2, AP_1 \cup AP_2, L)$, where*

- $L(\{s_1, s_2\}) = L(s_1) \cup L(s_2)$
- *$Trans$ is the smallest relation satisfying*
  - *for $\alpha \in Act_1 \cap Act_2$ (interleaving):* $\frac{s_1 \xrightarrow{\alpha} s_1'}{(s_1,s_2) \xrightarrow{\alpha} (s_1',s_2)}, \frac{s_2 \xrightarrow{\alpha} s_2'}{(s_1,s_2) \xrightarrow{\alpha} (s_1,s_2')}$
  - *for $\alpha \in Act_1 \cap Act_2$ (synchronization):* $\frac{s_1 \xrightarrow{\alpha!} s_1' \wedge s_2 \xrightarrow{\alpha?} s_2'}{(s_1,s_2) \xrightarrow{\alpha} (s_1',s_2')}$

The state-space after parallel composition is thus the product of the original state spaces (another reason for state explosion (see Sec. 3.1.4), so that a process can be in any state, independently from the other. The parallel composition of two automata results in an interleaving of their executions. This means that only one process at a time can fire a transition, after which his part of the global system state changes. However, processes may synchronize, *e.g.*, when sending/receiving a command or message. Parallel composition takes this into account by forcing transitions with the same action to execute synchronously. In this case, both processes fire their transition at the same time.

Given this definition, system variant B and D illustrated, respectively, in Fig. 6.1 and 6.2, can be formalized as the parallel composition of their application elements and platform resources automata:

$$B_A = D1_{ON_ROM}||D2_{512_ON_RAM}||TA_{DCU}||TD_{DCU}||TC||ROM||RAM_{512}||DCU$$

$$D_A = D1_{ON_ROM}||D2_{1024_ON_RAM}||TB||TD_{GPU}||TC||ROM||RAM_{2048}||DCU||GPU$$

where $D1_{ON_ROM}$ is the automaton capturing the behavior of the $D1$ data application element where mapped on ROM memory, TD_{DCU} and TD_{GPU} are the automata capturing the behavior TD task where mapped on, respectively, DCU or GPU processors. Similarly, RAM_{512} capture the automaton of a 512KB RAM memory.

As an example, Figure 6.5 illustrates two possible executions of the system variant B (see Fig. 6.1) in an sequence diagram form. Each automaton capturing the behavior of an application element or platform resource of the system variant has its own *local* execution trace. Interactions between automata (*e.g.*, input/output completion signals, hardware command) are indicated by directional messages between automata. At the initial state of the system (state 1 of B_A), all process automata are in their own local initial state (state 1). Next, in the execution of the Figure 6.5 (a), the automaton of the data $D1$ (noted $D1_A$)sends an allocation command to the ROM automaton (noted ROM_A) that receives the allocation command and reduces its free memory capacity according to the size of $D1$. Secondly, as their is no $ROMoverflow$, sends back an acknowledgment to $D1_A$. Finally, $D1_A$ will send a completion signal to TA_A that will allow the start of the task mechanism. Let us formalize this whole interaction scoped between $D1_A$, TA_A and ROM_A as:

$$\pi = (s_1, s_1, ..., s_1) \rightarrow (s_2, s_1, ..., s_2) \rightarrow (s_3, s_2, ..., s_1), \pi \in [\![D1_{ON_ROM}||TA||...||ROM]\!]_A$$

which is an execution trace (or path) of the parallel composition of the network of automata

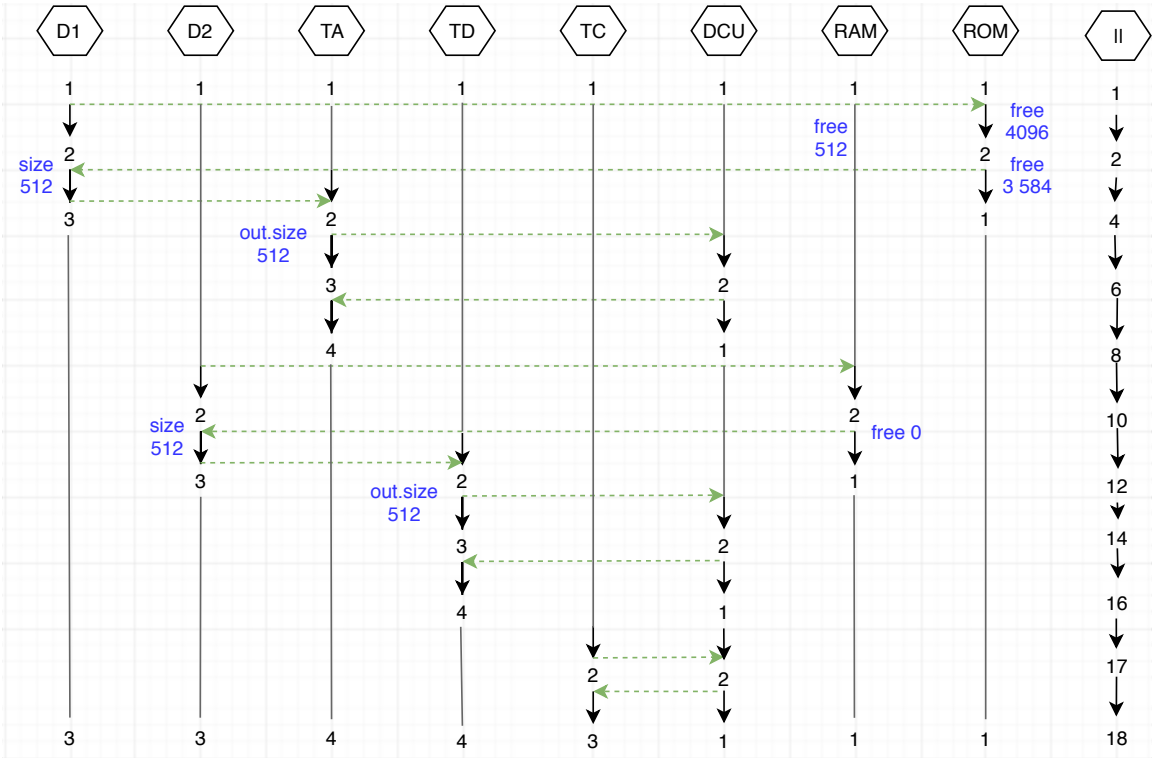
$$D1_{ON_ROM}||TA||...||ROM$$

The execution continues and TA_A will send a command to DCU_A. After validating the command, the DCU_A will acknowledge to TA_A the completion of the command. Similarly, $D2_A$ will send an allocation command to the RAM_A then, TD_A will also send a processing command to the DCU. TC_A will finally send the command that will flush the command buffer of the DCU_A in a way that respect (does not violate functional constraints and limitations) its pipeline. As all application elements have been processed using hardware resources correctly, the execution satisfies the end state reachability property. The HMI-rendering is then completed without any error. This design thus satisfies the **FC** requirements and can be considered functionally valid.

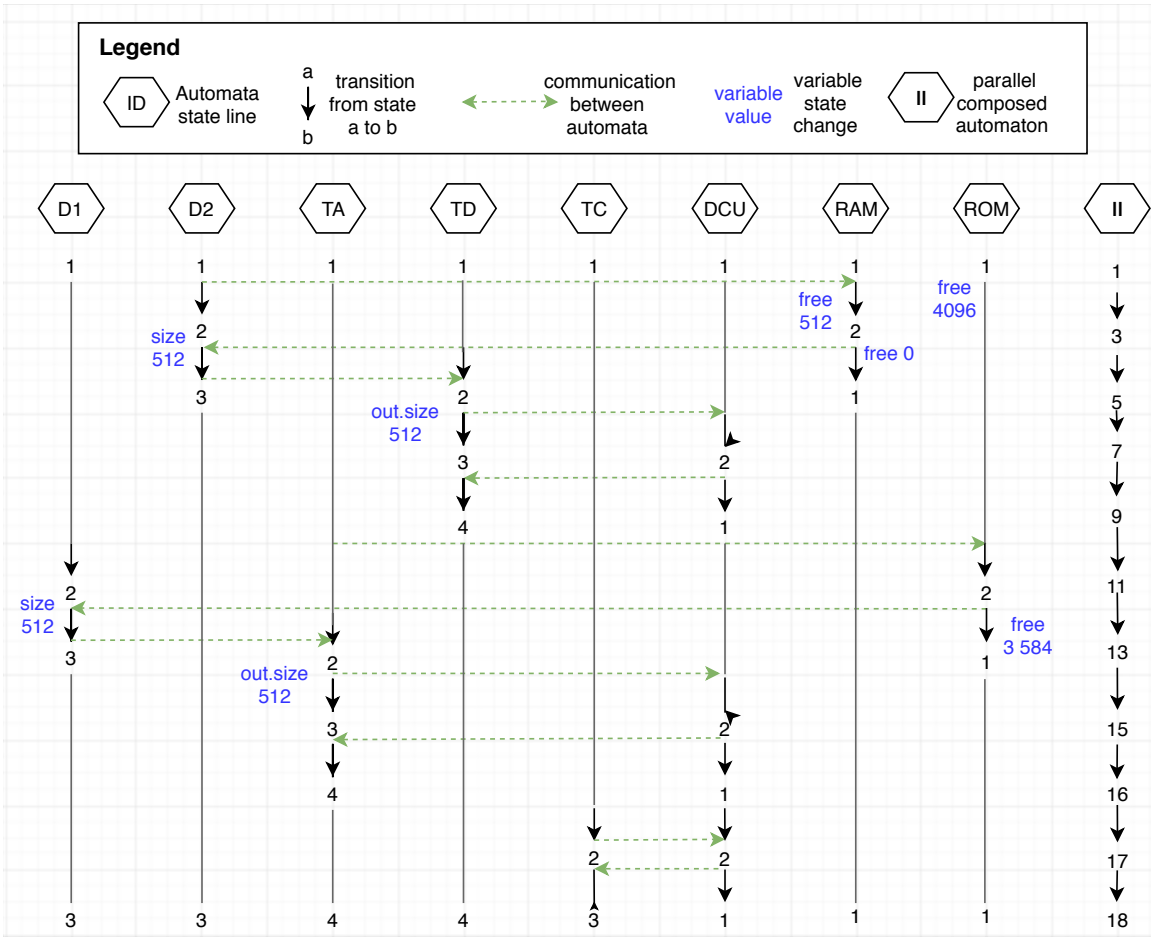
6.2.2 Model Checking Lots of System Designs

Assessing every possible execution of every system variant seems inefficient, especially when the number of system variants is huge. Yet, system design frameworks still assess their design space in iterative or enumeration-based manners (see Sec. 3.1.4). Reusing variability-aware model checking algorithms [Classen et al., 2010b], we propose to efficiently assess structural and behavioral functional feasibility of the system design space represented by a behavioral product line. The network of featured automata representing the structure and behavior of the possible design alternatives is checked against end-state reachability temporal property. Products (*i.e.*, system variants) that do not satisfy the property due to their behavior are then easily identified. As a result, the validation solves and extracts all valid variants in a single run.

This algorithm consists of verifying all execution paths of all products of the product line. There is a fundamental difference between family (or *variability-aware*) method and iterative (or *product-and-product*) techniques. Instead of exploring all



(a) An execution trace producible by system variant B



(b) An other execution trace producible by system variant B

Figure 6.5: Execution Traces of System Variant of Design B

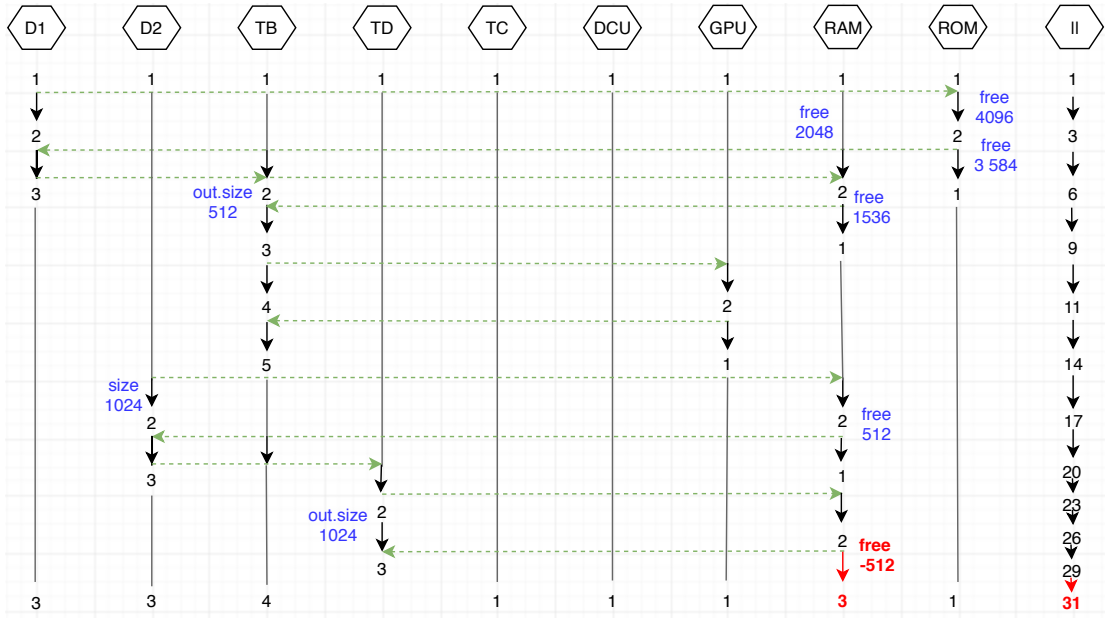


Figure 6.6: An execution trace producible by the System Variant of Design D

executions of each system variant, the model-checker explores an execution once for all variants able to produce this execution by exploiting commonalities between different products. Theoretically, the more the system variants share common behavior, the more efficient the variability aware model is checking in comparison to iterative model checking on individual systems [Classen et al., 2011c].

Automata formalism is a highly-tooled standard model of computation to model and verify concurrent systems. Such systems are generally composed of multiple concurrent processes: software or hardware , *etc.* Similarly to network of automata, a network of featured ones captures the state space of an entire product line composed by concurrent and communicating variable automata (or process). End state reachability property that captures functional requirements' satisfaction in a temporal logic form can be applied to assess the network of featured automata. However, as the end states can differ from a product to another, the temporal property is enriched with feature expressions. Similarly to featured automaton, featured Computation Tree Logic (*fCTL*) is automatically refined according to the features selection.

Definition 9 *An execution trace* π *of featured automaton* $fa = (S, s_0, s_f, Act, Trans, Ap, d, \gamma)$ *is a execution* $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 ... s_n \xrightarrow{\alpha_n} s_{n+1}$. *A feature expression* $\psi = \bigwedge_{i=0}^{n} \gamma(s_i \xrightarrow{\alpha_i} s_{i+1})$ *(simply reduced to* $\gamma(\pi)$*) encodes the set of products* $[\![\psi]\!]_{FD}$ *that can produce* π. *On the other hand,* $\{\pi_0, ...\pi_m\} = [\![fa_{|p}]\!] \cap [\![fa_{|p'}]\!]$ *denotes the set, possibly empty, of shared executions between two products* p, p'. *Finally, the semantics of featured automaton is defined as:*

$$[\![fa]\!]_{FA} = \bigcup_{p \in [\![d]\!]_{FD}} [\![fa_{|p}]\!]_A$$

An fCTL property ϕ *is an expression* $\phi := [\chi]\phi'$ *where* ϕ *is an CTL property and* $\chi \in \mathbb{B}$ *a feature expression quantifier. Then,*

$$fa \models \phi \iff \forall p \in [\![\chi]\!]_{FD} \cap [\![d]\!]_{FD}, fa_{|p} \models \phi'$$

That is, each product $p \in [\![\chi]\!]$ *included in the quantifier exhibits a behavior that satisfies the CTL property* $fa_{|p} \models \phi'$.

Definition 10 *Given two featured automata fa_1 and fa_2 their parallel composition, synchronized over the set of shared communication actions $Act_1 \cap Act2$, written $fa_1||fa_2$, is the automata $(S_1 \times S_2, Act_1 \cup Act_2, Trans, I_1 \times I_2, AP_1 \cup AP_2, L, d, \gamma)$, where*

- $L(\{s_1, s_2\}) = L(s_1) \cup L(s_2)$
- *$Trans$ is the smallest relation satisfying*
  - *for $\alpha \in Act_1 \cap Act_2$ (interleaving):* $\frac{s_1 \xrightarrow{\alpha} s_1'}{(s_1,s_2) \xrightarrow{\alpha} (s_1',s_2)}, \frac{s_2 \xrightarrow{\alpha} s_2'}{(s_1,s_2) \xrightarrow{\alpha} (s_1,s_2')}$
    where, $\begin{cases} \lambda((s_1, s_2) \xrightarrow{\alpha} (s_1', s_2)) = \lambda_1(s_1 \xrightarrow{\alpha} s_1') \\ \gamma((s_1, s_2) \xrightarrow{\alpha} (s_1, s_2')) = \lambda_2(s_2 \xrightarrow{\alpha} s_2') \end{cases}$
  - *for $\alpha \in Act_1 \cap Act_2$ (synchronization):* $\frac{s_1 \xrightarrow{\alpha!} s_1' \wedge s_2 \xrightarrow{\alpha?} s_2'}{(s_1,s_2) \xrightarrow{\alpha} (s_1',s_2')}$
    where, $\lambda((s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')) = \lambda_1(s_1 \xrightarrow{\alpha!} s_1') \wedge \lambda_2(s_2 \xrightarrow{\alpha?} s_2')$
- *$d = [\![\mathbb{B}(d_1) \wedge \mathbb{B}(d_2)]\!]_{FD}$ also denoted $d_1 \oplus d_2$, is the inter-product line feature diagram composition*[7].

Then, given $p_1 \in [\![d_1]\!]_{FD}, p_2 \in [\![d_2]\!]_{FD}$, and, $p = p_1 \wedge p_2, p \in d_1 \oplus d_2$, the projected systems, $fa_{1_{|p_1}}||fa_{2_{|p_2}}$ and $(fa_1||fa_2)_{|p}$ are equivalent.

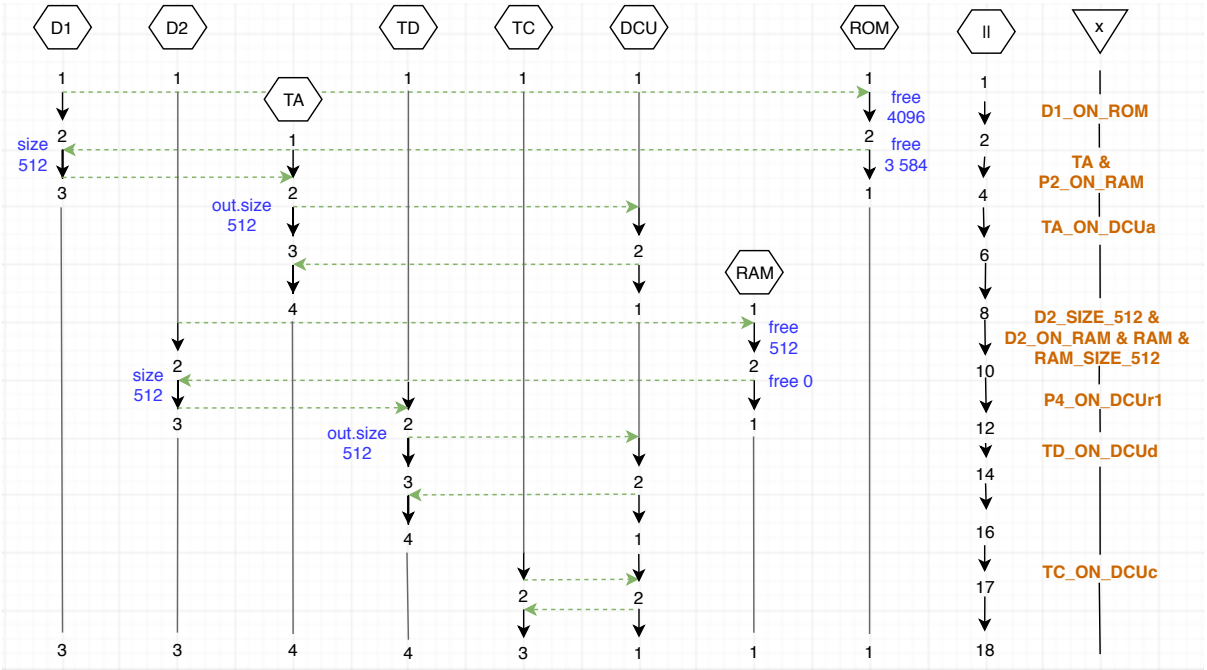


Figure 6.7: An FTS execution trace producible by system variant B

The featured automaton noted as:

$$FA_{uc} = D1_{FA}||D2_{FA}||TA_{FA}||TB_{FA}||TD_{FA}||TC_{FA}||GPU_{FA}||RAM_{FA}||ROM_{FA}$$

is the composition of the network of featured automata. Figure 6.7 illustrates a possible execution of the featured automaton that captures the system design space of our case

[7]This definition is suitable in our context as every configurable process have its own independent feature diagram.

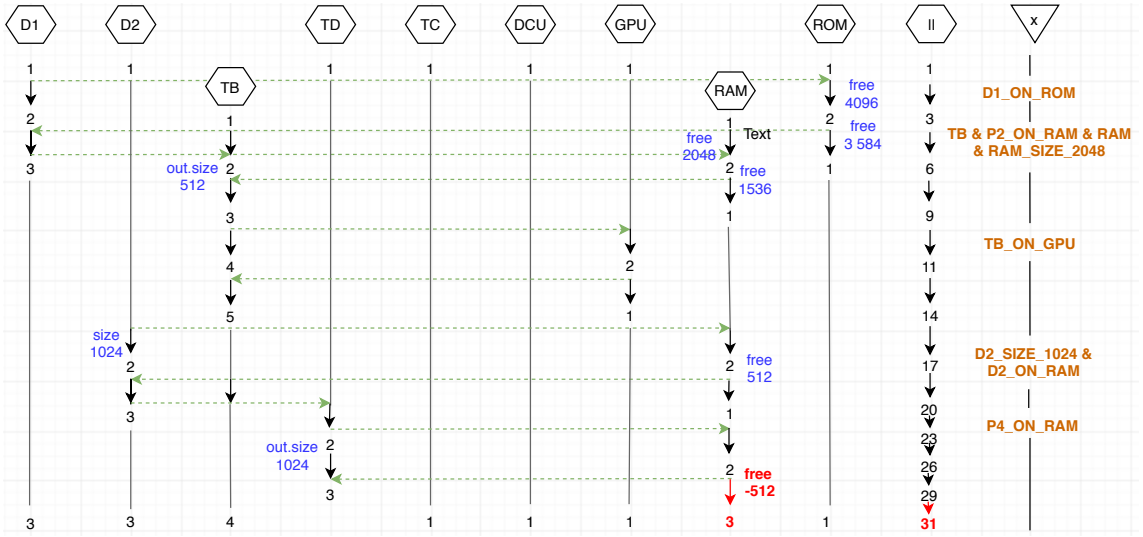


Figure 6.8: An FTS execution trace producible by system variant D

study (see fig. 6.3). We observe that this execution is indistinguishable from the first execution of the system variant B in Figure 6.5(a). Both executions are equivalent. The two figures are identical except that design decisions (represented by features) such as data size, memory capacity, application mappings over platform resources, *etc.* are selected through some state transitions. We can see that while $D1$ is allocating on ROM the related feature $D1_ON_ROM$ is selected, which represent the design decision to map $D1$ on ROM. More formally, as :

$$\pi = (s_1, ..., s_1) \rightarrow (s_2, ..., s_2), \pi \in [\![D1||...||ROM]\!]_{FA}, \gamma(\pi) = D1_ON_ROM^{8}$$

furthermore, in this execution, the $D1$ completion signal is received by TA (rather than TB) automaton. This transition require TA feature as:

$$\pi = (s_2, s_1..., s_2) \rightarrow (s_3, s_2, ..., s_1), \pi \in [\![D1||TA||...||ROM]\!]_{FA}, \gamma(\pi) = TA$$

.

The complete execution can be denoted as:

$$\pi = s_1 \rightarrow s_2, ..., s_{17} \rightarrow s_{18}, \gamma(\pi) \text{ and, } \psi = {}_{D1_ON_ROM\&TA\&...\&TC_ON_DCU_c}$$

The required design decisions ψ of ψ are leading to capture the state space of system variant B noted:

$$[\![UC_{|\psi}]\!]_A = [\![B]\!]_A$$

At automata level, the automata network of system variant B in Figure 6.1 would be equivalent to the ψ projected featured automata network capturing the case study. For example:

$$TD_{|\{TD_ON_DCU\}} = TD_{ON_ON_DCU}$$
$$D2_{|\{D2_SIZE_512\&D2_ON_RAM\}} = D2_{SIZE_512_ON_RAM}$$

Similarly, the Figure 6.8 illustrates, the invalid execution (previously explained see Fig. 6.6) of system variant D (see Fig. 6.1) obtained by D_ψ projection of the use case featured automaton.

[8]More precisely, $\lambda((s_1, ..., s_1) \rightarrow (s_2, ..., s_2)) = \gamma(s_1 \rightarrow s_2) \wedge ... \wedge \gamma(s_1 \rightarrow s_2)) = \top \wedge ... \wedge D1_ON_ROM$

The property that captures the functional requirement in a temporal logic form interpreted over the featured automaton can be denoted as:

$$\phi_{fa} = E\Diamond(D1.end \wedge D2.end \wedge [TA]TA.end \wedge [TB]TB.end \wedge TD.end \wedge TC.end \wedge \\ \neg DCU.error \wedge [GPU]\neg GPU.error \wedge [RAM]\neg RAM.error \wedge \neg ROM.error)$$

The end-state reachability is automatically refined according to tasks and resources presence. Only tasks that are present in the system variant have to complete. Similarly, only resources of the system variant have to behave without ending in an error state.



Figure 6.9: The shortest FTS invalid execution trace producible by a huge amount of system variants

Finally, Figure 6.9 illustrates how such formalism can efficiently assess a huge amount of system variants. This execution reaches an error state after allocating the two data $D1$ and $D2$. We can observe that the design decisions that lead to this erroneous execution are only affecting the mappings of $D1$ and $D2$ which are allocated on RAM, the size of $D2$ (*i.e.*, 1024KB) and the capacity of RAM (*i.e.*, 1024KB).

To be reachable or not to be

We use *featured* state reachability property to assess our design space represented by a behavioral product line. The verification of such property can be reduced to the computation of *traditional* reachability relations [Classen et al., 2010b]. Rather than cover all the details of every step of the model checking technique [9], we *i)* introduce the concept of reachability relations in featured automaton *ii)* present the algorithm that efficiently computes the reachability relations and *iii)* exemplify the algorithm with our case study.

Comparing single system and system product line model-checking, the major difference is the definition of *successor state*. Successor state is an essential function to determine the reachability relation in any state transition based system. In an automaton that captures a single system, state s' is a successor of state s if and only if there a transition s to s'. In a featured automaton that captures a system product line, each product of the featured automaton can have a different state space. To this extent, a transition can be enriched by a feature expression such as a transition only exists for a subset of the products. Consequently, the model checking algorithm has to keep track of the successor states according to the products in which they exist. We assume that a transition $s_i \xrightarrow{\alpha} s_{i+1}$ without a feature expression implies that s_{i+1} is a successor state of s_i in all products $[\![d]\!]_{FD}$ as no particular feature is required $\lambda(s_i \xrightarrow{\alpha} s_{i+1}) = \top$

[9] To the interested readers, we recommend this review [Cordy et al., 2019]

to take the transition. While the products $[\![\gamma(s_i \xrightarrow{\alpha} s_{i+1})]\!]_{FD} \subseteq [\![d]\!]_{FD}$ is the set of products on which the transition exists and can be taken during executions[10].

For a given pair of states (s, s'), the function $Post(s)(s')$ is the feature expression encoding the variants $[\![Post(s)(s')]\!]_{FD}$ that can reach s' from s using one from possibly many transitions that rely s' to s such as $\exists \alpha \in Act, s \xrightarrow{\alpha} s' \in Trans$. If there is only one transition from state s to its successor s', $Post(s)(s')$ is equal to $\lambda(s \xrightarrow{\alpha} s')$. We could define the reachability relation from the successor function. Given two states, say s_0 (the initial state) and s_n, the reachability relation that return the feature expression encoding the variants exhibiting, at least, one path $s_0 \rightarrow ... \rightarrow s_n$ from s_0 to s_n is given by $\bigwedge_{i=0}^{n-1} Post(s_i, s_{i+1})$ and where $\forall p \in [\![d]\!]_{FD} \wedge s_0 \rightarrow ... \rightarrow s_n \in [\![fa_{|p}]\!]_A$.

Definition 11 *The reachability relation function $R : S \rightarrow (S \rightarrow \mathcal{P}(\mathcal{P}(F))$ from states s_0 to state s_n in a featured automaton returns the feature selections required to reach s_n from s_0 and is defined as*

$$R(s_0)(s_n) = \bigvee_{s_0,...,s_n \in [\![fa]\!]_{FA}} \bigwedge_{i=0}^{n} Post(s_i, s_{i+1})$$
$$\forall p \in [\![R(s_0)(s_n)]\!]_{FD} \iff \exists s_0 \rightarrow ... \rightarrow s_n \in [\![fa_{|p}]\!]_A$$

where $\forall j, 0 \leq j < n, \exists \alpha \in Act, (s_j, \alpha, s_{j+1}) \in Trans$ and where The successor function in a featured automaton is defined as:

$$Post(s_i)(s_{i+1}) = \bigvee_{s_i \xrightarrow{\alpha} s_{i+1} \in Trans} \lambda(s_i \xrightarrow{\alpha} s_{i+1})$$

Let us illustrate this reachability relation using the featured automaton capturing the case study in Fig. 6.3. First, we only focus on the featured automaton formed by followed the parallel composition $(D1||TA||RAM||ROM||DCU)_{FA}$ automata. Other automata and their possible interactions are voluntarily ignored to avoid adding unnecessary complexity to the example. The reachability relation of state $s_n = (s_4, s_6, s_1, s_1, s_1)$ D1 data and task TA have reached their end state and RAM, ROM and DCU resources are at their idle (and initial), from $s_0 = (s_1, s_1, s_1, s_1, s_1)$ the initial state. There exists two paths in the featured automaton $\pi, \pi' \in [\![D1||TA||RAM||ROM||DCU]\!]_{FA}$ that connect s_n from s_0. Only differing by $D1$ allocation mapping behavior *i.e.*, In π the ROM is used whereas π') use the RAM. More formally:

$$\pi = (s_1, s_1, s_1, s_1, s_1) \rightarrow (s_2, -, -, s_2, -) \rightarrow (s_4, s_3, -, s_1, -) \rightarrow (-, s_4, -, -, s_2) \rightarrow (s_4, s_6, s_1, s_1, s_1)$$
$$\pi' = (s_1, s_1, s_1, s_1, s_1) \rightarrow (s_2, -, s_2, -, -) \rightarrow (s_4, s_3, s_1, -, -) \rightarrow (-, s_4, -, -, s_2) \rightarrow (s_4, s_6, s_1, s_1, s_1)$$

$$R(s_1, s_1, s_1, s_1)(s_4, s_6, s_1, s_1) = (\bigwedge_{i=0}^{|\pi|-1} Post(\pi_i)(\pi_{i+1})) \quad \bigvee (\bigwedge_{i=0}^{|\pi'|-1} Post(\pi'_i)(\pi'_{i+1}))$$
$$= \psi \quad \bigvee \psi'$$

$$\psi = \{D1_ON_ROM \bigwedge (TA\&P2_ON_DCUr0) \bigwedge TA_ON_DCUa \bigwedge \top\}$$
$$\psi' = \{(\{D2_ON_RAM\&RAM\&RAM_SIZE_512\} \vee \{D2_ON_RAM\&RAM\&RAM_SIZE_1024\} \vee$$
$$\{D2_ON_RAM\&RAM\&RAM_SIZE_2048\}) \bigwedge (TA\&P2_ON_DCUr0) \bigwedge TA_ON_DCUa \bigwedge \top\}$$

[10]Note that if $[\![\gamma(s_i \xrightarrow{\alpha} s_{i+1})]\!]_{FD} = \emptyset$ then $s_i \xrightarrow{\alpha} s_{i+1}$ does not exist in any product and can be considered as a *dead* transition.

According to the reachability relation results, the first path π can be produced by products that have $\psi = _{D1_ON_ROM \& TA \& P2_ON_DCUr0 \& TA_ON_DCUa}$ features selection as design decisions. There are 4 products (system variants) that share such structure $|[\![\psi]\!]_{FD}| = 4$. The first can be noted $p_1 = _{\psi \& \neg D1_ON_RAM \& \neg RAM}$ while the rest are differing from the first product by an unused RAM with a different capacity such as $\{p_2, p_3, p_4\} = _{p_1 \& (RAM_SIZE_512 \vee RAM_SIZE_1024 \vee RAM_SIZE_2048)}$. The 3 products that can produce π' (*i.e.*, $|[\![\psi']\!]_{FD}| = 3$) are similar to the one of π mainly differing by the mapping and allocation of $D1$ on RAM (with 3 different possible capacities) rather than ROM.

As said previously, the verification of the end state reachability property expressed in fCTL is reduced to compute the reachability relation of the required end state. Then, the fCTL property $\phi = _{E\Diamond(D1.end \wedge TA.end \wedge [RAM](\neg RAM.error) \wedge \neg ROM.error \wedge \neg DCU.error)}$ that determines the functional requirements satisfaction (*i.e.*, complete the HMI-rendering without any error) are satisfied by the products above as they reach a an end state. Consequently, every system variant p fulfills the functional requirement as they all exhibit a execution that reach the end state such as:

$$p \in [\![(D1||TA||RAM||ROM||DCU)_{FA}]\!]_{FD} \wedge p \in [\![R(s_0)(s_{end})]\!]_{FD}$$

Proposed algorithm

We reuse the algorithm that computes the reachability relation of a featured automaton [Classen et al., 2010b] efficiently. A simpler but inefficient method would be to derive every product automaton and iteratively compute their reachability relations. On the contrary, exploiting behavioral commonalities between products as long as they do not exhibit a behavioural divergence[11] will drastically improve the verification process's efficiency. To this ambition, the design decisions represented by features have to remain undetermined as long as they do not impact the execution. This optimization has been called late splitting [Apel et al., 2013] as we split a group of products in others by taking a particular design decision (features selection), if and only if it is requested by the transition to explore.

Algorithm 2, based on the formalization of [Cordy et al., 2019], compute the reachability function of the featured automaton from the initial state s_0 to any other state $s \in S$ using a depth-first search. The algorithm consists of a loop that iterates over a stack of couples (s, γ) where s is a state and $\gamma \in \mathbb{F}$ is an associated feature expression that notably encodes the set of variants $[\![\gamma]\!]_{FD} \subseteq [\![d]\!]_{FD}$. Initially, the stack contains only the element $(s_0, \mathbb{B}(d))$ in order to start the search from the initial s_0 while considering all variants $\subseteq [\![d]\!]_{FD}$ (the initial state is effectively reachable by every variant). At each iteration, the algorithm takes the top element (s, γ) of the stack, computes the successors of s and associates each successor with the featured expression γ' that encodes variants that only satisfy current required features γ plus required features to reach s' from s noted $\gamma' = \gamma \wedge Post(s)(s')$ (Lines 4-5). This results in a set couples $(s', \gamma') \in S \times (\mathcal{P}(\mathcal{P}(F)))$. For each pair, the algorithm first determines whether $[\![\gamma]\!]_{FD}$ contains at least one valid product; otherwise, it is not needed to pursue the search from s' as the next state cannot be reached by the variants encoded by the feature expression. This verification is achieved by checking the satisfiability of γ' (Line 6). If that is the case, the algorithm enters an inner loop (Lines 7-17).

Likely, the more the products share structural commonalities, the more they share behavioral commonalities. But behavioral commonalities between totally different

[11] a state that does not exist in all products.

Input: $fa = (S, s_0, s_f, trans, Ap, L, d, \gamma)$.
Output: $\bigcup_{s \in S} R(s_0)(s)$.

1 $R \leftarrow \bot$;
2 $Stack \leftarrow push((s_0, \mathbb{B}(d), []);$
3 **while** $Stack \neq []$ **do**
4 $\quad (s, \gamma) \leftarrow pop(Stack);$
5 $\quad succ \leftarrow \begin{cases} (s', \gamma') \mid & s' \in dom(Post(s)) \wedge \\ & \gamma' = \gamma \wedge Post(s)(s') \wedge \\ & \gamma' = \gamma' \neg R(s_0)(s') \wedge \\ & \gamma' \not\models \bot \end{cases}$
6 $\quad$ **foreach** $(s', \gamma') \in succ$ **do**
7 $\quad\quad R(s_0)(s') \leftarrow R(s_0)(s') \vee \gamma';$
8 $\quad\quad push((s', \gamma'), Stack);$
9 $\quad$ **end**
10 **end**
11 **return** R

Algorithm 2: Reachables(fa)

variants could also exist. Formally, $[\![\gamma]\!]_{FD}, [\![\gamma']\!]_{FD} \subseteq [\![d]\!]_{FD}$, encoding common variants such as $[\![\gamma]\!]_{FD} \cap [\![\gamma']\!]_{FD} \neq \emptyset$ will likely exhibit common behaviors $[\![fa_\gamma]\!]_{FA} \cap [\![fa_{\gamma'}]\!]_{FA} \neq \emptyset$ but it could be also true for disjoint variants $[\![\gamma]\!]_{FD} \cap [\![\gamma']\!]_{FD} = \emptyset$. Consequently, the algorithm may visit a given state more than once (Lines 7-13). Whereas, in single-system model checking, it should not pursue the search since it already knows that the revisited state is reachable. In our case, however, it may happen that the algorithm discovers a new path from and to an already visited state s' which is executable by variants that were not known to be able to reach s'. Formally, let $R(s_0)(s')$ be the feature expression encoding the set of variants that were known to reach s' and γ' the feature expression encoding the another set of variants able to reach s' that may contain variants that were not known. Then $\gamma_{new} = \gamma' \wedge \neg R(s_0)(s')$ encodes only the set of variants $[\![\gamma_{new}]\!]_{FD} = [\![\gamma']\!]_{FD} \setminus [\![R(s_0)(s')]\!]_{FD}$ that are newly known to reach s' (Line 8). If there is at least one valid product satisfying this feature expression $[\![\gamma_{new}]\!]_{FD} \neq \emptyset$, the search continues from s' considering only the `new` variants encoded by γ_{new} (Lines 9-12). Therefore the paths starting from s are worth re-exploring only for the variants in γ_{new}. Before pursuing the exploration, the feature expression $R(s_0)(s')$ is updated accordingly.

Illustration

Let us illustrate this using the scoped featured automaton $(D1||D2||RAM)_{FA}$, the end state reachability $fCTL$ property $\phi = E\diamond(D1.end \wedge D2.end \wedge [RAM]\neg RAM.error)$ is reduced to compute the reachability relation from $s_0 = (s_1, s_1, s_1)$ to $s_{end} = (s_4, s_4, s_1)$. Figure 6.10 illustrates the depth first search of the whole state space of $(D1_{FA}||D2_{FA}||RAM_{FA})$. To propose a more realistic example, we add variable values to semantics of state as their values could also impact the possible state transitions. For example, given -512 to the value of the RAM free space $RAM.free$, the only possible will be transition $s_2 \rightarrow s_3$ while s_3 is the RAM error state. On the contrary, a value of 512 will lead to the idle state

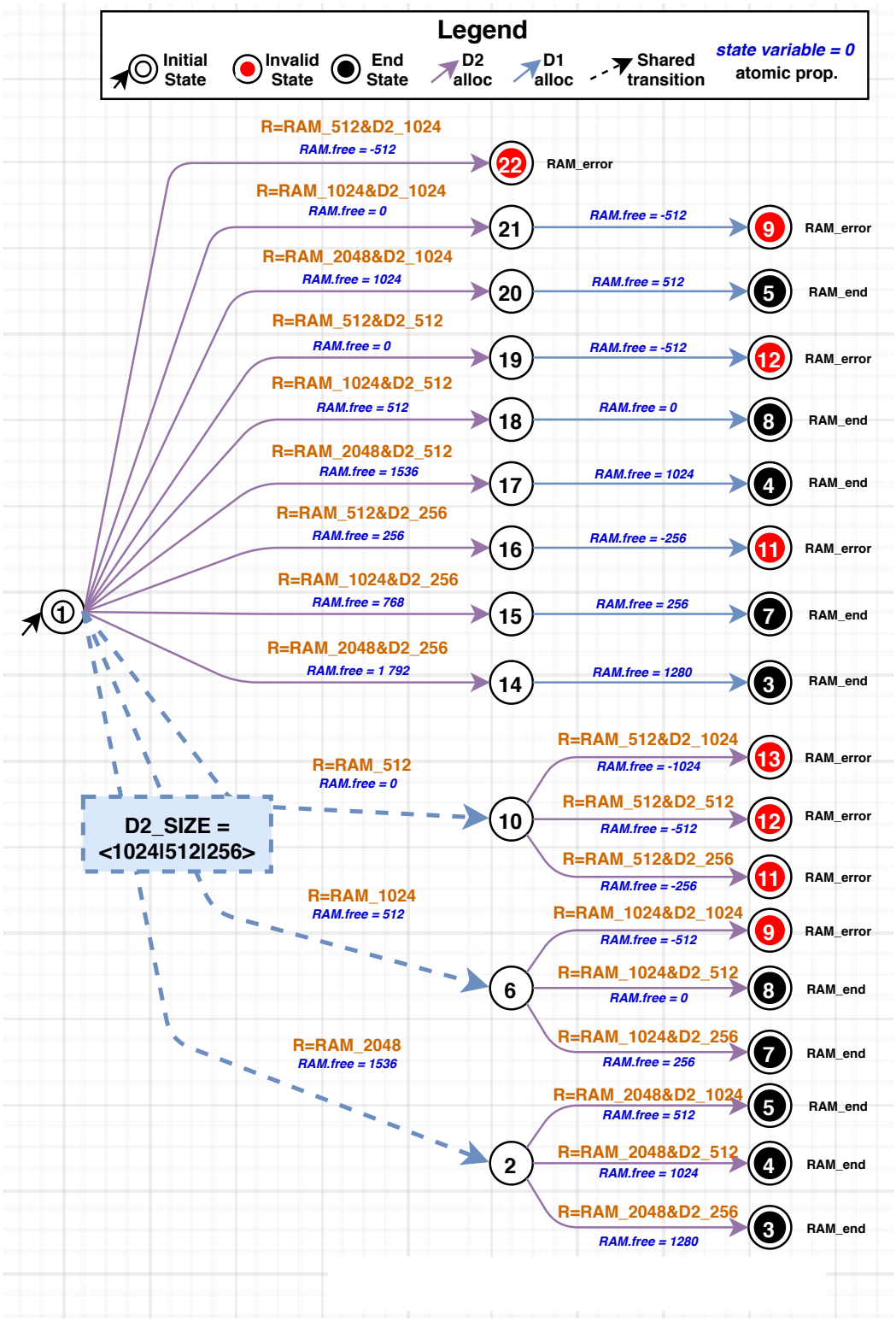


Figure 6.10: Depth First Search of Reachables($(D1||D2||RAM)_{FA}$) function.

$s_2 \rightarrow s_1$[12]. As variable value are now part of state semantic, the initial state can be noted syntactically such $s_1 = (s_1, < s_1, \psi_{size} >, < s_1, \psi_{free} >)$. Where ψ_{size} and ψ_{free} represent, respectively, the value of the D2 variable size $D2.size$ and $RAM.free$. Initially the value the variables are undetermined $\psi_{size} = \emptyset$ meaning that every value of its domain could be choose $\psi_{size} = \{256, 512, 1024\}$.

There are 12 state successors to the initial state (see Fig. 6.10): 3 states where we allocate $D1$ on RAM first rather than $D2$ (blue transitions) and 9 states where $D2$ is allocated first (violet transitions). Because the variable $RAM.free$ is used in an expression $_{free-=D1.size,overflow=free<0}$ the determination of the value of $RAM.free$ representing the free space of the RAM is required by transitions reaching any of these states. For example, $s_1 \xrightarrow{\alpha} s_2$ where $\alpha=\{(free=2048)-=(D1.size=512),overflow=free<0\},s_2)$.

Then, $s_2=(s_4,<s_1,\psi_{size}=\emptyset>,<s_1,\psi_{free}=2048>)$ represents the *designs* where we allocate $D1$ first (designs behavior) on a 2048KB RAM (designs structure). The state successors of s_2 allocate $D2$ on RAM. $D2$ size had to be determined in order to resolve the expression $_{free-=D2.size,overflow=free<0}$ such as $s_2 \xrightarrow{\alpha} s_3$ where $\alpha=\{free-=(D2.size=256),overflow=free<0\},s_3$. s_3 is a final state and the whole execution satisfy the functional requirement expressed in fCTL logic $_{s_1 \rightarrow s_2 \rightarrow s_3 \models \phi}$.

This execution can be produced by variants with $\psi =\{RAM_SIZE_2048\&D2_SIZE_256\}$ design decisions as $\pi \in UC_{|\psi}$. Finally, The system design (π, ψ) satisfies both structural requirements $[\![\psi]\!]_{FD} \subseteq [\![d]\!]_{FD}$ and behavioral ones $\pi \models \phi$[13]. On the contrary, designs using a 512KB RAM are not satisfying the behavioral requirements.

For the 9 states where D2 is allocated first (violet transitions), the transition from s_1 to one of these 9 states[14] need to determine both RAM capacity and $D2$ size to resolve the expression $_{RAM.free-=D2.size,RAM.overflow=RAM.free<0}$. Whereas $s_1 \rightarrow s_{30}$[15] are designs on which we try to allocate $D2$ first on a undersized RAM (a 1024KB D2 on a 512KB RAM), $s_1 \rightarrow s_{14}$ are those where 256KB $D2$ is allocated on 2048KB RAM.

Comparing to the single system reachability computation, we verify states $\{s_2, s_6, s_{10}\}$ once for every product. Exploiting commonalities between variant reducing thus the number of states to assess the whole design space from 36 to 30 states[16]. For example, s_2 would be verified for each variant $RAM_2048\&(D2_1024 \vee D2_512 \vee D2_256)$ in a product by product approach necessitates only one verification in our product line approach. If the state s_2 allows to satisfy (or not) the temporal logic property, this information is generalized to all variants that reach s_2.

In the complete case study (see Sec. 9.2), the error states discovered (in red) such as s_{30}, s_{29}, *etc.* can be generalized to all variants. s_{30} is a error state for every variant that has $RAM_512\&D2_1024$, no matter the other design decisions (mappings of $D1$, TD, variant with TA or TB, presence of GPU or not, *etc.*), as these properties would still be undetermined at s_{30}.

The framework provide designs as the execution traces (*i.e.*, how the application tasks are executed and scheduled onto the platform) and associated system variant that satisfy the requirements. Thereby helping the upcoming engineering of the designs.

[12] In fact it depends on positive $RAM.free \geq 0$ or negative value $RAM.free < 0$, a observation that could motivate future works such as symbolic or abstraction optimization techniques (see Sec. 10.3.4).

[13] More precisely, both executions of ψ $_{\{\pi, s_1 \rightarrow s_{14} \rightarrow s_{15}\}=[\![\psi]\!]_{FA}}$ satisfy the behavioral requirements

[14] $_{\{s_{14},s_{16},s_{18},s_{20},s_{22},s_{24},s_{26},s_{28},s_{30}\}}$.

[15] $s_{30} = (s_1, < s_3, \psi_{size} = 1024 >, < s_3, \psi_{free} = 512 >)$.

[16] Surprisingly, even in this pedagogical example, the gain is up of 16%

The design space that contains only valid design alternatives is:

$$\{(s_1 \rightarrow s_2 \rightarrow s_3, RAM_2048\&D2_256), \quad (s_1 \rightarrow s_2 \rightarrow s_4, RAM_2048\&D2_512),$$
$$(s_1 \rightarrow s_2 \rightarrow s_5, RAM_2048\&D2_1024), \quad (s_1 \rightarrow s_6 \rightarrow s_7, RAM_1024\&D2_256),$$
$$(s_1 \rightarrow s_6 \rightarrow s_8, RAM_1024\&D2_512), \quad (s_1 \rightarrow s_{14} \rightarrow s_{15}, RAM_2048\&D2_256),$$
$$(s_1 \rightarrow s_{16} \rightarrow s_{17}, RAM_1024\&D2_256), \quad (s_1 \rightarrow s_{20} \rightarrow s_{21}, RAM_2048\&D2_512),$$
$$(s_1 \rightarrow s_{22} \rightarrow s_{23}, RAM_1024\&D2_512), \quad (s_1 \rightarrow s_{26} \rightarrow s_{27}, RAM_2048\&D2_1024)\}$$

6.2.3 Conclusion

So far, we presented an end-to-end framework for system design engineering. In the chapter 5, we presented the modelling part of the framework and tried to pinpoint the main contributions. More precisely, we extends traditional specification models with variability concerns to capture variability at both application (see Sec. 5.1) and platform levels (see Sec. 5.2). We also developed a mapping strategy (see Sec. 5.3) that allows mapping these variable specifications to derive the system design space automatically from its specifications.

In the chapter 6, we present the assessing part of the framework. This part requires a mind-shift. Rather than assessing the design alternatives in a product by product method, we propose to capture and assess them as a product line. For this, we generate an FTS that encode the design space as a product line of system designs (see Sec. 6.1). FTS formalism can be seen as an automata-based model of computation extended with variability concerns.

We then reuse the variability-aware model checking algorithm in order to assess the design space efficiently against functional requirements (see Sec. 6.2). This answers the question *Which designs can be properly implemented and can render the case study HMI properly to the screen?*. Such algorithm theoretically exploits structural and behavioral commonalities between different but related designs to speed up the verification process. Basically, a common state is checked only once for every system variant that could reach it.

However, FTS is limited to functional verification. Still, we demonstrate how to verify efficiently two facets of our problem (**FC-S** and **FC-B**). While the valid design structures are guarantee though the feature model that contains only valid system variants, each valid design behavior satisfies the end-state temporal property.

To reach entirely the challenges, the designs must also meet non-functional requirements (**NF-S**, **NF-B** and **NFO**). In our case study, these include a maximal manufacturing cost, a minimal rendering quality, and a system responsiveness. Answering thus to questions such as *Which feasible designs, with an execution time less than 30.0ms, expose the highest rendering quality?* or even *Which feasible designs reach the best trade-off between rendering quality, manufacturing cost and execution time?, etc.* In the next chapter, we propose to extend our framework to manage also non-functional concerns. Finally, providing a framework foundation that reaches the challenges entirely.

Chapter 7

Toward a Complete Framework

Previously, we proposed a model-based design framework that combines *Y-Chart* pattern from embedded system design engineering and *Featured Transition System* (FTS) formalism from product line engineering. The *Y-Chart* pattern was extended at modeling stage to capture variable properties at both application and platform levels (see Chap. 5). At mapping stage, we proposed a new mapping algorithm that infers the variability-intensive system design space from the mapping of the variable application onto the configurable platform (see Chap. 5). We show that such design space can be transformed into FTS to reuse variability-aware model checking techniques, thus efficiently assessing functional requirements at both structural and behavioral levels of the entire design space (see Chap. 6).

This chapter extends the proposed framework to non-functional feasibility, non-functional satisfiability, and optimality through *Y-Chart*s models and FTS formalism with non-functional concerns. Combining, extending, and closing the gap between these two engineering philosophies, system design and product line, our novel framework allows us to reach entirely the three challenges defined in this thesis.

Each challenge is reached by extending specification models or reasoning techniques of our former modeling framework:

1. *For specifications models*: to capture non functional requirements and non functional properties at both variability-intensive application and highly-configurable platform specifications (**Challenge 1**), we extend application and platform models with non functional concerns. The resulting design space with non-functional concerns is derived (**Challenge 2**), reusing the same variability-aware mapping algorithm.

2. *For reasoning techniques*: to reason on all facets of the adapted design space efficiently (**Challenge 3**), we extend the *Featured Transition System* formalism (FTS) with quantitative properties. We also integrate complementary state-of-the-art reasoning tools such as a product-line multi-objective optimization framework.

7.1 Modeling Non Functional Concerns

Given non-functional requirements, high-level variable application and configurable platform inputs (which notably capture non-functional concerns,) the framework automatically infers the design space by implicitly mapping each data-flow variant on

each platform configuration, generating a behavioral product line extended with quantitative properties. We also extend the variability-aware model checking algorithm to assess the system design space against non-functional requirements.

At the application level, the functional expert is in charge of capturing, in addition to functional properties (*e.g.*, image size, task), non-functional properties(*e.g.*, image and task rendering quality) and quality properties (*e.g.*, overall quality) through an extended concurrent variable data-flow specification with non-functional concerns. On the platform side, the expert also captures non-functional properties (*e.g.*, memory and processor bandwidth, cost, frequency) and quality properties (*e.g.*, overall cost) of configurable hardware components such as non-programmable graphic processors and data storage units.

To support variability-intensive application and highly-configurable platform specifications, we previously proposed application and platform formal models extended with variability concerns. By mapping these specifications, we automatically infer an adapted system design space (**Challenge 2**). We now propose to extend these models with non functional properties to capture both both functional and non functional concerns (**Challenge 1**). In addition to the input specifications models, a model captures the NF requirements as constraints on quality properties and a cost function representing trade-offs between quality properties.

7.1.1 System Specifications with Non Functional Properties

Applications as variable data-flows

To capture non functional properties, we simply define additional properties Ψ such as $\Psi_{quality}$ in the *Concurrent Variable Data-Flows Graph* model VDG. However, the *quality* non functional property value of data is directly linked to its *size* functional property Ψ_{size}. For example, the 256KB size of $D2$ leads to a quality of 0, 512 to 1, and 1024 to 3. As we will see, our definition of χ allows one to define cross-cutting constraints over a given node's property values.

Definition 12 *A variable data-flow graph with non functional properties is a tuple* $VDG^* = (N, Path, E, \Psi, \chi)$ *where* $(N, Path, E,)$ *is defined as in* VDG; $\Psi = \{\Psi_{size}, \Psi_{quality}\}$ *is a set of properties;* $\chi : N \rightarrow (\Psi \rightarrow \bigcup_{\psi \in \Psi} \nu(\psi)) \rightarrow \{\top, \bot\}$ *is a function that associates a node* n *to the set of values that (all or a subset of) each property* ψ *in* Ψ *can take, where* $\nu(\psi)$ *is the finite set of values that* ψ *can take.*

Basically, χ defines which valuations of the functional and non functional properties are valid *altogether*. This flexible definition, akin to the notion of configuration of non-boolean parameters [Fleischanderl et al., 1998, Sabin and Weigel, 1998, Hubaux et al., 2012, Cordy et al., 2013b, Felfernig et al., 2014], can express that some property values are forbidden in n, and that the value of given property in n restricts the values of the others. For example:

$$\chi(D2)(\{\Psi_{size}, \Psi_{quality}\})(256, 0) = \top, \text{ but, } \chi(D2)(\{\Psi_{size}, \Psi_{quality}\})(256, 1) = \bot$$

The system quality emerges from the overall quality property of the application variant depends on *i)* the size of `D2` and *ii)* whether `A` or `B` consumes `D1` which impact their $\Psi_{quality}$ non functional property value. Furthermore, as we will see, the `size` of the data (a structural functional property) will also impact the execution *time* of the system behavior (a behavioral quality property). In our case study, the property

value of the system is obtained by summing the property values of its constituents. We make this assumption in the rest of this thesis without losing generality: one can use other aggregation functions (*e.g.*, average, maximum) instead.

Platforms as variable resource graphs

Similarly, we simply define additional properties Ψ such as Ψ_{cost}, Ψ_{freq}, *etc.* to give to the expert the means to capture non functional properties (*e.g.*, memory and processor bandwidth, cost, frequency) of configurable hardware components. We can see that Ψ_{cost} property value are linked to Ψ_{cap} property value.

Definition 13 *A variable resource graph with non-functional property is a tuple* $VRG^* = (R, C, \Theta, \Psi, \chi)$ *where* (R, C, Θ) *is defined as* VRG; $\Psi = \{\Psi cap, \Psi_{cost}, \Psi freq\}$ *is a set of properties;* $\chi : R \rightarrow (\Psi \rightarrow \bigcup_{\psi \in \Psi} \nu(\psi)) \rightarrow \{true, false\}$ *associates a resource* r *to the set of values that the properties in* Ψ *can take.*

The function χ is defined similarly to its counterpart in variable data-flow graphs and offers the same benefits. For example Ψ_{cap} and Ψ_{cost} value couples are linked:

$$\chi(RAM)(\{\Psi_{cap}, \Psi_{cost}, \Psi_{freq}\})(512, 20, 100) = \top$$
$$\chi(RAM)(\{\Psi_{cap}, \Psi_{cost}, \Psi_{freq}\})(1024, 20, 100) = \bot$$

The *overall cost* quality property of a system variant is the sum of non-functional property Ψ_{cost} of the platform resource components. Moreover, as we will see, the `freq` value of the resources (a structural functional property) will also impact the execution `time` of the system behavior (a behavioral quality property).

7.1.2 Non Functional Requirements

In addition to the input specifications models, the expert also defines the NF requirements as constraints and a cost function (to minimize or maximize) representing trade-offs between quality properties. NF constraints are constraints on manufacturing cost, execution time or even rendering quality and commonly include a maximal manufacturing cost, a minimal rendering quality, and a system responsiveness (*i.e.* time to render graphics on the visual display from which frame per seconds are determined).

Besides functional requirements, we have to identify designs that respect and optimize non-functional requirements. Some quality property of the design only depends on its structure (*i.e.*, structural quality property). Manufacturing cost and rendering quality, for example, depend on, respectively, the application variant (*e.g.* size of input data, the choice between alternative data-flow tasks) and the platform variant(*e.g.*, selected components of the platform, memory capacities). To guarantee a minimal rendering quality and a maximal manufacturing cost, the design should first of all exhibit a consistent system variant (a compatible triplet of application, mapping and platform variants to satisfy **FC-S** requirement) and satisfy both rendering quality and manufacturing cost constraints (**NF-S** requirements)[1].

At the behavioral aspect, execution time emerges from the particular mapping and scheduling of a given application over a specific platform. Both structural (*e.g.* data size, processor frequency) and behavioral aspects (*e.g.* scheduling of tasks and memory

[1] A system variant that satisfies non-functional requirements but not the functional ones is generally useless as the implementation of such systems fail.

access operations) influence[2] the resulting execution time. Consequently, every design should also respect the maximum run-time to complete the HMI-rendering (**NF-B** requirements).

Among the designs, we must also identify designs offering the best trade-off between the quality attributes (*i.e.*, minimize/maximize objectives function, *Multi-Objective Optimization* (MOO) **NF** requirements). However, a MOO NF requirements could only rely on structural quality properties (**NFO-S**) or behavioral ones (**NFO-B**) or even both (**NFO**). Optimizing both quality attributes (*e.g.* cost of manufacturing/rendering quality vs speed of execution) requires to optimize design structure and behavior simultaneously. Otherwise, this would lead to suboptimal solutions.

We thus propose a simple domain-specific language for the experts to capture such NF requirements so that the following constraints are expressed as follows:

Which feasible designs, with an execution time less than 30.0ms, have the cheapest manufacturing cost is capture by :

$$\boxed{exec_time < 30, minimize(manuf_cost)}$$

Which feasible designs, with a execution time less than 30.0ms, expose the highest rendering quality? :

$$\boxed{exec_time < 30, maximize(rend_quality)}$$

Which feasible designs, with a rendering quality score of, at least, 1 and a manufacturing cost lower than 20.0 dollars, exhibit the fastest execution time? :

$$\boxed{rend_quality_time \geq 1, manuf_cost < 20, minimize(exec_time)}$$

Which feasible designs reach the best trade-off between rendering quality, manufacturing cost and execution time? :

$$\boxed{maximize(rend_quality), minimize(manuf_cost), minimize(exec_time)}$$

7.2 Assessing Non Functional Concerns

To assess design alternatives against non-functional concerns, we extend the FTS formalism with quantitative properties to be able to capture the whole design space model with non-functional properties. Firstly, we show how the feature diagram is extended to capture non-functional properties that only depend on the design's structure and how the featured automaton added with weights can track system designs' run-time execution.

Secondly, we propose a variability-aware cost-optimal reachability model checking algorithm. We illustrate how our algorithm can be applied to assess simultaneously functional feasibility (**FC**), non-functional satisfiability (**NF**) and optimality (**NFO**) at both structural and behavioral aspect of the whole design space at once, meeting the **Challenge 3** entirely.

[2]And even directly determine for some class of systems but not in concurrent systems where, by definition, the computation of behaviors require a behavioral model to capture and assess the behavioral complexity.

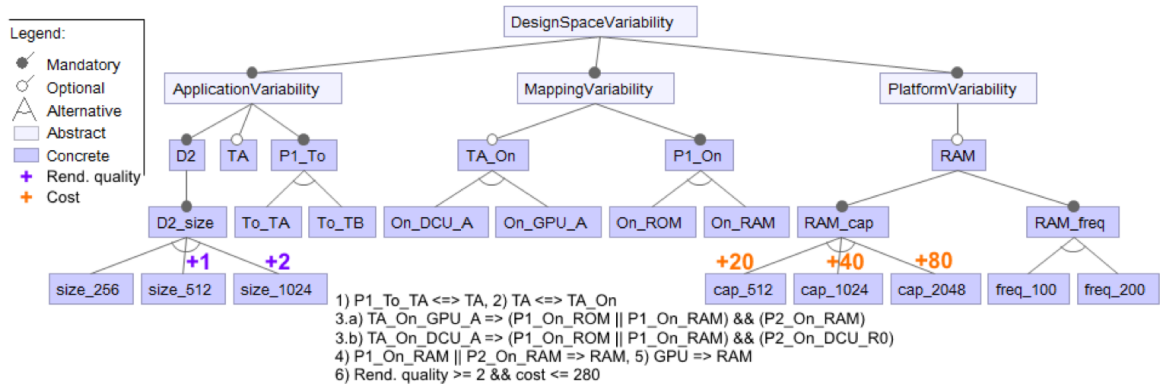


Figure 7.1: An excerpt of the PFM corresponding to the case study.

7.2.1 Design Space as a Behavioral Product Line with Quantitative Properties

To capture and assess formally non-functional concerns from system specifications, we propose to extend Featured Automaton formalism with quantitative properties. Non-functional properties that depend on design structure will be captured by the feature model with priced features so-called *Priced Feature Model* (PFM) [Cordy et al., 2013b, Benavides et al., 2007, Khalilov et al., 2016, Ross et al., 2017, Guo et al., 2017, Zhang et al., 2015, Siegmund et al., 2012, Siegmund et al., 2015, Jamshidi et al., 2017a, Valov et al., 2017, Olaechea et al., 2012, Zulkoski et al., 2014, Olaechea et al., 2014, Kugele and Pucea, 2014, Henard et al., 2015, Olaechea et al., 2016]. While execution time of hardware commands according to processing bandwidths of hardware components are captured through the featured automaton where a transition can have weights. Called *Weighted Featured Automaton* [Asirelli et al., 2011, Chechik et al., 2003, Classen et al., 2013b, Cordy et al., 2012, Cledou et al., 2017, Luthmann et al., 2017, Nunes et al., 2012, Rodrigues et al., 2015, Olaechea et al., 2016, Olaechea et al., 2018]. A weight could represent the cost of a particular behavioral quantitative property required to execute the transition. For example, the run-time required to process some data.

Priced feature model

Similarly to a feature model, a *Priced Feature Model* (PFM) naturally describes the system variant space by capturing possible design variations using features. This extension allows enriching features with prices. Adding possible prices (or costs) to features will be used to represent costs of particular design decisions (represented by the feature selection), such as selecting high-resolution data or high-quality graphical task, or high-end memory or processor. Furthermore, reasoning on the PFM allows for determining which design variants satisfy and optimize non-functional requirements (**NF-S** and **NFO-S**.

Definition 14 *A PFM is a tuple* $pfm = (F, \Phi_f, \Theta, \gamma_{\downarrow}, \Phi_\theta,)$ *where* (F, Φ_f) *is a feature model;* Θ *is a set of positive real-valued quality properties;* $\gamma_{\downarrow} : F \to \mathbb{R}_0^+$ *is a function defining how* $f \in F$ *changes the value of quality properties* $q \in \Theta$*;* Φ_θ *is a set of constraints over* Θ *defining what are the valid aggregated values for a given property* q*. given a feature combination* F'*, the aggregated value of each property* $q \in \Theta$ *satisfies the constraints* $F' \models \Phi_\theta$ *if:* $\gamma_{\downarrow}(F') \models \Phi_\theta$*; The semantics of a PFM* $[\![pfm]\!]_{PFM} : F \to$

$\mathbb{R}^n_{\geq 0}$ *is the set of valid feature combinations with associated quality properties such as* $\forall p \in [\![d]\!]_{PFD}, p \models \Phi_f \wedge p \models \Phi_\theta$ *while* $[\![pfm]\!](p)_{PFM} = \sum_{f \in p} \gamma_{\downarrow}(f)$

An excerpt of PFM is shown in Fig. 7.1. First, possible frequencies of each platform components are simply transformed into an XOR features group such as other functional properties (*e.g.*, data size, memory capacity). *Cost* price captures the associated manufacturing cost of the component. High-resolution data quality is also captured by a *Rend.Quality* price associated to the data resolution. These are derived from constraints over property values encoded in χ_n and χ_r while Φ_τ correspond to the NF requirements defined by the engineers.

Featured weighted automata

In chapter 6 we show how featured automaton formalism are able capture every possible functional executions of every design alternatives. Unfortunately, featured automaton is not able to represent and track behavioral cost consumption such as execution time. We extend the Featured Automaton formalism with non-functional concerns. Therefore, we capture not only the functional behavior of every design (*i.e.*, what we can do) but also the non-functional behavior (*i.e.*, how we can do it).

Non-functional behavioral properties, such as execution time or energy consumption, mainly depend on low-level hardware behavior. Thus, the internal mechanism of processing hardware commands such as data transfer (memory read/write commands) or data processing (gpu instructions) are modeled. We then propose a Featured Weighted Automaton, an extension that capture time consumption of hardware processing commands by adding weight to transitions.

Definition 15 *A FWA is a tuple* $fwa = (S, s_0, S_f, Act, Trans, Ap, L, d, \gamma_{\rightarrow}, \Delta, \Phi_\delta)$ *where* $(S, s_0, S_f, Act, Trans, Ap, L, d, \gamma_{\rightarrow})$ *is a featured automaton except that* d *is a PFM and* $\gamma_{\rightarrow} : Trans \rightarrow F \rightarrow \mathbb{R}^n_{\geq 0}$ *label a transition with a weighted feature expression.* $w = \gamma_{\rightarrow}(t)$ *means that taking the transition* $t \in Trans$ *,available for products* $[\![\gamma_{\rightarrow}(t)]\!]$*, will cost* $w \in \mathbb{R}^n_{\geq 0}$*.* Δ *is a set of positive real-valued quality properties;* Φ_δ *is a set of constraints over* Δ *defining what is the valid aggregated value for a given property* q*. Given a path* π*, the aggregated value of each property* $q \in \Delta$ *satisfies the constraints such as* $\pi \models \Phi_\delta$ *if:* $\sum_{t \in \pi} \gamma_{\rightarrow}(t) \models \Phi_\delta$*.*

Figure 7.2 illustrates the Featured Weighted Automaton of the hardware platform enriched with non-functional hardware processing mechanisms and associated concurrent cost consumption. The application automaton will be executed over this hardware platform automaton to explore all executions of all variants, now also considering non-functional concerns. While a featured application automaton does not need modeling modification to capture new (possibly weighted) semantic, the platform automaton becomes more complex as it models hardware processing mechanisms and their cost in term of execution times.

For example, the featured weighted automaton that captures the RAM storage not only offers memory allocation mechanism needed to assess functional concerns but weighted store and fetch data transfer mechanisms are now also provided to capture the execution time required for these commands. The data transfer performance will depend on the memory storage technology (latencies and synchronization cycles, protocol, memory controller, , *etc.*) and the configuration of its frequency. In our case, the RAM have two alternative frequencies to be set at design time. The design

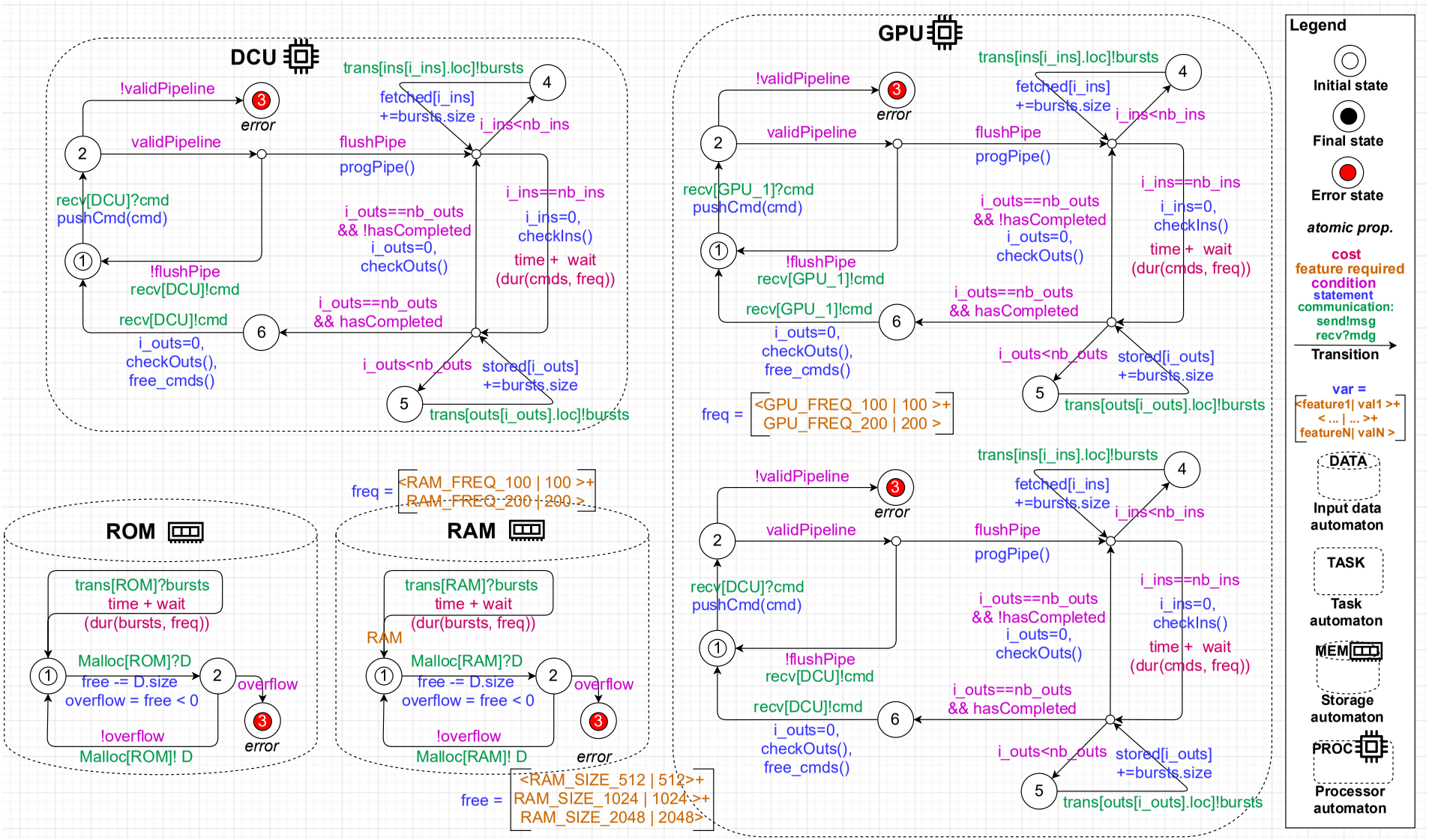


Figure 7.2: Platform Featured Weighted Automaton.

choice between $100Mhz$ or $200Mhz$ will impact the storage bandwidth and then the execution time of the data transfer commands the hardware component will receive and process.

For the processor, the mechanisms are even more complex. To assess only functional concerns, the processor automaton gathers graphic processing commands into a buffer and then validates the command buffer consistency. To assess the non-functional concerns, we need to model the processing bandwidth of each processing stage of the processor pipeline[3].

Similarly to a memory automaton, the possibly variable frequencies will impact the processing capacity of graphical processors. The interaction between the hardware components aims to mimic the real executions of the design implementations. The processor will receive commands from the application automaton that act like software. To process these commands, the processor will fetch, process data, and then store data.

7.2.2 Variability-aware Validation

We extend feature-model reasoning and model-checking techniques to identify the system designs that meet the requirements (**FC-S**, **FC-B**, **NF-S** and **NF-B**). Among the designs, the framework also finds those offering the best trade-off between the quality properties and/or are optimal (**NFO**). The behavioral product line with Quantitative Properties is checked against variability-aware cost-optimal end-state reachability temporal property to identify optimal designs *i.e.*, the set of optimal data-flow variant mapping and scheduling onto platform configuration exhibiting the most suitable quality properties.

We finally provide the execution traces that optimize the quality properties while satisfying the requirements. Such a trace shows not only the system variants able to execute it but also the behaviour they exhibit (*i.e.*, how the application tasks are executed and scheduled onto the platform), thereby helping the upcoming engineering of the designs. Identifying the system designs that meet (**FC-S**, **FC-B**, **NF-S**, **NF-B** and **NFO**) is reduced to find the FWA execution traces that exhibit the best trade-off between structural (system variant quality and manufacturing cost) and behavioral costs (run-time of the execution trace), and that also satisfy functional and non-functional constraints.

Behaviorally-Induced System Design

We now show how our *Featured Weighted Automaton* (FWA) extension can be applied to evaluate simultaneously and efficiently all facets of the problem **FC-S, FC-B, NF-S, NF-B, NFO-S, NFO-B**; the functional feasibility, the non-functional satisfiability and optimality at both structural and behavioral aspects of each design alternative while optimizing possibly-antagonistic quality attributes (*e.g.* cost of manufacturing vs speed of execution) by finding the most suitable executions of the FWA.

Definition 16 *An execution trace* π *of a weighted featured automaton* $fwa = (S, s_0, S_f, Act, Trans, Ap, L, d, \gamma_{\rightarrow}, \Delta, \Phi_{\delta})$ *has a behavioral cost of* $\sum_{t \in \pi} \gamma_{\rightarrow}(t)$ *and a*

[3]Basically, we modeled read/write latency cycles and burst memory transfers in order to capture non functional behavior of memory and the different computation cycles and parallelism of the processing stages of the processor pipelines which was enough to get a precise idea of the execution time of each design.

minimal structural cost of $\sum_{f \in \bigcup_{t \in \pi} \gamma_{\rightarrow}(t)} \gamma_{\downarrow}(f)$. *The semantics of FWA is defined as:*

$$\forall p \in [\![d]\!]_{PFM}, [\![fwa]\!](p)_{FWA} = \bigcup_{\pi \in [\![fwa_{|p}]\!]_A} \pi, \sum_{t \in \pi} \gamma_{\rightarrow}(t), [\![d]\!](p)_{PFM}$$

The Figure 7.3 shows an execution trace from the FWA that represent the execution of the system design C (see Fig. 2.6(c)). Featured transition not only impact the feature combinations needed to produce the explored execution but also the behavioral cost of the transition. Moreover, when a feature is taken as it is required to execute a particular transition, its cost is added to the structural cost of the design structure.

For example, when the system transition $s_3 \rightarrow s_6$ is fired, the start transition $s_1 \rightarrow s_2$ of TB automaton is taken. The TB feature is selected and the overall rendering quality of the system design as $\gamma_{\rightarrow}(s_1 \rightarrow s_2) = TB$ and $\gamma_{\downarrow}(TB) = +2$ rendering quality. The $RAM \wedge RAM_SIZE_2048$ allocation transition $s_1 \rightarrow s_2$ is also taken thus increasing the manufacturing cost of the design decisions as $\gamma_{\downarrow}(\gamma_{\rightarrow}(s_1 \rightarrow s_2)) = +80$ manufacturing cost. The structural cost of the entire system transition $s_3 \rightarrow s_6$ is:

$$\begin{aligned}
\gamma_{\downarrow}(\gamma_{\rightarrow}(s_3 \rightarrow s_6)) &= \gamma_{\downarrow}(\gamma_{\rightarrow}(\{s_2 \rightarrow s_3 || s_1 \rightarrow s_2 || s_1 \rightarrow s_2 || s_2 \rightarrow s_1\})) \\
&= \gamma_{\downarrow}(TB\&P2_ON_RAM\&RAM\&RAM_SIZE_2048) \\
&= \sum_{f \in TB\&P2_ON_RAM\&RAM\&RAM_SIZE_2048} \gamma_{\downarrow}(f) \\
&= (\underbrace{+2}_{\text{rend.quality}}, 0) + (0,0) + (0,0) + (0, \underbrace{+80}_{\text{manuf.cost}})
\end{aligned}$$

To model the time consumption of the various hardware processing mechanisms, transitions can also have a behavioral cost, so-called weight. The execution time is a *special* behavioral cost as its implementation takes into account the interleaving and synchronization over the different automata processes. Based on [Cordy et al., 2012] work, we mimic the time as a simplified featured clock. Each system transition as a determined time valuation according to the state and transition of the different automata processes. Figure 7.4 illustrate how run-time consumption is calculated according to the time valuation of hardware component transitions. However, this valuation can vary according to selected features. For example, as a higher frequency will increase the bandwidth of a hardware component, the selected feature representing frequency configuration will directly impact the cost of *processing* transitions.

For example, the path $s_{31} \rightarrow s_{33} \rightarrow s_{35} \rightarrow s_{37} \rightarrow s_{39} \rightarrow s_{41} \rightarrow s_{43} \rightarrow s_{45}$ represents the processing of tasks TB and TD by the hardware pipeline $GPU1$ and $GPU1$ of GPU. To do so, the hardware pipelines will fetch and process $D1$ and $D2$ data from ROM and RAM memory (transition $s_4 \rightarrow s_5$). It will finally store the processing result onto RAM (transition $s_5 \rightarrow s_4$). GPU fetch and store data on memory storage using the transition $s_1 \rightarrow s_1$ of the associated storage.

The Δ_{time} cost of a transition is the time needed to reach a state from another. The main difference between time-weighted and non-weighted transition is that a fired transition at automaton process level can take several system transitions to be fired completely. But, at each system transition, the time needed to complete the process transition is reduced according to the elapsed time during each system transition.

Let us illustrate the time valuation $\pi = s_{31} \rightarrow s_{33} \rightarrow s_{35} \rightarrow s_{37} \rightarrow s_{39}$. Where $s_{31} = (4,4,1,1), s_{33} = (4,5,1,1), s_{35} = (4,4,1,1), s_{37} = (4,5,1,1), s_{39} = (5,1,1,1)$.

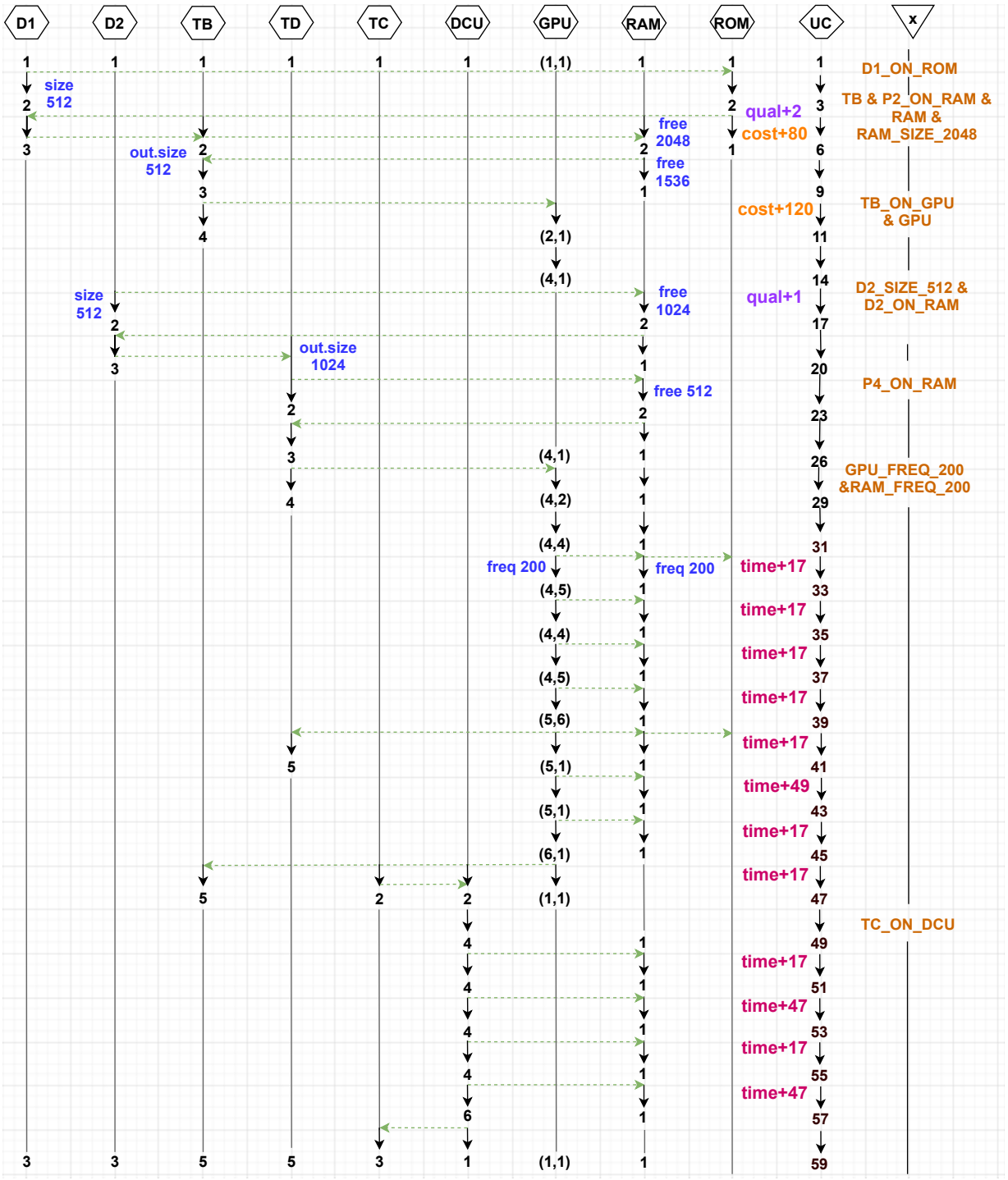


Figure 7.3: The execution of Design C

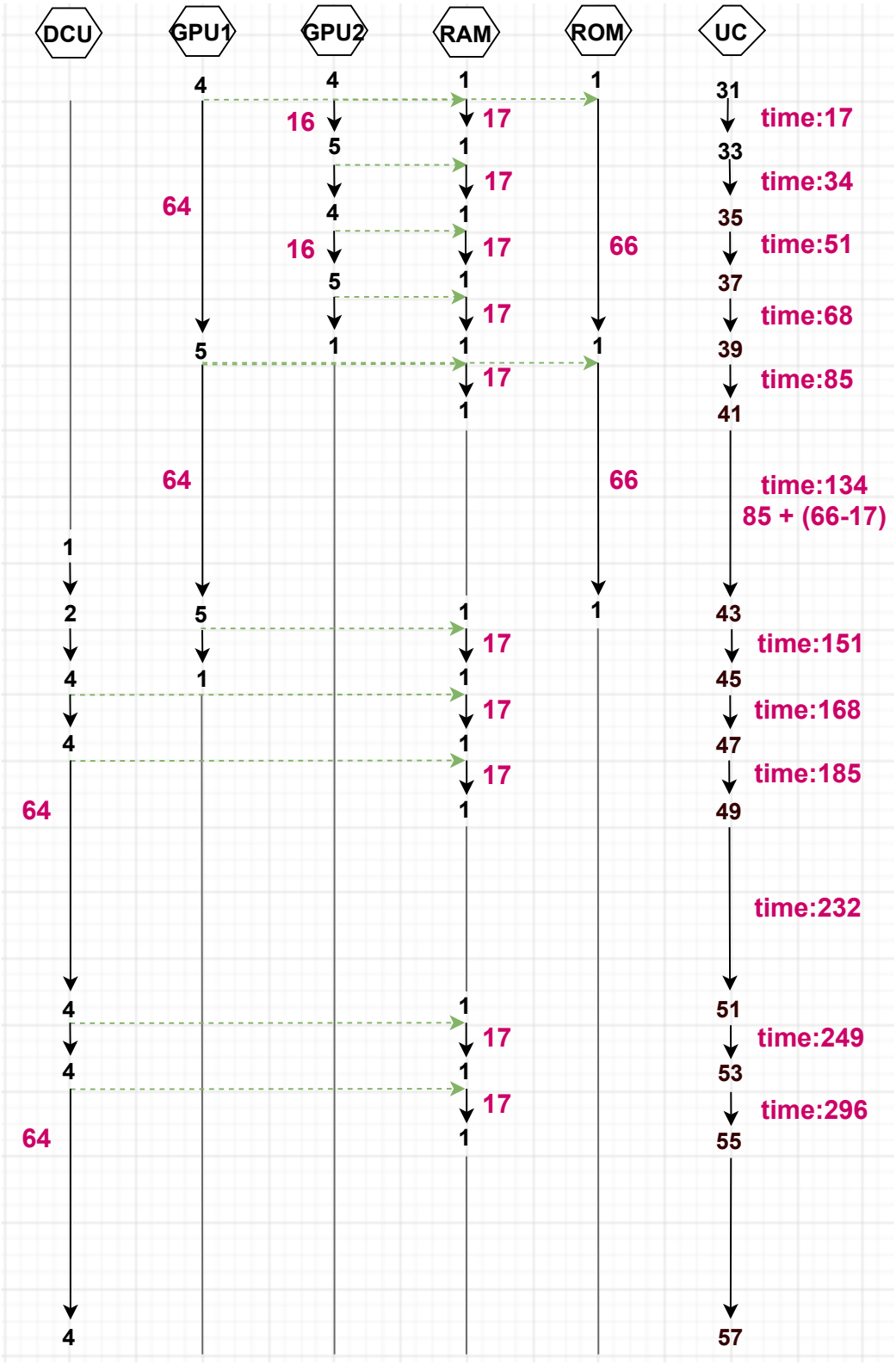


Figure 7.4: Run-time consumption of Design C

In π the minimum time from a system transition to another is 17 which is the time that the states of GPU_D and RAM evolve[4]. During π execution, the time elapsed has increased so that the process transitions that need more time can be completed (*e.g.*, GPU_B and ROM state transitions, fired at s_{31}, finally effective at s_{39}).

$$
\begin{aligned}
\gamma_{\rightarrow}(\pi) &= \gamma_{\rightarrow}(s_{31} \rightarrow s_{33}) \otimes \gamma_{\rightarrow}(s_{33} \rightarrow s_{35}) \otimes \gamma_{\rightarrow}(s_{35} \rightarrow s_{37}) \otimes \gamma_{\rightarrow}(s_{37} \rightarrow s_{39}) \\
&= \gamma_{\rightarrow}(\underbrace{4 \xrightarrow{[\Delta_{time}:64]} 5}_{GPU_B} \,||\, \underbrace{4 \xrightarrow{[\Delta_{time}:16]} 5}_{GPU_D} \,||\, \underbrace{1 \xrightarrow{[\Delta_{time}:17]} 1}_{RAM} \,||\, \underbrace{1 \xrightarrow{[\Delta_{time}:66]} 1}_{ROM}) \otimes \\
&\quad \gamma_{\rightarrow}(\underbrace{4 \xrightarrow{[\Delta_{time}:47]} 5}_{GPU_B}) \,||\, \underbrace{5 \xrightarrow{[\Delta_{time}:0]} 4}_{GPU_D} \,||\, \underbrace{1 \xrightarrow{[\Delta_{time}:17]} 1}_{RAM}) \,||\, \underbrace{1 \xrightarrow{[\Delta_{time}:49]} 1}_{ROM}) \otimes \\
&\quad \gamma_{\rightarrow}(\underbrace{4 \xrightarrow{[\Delta_{time}:30]} 5}_{GPU_B}) \,||\, \underbrace{4 \xrightarrow{[\Delta_{time}:16]} 5}_{GPU_D} \,||\, \underbrace{1 \xrightarrow{[\Delta_{time}:17]} 1}_{RAM}) \,||\, \underbrace{1 \xrightarrow{[\Delta_{time}:32]} 1}_{ROM}) \otimes \\
&\quad \gamma_{\rightarrow}(\underbrace{4 \xrightarrow{[\Delta_{time}:13]} 5}_{GPU_B}) \,||\, \underbrace{5 \xrightarrow{[\Delta_{time}:0]} 1}_{GPU_D} \,||\, \underbrace{1 \xrightarrow{[\Delta_{time}:17]} 1}_{RAM}) \,||\, \underbrace{1 \xrightarrow{[\Delta_{time}:15]} 1}_{ROM}) \\
&= 17 + 17 + 17 + 17 \\
&= 68
\end{aligned}
$$

Finally, by exploring the execution in Figure 6.4 that require a system structure with a manufacturing cost of 34$, a rendering quality of 3 and has a run-time execution of 296 cycles, we found one suitable design for the engineering question 3 *Which feasible designs, with an execution time less than 30.0ms, expose the highest rendering quality?*

All-in-one verification algorithm

Chapter 6 we show that computing the end-state reachability relation of a featured automaton allows assessing the functional feasibility of the whole design space. Basically, for each state, the reachability relation records the design decision required to reach that particular state. Finally, the design decisions encode products able to reach that state. But, this reachability relation does not record structural or behavioral cost of reaching a particular state. To this extent, we now also add the structural cost of the required design decisions and associated behavioral cost into the reachability relation.

Definition 17 *The reachability relation function from state s_0 to state s_n in a weighted featured automaton $R : S \rightarrow (S \rightarrow (F \rightarrow \mathbb{R}^n_{\geq 0}))$ returns the costs and the required feature selections in order to reach s_n from s_0*

Based on our Featured Weighted Automata (FWA) formalism, we design an algorithm to efficiently compute the reachability relation in order to solve all facets of the problem **FC-S**, **FC-B**, **NF-S** and **NF-B** for all variants at once. The key idea is to perform an exploration of the state space of the automaton in search of cost-optimal paths that can reach the accepting state while satisfying the FC and NF requirements.

The first step filters out variants that do not satisfy constraints from the priced feature model PFM, Φ_ρ and Φ_τ, thereby ensuring that the structure of the variants

[4]In fact, we should have separate $(s_{31} \rightarrow s_{33})$ into 2 transitions as the state of GPU_D evolve 1 time cycle before RAM. But for the sake of readability we simplify to offer a more concise representation.

does not violate the requirements. We then explore all paths starting from the initial state s_0. As we visit a new state, we retain the set of variants able to execute the sequence of transitions that led to the state.

We also accumulate the sum of the weights over all executed transitions. We assert that these values satisfy the NF requirements. In the end, we obtain a set of paths going from the initial to the accepting final states, together with, for each path π, (a) the valid variants that can execute π and (b) the values of the quality properties of each variant p that can execute π.

This first algorithm finds all variants satisfying the requirements. We have to find the *optimal* designs that satisfy the requirements while providing the best values for behavioral and structural quality properties. Since these properties can be antagonistic (*e.g.*, manufacturing costs can decrease to the detriment of rendering quality or rendering quality can increase to the detriment to responsiveness), the problem is assimilated to a multi-objective optimization.

To drive our search for optima, we derive from non functional requirements a cost function over all quality properties: let $\zeta(\tau_1, \ldots, \tau_n) = \theta_1 \times \tau_1 \ldots \theta_n \times \tau_n$ be our cost function, such that $\theta_i \in \mathbb{R}^+$ is the coefficient associated to the property τ_i. Then, our objective is to discover the designs that minimize ζ. This is achieved by modifying our exploration algorithm in order to *(i)* record the optimal property values, and *(ii)* stop exploring a path as soon as all quality properties reach a worse (*i.e.*, higher) value. This latter heuristics require the cost function to be monotonic as more states are explored along a given path; hence why we assume that all θ_i and τ_i are ≥ 0. This, however, is not mandatory and only allows us to stop exploring sooner.

Algorithm 3 details this exploration procedure. It takes as input a FWA, and a cost function ζ. It iteratively computes R, the reachability relation that associates each state s to the γ function encoding variants that can reach s and their associated property values. At first (Line 1), R contains s_0 together with γ_0, such that $dom(\gamma_0) = dom([\![pfm]\!])$ and $\gamma_0(F') = 0$ for any variant F'. Then, we start the exploration from s_0 (L3) and iterate over the states encountered successively (L4–L18).

At each iteration, we retrieve the state s reached last, together with its associated γ function (L5). Here, γ encodes the variants that can reach s and associates to each variant the values of its property values when following the path that led to s. If all these variants yield a value for ζ greater than the current optimum ζ^* (L6), we do not pursue the exploration further from this state. Otherwise, we distinguish between the cases where s is s_f (L7–9) and where it is not (L10–16).

In the first case, we assign ζ^* to the minimal cost over all variants that can reach the accepting state. In the second case, we compute the set of successors of (s, γ) (L12), given by $Post(s, \gamma) = \{(s', \gamma') | (s, \alpha, s') \in Trans \wedge \gamma' = \gamma \otimes \gamma_{\rightarrow}(s, s')\}$ where $dom(\gamma_1 \otimes \gamma_2) = dom(\gamma_1) \cap dom(\gamma_2)$ and $\forall F' : (\gamma_1 \otimes \gamma_2)(F') = \gamma_1(F') + \gamma_2(F')$. This means that γ' is defined only for the variants that can reach s and execute the transition from s to s', and it sums the property values of each variant in γ with its property values on the transition (s, s').

Then, we add each successor *iff* it *improves* the reachability relation (L13), that is, if for at least one variant, there is no element in R that gives a better value for all properties. This rule is captured by the comparison operator $\succeq$ over weighted feature expressions, defined as $\gamma_1 \succeq \gamma_2 \equiv \forall F' \in dom(\gamma_1) : \gamma_1(F') \geq \gamma_2(F')$. If R is improved, we continue the exploration (L14) and add the successor to R (L15) using a particular union operator $\sqcup$ that keeps R as an antichain. This is achieved by a split-and-combine algorithm along the lines of [Cordy et al., 2012,Fahrenberg and Legay, 2017b]. Finally,

Input: $fwa = (S, s_0, s_f, \rightarrow, \gamma_{\rightarrow})$;
$pfm = (F, Q, \Phi_\rho, \eta, \Phi_\tau)$; $\zeta : \zeta(\tau_1, \ldots, \tau_n) \in \mathbb{R}^+$
Output: $\mathcal{F}^*$, the set of optimal variants that reach $s_f \in S_f$
ζ^*, their associated minimal cost

1 $R \leftarrow \bot$;
2 $\zeta^* \leftarrow +\infty$;
3 $Stack \leftarrow push(\{s_0, \gamma_0\}, [])$;
4 **while** $Stack \neq []$ **do**
5 $\quad (s, \gamma) \leftarrow pop(Stack)$;
6 $\quad$ **if** $\exists F' \in dom(\gamma) : \zeta(\gamma(F')) \leq \zeta^*$ **then**
7 $\quad\quad$ **if** $s \in S_f$ **then**
8 $\quad\quad\quad \zeta^* \leftarrow \min_{F' \in dom(\gamma)} \zeta(\gamma(F'))$;
9 $\quad\quad$ **end**
10 $\quad\quad$ **else**
11 $\quad\quad\quad succ \leftarrow \begin{cases} (s', \gamma') \mid & s' \in dom(Post(s)) \wedge \\ & \gamma'' \in dom(Post(s)(s')) \wedge \\ & \gamma' = \gamma \otimes \gamma'' \wedge \\ & \nexists \gamma''' \in R(s_0)(s'), \gamma' \succeq \gamma''' \wedge \\ & \gamma' \not\models \bot \end{cases}$
12 $\quad\quad\quad$ **foreach** $(s', \gamma') \in succ$ **do**
13 $\quad\quad\quad\quad R(s_0)(s') \leftarrow R(s_0)(s') \sqcup \gamma'$;
14 $\quad\quad\quad\quad push((s', \gamma'), Stack)$;
15 $\quad\quad\quad$ **end**
16 $\quad\quad$ **end**
17 $\quad$ **end**
18 **end**
19 $\mathcal{F}^* \leftarrow \{F^* \subseteq F | (s_f, \gamma_f) \in R \wedge \gamma_f(F^*) = \zeta^*\}$;
20 **return** $(\mathcal{F}*, \zeta^*)$

Algorithm 3: optima(fwa, pfm, ζ)

we return the set of variants $\mathcal{F}^*$ that can reach s_f while minimizing ζ, together with the optimal cost (L19–20).

Illustration

The Figure 7.5 illustrates this algorithm on our case study. The transition $s_1 \rightarrow s_2$ can be taken by variants that have a 2048 Byte RAM that cost 8.0$. This is represented by selecting the *RAM_2048* priced feature. Then, $dom(\gamma_{\rightarrow}(s_1 \rightarrow s_2)) =$*RAM_2048*. While the structural cost of $s_1 \rightarrow s_2$ is the cost vector $v_{\downarrow}\{0, 80\} = \gamma_{\downarrow}($*RAM_2048*$)$ the behavioral cost is the zero vector as the transition does not consume time $v_{\rightarrow}\{0\} = \gamma_{\rightarrow}(s_1 \rightarrow s_2))$. $v_{\downarrow} \cup v_{\rightarrow} = \{0, 80, 0\}$ and $\zeta(\{0, 80, 0\}) = \tau_{quality} \times 0 + \tau_{manuf} \times 80 + \tau_{time} \times 0$ compute the global cost taking into account the preferences represented by $\tau_0 \cdots \tau_n$ of the customer over the different quality attributes. $R(s_1)(s_2)$ is updated to $\gamma_a \in Post(s_1)(s_2)$ defined for variants $[\![$*RAM_2048*$]\!]$ and equal to the cost vector $v = \{0, +80, 0\}$.

Similarly, the transition $s_2 \rightarrow s_3$ is just requiring a low resolution $D2$ and low quality TA processing. Consequently the quality loss of that design choices is added

to the cost of that transition. As $Post(s_2)(s_3) \subseteq \gamma_b(_{D2_512\&TA\&TA_ON_DCU\&TD_ON_DCU})$ where the function application is equal to $\{+3, 0, 0\}$ then $R(s_1)(s_2)$ is updated to $\gamma_c = \gamma_a \otimes \gamma_b$. γ_c is thus defined (*i.e.*, $dom(\gamma')$) for variants that have the design decisions $F_c =_{RAM_2048\&D2_512\&TA\&TA_ON_DCU\&TD_ON_DCU}$ and has for value the cost vector $\{+3, +80, 0\} = \gamma_a(F_a) + \gamma_b(F_b), F_c = F_a \cup F_b$.

The transition $s_3 \rightarrow s_4$ has a behavioral cost, but no structural one. It requires the RAM frequency to be configured to $200MhZ$ and requires also $144ms$ of time. $R(s_1)(s_2)$ is then updated to $\gamma_d = \gamma_c \otimes \gamma_e, \gamma_e \in Post(s_3)(s_4), \gamma_e(F_e) = \{0, 0, +144\}, F_e =_{RAM_100MhZ}$. $s_3 \rightarrow s_5$ is similar than $s_3 \rightarrow s_4$ but require a higher RAM frequency which lead to a lower time consumption to execute the entire application. As s_4 and s_5 are final states, the algorithm computes the associated cost using the cost function.

Increasing the RAM frequency will not lead to improve the time consumption of the design execution. A higher $D2$ resolution $s_2 \rightarrow s_6$ will overload the DCU and ROM hardware components taking $256ms$ to execute the application elements regardless the RAM frequency (*i.e.*, $s_6 \rightarrow s_7$, $s_6 \rightarrow s_8$). In this illustration, this will violate the time constraint of $200ms$. On the other hand, taking a RAM of $512KB$ capacity $s_1 \rightarrow s_9$ will lead (*i.e.*, $s_9 \rightarrow s_10$, $s_9 \rightarrow s_11$, $s_9 \rightarrow s_12$) to an overflow. This shows that the algorithm is still checking functional requirements while checking also non functional ones.

Finally, taking a $1024KB$ RAM will exhibit highly efficient paths at a lower price. The algorithm will return such designs as optimal as the cost function will be minimized by both low-cost system structure and behavior. Regarding the non-functional constraints over the quality attributes, that forbid, for example to have a $1024KB$ RAM and a GPU because it is too costly as the maximum cost is 32.0$. That notably limit designs to only use DCU. Using only this graphical controller unit will constraint the $D1$ image processing by the lower quality TA task. This shows that the different non-functional constraints are interleaved and constrain the design space in a complex manner. In the end, in this illustration the most promising design is with respect to $time \leq 200$, $munuf_cost \leq 320$ and $rend_quality_loss \leq 3$ constraints and the function cost $time + munuf_cost \times 1 + rend_quality_loss \times 100$:

$$(s_1 \rightarrow s_{13} \rightarrow s_{14} \rightarrow s_{15}, RAM_1024\&D2_512\&TA_ON_DCU\&TD_ON_DCU)$$

7.3 Conclusion

We finally extend our previously framework to model and assess the non-functional concerns to reach the challenges entirely. We reach the (**Challenge 1**) through the modelling part of our framework. We capture the non functional requirements (see Sec. 7.1.2) and extend our variable application and configurable platform specification models with non functional properties (see Sec. 7.1.1. Similarly to functional properties, non-functional ones could also have variable values. The variability extension we proposed to capture variable functional properties is expressive enough also to manage non-functional ones. Although the non-functional properties of the specification models do not impact their mappings. Application elements and platform resources exhibit non-functional properties but do not impact their possible mappings. Thus, we reused our mapping strategy to finally derive the design space that captures both functional and non-functional properties. (**Challenge 2**) is then reach.

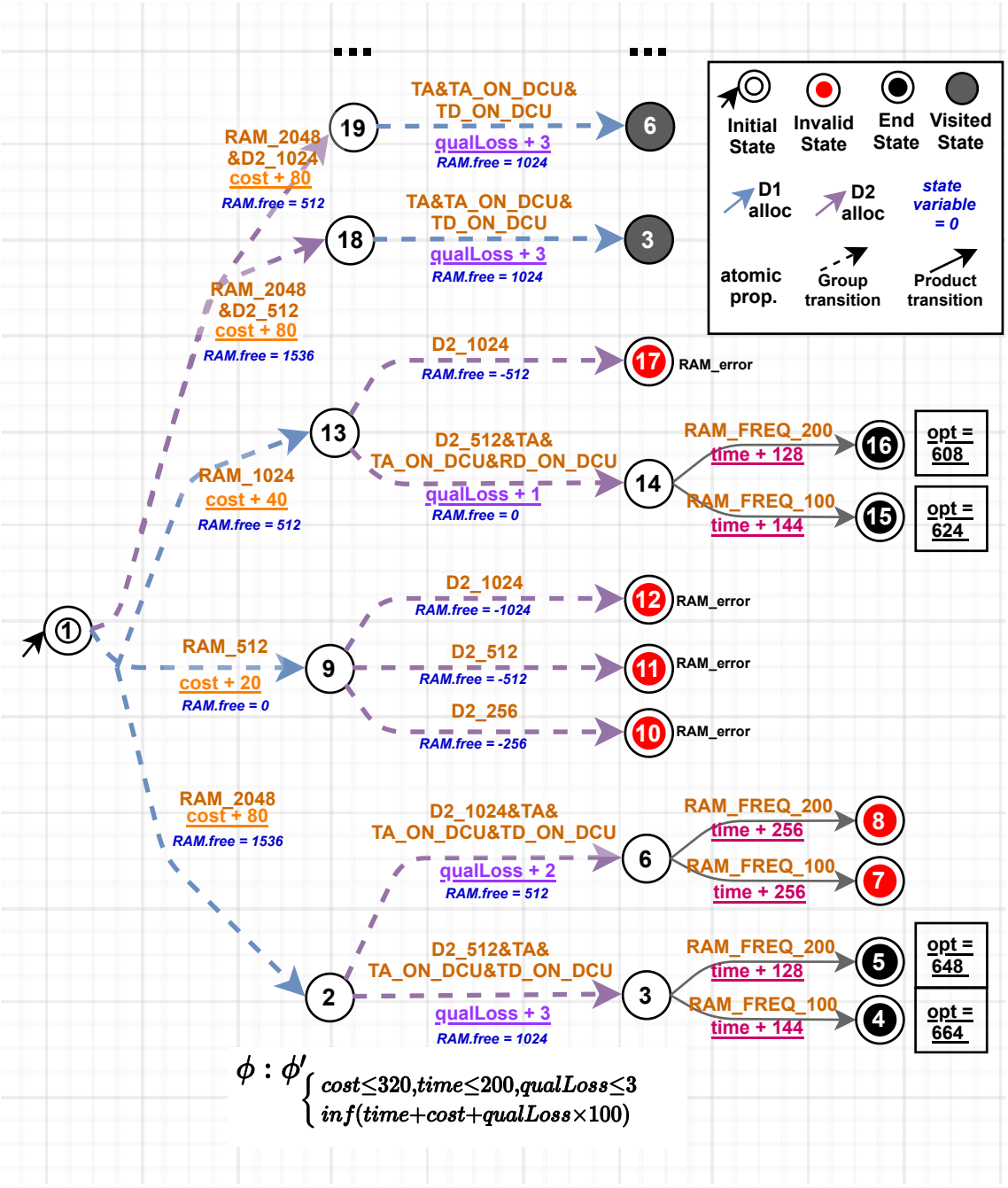


Figure 7.5: Depth First Search of Optima(UC) function.

At the assessing part of the framework, we proposed an FTS extension we called *Featured Weighted Automata* (FWA) to capture and assess our system design space against non-functional concerns. To this extent, we both extend the feature model and the featured automata with quantitative properties (see Sec. 7.2.1). The priced feature model allows to assess the non functional requirements at a structural aspect (facet **NF-S**). Relying on analytical methods, questions such as *Which feasible designs have a manufacturing cost less than 30$?* can be answered through the priced feature model. On the other hand, our FWA extension adds non-functional attributes to the featured state space. Consequently, the non-functional attributes of system executions, such as run-time of each execution can also be checked against non-functional requirements at a behavioral level (facet **NF-B**). *Which feasible designs exhibit an execution time less than 30.0 ms?*.

We finally proposed an algorithm that efficiently assesses our design space represented by an FWA against the whole set of requirements. This algorithm can also identify the designs that optimize a function cost previously defined by the engineers. Giving the ability to answering to any questions depicted in Section 2.1.2. By evaluating simultaneously and efficiently all facets of the problem **FC-S, FC-B, NF-S, NF-B, NFO**; the functional feasibility, the non-functional satisfiability, and optimality at both structural and behavioural aspects of each design alternative, we reach the **Challenge 3**.

So far, we presented a formal theoretical framework. The modeling method we present allows capturing and inferring the design space from variable application and configurable platform. The assessing part of the framework we present relies on an extended FTS behavioral product line. Such formalism seems to exploit structural and behavioral commonalities between products in order to speed up the verification process. Behavioral commonalities are the states shared by a group of products while structural commonalities are design choices present on a group of products. To speed-up the verification process, design choices that do not respect structural constraints such as manufacturing cost or rendering quality are efficiently pruned. Moreover, each state is checked once for every system variant that could produce it. In the next chapter (see Chap. 8), we implement our framework and evaluate its efficiency on our industrial case study and other instrument clusters.

Part III

Validation

Chapter 8

Framework Implementation

In order to evaluate our framework, we implemented it as a toolchain. In this chapter, we detailed our implementation, which can map and capture variable data-flow applications and configurable platforms to infer the design space and transform the design space in our *Featured Weighted Automaton* (FWA) formalism. Our implementation combines original development with state-of-the-art tools or extensions and returns the most suitable systems designs.

8.1 Implementation Overview

Separated in two main parts, front-end and back-end (see Fig. 8.1), our implementation follow the *Y-chart* model-based design space exploration pattern in order to realize the three processes, modeling, mapping and analysis. The front end[1] is composed of two modules, each of which implements a process depicted in Fig. 8.1.

The first one allows for specifying a variable data-flow graph (see Listing 8.1) and a variable resource graph (see Listing 8.2) and non functional requirements via fluent *Java* APIs. *Java* language is used to capture, the variable applications (see Fig. 8.2(a)), configurable platform (see Fig. 8.2(b)) specifications models, NF requirements and the variability-aware mapping algorithm. *Java* meta-models capture variable applications and configurable platforms so that application elements or platform components are directly represented by *Java* objects. The *Java* mapping algorithm takes a variable application and configurable instance in order to create the mapping model (see Fig. 8.2(c)) and then represent the whole design space through a *Java*-based model.

The second one is an FWA generator that consumes inputs, design space and NF

[1] `https://bitbucket.org/SamiLazreg/enlighter`

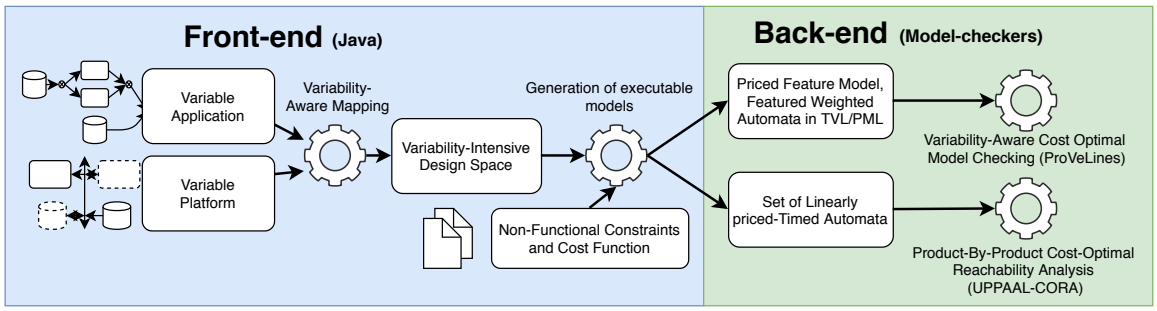


Figure 8.1: Framework Models and Processes

requirements to generate assessing models to our back-end automated reasoning tools. We implemented our FWA Algorithm 1 over the *ProVeLines* model checkers[2]. We are also able to generate our design space to a set of *UPPAAL-CORA*[3] model checker automata.

UPPAAL-CORA [Behrmann et al., 2005] is an established tool to carry out *Cost-Optimal Reachability Analyses* (CORA) that we reuse as is. It takes as input a network of *Linearly Priced Timed Automata* (LPTA) [Behrmann et al., 2001]. LPTA can be regarded as FWA without variability, can only encode the behaviour of the variants separately. Our automata generator actually transforms our design space into a network of LPTAs in the *UPPAAL-CORA* format. Using SPLOT's feature model reasoning library [Mendonca et al., 2009], we also generate an additional automaton dedicated to configuring the other LPTAs *before* their execution starts by setting variables that correspond to the variation points of the design space. We thus follow the 150% model approach [Thüm et al., 2014]. Then, *UPPAAL-CORA* can find an execution of a variant that reaches the accepting state while satisfying all the NF requirements and optimizing the cost function.

The other model checker is *ProVeLines* [Cordy et al., 2013a], which can check variability-intensive systems. We chose this tool because it was extended over the years, by both its original developers [Cordy et al., 2013a] and others [Olaechea et al., 2016, Olaechea et al., 2018], to solve multiple model-checking problems including real-time verification [Cordy et al., 2012]. This gave us confidence that we could extend *ProVeLines* to implement our own algorithms. We then fully implemented Alg. 3 in a new version of *ProVeLines* name *ProVeLines-CORA*. To achieve this, we first extended *ProVeLines* input language *Promela* [Holzmann, 2004] to associate *Promela* statements with weighted feature expressions. Actually, each *Promela* process encodes a single FWA. Like *UPPAAL-CORA*, our *ProVeLines* extension is able to provide the execution trace associated to an optimal variant. The difference lies in that weighted feature expressions allow an all-at-once verification of all variants. To encode the structural variability, we generate a PFM in the *Textual Variability Language* (TVL) format supported by *ProVeLines* [Classen et al., 2011a, Cordy et al., 2013b].

8.2 System Specifications and Mapping Models

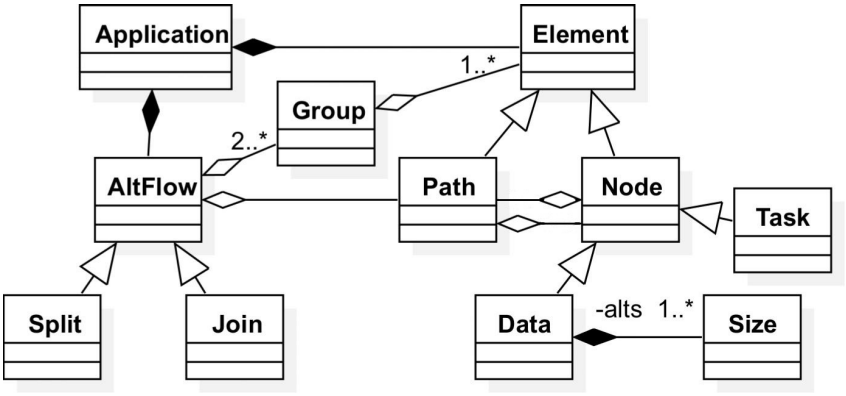
We now detail how we implement the variable application model and the configurable platform model. We also show how we implement the mapping model and the mapping application mapping algorithm onto the platform.

8.2.1 Applications as Variable Data-Flows

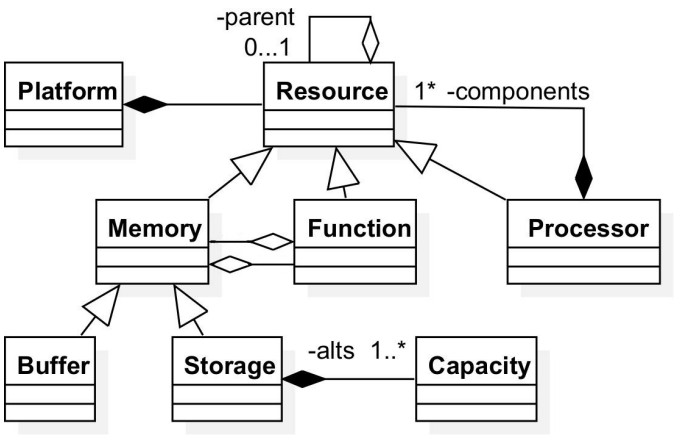
Listing 8.1: Running Example Application

```
1 Application app = new Application("WarpWithWhat");
2
3 Path p1 = app.addPath("P1"); Path p2 = ...; Path p3 = ...; Path p4 = ...;
4
5 DataSource d1 = app.addDataSource("D1").addSize(512).connect("o", p1);
6 Task ta = app.addTask("ta", "A").connect(p1, "i").connect("o", p2);
```

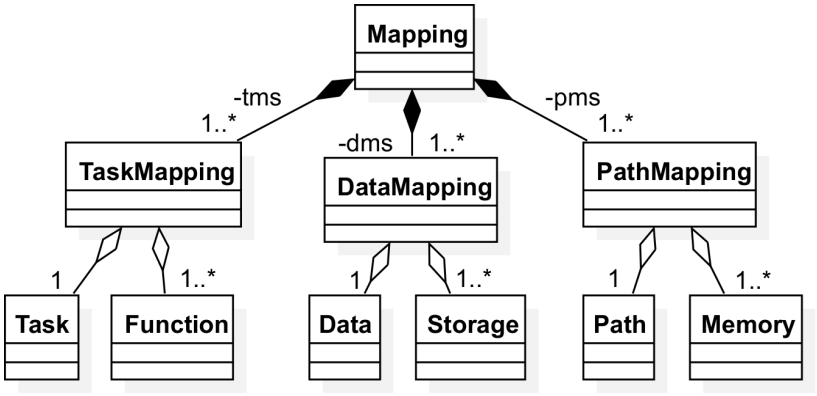
[2] https://bitbucket.org/maxcordy/provelines-cora
[3] https://people.cs.aau.dk/~adavid/cora/introduction.html



(a) Application Meta-model



(b) Platform Meta-model



(c) Mapping Meta-model

Figure 8.2: Meta-models implementations

```
7  Task tb = app.addTask("tb", "B").connect(p1, "i").connect("o", p2);
8
9  DataSource d2 = app.addDataSource("D2").addSize(256, 2).addSize(512,
       1).addSize(1024).connect("o", p3);
10 Task td = app.addTask("td", "D").connect(p3, "i").connect("o", p4);
11 Task tc = app.addTask("tc", "C").connect(p2, "i0").connect(p4, "i1");
12
13 app.split(p1).to(ta).to(tb);
14 app.join(p2).from(ta).from(tb);
```

The *Java* data-flow meta-model implementation (see Fig. 8.2(a)) allows to capture the functional and non functional properties of a variable application as concurrent data-flow model (see Fig. 2.4). Structural and behavioral elements are captured through java objects. Tasks are represented by Task objects, *etc.* Alternatives data size can be captured by allowing Data object to have multiple attached Size objects. Listing 8.1 illustrates how we capture the functional and non functional requirements (see Fig. 2.4) of the embedded system through a *Java* data-flow API. Data are instantiated at lines 5, 9, tasks at line 6, 7, 10, 11 data-path at line 3). This API allows to capture the variability in both structural (*e.g.*, data sizes at line 5, 9) and behavioral aspect (*e.g.*, alternative flows by allowing data-paths to have multiple inputs and output tasks connected at line 13, 14). Also, image data *D2* resolutions that influence the overall quality loss are captured at line 9.

8.2.2 Platforms as Variable Resource Graphs

Listing 8.2: Running Example Platform

```
1  Platform plt = new Platform("Kepler");
2
3  Storage rom = plt.addStorage("ROM",
       Type.READ_ONLY).addCapacity(4096).setAccessLatency(2).
       setBytesPerCycle(4).setFrequency(100).setCost(60);
4
5  Storage ram = plt.addStorage("RAM", Type.READ_AND_WRITE).addCapacity(512,
       40).addCapacity(1024, 60).addCapacity(2048,
       80).setAccessLatency(2).setBytesPerCycle(8).setFrequencies(100,
       200).setOptional("true");
6
7  Component dcu = plt.addComponent("DCU").setFrequency(100).setCost(80);
8  Processor a_dcu = dcu.addProcessor("a", "A").setBytesPerCycle(4);
9  Memory r0_dcu = dcu.addFIFOBuffer("R0");
10 a_dcu.connectToInputPort("i", ram, rom).connectToOutputPort("o", r0_dcu);
11 ...
12 Component gpu = plt.addComponent("GPU").setFrequencies(100,
       200).setOptional("true").setCost(120);
13 Processor a_gpu = ...; Memory r0_gpu = ...; Processor b_gpu = ...;
14 gpu.requires(ram);
```

Listing 8.2 illustrates how we express the configurable platform specification of the case study (see Fig. 2.5) of the embedded system through our resource component-based *Java* API implementation (see Fig. 8.2(b)). This captures functional and non functional properties of templated resource components such as optional GPU multi-pass processors with configurable frequency instantiated at line 17-18 and streaming

DCU processor at line 7-15[4]. Other elements in the processors can be the processors' hardware processing functions instantiated at line 8, 11, 14, 18 and FIFO buffers at line 9, 12, 18 relevant elements being connected with each other. Read-only ROM memory is instantiated at line 3, optional read-write RAM memory with variable capacity (affecting the RAM manufacturing cost) at line 5. In addition, a platform can have variability dependency (line 19) on resources.

8.2.3 Non-Functional Requirements

In addition to the input specification models, NF requirements as constraints and the cost function representing trade-offs between quality properties can also be defined using our *Java* API (see Listing 8.3). The non-functional constraint is represented by a *Java* object that can capture constraints over quality attributes such as the rendering quality, manufacturing cost and run time. The cost function also allows to specify particular weights for each quality attribute to express the quality preferences or priorities.

Listing 8.3: Non-Functional Requirements

```
1 NonFunctionalCst nfRequirements =
2            NonFunctionalCst.AND(
3                 NonFunctionalCst.QUALITY_LOSS("<=", 2),
4                 NonFunctionalCst.COST("<=", 280),
5                 NonFunctionalCst.RUN_TIME("<", 840));
6
7 FunctionCost cfct = new FunctionCost().QUALITY(100).COST(10).RUN_TIME(1);
```

8.2.4 Variability-Aware Mapping Process

The mapping algorithm, implemented in *Java* (see Listing 8.4) takes as inputs the variable data-flow and configurable platform *Java* models, and generates the *Variability-Aware Mapping Space*(see Fig. 8.2(c) for metamodel). It then represents all mapping of application elements onto platform resources.

The process is composed of two steps: *i)* it maps each data source and output data path on storage memories (line 4-7) *ii)* it maps each task on appropriate processor function (*i.e.*, processor function can implement the task while data path inputs can be mapped on reachable memory) and maps task output on memory (line 8-11). Then, the algorithm prunes unfeasible mappings *w.r.t.* structural and variability constraints at line 12 (*e.g.*, data-path mapping are not reachable by any task mapping or vice versa), finally adding appropriate consistency constraints to ensure mapping space consistency (line 13).

Listing 8.4: Mapping Algorithm

```
1 Mapping mapping = new MappingAlgorithm().map(app, plt);
2 ...
3 public Mapping map(Application app, Platform plt) {
4    for(DataSource ds : app.getDataSources()) {
5       addDataSourceMappings(ds, plt.getStorages())
6       addDataPathMappings(ds);
7    }
```

[4]This resource template are linked to FWA implementations

```
8     for(Task t : app.getSortedTasks()) {
9        addTaskMappings(t, getProcessors(plt, t));
10       addDataPathMappings(t);
11    }
12    do while(removeUselessMappingChoices());
13    addMappingConstraints();
14    return new Mapping(dms, tms, pms);
15 }
```

8.3 Assessing the System Design Space

From the system design space model (see listing 8.5), we generate a *fPromela* FWA *Behavioral Product Line* that capture structure, behavior and variability aspects of the design space. To this extent, we first extended *ProVeLines* input language Promela [Holzmann, 2004] to associate *Promela* statements with weighted feature expressions. This extensions allow to capture behavioral costs. While the *Weighted Featured Automaton* is encoded in *fPromela* language (see listing 8.7), the *Priced Feature Model* is described in *TVL* (see listing 8.6). Secondly, we fully implemented Alg. 3 in a new version of *ProVeLines* name *ProVeLines-CORA*. Thus, we can use our model checker *ProVelines-CORA* to assess the functional feasibility, non functional satisfiability and even optimality of the whole system design space.

8.3.1 Design Space as a Behavioral Product Line

The PFM variability model expressed in TVL [Classen et al., 2011a, Cordy et al., 2013b] is a textual representation of the PFM (see Fig. 7.1). The structure of each system element and its variable properties, functional and non-functional, are captured through features. For example, at platform level the *RAM* which is represented by an optional feature starting at line 19 have alternative capacities and alternatives frequencies represented by *XOR* feature groups at, respectively lines 19-24 and 25-28. The overall manufacturing cost is different for each capacity. Data mapping is captured in lines 34-37, task mapping 38-41, alternative $P1$ flow variability 5-8, 14, 15, resource optionality 10, 30, and resource dependency at line 53. Application to mapping is captured at line 45, mapping to mapping 47-49, and platform to mapping consistency constraints (line 51) are also represented as featured constraints.

Besides, the FWA (see Fig. 7.2 is captured by an executable network of featured automata in our extension of *fPromela* textual language. *fPromela* is an imperative language that captures state and transition through instructions. There are several type of instructions such as statements, message passing/receiving, *if* and *do* blocks, *etc.* In *fPromela* each featured automaton that composes the network is a process. For instance, *RAM* storage 3-33, $D2$ image 92-120, task *TA* 50-72 and *DCU* processor 35-70 are different and concurrent process. Each line execution is a state transition, while each process has a respective execution thread. Featured transitions are guarded by feature selections. For example, to set the initial capacity of *RAM* to a particular value, the associated feature selection of the feature guard must be selected.

Listing 8.5: Generate Running Example Formal Models from Design Space

```
1 DesignSpace ds = new DesignSpace(app, mapping, plt);
2 ToTVL toTVL = new ToTVL().generate(ds);
```

```
ToPML tofPML = new TofPML().generate(ds);
```

Listing 8.6: Part of Running Example Design Space variability in TVL

```
root DesignSpaceVariability{
   group allOf{
      ApplicationVariability group allOf{
         ...
         P1_to group oneOf{
            P1_to_TA,
            P1_to_TB
         },
         D2_size group oneOf{
            D2_size_256 => qualityLoss + 2,
            D2_size_512 => qualityLoss + 1,
            D2_size_1024
         },
         opt TA => qualityLoss + 2,
         opt TB
      }
      group PlatformVariability{
         ...
         opt RAM group allOf{
            RAM_size group oneOf{
               RAM_size_512 => cost + 40
               RAM_size_1024 => cost + 60
               RAM_size_2048 => cost + 80
            },
            RAM_frequency group oneOf{
               RAM_frequency_100,
               RAM_frequency_200
            }
         },
         opt GPU => cost + 120
      }
      group MappingVariability{
         ...
         D2_On group oneOf{
            D2_On_RAM,
            D2_On_ROM
         }
         opt TA_On group oneOf{
            TA_On_DCU_a,
            TA_On_GPU_a
         },
      }
   }
   ...
   TA <=> TA_On;
   ...
   D2_On_RAM => P3_On_RAM
   TA_On_SoC_GPU_1_A => (P1_On_SoC_ROM || P1_On_SoC_RAM) && P2_On_SoC_RAM;
   TA_On_SoC_DCU_A => (P1_On_SoC_ROM || P1_On_SoC_RAM) && P2_On_SoC_DCU_R0;
   ...
   D2_On_RAM => RAM;
   ...
   GPU => RAM;
   ...
}
```

Listing 8.7: Part of Running Example Design Space behavior in fPromela

```
check <> (d1_end && ... && soc_gpu_1_idle) qualityLoss 100 cost 10 time 1
    within qualityLoss <= 2 cost <= 280 time < 840
...
active proctype Storage_RAM(){ atomic{

   bool ram_idle = true;
   int cons = 0;
   Data in;

   gd
   :: RAM ->
      ...
      do
      :: Malloc[RAM_ID]?in -> cons = cons + in.size;
         if
         :: RAM_capacity_512  -> size = 512;
         :: RAM_capacity_1024 -> size = 1024;
         :: RAM_capacity_2048 -> size = 2048;
         fi;
         assert(cons <= size);
         Malloc[RAM_ID]!in;

      :: Trans[RAM]?in -> ram_idle = false;
         if
         :: RAM_frequency_100 -> freq = 100;
         :: RAM_frequency_200 -> freq = 200;
         fi;
         wait((latency*main_freq/freq)+(burst/bpc*main_freq/freq));
         Trans[RAM]!in;
         ram_idle = true;
      od;
   :: else -> skip;
      ...
};}
...
active proctype processor_DCU(){ atomic{

   bool dcu_idle = true;
   int freq = 100;
   command cmd;

   do :: recv[DCU_ID]?cmd -> dcu_idle = false;
           ...
           setOutputs() cmd_list_push() is_flush_cmd()
           ...
           if
          :: !_is_flush_cmd -> Recv[DCU_ID]!cmd;
           :: else -> prog_pipeline()
               do
               :: !hasCompleted() ->
                    ...
                    do
                    ::i_ins < cmd.nb_ins ->
                        Trans[cmd.ins[i_ins].loc]!cmd.ins[i_ins];
                      fetch[i_ins] = fetch[i_ins++] + burst;
```

```
                    ::i_ins == cmd.nb_ins -> checkIns() break;
                    od;
                    ...
                    wait(time_cmds());
                    ...
                    do
                    ::i_outs < cmd.nb_outs ->
                        Trans[cmd.outs[i_outs].loc]!cmd.outs[i_outs];
                       store[i_outs] = store[i_outs++] + burst;
                    ::i_outs == cmd.nb_outs -> checkOuts() break;
                    od;
                    ...
                :: else -> Recv[DCU_ID]!cmd; free_cmd_list() -> break;
              od;
          fi;
        soc_dcu_idle = true;
   od;
};}
...
active proctype Data_D2(){ atomic{
   Data out;
   bool d2_end = false;
   ...
   if
   :: D2_size_256 -> qualityLoss + 2, out.size = 256;
   :: D2_size_512 -> qualityLoss + 1, out.size = 512;
   :: D2_size_1024 -> out.size = 1024;
   fi;
   if
   :: D2_On_RAM -> Malloc[RAM_ID]!out;
                        Malloc[RAM_ID]?eval(out);
   :: D2_On_ROM -> Malloc[ROM_ID]!out;
                        Malloc[ROM_ID]?eval(out);
   fi;
   ...
   P3!out;
   d2_end = true;
};}
...
active proctype Task_TA(){ atomic{
   Data in, out;
   bool ta_end = false;
   ...
   if
   :: TA -> P1?in;
       ...
       if
       :: TA_On_GPU_a ->
            ...
            Malloc[RAM_ID]!out;
            Malloc[RAM_ID]?eval(out);
            recv[GPU_1_ID]!cmd("a", in, out);
            ...
            recv[GPU_1_ID]?eval(cmd);

       :: TA_On_DCU_a ->
            ...
            Recv[DCU_ID]!cmd("a", in, out);
            ...
```

```
112             Recv[DCU_ID]?eval(cmd);
113         fi;
114         ...
115         P2!out;
116     :: else -> skip;
117     fi;
118     ...
119     ta_end = true;
120 };}
121 ...
```

8.3.2 Variability-Aware Validation Process

Our generated formal models (*fPromela* and *TVL*) are checked through our model-checker *ProVeLines-CORA* with specific command lines (see Listing 8.8 line 1). It returns the system variants that do not exhibit a an execution that respects non functional requirements. For instance the design $(\pi_{inv1}, \psi_{inv1})$ (line 3-18):

$$\pi_{inv1} = @0 \rightarrow ... \rightarrow @13418292 \rightarrow ... \rightarrow @15418392 \rightarrow ... \rightarrow @19418392 \rightarrow ...$$
$$\psi_{inv1} = RAM\&RAM_CAP_512\&D2_SIZE_1024$$

will exceed the *RAM* storage capacity. Consequently the design does not fulfill the functional requirements. About the design $(\pi_{inv2}, \psi_{inv2})$ (line 19-35):

$$\pi_{inv2} = @0 \rightarrow ... \rightarrow ... \rightarrow ... \rightarrow ... \rightarrow ... \rightarrow ... \rightarrow ...$$
$$\psi_{inv2} = D2_size_1024\&D2_On_ROM\&D1_On_ROM\&$$
$$TB_On_GPU\&TD_On_GPU\&GPU\&GPU_freq_100$$

will exceed the maximum execution time. Hence, the design does not fulfill the non functional requirements ($time <= 840$). Given this output, we can remove the invalid system variant from the design space by constraining the PFM (see Listing 8.9).

The model-checker output also returns output containing optimal designs satisfying functional and non functional requirements (line 38 - 50). For instance, the optimal design (π_{opt}, ψ_{opt}):

$$\pi_{opt} = @0 \rightarrow ... \rightarrow @13418292 \rightarrow ... \rightarrow @22528392 \rightarrow ... \rightarrow @134189119 \rightarrow ...$$
$$\psi_{opt} = D1_ON_ROM\&RAM\&RAM_CAP_1024\&$$
$$D2_SIZE_1024\&TA\&TA_ON_DCU\&RAM_FREQ_200$$

Listing 8.8: Part of Running Exemple ProVeLines output

```
1  .\provelines -check -cora -opt running_example.pml
2  ...
3  assertion failled : (assert(cons <= size);) at line... for products:
4     D2_size_1024 && D2_On_RAM && RAM_capacity_512 && ...
5        : of the trace ... :
6     System - Storage_RAM - Processor_DCU - ... - Data_D2 - Task_TA | QualityLoss
          - Cost - Time [ Features ]
7     \@00000000 : \@3 - \@35 - ... - \@72 - \@92 | 0 - 140 - 0
```

```
    [ NONE ]
    ...
    \@19418392 : \@22 - \@41 - ... - \@83 - \@92 | 0 - 180 - 0
    [ RAM && D2_size_1024 && D2_On_RAM && RAM_capacity_512 ]
    ...

assertion failed : (assert(!(time < 840);) at line... for products:
  D2_size_1024 && D2_On_ROM && D1_On_ROM && TB_On_GPU && TD_On_GPU && GPU &&
      GPU_frequency_100
    \@00000000 : \@3 - \@35 - ... - \@72 - \@92 | 0 - 140 - 0
    [ NONE ]
    ...
    \@10528892 : \@10 - \@52 - ... - \@84 - \@92 | 0 - 140 - 1024
    [D2_size_1024 && D2_On_ROM && D1_On_ROM && TB_On_GPU && TD_On_GPU && GPU &&
       GPU_frequency_100]
    ...

optimal product (global cost = 608) : D1_ON_ROM && RAM && RAM_Capacity_1024 &&
    D2_Size_1024 && TA && TA_On_DCU && RAM_frequency_200
  Storage_RAM - Processor_DCU - ... - Data_D2 - Task_TA | QualityLoss - Cost -
      Time [ Features ]
    \@00000000 : \@3 - \@35 - ... - \@72 - \@92 | 0 - 140 - 0
    [ NONE ]
    ...
    \@134189119: \@13 - \@41 - ... - \@89 - \@119 | 2 - 200 - 256
    [ RAM && D2_Size_1024 && RAM_Capacity_1024 && TA && TA_On_DCU &&
       RAM_frequency_200 ]
    ...
```

Listing 8.9: Part of Valid Design Space Variability in TVL

```
root DesignSpaceVariability{
   group allOf{
      ...
   }
   ...
   //invalid products constraints
   !(D2_size_1024 && D2_On_RAM && RAM_capacity_1024 && D1_On_RAM && P1_On_RAM);
   ...
}
```

In this chapter, we proposed an end-to-end implementation of our formal framework. The models and process of the framework have been developed *i)* to infer the system design space from a variable data-flow oriented application and a configurable non-programmable hardware platform and *ii)* assess the resulting design space efficiently against requirements. We implement the modeling part of the framework in a *Java* front-end.

In the next chapter, we use this implementation to model and assess qualitatively the mid-end Visteon use case. We also generate other mid-end instrument clusters in a realistic way. This dataset of models will be used to assess the scalability of our implementation.

Chapter 9

Industrial Evaluation

We realized three evaluations in collaboration with Visteon Electronics. Our first evaluation assesses the functional feasibility and aims to show the relevance of our approach for practitioners based on our partner's instrument cluster system (see Chap. 2). The second evaluation takes into account the non-functional concerns in order to assess also the non-functional satisfiability and optimality. The third one aims to obverse the scalability of our approach.

9.1 Case Study

With the assistance of an expert system engineer, a functional and a platform manager, we successfully modeled this system module's application and platform specifications using our framework implementation. The application and the platform were captured using our variable data-flow graph and variable resource graph model implementation. The running example presented throughout this thesis is a pedagogical simplification of this module from which we abstract functional and hardware properties that could be considered as details for non specialist[1].

9.2 Preliminary Experiment

In order to validate our framework implementation and the relevance of our variability-aware approach, we firstly assess only the functional feasibility (facets **FC-S** and **FC-B**) of the instrument cluster system from their variable specifications. In this first evaluation, we conduct the modeling and assessing of the module's system design space where only functional properties of the variable application and configurable platform have been modeled. Our framework generates the design space in FA formalism of *ProVeLines* to perform the behavioral analyzes while checking the designs that do not exhibit a behavior and structure that meet the functional requirements. Table 9.1 shows the time measurements of the use case analyzes while the different variability dimensions vary. The metrics we focus on are *i)* the number of states we have to explore *ii)* the time needed to explore every system design alternatives. In

[1]For example, the complete platform model integrates the memory burst access mechanisms, which help to get a more precise idea of run-time execution, memory access latency and bandwidth consumption. The real hardware pipelines of processor s are also more complex as they provide more concurrent graphical operations.

Variability	app. var.	plt. var.	Sys. var. (*)	S. expl. (*)	S./ var.	Time (in ms)(*)	ms/ var.
NONE	1	1	78 (1)	2453 (1)	31.48	27 (1)	0.34
data size	8	1	624 (8)	15254 (6.21)	35.97	201 (7.4)	0.32
plt.	1	24	424 (5.4)	5673 (2.31)	13.37	65 (2.4)	0.15
plt.&data size	8	24	3392 (43)	37435 (15)	11.03	602 (22)	0.17
data-path	2	1	150 (1.9)	4727 (1.92)	31.51	74 (2.7)	0.49
data-path&size	16	1	1200 (15)	29066 (11)	24.22	587 (21)	0.48
plt. mult. mem.	1	40	16408 (210)	134941 (55)	8.22	4010 (148)	0.24
plt. mult. proc.	1	80	2848 (36)	19625 (8)	6.89	337 (12)	0.11

Table 9.1: Result for Preliminary Experiment using ProVeLines Family-Based model checker.

a traditional product by product system design framework, the states and time required are generally correlated to the sum of the verification of every different design alternatives.

We propose to compute average states explored per variant and average time (in ms) required per variant metrics to indicate the scalability improvement of a variability-oriented system design framework. If these metrics do not drastically decrease as the number of variants increases, this experiment will invalidate the efficiency of our approach. On the contrary, if the states required by variant decrease while adding more variants, this will give some credits to such approaches as we expect that our variability-aware algorithm will exploit possible commonalities between system design alternatives. Benchmarks were run on a Dell Latitude 7400 with a 3,2 GHz Intel Core i5 processor and 16 GB of DDR3 RAM. To avoid random variations, we repeated each experiment ten times and computed the average.

Adding the data size variability dimension (row 2 *data size*) increases the explored states by a factor of 6.21 while the number of system variants increases by 8. In the end, the algorithm spends 201 ms (7.4 times more time) to verify 8 times more system variants encoded in a 6.21 bigger featured state space. Therefore, verification time per variant is similar. This shows that adding the data size variability to mapping one does not improve (or decrease) significantly the efficiency of the verification.

In the row 3 *plt.*, we only add the platform variability such as optional hardware component and variable memory capacities. In this experiment, we observe a substantial speed-up of the verification process. The number of variants increases by 5.4, whereas the verification time only increases by 2.4. Adding also the data size variability in addition to platform variability (row 4 *plt. & data size*) will slightly decrease the performance of the verification. Still, the variability-ware algorithm spend fewer times (factor of 22) in order to verify much more system variants (factor of 43) compared to the case were only mapping variability is assessed (row 1).

Concerning the variability of alternative data-path and tasks (row 5 *data-path*), this variability is less effective to verify than the other variability dimensions. Adding the data-path alternatives (the one processing by task A rather than task B) will increase both the system variant space and the explored state space by 1.92, while the verification will spend 2.74 more times.

Adding optional hardware memories (row 7) or processors (row 8) to simulate larger platforms drastically improves the efficiency of the verification process. For both cases, we observe that the required state space to explore per variant decreases while the number of variants increases. For instance, in the case 7, the explored

state space only increases by a factor of 55 to verify 210 times more system variants. Similarly, for the case 8, the state space increases by a factor 8 to verify 36 times more variant.

If only mapping variability is present (row 1), verifying a variant takes approximately 31 states. Whereas, in the case of larger platforms (row 7-8), this metrics falls to less than 10. This shows that fewer states' exploration is needed while increasing the number of platform configurations in our case study. However, our variability-aware algorithm behaves unfortunately like an iterative *product-by-product* model checker while adding application data-flow variability (row 5-6).

We finally observe that each variability dimension impact the verification metrics. We now try to synthesize these results. Firstly, the size of the state space the algorithm has to explore to verify the whole design space is the most impacting factor to the verification time. The most impressive gains have been in the cases where the required state space to explore was increasing slower than the variant space. This is showing that in cases 3-4, 7-8, the variability leads to a design space sharing a lot of commonalities between design alternatives. Whereas the verification data size and data paths variability (application variability) behaves like an iterative *product-by-product* model checker while adding application variants (row 2, 5-6).

9.3 Qualitative Experiment

In this second evaluation, we aim to find the optimal designs that both meet functional and non-functional requirements of the mid-end instrument cluster module (row 1 in Table. 9.2 and 9.3). The application and the platform's functional and non-functional properties were captured using, respectively, our variable data-flow graph and variable resource graph formal model implementation.

Some technical simplifications were also made as we aim to facilitate early design decisions: we do not consider data and parameters that only have a minor impact on the system run-time; we abstract away from data content and consider data sizes as the most influential factor for run-time performance; we do not model mechanisms like internal cache replacement policies, AXI[2] bus [Kyung et al., 2010] and internal backup communication buffers as these implementation details have no fundamental impacts and are handled by engineers later in the development process. These simplifications were deemed harmless by our industrial expert and did not endanger the correctness of our results. This results in an application model with two input data, 16 tasks and 32 flow processing, and in a platform model with one `ROM`, one `RAM`, one `DCU` and two `GPUs`.

In total, the variability yields 1,548,288 candidate designs. Our mapping strategy reduced this number to 1,878. By adding NF properties and requirements *rendering quality* ≥ 2 and *manufacturing cost* ≤ 280, we further diminish this number to 894. By checking the designs against the constrained state reachability property (*i.e.*, the end of execution is reached within 840 processing cycles) we obtain 279 variants that satisfy all requirements. Incorporating the cost function representing trade-offs defined with the expert (*i.e.*, with $\theta_{time} = 1$, $\theta_{m.cost} = 10$, $\theta_{r.qual.} = 100$) yielded 6 optimal variants with time = 642, manufacturing cost = 140 and rendering quality = 2.

Our framework firstly generates the design space in *Linear Priced Timed Automata* (LPTA) formalism in order to perform the behavioral analyzes using *UPPAAL-CORA*

[2] `https://developer.arm.com/documentation/dui0305/c/Glossary`

tool-chain. System expert validated that the six optima returned by *UPPAAL-CORA* are the most suitable designs and conform to the best designs that the company prototyped. One of the six designs is actually the final implementation of Visteon, while one of the six surprised the expert at it was not previously identified[3]. The quality attributes values also corresponded to what was observed in the concrete system implementations. There are slight differences regarding execution run-time (cycles) due to our modelling simplifications (around 15% on average); however, the numbers are close to reality, and the relative orders between promising variants are preserved.

Our framework secondly generates the design space in *fPromela* (FWA) in order to perform the behavioral analyzes using our model variability-aware model checker *ProVeLines-CORA*. These two separate verification processes provided the same results for all variants. This increases our confidence that the transformation of the design space into LPTA and FWA is consistent. Expert's confirmation was also needed, though, as mistakes may originate from the input models themselves. In the end, the experts validated that our Variability-Oriented approach helps make optimal design decisions that could provide significant gains at all stages of development.

9.4 Scalability Experiment

The third part of our evaluation focuses on the efficiency and the scalability of our approach in terms of verification time. This is essential, as the total number of variants can be high in real-world systems while each variant can exhibit a large state space. In addition to the instrument cluster case study (row 1), we constituted a dataset of models that were generated in a realistic way. To do so, our model generator relies on multivariate Gaussian distributions whose parameters were settled on the basis of the characteristics of our partner's industrial systems. Thereby, the characteristics of our generated data-flow and platform models are similar to real-world cases. For instance, the number of input data, the number of tasks and their average number of input/output data paths correspond to the topology of Visteon mid-end instrument clusters. Moreover, the variability of the data size, data-flow, memory capacities and optional platform component are built to correspond to real variability ratios in mid-end instrument cluster specifications.

Among all the models we generated, we selected 11 of them that appropriately summarize our findings. These models exhibit different state densities (*i.e.*, average number of system states per variant) and variability intensities (*i.e.*, numbers of system variants). We carried out three series of experiments presented hereunder. Table 9.1 provides the results, such that double borders separate the results of the different series of experiments. Benchmarks were run on a MacBook Pro 2014 with a 2,8 GHz Intel Core i7 processor and 16 GB of DDR3 RAM. To avoid random variations, we repeated each experiment ten times and computed the average.

9.4.1 Product-Based Versus Family-Based Verification

We firstly evaluate the efficiency of our method to verify that *all* variants satisfy the requirements, that is, we consider only the facets **FC-S**, **FC-B**, **NF-S** and **NF-B** of the problem to check the functional feasibility and the non functional satisfiability. To do so, we compare the run-time of our family-based verification algorithm with

[3]The system design was counter-intuitive.

		FCS + FCB + NFS + NFB					
		Product-based			**Family-based**		
case	system variants	time (in ms)	states explored	states/ variant	time (in ms)	states explored	states/ variant
Ins. Cl. #0	1,878	22.60	693,178	369	3.33	305,114	162
Gen. #1	32	0.80	49,952	1,561	0.48	45,681	1,427
Gen. #2	64	2.72	143.968	2,249	2.10	124,004	1,937
Gen. #3	243	22.11	743,823	3,061	80.54	660,348	2,717
Gen. #4	516	16.32	854,996	1,656	10.95	777,616	1,507
Gen. #5	1,152	55.48	2,599,393	2,256	40.17	2,217,593	1,924
Gen. #6	1,280	38.49	1,915,264	1,496	11.81	840,711	656
Gen. #7	2,187	144.48	5,283,792	2,416	120.83	4,593,249	2,104
Gen. #8	2,592	109.96	3,785,940	1,460	35.01	1,032,639	398
Gen. #9	3,168	151.87	7,201,320	2,273	69.14	3,926,395	1,239
Gen. #10	12,288	395.64	19,777,536	1,609	148.71	6,570,642	534
Gen. #11	34,560	1344.23	59,858,304	1,732	227.31	10,871,741	314

Table 9.2: Results for Product-Based Versus Family-Based Verifications.

an alternative, ProVeLines-CORA product-based algorithm that checks each variant separately.

The results are presented on Table 9.2. It depicts, for each approach and model, the time needed to check all variants and the total number of states explored by each algorithm. In the family-based case, fewer states are explored since one state common to multiple variants is explored only once. For case #0 which is the module of the studied instrument cluster, the family-based method outperforms the product-based one, reducing the verification time from 22.60 seconds to 3.33. The gain of the family-based method tends to show that the design alternatives of an industrial instrument cluster could exhibit a lot of commonalities at both structural and behavioral aspects.

The generated models allow us to observe that the benefits of the family-based method tends to grow with the number of variants. When this number is low (#1–3), our family-based algorithm either brings insignificant improvements (#1 and #2) or performs way worse than the product-based approach (#3). This could be explained by the fact that fewer states are shared across the design alternatives. The family-based version has to explore similar state space to assess the whole product line compared to product-by-product version. Moreover, there is also an inherent overhead induced by the verification of the design choices' consistency (SAT-solving) required while exploring states.

However, for models with higher variability (#4–11) we obtain reductions in verification time of minimum 16% (#6), most often substantial ones. The most impressive results are obtained for the case with the most variants (#11 – which is also the case where variants share the most commonalities): our algorithm is 5.9 times faster, achieving an absolute reduction of 1,116.92 seconds. We also observe that a higher state density often reduces the gain offered by our algorithm (*e.g.*, cases #5 and #7). To explain this, we analyzed the models and observed that a small number of variants exhibit a large state space, while all the others encompass far fewer states. The variants thus have fewer states in common. In the end, a family-based algorithm performs better as variants share more commonalities.

			NFO	
			P.V.L.-CORA	**UPP.-CORA**
case	density	variants	time	time
Ins. Cl. #0	360	1,878	2.42	5.64
Gen. #1	1,561	32	0.80	18.61
Gen. #2	2,250	64	1.27	39.59
Gen. #3	3,061	243	75.79	OoM.
Gen. #4	1,656	516	8.23	17.60
Gen. #5	2,256	1,152	29.77	OoM.
Gen. #6	1,496	1,280	8.65	19.89
Gen. #7	2,416	2,187	89.56	OoM.
Gen. #8	1,461	2,592	24.40	OoM.
Gen. #9	2,273	3,168	19.58	OoM.
Gen. #10	1,609	12,288	75.97	OoM.
Gen. #11	1,732	34,560	72.17	OoM.

Table 9.3: Results of ProVeLines-CORA and UPPAAL-CORA for designs optimizations.

9.4.2 Optimal Design in ProVeLines and UPPAAL

We secondly compare the efficiency of the two model checkers to solve the **NFO** problem facet, that is, we compare *UPPAAL-CORA* against our algorithm 3. The two model checking tools present clear differences in input syntax and semantics (LPTA vs. fPromela FWA). Moreover, our *ProVeLines-CORA* implementation performs family-based cost-optimal reachability in discrete-time models while *UPPAAL-CORA* was developed in order to perform cost-optimal reachability in continuous-time models in a product-based approach. Consequently, the generated design space models in *fPromela* in a behavioral product line are highly optimized compared to LPTA models of UPPAAL (an automaton capturing all designs).

Results are given in the center of Table 9.3. Like any *product-by-product* approach, *UPPAAL-CORA* suffers from every increase in the number of variants. It systematically performs worse than ProVeLines-CORA. Even worse, apart from cases #0–2, #4 and #5, *UPPAAL-CORA* consumes all the allowed memory and raises a fatal error. We assume two reasons behind this. First, *UPPAAL-CORA* does not implement partial-order reduction [Godefroid, 1996] and thus considers all possible interleaving between process actions while our *fPromela* models do not consider unnecessary interleaving[4]. Second, we use a model that encodes all variants' behaviour and thus accumulates all their state space. Yet, applying an alternative, product-based approach where each variant is turned into a separate model did not solve the problem and even led to slower times[5].

Through these experiments, we also observe that looking only for valid optima in *ProVeLines-CORA*, as opposed to verifying all variants, can yield significant reductions in execution time (up to 68% in case #11). More importantly, computing the optima without a family-based algorithm boils down to applying the product-based algorithm used in the first series of experiments in Table 9.2. In this regard, our Algorithm 3 exacerbates the benefits of a family-based algorithm. For the models with the most variants (#9–11), Algorithm 3 is 5 to 128 times faster than the product-based method.

[4]One way to circumvent this is to assign priorities over the automata to reduce the state space. However, in most cases, this will cause to miss better scheduling opportunities and will yield suboptimal results.

[5]We stopped the exploration of separated model at one hour of verification time

Interestingly, it seems to be way less affected by the number of variants or the design space density than the all-variant verifications. This may be affected by our *branch-and-bound* optimization that discard a execution (maybe shared by multiple variants) as soon as the cost exceed the *best* solution (see Alg. 3). Contrary to *product-by-product* methods, *bad* executions are explored once and then discarded for all variants.

9.5 Threats to Validity

In this section, we identify and discuss the main threats to the validity of our results. First, we discuss the internal threats.

We collaborated with Visteon Electronics for two years between 2016 and 2018. This close collaboration was in place during every step of the elaboration of this thesis with monthly meetings. We drove our research to assess a real mid-end instrument cluster. However, we aimed to generalize both the motivations and the contribution to early design of data-flow-oriented embedded systems. The selection of the case study, the fact we retro-engineered the case study, and the way we transferred our approach to system engineers are the main sources of internal threats to validity.

Second, the technical ground on which of the framework is built-on raises some external validity. First, model checking and SAT-solving are inherently exponential to problem size. Results are also highly dependent on the level of details of the system design space model. On the one hand, obtaining a highly detailed model tends to increase the assessing time. On the other hand, assessing highly abstracted model will give spurious, or irrelevant results. In our case study, we continuously help engineers to model the system specification at a level of details that allows to obtain the designs that effectively meet and optimize the requirements.

9.5.1 Internal validity

The main threat to internal validity is the selection of the case study. Although it was built, with other case studies, from specification and feedback meetings with system and platform experts from our partner company, it is a single case. Still, we choose it as being representative in terms of topology and complexity of the data flow, the platform and their variability within the company.

We helped the functional and platform experts to model the variable application and configurable platform specifications. As we also have some expertise in this field, we have helped experts to fit their system specification models in our model implementations. We also facilitated the modelling of the platform in a featured weighted automaton (see Fig. 7.2). Without our help, they might not have been able to do so. However, the results given by our framework were easily understandable, especially the structural aspect of the designs (*i.e.*, system variants). Moreover, we conducted our scalability experiments over large but simulated data sets that were also understandable and validated by our industry partner[6].

Finally, even if we helped functional and platform experts to model systems specifications, the results of our variability-aware model checking algorithm easily give insight to engineers in order to make appropriate design decisions at early stages of development.

[6]The data sets respect the structure, behaviour and variability of the potential applications and platforms of our industrial partner.

9.5.2 External validity

Several threats to external validity exist. More generally, we also expect the proposed approach to be applicable to other domains where data flows can be an appropriate modelling formalism, like image or signal processing in aerospace or underwater systems. On the platform side, the component-based representation seems generic enough to cover different forms of deployment architectures. Yet, we do not have any evidence of this possibility of generalization.

Satisfiability and optimality checking processes depend highly on the level of details of application and platform models. A not detailed enough platform model could lead to loss of relevance in the performance metrics and could even produce false optimal results. At the application level, the system will not reflect what is expected. From the feedback we got from our industrial partner, the level of details of the generated FWA is currently sufficient[7]. Still, in other contexts, some details may be added to have more precise performance metrics or hidden due to intellectual property management or confidentiality issues.

The most important threat to validity may be the modelling of the system specifications. This task could be time-consuming, especially when a deep level of detail is required to get relevant performance metrics. Still, we think that our variability-aware approach could ease the modelling of different application and platform variants using variability-aware models. As for applications, they are described with data-flows that are well mastered by *system experts* in our case study, but they have been built manually. Reverse engineering from existing application code could be envisaged to build data-flow or at least templates of them, but their quality would be directly related to the quality of organization of existing software.

9.5.3 Conclusion Validity

Besides these threats to validity, we now review the two main conclusions from our industrial evaluation.

First, the modelling of the structure, behaviour and variability aspects the system specifications could require many efforts. Such efforts could be a significant obstacle in industry adoption. Still, we argue that it will require less effort than implementing each design's software and hardware. Nevertheless, the expertise required to abstract the system specifications at an appropriate level of detail could be problematic for engineers without knowledge in variability modelling and model-based design methodology.

Mapping the system specifications to derive and transform the design space in a featured automaton formalism also needs knowledge in formal methods and system engineering techniques. Moreover, lack of knowledge at platform or even platform part could lead to irrelevant results at the assessing stage. For instance, some hardware knowledge may be unavailable due to confidentiality issues. Nevertheless, challenges to model-based system design adoption in industry is an entire research topic [Vogelsang et al., 2017, Wu et al., 2018, Amorim et al., 2019, Laing et al., 2020, Chami et al., 2018, Chami and Bruel, 2018, Huldt and Stenius, 2019]. Yet, investigate how the variability concern could be efficiently identified and modelled to optimize the design process of a particular company culture is an interesting perspective of our results.

[7] As it is at the level they use to specify the platform when selecting suppliers.

Chapter 10

Conclusion and Perspectives

A tremendous amount of variability can be observed in embedded systems, especially when they are built from highly variable specifications targeting diverse hardware platforms configurable in a fine-grained way. Such variability induces a combinatorial explosion in the number of design alternatives, often intractable for engineers. Furthermore, each system could also exhibit multiple scheduling executions. Consequently, even for system experts, finding suitable designs *w.r.t.* functional and non-functional requirements is very complex in addition to being time-consuming and error-prone.

In this thesis, we aimed to proactively assist system engineers in making appropriate design choices at early stages of development. To this aim, we proposed an end-to-end framework that deals efficiently with a variability-induced combinatorial explosion and reason efficiently at both structural and behavioural levels.

In this last chapter, we conclude this thesis with a review of the challenges and the results (see Sec. 10.1). We also share final remarks about the proposed approach in the Sec. 10.2. Next, we present the short term and long term perspectives opened by this work. These perspectives reflect the fundamental purpose of our framework by focusing both on theoretical and industrial aspects.

10.1 Review of the Challenges

Part I presents motivations and challenges raised by our class of problem. Assessing the design space *w.r.t.* functional and non-functional requirements in order to find suitable designs requires not only a way to deal efficiently with a variability-induced combinatorial explosion but also a way to reason at both structural and behavioural levels (facets **FC-S, FC-B, NF-S, NF-B, NFO**). The first challenge **C1** was a **modelling challenge** as it requires models able to *"Capture functional and non-functional high-level variable requirements and specifications of embedded systems that can vary at both application and platform levels"*. The second challenge **C2** was a **model transformation challenge** as it requires to derive the system design space from system specifications automatically. Manually deriving the design space is not a viable solution in industry. The third and last challenge **C3** was a **reasoning challenge**. It requires to evaluate simultaneously and efficiently the functional feasibility, the non-functional satisfiability, and optimality at both structural and behavioural levels of the design alternatives(facets **FC-S, FC-B, NF-S, NF-B, NFO**).

Part I ended with a review of the state of the art[1].It was established that embedded

[1]References can be found in chapter 3.

system design engineering methods find the best designs of systems for decades. On the other hand, approaches that manage variability have been successfully applied to product line engineering and variability-intensive system development. Variability-aware approaches manage this combinatorial explosion but are either limited to assess structural or behavioral concerns. Moreover, they do not directly capture embedded system specifications, which are traditionally used to derive the design space.

On the contrary, system design approaches capture system specifications in order to derive the design space. Moreover, they also assess both structural and behavioral aspects. Still, these approaches do not capture nor manage the variability dimensions present in our system specifications. Worse, when the design space size is exponential in terms of design variations, the behaviour of each design alternative is still assessed iteratively. We then concluded that these previous approaches do not well cover the three challenges.

Part II presented the original framework we proposed in order to meet all the challenges. We applied and extended the Y-Chart model pattern for the modelling stage and the behavioural product line paradigm for the assessing stage. This framework reduces the gap between system design engineering and product line methods. We proposed a variability-aware modelling method to capture variable system specifications in order to reach the first challenge **C1**. More precisely, we extend the data-flow-oriented application and concurrent component-based platform models with variability concerns. Hence, in addition, to capture structural and behavioural aspects, our extended models also capture variability present at both specifications levels. We are thus giving the means to capture, in a time-efficient way, every data-flow variants as a variable data-flow model and every platform configurations in a configurable platform model. Alternative data-flows and variable data sizes can be captured at the application level whereas optional hardware components, variable clock frequencies, and capacities can be modeled at the platform one.

To reach the second challenge **C2**, we proposed a mapping algorithm able to map our variable data-flow model over a configurable platform to infer the resulting design space automatically. This algorithm maps every application element over platform resources. Because application elements and platform resources could be optional, the algorithm generates consistency constraint to only capture valid system variants.

The resulting design space grows exponentially with the number of design variation points and scheduling opportunities. Hence, evaluating iteratively every design alternative *w.r.t.* requirements may hamper scalability. We proposed to reach the **reasoning** challenge **C3** by applying and extending behavioural product line formalism. As a behavioural product line encodes the behaviour of every product of the product line in a single automaton, it allows exploiting behavioural and structural commonalities between different but related system design alternatives. Common states are checked once for every design that reaches it. Common design decisions (data size, data flow, mapping, memory capacity, hardware resource selection, *etc.*) are checked once for every design that contains them.

Our proposed concept of featured weighted automaton extends featured automaton with non-functional concerns. Feature model was extended with *priced features* to reason on non-functional requirements that impact the structure of the design such as manufacturing cost **NF-S** while automaton was extended with *weighted featured transitions* to reason on the behavioral ones (*e.g.*, execution time). In the end, we proposed a variability-aware model checking algorithm that assesses simultaneously and efficiently all facets of the problem **FC-S, FC-B, NF-S, NF-B, NFO** by ex-

ploiting structural and behavioural commonalities between design alternatives.

- *Satisfying the functional requirements*, that is, is the design consistent in order to be implemented? (facet **FC-S**) and is the design execution satisfies the temporal properties? (facet **FC-B**).
- *Satisfying the non-functional requirements*, that is, can the design be implemented within this budget? (facet **NF-S**) and is the design executes in less than 30ms? (facet **NF-B**).
- *Optimizing multiple quality attributes*, that is, is the design reaches the *best* trade-off between both structural and behavioural quality attributes (facet **NFO**), such as minimizing the manufacturing cost and the execution time while maximizing the rendering quality.

In Part III, we reported on the implementation of our framework as a model-driven tool-chain. We implemented our variability-aware model checking algorithm on top of ProVeLines. Through our modelling tool-chain, we give the means to capture system specifications through *Java* APIs that are then mapped to derive the design space. The generated behavioural product line that captures the design space is in ProVeLines input formats. We also reported on the application of the proposed approach to a real-world industrial use case of an automotive instrument cluster.

The variable specification of the instrument cluster was properly modelled using our specifications models. Our framework implementation automatically inferred the correct design space and identified the optimal designs *w.r.t.* requirements, giving positive hints on potential applicability. Our experimental validation with large simulated datasets also shows sufficient scalability of the prototype implementation to design industrial-scale instrument clusters or systems with similar complexity. We also observed that our family-based verification process is more efficient than *product-by-product* ones.

10.2 Final Discussion

In this final discussion, we conclude about the efficiency of our variability-aware approach. At the modelling process, we claim that modelling a single variable specification is more efficient that model each variant separately. Still, it requires to be able to clearly identify explicit variability points for both functional and non-functional property at both structural and behavioural aspect. This activity was possible in our context, but the characterization of this generalization to other system domains requires future works (see Sec. 10.3).

At the assessing process, both theoretical and experimental results are aligned. The amount of states shared among the different system variant behaviours (commonalities) are correlated to the speed-up we observed in our *variability-aware* model checking algorithm compared to *product-by-product* ones. However, these commonalities highly depend on the use cases and their variability. As observed in Sec. 9, for the same kind of systems, application variability reduces the gain as it cause *early-splitting* [Classen et al., 2013a] of systems variants (and even behave - at worse - like the *product-by-product* version). In contrast, other variability dimensions lead to substantial verification speed-up.

The efficiency of our assessing process is correlated to the amount of the shared states among the variants of the different system. We claim that if all the states are shared by all systems variant, then our variability-ware methods will be linear in the number of variant as it will require to only assess a single state space once for every variants. On the contrary, if no states are shared, our *variability-aware* model-checking algorithm will perform similarly (or even worse[2]) that a *product-by-product* version.

To overcome this drawback, we present long term perspectives (see Sec. 10.3). The main idea is to merge *similar* states. The *similarity* notion could be diverse. For example, for variability-aware abstraction methods (see Sec. 10.3.3), an execution is similar to another one if they share the same (or equivalent) temporal property. For statistical-model-checking (see Sec. 10.3.3), previous executions can be used to predict with a high confidence ratio the feasibility of others.

All these optimization techniques aim to the same objective: to assess states or executions once for multiple products.

10.3 Perspectives

We now discuss the perspectives of this thesis, from short to long term. The short term perspectives are fundamental in order to validate our framework to other systems and industrial contexts, while long term perspectives aim to depict future directions in fundamental research.

10.3.1 Integration to Automotive Industry

Using our framework, we were able to model the variable system specifications of the case study from Visteon as a variable data-flow and a configurable hardware platform. The framework then derived the system design space and assessed it to identify the most suitable designs *w.r.t.* requirements. This case was part of a mid-end instrument cluster. Mid-end and low-end instrument clusters are the vast majority of automotive systems showing potential integration in industry. There are numerous standard HMI design and development tools for instrument cluster systems, such as Kanzi Studio[3], Altia[4], Crank Software[5], CGI Studio[6], EB Guide[7], Qt Automotive[8]. These tools offer visual models to capture the embedded HMI of the instrument cluster. The HMI elements are widgets, FX effects, UI Animations, *etc.* In order to customize the HMI for the different range of cars(*i.e.*, from low-end to high-end configurations), UI elements can be enabled or disabled by engineers. The choice of a particular API platform configuration, as well as every mapping of each UI element over a specific hardware resource, must be determined by engineers. In the end, these tools map the selected UI flow over the platform API chosen to generate and benchmark a C/C++/ASM system design implementation. However, these tools do not provide support to make appropriate design choices. Consequently, to optimize the HMI implementation on a

[2]Due to SAT-Solving overhead

[3]`https://www.rightware.com/kanzi`

[4]`https://www.altia.com/`

[5]`https://www.cranksoftware.com/industries/automotive`

[6]`https://cgistudio.at/`

[7]`https://www.elektrobit.com/ebguide/`

[8]`https://www.qt.io/qt-automotive-suite/`

particular configurable platform, engineers usually find suitable designs by prototyping various design candidates.

In these UI designer tools, a design is composed of an HMI variant and a API platform configuration. First of all, the design space may be derived from the HMI and API platform models previously defined by the engineers. Such input models are similar in some points to our application and platform models. Basically, the HMI widgets are data, while FX effects and UI Animations are tasks. The platform API can be seen as a declarative description of the platform's possible hardware functions and memory storage. This description may lack behavioural information[9]. The integration of hardware IP at behavioural level may allow capturing the design space as a variable automaton to apply our technique. Still, this close collaboration between instrument cluster HMI design tools and hardware providers such as NXP[10], Renesas[11], Infineon[12], or STMicroelectronis[13] could raise confidentiality issues.

10.3.2 Applications to Other Systems

Even if our framework can efficiently model and assess the Visteon case study, the need to manage other system design engineering problems in different contexts is fundamental to further investigate our framework's applicability. Our framework captures data-flow oriented applications and non-programmable platforms and uses variability-model checking of temporal and quantitative properties to assess embedded and concurrent system designs. We drive our framework development not only to handle the specific Visteon case but also to handle the whole considered class of problems.

In our vision, the modelling part of the framework can handle other application and or platform specification models to capture different systems. The mapping algorithm may also adapt depending on the models to map. The assessing part of the framework is relying on automata theory. This theory is highly-expressive and has been extensively applied to a vast amount of systems. However, such formalisms either suffer from scalability or undecidability issues which can limit the verification.

We plan to apply our framework to the design of systems such as aerospace, underwater, medical systems, Internet of Things, grid computing, and even production plans. These concurrent systems are often highly constrained in terms of hardware resources and capacities and aim at optimizing their usage. They also may need formal verification of temporal properties to identify potential errors at early stage of development.

Our goal in these applications to other domains would be to extend the framework to capture multiple system specifications and derive and generate the resulting design space in our behavioural product line formalism. Finally, extensions should also be conducted in order to assess new system properties. For example, aerospace systems and underwater systems must take into account their environment. Environments are usually modelled as a stochastic automaton communicating with the embedded system. Finding the most suitable system design alternatives in certain environmental stochastic conditions requires assessing stochastic temporal properties.

[9]The level of detail of the platform behaviour strongly impacts the relevance of the design evaluations.

[10]`https://www.nxp.com`

[11]`https://www.renesas.com`

[12]`https://www.infineon.com`

[13]`https://www.st.com`

10.3.3 Variability-Aware Statistical Model-Checking

The experiments we conducted show that a massive amount of states is shared by design alternatives. Our variability-aware model checking algorithm allows exploring a state once for every system variant (or product) that could produce it. Hence, this optimization speeds-up remarkably the verification process. Nevertheless, the state space explosion problem could rise for designing larger systems such as high-end instrument clusters or other embedded systems. In this case, we will lack to assist engineers in making appropriate design decisions.

Statistical model checking (SMC) [Legay et al., 2010] is an approximate technique that, rather than covering every execution of a system, verifies a small sample of executions. Results are then used to statistically compute the probability that the system satisfies the property with some degree of confidence. Simulation-based methods are known to be far less memory time-consuming than exhaustive ones and are sometimes the only viable option. Over the past years, SMC has been used to, *e.g.* assess the absence of errors in various areas from aeronautic to systems biology; measure cost average and energy consumption for complex applications such as nanosatellites [Heidt et al., 2000]; detect rare bugs in concurrent systems [Jegourel et al., 2013, Larsen and Legay, 2018].

Recent works apply variability-aware statistical model-checking [Cordy et al., 2020] to well known academic configurable systems. They identified all system configurations that do not fulfil functional requirements with a speed-up of 924% on average compared to the exhaustive variability-aware model checking method. This shows that verifying a *sampled* execution for multiple designs can speed up the verification process, increase the confidence that a particular system variant satisfies a requirement, or reduce the number of sampled execution to reach a specific confidence level.

10.3.4 Abstractions of Behavioral Product Line

Another direction to improve the efficiency of our work is through abstractions [Cordy and Legay, 2019, Dimovski and Wasowski, 2017, Dimovski et al., 2019]. A first abstraction consists of reducing the state space by merging similar states. The second reduces the variant space by joining features expressions. These abstractions reduce the model's size to optimize the verification of the behavioural product line that encodes our design space.

These reductions often come at the cost of inaccuracies in the model. Therefore, a design can be *spurious* as it exists within only the design space's abstracion. The spurious design exhibits a system variant that is not in the variant space or execution that is not in the featured state space, or both. In this case, the abstraction must be refined to eliminate these false positives. Common methods achieve this refinement by *Counterexample Guided Abstraction Refinement* [Cordy et al., 2014] (CEGAR).

Another well-known technique is symbolic model checking [Classen et al., 2011c]. Rather than enumerating every state, states are automatically grouped as long as they share similar properties. Different algorithms commonly used in symbolic model checking rely on a fixed-point computation implemented on symbolic data structures and breadth-first searches. This technique highly differs from our depth-first search that explores every state individually. However, extending these techniques seems to be a natural perspective.

10.3.5 Multi-Level Design Space Exploration

Modern System Design frameworks generally use multiple system models at different level of details to improve the design space exploration method [Pimentel et al., 2006, Bakshi et al., 2001]. The exploration starts typically with the most analytical model to quickly prune the design space by removing *obviously-bad* design space regions. A model of computation can then be used to guarantee or optimize the design behaviour formally. Finally, promising designs are simulated or prototyped to have a precise idea of the design's quality and performance.

Statistical learning techniques [Valov et al., 2017, Guo et al., 2017, Zhang et al., 2015, Siegmund et al., 2015, Jamshidi et al., 2017a, Jamshidi et al., 2017b, Kolesnikov et al., 2019, kal,] have been applied to identify the system configurations that exhibit the best performance. They can derive performance influence of the combinations of the features. A small sample of configurations performances can be used to predict the performance of the rest of the configurations. However, the use cases are not concurrent systems.

Recently, they propose a white-box approach [Velez et al., 2020] using the variable source code to improve the quality of the predictions. Using such techniques, we may learn from prototypes or simulations runs to predict outcomes for unexplored design alternatives. We consider applying such models in a black-box way (the model predicts the end value of the quality attributes) and a white-box way (the model predicts execution traces), as the latter is more likely to provide accurate predictions and guide the design space exploration.

The global contribution of this thesis is the elaboration of an end-to-end framework to tackle an industrial problem but also to challenge existing methods. Given variable applications and configurable platforms, we contribute to modelling and assessing the functional, non-functional and even optimality of the different designs alternatives. The variability presents in system specifications dramatically increases the size of the design space. Still, we were able to optimize the verification of the whole design space by exploiting structural and behavioural commonalities between different but related designs. We hope that this thesis results encourage the development of further methods to manage variability in system and software engineering.

Bibliography

[kal,]

[Abdelzaher et al., 2008] Abdelzaher, T., Diao, Y., Hellerstein, J. L., Lu, C., and Zhu, X. (2008). Introduction to control theory and its application to computing systems. In *Performance Modeling and Engineering*, pages 185–215. Springer.

[Aho, 2012] Aho, A. V. (2012). Computation and computational thinking. *The Computer Journal*, 55(7):832–835.

[Alur and Henzinger, 1994] Alur, R. and Henzinger, T. A. (1994). A really temporal logic. *Journal of the ACM (JACM)*, 41(1):181–203.

[Alur et al., 2001] Alur, R., La Torre, S., and Pappas, G. J. (2001). Optimal paths in weighted timed automata. In *International Workshop on Hybrid Systems: Computation and Control*, pages 49–62. Springer.

[Amorim et al., 2019] Amorim, T., Vogelsang, A., Pudlitz, F., Gersing, P., and Philipps, J. (2019). Strategies and best practices for model-based systems engineering adoption in embedded systems industry. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 203–212. IEEE.

[Apel et al., 2013] Apel, S., von Rhein, A., Wendler, P., Größlinger, A., and Beyer, D. (2013). Strategies for product-line verification: case studies and experiments. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 482–491. IEEE.

[Asirelli et al., 2011] Asirelli, P., ter Beek, M. H., Gnesi, S., and Fantechi, A. (2011). Formal description of variability in product families. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 130–139. IEEE.

[Attarzadeh-Niaki and Sander, 2017] Attarzadeh-Niaki, S.-H. and Sander, I. (2017). Automatic construction of models for analytic system-level design space exploration problems. In *Proceedings of the Conference on Design, Automation & Test in Europe*, pages 670–673. European Design and Automation Association.

[Bakshi et al., 2001] Bakshi, A., Prasanna, V. K., and Ledeczi, A. (2001). Milan: A model based integrated simulation framework for design of embedded systems. *ACM Sigplan Notices*, 36(8):82–93.

[Balarin, 1997] Balarin, F. (1997). *Hardware-software co-design of embedded systems: the POLIS approach*. Springer Science & Business Media.

[Basten et al., 2010] Basten, T., Van Benthum, E., Geilen, M., Hendriks, M., Houben, F., Igna, G., Reckers, F., De Smet, S., Somers, L., Teeselink, E., et al. (2010). Model-driven design-space exploration for embedded systems: The octopus toolset. *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 90–105.

[Batory, 2005] Batory, D. S. (2005). Feature Models, Grammars, and Propositional Formulas. In *SPLC'05*, volume 3714 of *LNCS*, pages 7–20. Springer.

[Bauer et al., 2013] Bauer, S. S., Fahrenberg, U., Juhl, L., Larsen, K. G., Legay, A., and Thrane, C. (2013). Weighted modal transition systems. *Formal Methods in System Design*, 42(2):193–220.

[Behrmann et al., 2006] Behrmann, G., David, A., and Larsen, K. G. (2006). A tutorial on uppaal 4.0. *Department of computer science, Aalborg university*.

[Behrmann et al., 2001] Behrmann, G., Fehnker, A., Hune, T., Larsen, K. G., Pettersson, P., Romijn, J., and Vaandrager, F. W. (2001). Minimum-cost reachability for priced timed automata. In *Proceedings of HSCC '01*, pages 147–161.

[Behrmann et al., 2005] Behrmann, G., Larsen, K. G., and Rasmussen, J. I. (2005). Optimal scheduling using priced timed automata. *ACM SIGMETRICS Performance Evaluation Review*, 32(4):34–40.

[Benavides et al., 2007] Benavides, D., Segura, S., Trinidad, P., and Cortés, A. R. (2007). Fama: Tooling a framework for the automated analysis of feature models. In *VaMoS'07*, pages 129–134.

[Bengtsson et al., 1996] Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., and Yi, W. (1996). Uppaal-a tool suite for automatic verification of real-time systems. *Hybrid Systems III*, pages 232–243.

[Biere et al., 1999] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. (1999). Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer.

[Bokhari, 1981] Bokhari, S. H. (1981). On the mapping problem. *IEEE Transactions on Computers*, (3):207–214.

[Botterweck et al., 2010] Botterweck, G., Polzer, A., and Kowalewski, S. (2010). Variability and evolution in model-based engineering of embedded systems.

[Boudjadar et al., 2013] Boudjadar, A., David, A., Kim, J. H., Larsen, K. G., Mikučionis, M., Nyman, U., and Skou, A. (2013). Hierarchical scheduling framework based on compositional analysis using uppaal. In *International Workshop on Formal Aspects of Component Software*, pages 61–78. Springer.

[Bouyer et al., 2007] Bouyer, P., Larsen, K. G., and Markey, N. (2007). Model-checking one-clock priced timed automata. In *International Conference on Foundations of Software Science and Computational Structures*, pages 108–122. Springer.

[Brihaye et al., 2004] Brihaye, T., Bruyere, V., and Raskin, J.-F. (2004). Model-checking for weighted timed automata. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 277–292. Springer.

[Brink et al., 2014] Brink, C., Kamsties, E., Peters, M., and Sachweh, S. (2014). On hardware variability and the relation to software variability. In *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, pages 352–355. IEEE.

[Broy, 2006] Broy, M. (2006). Challenges in automotive software engineering. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 33–42.

[Broy et al., 2011] Broy, M., Chakraborty, S., Goswami, D., Ramesh, S., Satpathy, M., Resmerita, S., and Pree, W. (2011). Cross-layer analysis, testing and verification of automotive control software. In *Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*, pages 263–272.

[Brugali and Hochgeschwender, 2017] Brugali, D. and Hochgeschwender, N. (2017). Managing the functional variability of robotic perception systems. In *2017 First IEEE International Conference on Robotic Computing (IRC)*, pages 277–283. IEEE.

[Chami et al., 2018] Chami, M., Aleksandraviciene, A., Morkevicius, A., and Bruel, J.-M. (2018). Towards solving mbse adoption challenges: the d3 mbse adoption toolbox. In *INCOSE International Symposium*, volume 28, pages 1463–1477. Wiley Online Library.

[Chami and Bruel, 2018] Chami, M. and Bruel, J.-M. (2018). A survey on mbse adoption challenges.

[Charette, 2009] Charette, R. N. (2009). This car runs on code. *IEEE spectrum*, 46(3):3.

[Chechik et al., 2003] Chechik, M., Devereux, B., Easterbrook, S. M., and Gurfinkel, A. (2003). Multi-valued symbolic model-checking. *ACM Trans. Softw. Eng. Methodol.*, 12(4):371–408.

[Clarke et al., 1999] Clarke, E., Grumberg, O., and Peled, D. (1999). *Model Checking*. MIT Press.

[Clarke et al., 1994] Clarke, E. M., Grumberg, O., and Long, D. E. (1994). Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542.

[Classen, 2011] Classen, A. (2011). *Modelling and Model Checking Variability-Intensive Systems (Ph. D. dissertation)*. PhD thesis, University of Namur (FUNDP).

[Classen et al., 2011a] Classen, A., Boucher, Q., and Heymans, P. (2011a). A text-based approach to feature modelling: Syntax and semantics of tvl. *Science of Computer Programming*, 76(12):1130–1143.

[Classen et al., 2013a] Classen, A., Cordy, M., Schobbens, P.-Y., Heymans, P., Legay, A., and cois Raskin, J.-F. (2013a). Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *Transactions on Software Engineering*, pages 1069–1089.

[Classen et al., 2013b] Classen, A., Cordy, M., Schobbens, P.-Y., Heymans, P., Legay, A., and Raskin, J.-F. (2013b). Featured transition systems: Foundations for verifying variability-intensive systems and their application to ltl model checking. *IEEE Transactions on Software Engineering*, 39(8):1069–1089.

[Classen et al., 2011b] Classen, A. et al. (2011b). *Modelling and model checking variability-intensive systems*. PhD thesis, Ph. D. dissertation.

[Classen et al., 2011c] Classen, A., Heymans, P., Schobbens, P.-Y., and Legay, A. (2011c). Symbolic model checking of software product lines. In *ICSE'11*, pages 321–330. ACM.

[Classen et al., 2010a] Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., and Raskin, J.-F. (2010a). Model checking lots of systems: efficient verification of temporal properties in software product lines. In *ICSE'10*, pages 335–344. ACM.

[Classen et al., 2010b] Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., and Raskin, J.-F. (2010b). Model checking lots of systems: efficient verification of temporal properties in software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 335–344. ACM.

[Cledou et al., 2017] Cledou, G., Proença, J., and Soares Barbosa, L. (2017). Composing families of timed automata. In Dastani, M. and Sirjani, M., editors, *Fundamentals of Software Engineering*, pages 51–66, Cham. Springer International Publishing.

[Clements, 2001] Clements, P. (2001). Software product lines. *Practices and patterns*.

[Cordy, 2014] Cordy, M. (2014). *Model Checking for the Masses*. PhD thesis, University of Namur.

[Cordy et al., 2013a] Cordy, M., Classen, A., Heymans, P., Schobbens, P.-Y., and Legay, A. (2013a). Provelines: a product line of verifiers for software product lines. In *Proceedings of the 17th International Software Product Line Conference co-located workshops*, pages 141–146. ACM.

[Cordy et al., 2019] Cordy, M., Devroey, X., Legay, A., Perrouin, G., Classen, A., Heymans, P., Schobbens, P.-Y., and Raskin, J.-F. (2019). A decade of featured transition systems. In *From Software Engineering to Formal Methods and Tools, and Back*, pages 285–312. Springer.

[Cordy et al., 2014] Cordy, M., Heymans, P., Legay, A., Schobbens, P.-Y., Dawagne, B., and Leucker, M. (2014). Counterexample guided abstraction refinement of product-line behavioural models. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 190–201.

[Cordy and Legay, 2019] Cordy, M. and Legay, A. (2019). Verification and abstraction of real-time variability-intensive systems. *International Journal on Software Tools for Technology Transfer*, 21(6):635–649.

[Cordy et al., 2020] Cordy, M., Papadakis, M., and Legay, A. (2020). Statistical model checking for variability-intensive systems. In *FASE*, pages 294–314.

[Cordy et al., 2012] Cordy, M., Schobbens, P.-Y., Heymans, P., and Legay, A. (2012). Behavioural modelling and verification of real-time software product lines. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 66–75. ACM.

[Cordy et al., 2013b] Cordy, M., Schobbens, P.-Y., Heymans, P., and Legay, A. (2013b). Beyond boolean product-line model checking: Dealing with feature attributes and multi-features. In *ICSE'13*, pages 472–481. IEEE.

[Czarnecki and Antkiewicz, 2005] Czarnecki, K. and Antkiewicz, M. (2005). Mapping features to models: A template approach based on superimposed variants. In *GPCE'05*, pages 422–437.

[Dalsgaard et al., 2010] Dalsgaard, A. E., Olesen, M. C., Toft, M., Hansen, R. R., and Larsen, K. G. (2010). Metamoc: Modular execution time analysis using model checking. In *OASIcs-OpenAccess Series in Informatics*, volume 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[Davare et al., 2007] Davare, A., Densmore, D., Meyerowitz, T., Pinto, A., Sangiovanni-Vincentelli, A., Yang, G., Zeng, H., and Zhu, Q. (2007). A next-generation design framework for platform-based design. In *Conference on using hardware design and verification languages (DVCon)*, volume 152.

[Davis, 1989] Davis, S. M. (1989). From "future perfect": Mass customizing. *Planning review*, 17(2):16–21.

[De Michell and Gupta, 1997] De Michell, G. and Gupta, R. K. (1997). Hardware/-software co-design. *Proceedings of the IEEE*, 85(3):349–365.

[Densmore and Passerone, 2006] Densmore, D. and Passerone, R. (2006). A platform-based taxonomy for esl design. *IEEE Design & Test of Computers*, 23(5):359–374.

[Dimovski et al., 2019] Dimovski, A. S., Brabrand, C., and Wasowski, A. (2019). Finding suitable variability abstractions for lifted analysis. *Formal Aspects of Computing*, 31(2):231–259.

[Dimovski and Wasowski, 2017] Dimovski, A. S. and Wasowski, A. (2017). Variability-specific abstraction refinement for family-based model checking. In *International Conference on Fundamental Approaches to Software Engineering*, pages 406–423. Springer.

[Eddy, 2015] Eddy, N. (2015). Gartner: 21 billion iot devices to invade by 2020. *InformationWeek, Nov*, 10.

[Edwards et al., 1997] Edwards, S., Lavagno, L., Lee, E. A., and Sangiovanni-Vincentelli, A. (1997). Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390.

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

[Fahrenberg and Legay, 2017a] Fahrenberg, U. and Legay, A. (2017a). Featured weighted automata. In *Proceedings of the 5th International FME Workshop on Formal Methods in Software Engineering*, pages 51–57. IEEE Press.

[Fahrenberg and Legay, 2017b] Fahrenberg, U. and Legay, A. (2017b). Featured weighted automata. In *5th IEEE/ACM International FME Workshop on Formal Methods in Software Engineering, FormaliSE@ICSE 2017, Buenos Aires, Argentina, May 27, 2017*, pages 51–57.

[Felfernig et al., 2014] Felfernig, A., Hotz, L., Bagley, C., and Tiihonen, J. (2014). *Knowledge-based Configuration: From Research to Business Cases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1 edition.

[Fischer et al., 2014] Fischer, T., Kollner, C., Hardle, M., and Muller-Glaser, K. D. (2014). Product line development for modular fpga-based embedded systems. In *Rapid System Prototyping (RSP), 2014 25th IEEE International Symposium on*, pages 9–15. IEEE.

[Fleischanderl et al., 1998] Fleischanderl, G., Friedrich, G. E., Haselböck, A., Schreiner, H., and Stumptner, M. (1998). Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems*, 13(4):59–68.

[Freuder, 1997] Freuder, E. C. (1997). In pursuit of the holy grail. *Constraints*, 2(1):57–61.

[Gheorghita et al., 2008] Gheorghita, S. V., Basten, T., and Corporaal, H. (2008). Application scenarios in streaming-oriented embedded-system design. *IEEE Design & Test of Computers*, 25(6).

[Godefroid, 1996] Godefroid, P. (1996). *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer.

[Graf et al., 2014] Graf, S., Glaß, M., Teich, J., and Lauer, C. (2014). Multi-variant-based design space exploration for automotive embedded systems. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE.

[Graf et al., 2013] Graf, S., Glaß, M., Wintermann, D., Teich, J., and Lauer, C. (2013). Ivam: Implicit variant modeling and management for automotive embedded systems. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*, pages 1–10. IEEE.

[Graf et al., 2015] Graf, S., Reinhart, S., Glaß, M., Teich, J., and Platte, D. (2015). Robust design of e/e architecture component platforms. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6. IEEE.

[Gries, 2004] Gries, M. (2004). Methods for evaluating and covering the design space during early design development. *Integration, the VLSI journal*, 38(2):131–183.

[Grun et al., 1998] Grun, P., Halambi, A., Khare, A., Ganesh, V., Dutt, N., and Nicolau, A. (1998). Expression: An adl for system level design exploration. Technical report, Citeseer.

[Guo et al., 2017] Guo, J., Yang, D., Siegmund, N., Apel, S., Sarkar, A., Valov, P., Czarnecki, K., Wasowski, A., and Yu, H. (2017). Data-efficient performance learning for configurable systems. *Empirical Software Engineering*, pages 1–42.

[Heidt et al., 2000] Heidt, H., Puig-Suari, J., Moore, A., Nakasuka, S., and Twiggs, R. (2000). Cubesat: A new generation of picosatellite for education and industry low-cost space experimentation.

[Henard et al., 2015] Henard, C., Papadakis, M., Harman, M., and Le Traon, Y. (2015). Combining multi-objective search and constraint solving for configuring large software product lines. In *Proceedings of ICSE '15*, pages 517–528. IEEE Press.

[Henzinger and Sifakis, 2006] Henzinger, T. A. and Sifakis, J. (2006). The embedded systems design challenge. In *International Symposium on Formal Methods*, pages 1–15. Springer.

[Henzinger and Sifakis, 2007] Henzinger, T. A. and Sifakis, J. (2007). The discipline of embedded systems design. *Computer*, 40(10):32–40.

[Herrera et al., 2014] Herrera, F., Posadas, H., Peñil, P., Villar, E., Ferrero, F., Valencia, R., and Palermo, G. (2014). The complex methodology for uml/marte modeling and design space exploration of embedded systems. *Journal of Systems Architecture*, 60(1):55–78.

[Holzmann, 2004] Holzmann, G. J. (2004). *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley.

[Hopcroft et al., 2001] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65.

[Hu et al., 2013] Hu, X.-B., Wang, M., and Di Paolo, E. (2013). Calculating complete and exact pareto front for multiobjective optimization: a new deterministic approach for discrete problems. *IEEE Transactions on cybernetics*, 43(3):1088–1101.

[Hubaux et al., 2012] Hubaux, A., Xiong, Y., and Czarnecki, K. (2012). A user survey of configuration challenges in linux and ecos. In *VaMoS '12*, pages 149–155. ACM.

[Huldt and Stenius, 2019] Huldt, T. and Stenius, I. (2019). State-of-practice survey of model-based systems engineering. *Systems Engineering*, 22(2):134–145.

[Jaghoori et al., 2008] Jaghoori, M. M., Longuet, D., De Boer, F. S., and Chothia, T. (2008). Schedulability and compatibility of real time asynchronous objects. In *Real-Time Systems Symposium, 2008*, pages 70–79. IEEE.

[Jamshidi et al., 2017a] Jamshidi, P., Siegmund, N., Velez, M., Kästner, C., Patel, A., and Agarwal, Y. (2017a). Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 497–508, Piscataway, NJ, USA. IEEE Press.

[Jamshidi et al., 2017b] Jamshidi, P., Velez, M., Kästner, C., Siegmund, N., and Kawthekar, P. (2017b). Transfer learning for improving model predictions in highly configurable software. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2017 IEEE/ACM 12th International Symposium on*, pages 31–41. IEEE.

[Jegourel et al., 2013] Jegourel, C., Legay, A., and Sedwards, S. (2013). Importance splitting for statistical model checking rare properties. In *International Conference on Computer Aided Verification*, pages 576–591. Springer.

[Kavi et al., 1986] Kavi, K. M., Buckles, B. P., and Bhat, U. N. (1986). A formal definition of data flow graph models. *IEEE Transactions on computers*, (11):940–948.

[Khalilov et al., 2016] Khalilov, E., Ross, J., Antkiewicz, M., Völter, M., and Czarnecki, K. (2016). Modeling and optimizing automotive electric/electronic (e/e) architectures: Towards making clafer accessible to practitioners. In *International Symposium on Leveraging Applications of Formal Methods*, pages 447–464. Springer.

[Kim et al., 2016] Kim, J. H., Legay, A., Traonouez, L.-M., Acher, M., and Kang, S. (2016). A formal modeling and analysis framework for software product line of preemptive real-time systems. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1562–1565. ACM.

[Kolesnikov et al., 2019] Kolesnikov, S., Siegmund, N., Kästner, C., Grebhahn, A., and Apel, S. (2019). Tradeoffs in modeling performance of highly configurable software systems. *Software & Systems Modeling*, 18(3):2265–2283.

[Krzysztof and Eisenecker, 2000] Krzysztof, C. and Eisenecker, U. W. (2000). *Generative Programming: Methods, Tools and Applications*. Addison-Wesley.

[Kugele and Pucea, 2014] Kugele, S. and Pucea, G. (2014). Model-based optimization of automotive e/e-architectures. In *Proceedings of CSTVA 2014*, pages 18–29. ACM.

[Kyung et al., 2010] Kyung, H.-m., Park, G.-h., Kwak, J. W., Kim, T.-j., and Park, S.-B. (2010). Design and implementation of performance analysis unit (pau) for axi-based multi-core system on chip (soc). *microprocessors and microsystems*, 34(2-4):102–116.

[Laing et al., 2020] Laing, C., David, P., Blanco, E., and Dorel, X. (2020). Questioning integration of verification in model-based systems engineering: an industrial perspective? *Computers in Industry*, 114:103163.

[Larsen et al., 2001] Larsen, K., Behrmann, G., Brinksma, E., Fehnker, A., Hune, T., Pettersson, P., and Romijn, J. (2001). As cheap as possible: effcient cost-optimal reachability for priced timed automata. In *International Conference on Computer Aided Verification*, pages 493–505. Springer.

[Larsen and Legay, 2018] Larsen, K. G. and Legay, A. (2018). Statistical model checking the 2018 edition! In *International Symposium on Leveraging Applications of Formal Methods*, pages 261–270. Springer.

[Larsen et al., 2007] Larsen, K. G., Nyman, U., and Wasowski, A. (2007). Modal I/O automata for interface and product line theories. In *ESOP*, pages 64–79.

[Lee and Sangiovanni-Vincentelli, 1998] Lee, E. A. and Sangiovanni-Vincentelli, A. (1998). A framework for comparing models of computation. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 17(12):1217–1229.

[Legay et al., 2010] Legay, A., Delahaye, B., and Bensalem, S. (2010). Statistical model checking: An overview. In *International conference on runtime verification*, pages 122–135. Springer.

[Liebig et al., 2009] Liebig, J., Apel, S., Lengauer, C., and Leich, T. (2009). Robby-dbms: a case study on hardware/software product line engineering. In *Proceedings of the First International Workshop on Feature-Oriented Software Development*, pages 63–68. ACM.

[Luthmann et al., 2017] Luthmann, L., Stephan, A., Bürdek, J., and Lochau, M. (2017). Modeling and testing product lines with unbounded parametric real-time constraints. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A*, SPLC '17, pages 104–113, New York, NY, USA. ACM.

[Macaulay, 2012] Macaulay, L. A. (2012). *Requirements engineering*. Springer Science & Business Media.

[Mendonca et al., 2009] Mendonca, M., Branco, M., and Cowan, D. (2009). S.p.l.o.t.: Software product lines online tools. In *Proceedings of OOPSLA '09*, pages 761–762, New York, NY, USA. ACM.

[Murashkin et al., 2013] Murashkin, A., Antkiewicz, M., Rayside, D., and Czarnecki, K. (2013). Visualization and exploration of optimal variants in product line engineering. In *Proceedings of the 17th International Software Product Line Conference*, pages 111–115. ACM.

[Muschevici et al., 2010] Muschevici, R., Clarke, D., and Proença, J. (2010). Feature petri nets. In *Proceedings of the 14th International Software Product Line Conference (SPLC 2010)*, volume 2. Lancaster University; Lancaster, United Kingdom.

[Negrean et al., 2013] Negrean, M., Klawitter, S., and Ernst, R. (2013). Timing analysis of multi-mode applications on autosar conform multi-core systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 302–307. EDA Consortium.

[Noir et al., 2016] Noir, J. L., Madelénat, S., Gailliard, G., Labreuche, C., Acher, M., Barais, O., and Constant, O. (2016). A decision-making process for exploring architectural variants in systems engineering. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 277–286. ACM.

[Nunes et al., 2012] Nunes, V., Fernandes, P., Alves, V., and Rodrigues, G. N. (2012). Variability management of reliability models in software product lines: An expressiveness and scalability analysis. In *SBCARS '12*, pages 51–60.

[Olaechea et al., 2018] Olaechea, R., Atlee, J., Legay, A., and Fahrenberg, U. (2018). Trace checking for dynamic software product lines. In *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '18, pages 69–75. ACM.

[Olaechea et al., 2016] Olaechea, R., Fahrenberg, U., Atlee, J. M., and Legay, A. (2016). Long-term average cost in featured transition systems. In *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, pages 109–118, New York, NY, USA. ACM.

[Olaechea et al., 2014] Olaechea, R., Rayside, D., Guo, J., and Czarnecki, K. (2014). Comparison of exact and approximate multi-objective optimization for software product lines. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 92–101. ACM.

[Olaechea et al., 2012] Olaechea, R., Stewart, S., Czarnecki, K., and Rayside, D. (2012). Modelling and multi-objective optimization of quality attributes in variability-rich software. In *Proceedings of the Fourth International Workshop on Nonfunctional System Properties in Domain Specific Modeling Languages*, NFPinDSML '12, pages 2:1–2:6. ACM.

[Palermo et al., 2008] Palermo, G., Silvano, C., and Zaccaria, V. (2008). Robust optimization of soc architectures: A multi-scenario approach. In *Embedded Systems for Real-Time Multimedia, 2008. ESTImedia 2008. IEEE/ACM/IFIP Workshop on*, pages 7–12. IEEE.

[Palermo et al., 2009] Palermo, G., Silvano, C., and Zaccaria, V. (2009). Variability-aware robust design space exploration of chip multiprocessor architectures. In *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, pages 323–328. IEEE.

[Passos et al., 2016] Passos, L., Teixeira, L., Dintzner, N., Apel, S., Wasowski, A., Czarnecki, K., Borba, P., and Guo, J. (2016). Coevolution of variability models and related software artifacts. *Empirical Software Engineering*, 21(4):1744–1793.

[Peled, 1998] Peled, D. (1998). Ten years of partial order reduction. In *International Conference on Computer Aided Verification*, pages 17–28. Springer.

[Pimentel et al., 2006] Pimentel, A. D., Erbas, C., and Polstra, S. (2006). A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55(2):99–112.

[Pnueli, 1977] Pnueli, A. (1977). The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE.

[Pohl et al., 2005] Pohl, K., Böckle, G., and van Der Linden, F. J. (2005). *Software product line engineering: foundations, principles and techniques.* Springer Science & Business Media.

[Polzer et al., 2009] Polzer, A., Kowalewski, S., and Botterweck, G. (2009). Applying software product line techniques in model-based embedded systems engineering. In *Model-Based Methodologies for Pervasive and Embedded Software, 2009. MOMPES'09. ICSE Workshop on*, pages 2–10. IEEE.

[Pretschner et al., 2007] Pretschner, A., Broy, M., Kruger, I. H., and Stauner, T. (2007). Software engineering for automotive systems: A roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 55–71. IEEE Computer Society.

[Rodrigues et al., 2015] Rodrigues, G. N., Alves, V., Nunes, V., Lanna, A., Cordy, M., Schobbens, P., Sharifloo, A. M., and Legay, A. (2015). Modeling and verification for probabilistic properties in software product lines. In *16th IEEE International Symposium on High Assurance Systems Engineering, HASE 2015, Daytona Beach, FL, USA, January 8-10, 2015*, pages 173–180.

[Ross et al., 2017] Ross, J. A., Murashkin, A., Liang, J. H., Antkiewicz, M., and Czarnecki, K. (2017). Synthesis and exploration of multi-level, multi-perspective architectures of automotive embedded systems. *Software & Systems Modeling*, pages 1–29.

[Rossi et al., 2006] Rossi, F., Van Beek, P., and Walsh, T. (2006). *Handbook of constraint programming*. Elsevier.

[Sabin and Weigel, 1998] Sabin, D. and Weigel, R. (1998). Product configuration frameworks-a survey. *IEEE Intelligent Systems and their Applications*, 13(4):42–49.

[Sabouri et al., 2012] Sabouri, H., Jaghoori, M. M., de Boer, F., and Khosravi, R. (2012). Scheduling and analysis of real-time software families. In *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, pages 680–689. IEEE.

[Sangiovanni-Vincentelli, 2007] Sangiovanni-Vincentelli, A. (2007). Quo vadis, sld? reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, 95(3):467–506.

[Saraswat et al., 1994] Saraswat, V. A., Jagadeesan, R., and Gupta, V. (1994). Foundations of timed concurrent constraint programming. In *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71–80. IEEE.

[Saraswat et al., 1991] Saraswat, V. A., Rinard, M., and Panangaden, P. (1991). The semantic foundations of concurrent constraint programming. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–352.

[Schobbens et al., 2007] Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., and Bontemps, Y. (2007). Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479.

[Schor et al., 2012] Schor, L., Bacivarov, I., Rai, D., Yang, H., Kang, S.-H., and Thiele, L. (2012). Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, pages 71–80. ACM.

[Schrijver, 1998] Schrijver, A. (1998). *Theory of linear and integer programming*. John Wiley & Sons.

[Sgroi et al., 2000] Sgroi, M., Lavagno, L., and Sangiovanni-Vincentelli, A. (2000). Formal models for embedded system design. *IEEE Design & Test of Computers*, 17(2):14–27.

[Siegmund et al., 2015] Siegmund, N., Grebhahn, A., Apel, S., and Kästner, C. (2015). Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 284–294. ACM.

[Siegmund et al., 2012] Siegmund, N., Rosenmüller, M., Kuhlemann, M., Kästner, C., Apel, S., and Saake, G. (2012). Spl conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, 20(3-4):487–517.

[Sigdel et al., 2009] Sigdel, K., Thompson, M., Pimentel, A. D., Galuzzi, C., and Bertels, K. (2009). System-level runtime mapping exploration of reconfigurable architectures. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE.

[Sima and Bertels, 2009] Sima, V.-M. and Bertels, K. (2009). Runtime decision of hardware or software execution on a heterogeneous reconfigurable platform. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–6. IEEE.

[Singh et al., 2017] Singh, A. K., Dziurzanski, P., Mendis, H. R., and Indrusiak, L. S. (2017). A survey and comparative study of hard and soft real-time dynamic resource allocation strategies for multi-/many-core systems. *ACM Comput. Surv.*, 50(2):24:1–24:40.

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

[Streitferdt et al., 2005] Streitferdt, D., Sochos, P., Heller, C., and Philippow, I. (2005). Configuring embedded system families using feature models. In *Proc. of Net. ObjectDays*, pages 339–350.

[ter Beek et al., 2016] ter Beek, M. H., Fantechi, A., Gnesi, S., and Mazzanti, F. (2016). Modelling and analysing variability in product families: model checking of modal transition systems with variability constraints. *Journal of Logical and Algebraic Methods in Programming*, 85(2):287–315.

[Thüm et al., 2014] Thüm, T., Apel, S., Kästner, C., Schaefer, I., and Saake, G. (2014). A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6:1–6:45.

[Valov et al., 2017] Valov, P., Petkovich, J.-C., Guo, J., Fischmeister, S., and Czarnecki, K. (2017). Transferring performance prediction models across different hardware platforms. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 39–50. ACM.

[Van Stralen and Pimentel, 2010] Van Stralen, P. and Pimentel, A. (2010). Scenario-based design space exploration of mpsocs. In *Computer Design (ICCD), 2010 IEEE International Conference on*, pages 305–312. IEEE.

[Van Vliet et al., 2008] Van Vliet, H., Van Vliet, H., and Van Vliet, J. (2008). *Software engineering: principles and practice*, volume 13. Citeseer.

[Velez et al., 2020] Velez, M., Jamshidi, P., Sattler, F., Siegmund, N., Apel, S., and Kästner, C. (2020). Configcrusher: towards white-box performance analysis for configurable systems. *Automated Software Engineering*, pages 1–36.

[Vogelsang et al., 2017] Vogelsang, A., Amorim, T., Pudlitz, F., Gersing, P., and Philipps, J. (2017). Should i stay or should i go? on forces that drive and prevent mbse adoption in the embedded systems industry. In *International Conference on Product-Focused Software Process Improvement*, pages 182–198. Springer.

[Wescott, 2011] Wescott, T. (2011). *Applied control theory for embedded systems*. Elsevier.

[Wildermann et al., 2011a] Wildermann, S., Reimann, F., Teich, J., and Salcic, Z. (2011a). Operational mode exploration for reconfigurable systems with multiple applications. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1–8. IEEE.

[Wildermann et al., 2011b] Wildermann, S., Reimann, F., Ziener, D., and Teich, J. (2011b). Symbolic design space exploration for multi-mode reconfigurable systems. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 129–138. ACM.

[Wu et al., 2018] Wu, Q., Gouyon, D., Levrat, E., and Boudau, S. (2018). A review of know-how reuse with patterns in model-based systems engineering. In *International Conference on Complex Systems Design & Management*, pages 219–229. Springer.

[Wymore, 2018] Wymore, A. W. (2018). *Model-based systems engineering*, volume 3. CRC press.

[Zhang et al., 2015] Zhang, Y., Guo, J., Blais, E., and Czarnecki, K. (2015). Performance prediction of configurable software systems by fourier learning (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 365–373. IEEE.

[Zhang et al., 2017] Zhang, Z., Nielsen, B., Larsen, K. G., Nies, G., Stenger, M., and Hermanns, H. (2017). Pareto optimal reachability analysis for simple priced timed automata. In *International Conference on Formal Engineering Methods*, pages 481–495. Springer.

[Zulkoski et al., 2014] Zulkoski, E., Kleynhans, C., Yee, M.-H., Rayside, D., and Czarnecki, K. (2014). Optimizing alloy for multi-objective software product line configuration. In *Proceedings of the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z - Volume 8477*, ABZ 2014, pages 328–333. Springer-Verlag New York, Inc.

Abstract

Software-intensive embedded systems, such as automotive systems, are increasingly built from highly-variable applications targeting evermore configurable hardware platforms. Moreover, there are often various ways to implement a given application on a specific platform. This threefold variability leads to an immense number of system design alternatives. The notorious problem is to establish, at early stages of development, which designs fulfill and optimize functional and non-functional requirements. Traditional system design frameworks capture system requirements and specifications to derive and evaluate every design automatically. However, they use enumeration-based techniques and offer poor scalability at both modelling and analysis stages. On the other hand, variability modelling approaches exploit commonalities between different but related products to efficiently evaluate the whole product line. Yet, given system specifications, they lack to automatically derive the design space while only specific facets of the problem are evaluated in isolation. We propose a model-driven framework that combines and extends both approaches. It captures requirements and specifications in the form of variable data-flows and configurable hardware platforms, with non-functional constraints and a cost function. An original mapping algorithm then derives the design space automatically and generates it in the form of a variability-aware model of computation, which encodes every system design. Novel verification algorithms then pinpoint suitable designs efficiently. The benefits of our approach are evaluated through a real-world case study from the automotive industry.

Keywords: Model-Based embedded-system design, Feasibility analysis, Optimization, Model of computation, Behavioral Product Line, Quantitative properties, Variability-aware model-checking