Functional feasibility analysis of variability-intensive data flow-oriented applications over highly-configurable platforms

Sami Lazreg, Philippe Collet, Sébastien Mosser

Functional Feasibility Analysis of Variability-Intensive Data Flow-Oriented Applications over Highly-Configurable Platforms

Sami Lazreg
Visteon Electronics and
Université Côte d'Azur, CNRS,
I3S, France
slazreg@visteon.com

Philippe Collet
Université Côte d'Azur, CNRS,
I3S, France
collet@i3s.unice.fr

Sébastien Mosser
Université Côte d'Azur, CNRS,
I3S, France
mosser@i3s.unice.fr

ABSTRACT

Data-flow oriented embedded systems, such as automotive systems used to render HMI (e.g., instrument clusters, infotainments), are increasingly built from highly variable specifications while targeting different constrained hardware platforms configurable in a fine-grained way. These variabilities at two different levels lead to a huge number of possible embedded system solutions, which functional feasibility is extremely complex and tedious to predetermine. In this paper, we propose a tooled approach that capture high level specifications as variable dataflows, and targeted platforms as variable component models. Dataflows can then be mapped onto platforms to express a specification of such variability-intensive systems. The proposed solution transforms this specification into structural and behavioral variability models and reuses automated reasoning techniques to explore and assess the functional feasibility of all variants in a single run. We also report on the validation of the proposed approach. A qualitative evaluation has been conducted on an industrial case study of automotive instrument cluster, while a quantitative one is reported on large generated datasets.

CCS Concepts

•General and reference → Design; Validation; •Computer systems organization → Embedded systems; •Software and its engineering → Software product lines; •Theory of computation → Verification by model checking;

Keywords

Embedded system design engineering; variability modeling; feature model; behavioral product lines model checking.

1. INTRODUCTION

Validating embedded systems design at early stages of development is of fundamental importance in industry. Ideally embedded system design should be modeled from high-level specifications, and then assess against possible implementations. Data-flow oriented embedded systems, such as automotive systems used to render HMI (e.g., instrument clusters, infotainments) are typically built from highly variable specifications. They are composed of a data-flow driving and feeding graphical processors to provide efficient and high-quality graphic rendering at a lower cost, while targeted hardware platforms are composed of heterogeneous and constrained hardware components. The variability is then two-fold, with multiple graphic data-flow variants that can meet functional requirements, and diverse targeted hardware platform, which are highly configurable in a fine-grained way. These dimensions of variability dreadfully increase the size of the design space of these embedded systems (i.e., the number of possible embedded system implementation designs), making the feasibility assessment of these systems extremely tedious and complex.

Generally, design spaces of variable systems and protocols are assessed through variability-aware model checking on variable transition-based systems [1, 8]. However these approaches are not capable of automatically map the variable data-flow specifications on configurable platform descriptions to apply their model-checking techniques. Consequently, this would imply to manually infer, model and assess embedded system design spaces from high level specifications, making this activity extremely tedious, time consuming, and error-prone.

Facing this issue, we determine three challenges to be tackled: (i) capturing and modeling from high-level specifications, structure, behavior and variability of these embedded systems (e.g., data-flow and platform alternatives, data sizes, memory capacities, graphic pipelines), (ii) inferring automatically all possible embedded system design implementations from specification models and, (iii) exploring and assessing the feasibility of all system implementations w.r.t. the predefined structural, behavioral and variability constraints. Current approaches [22, 15, 16] assess functional feasibility of constrained data-flow-oriented embedded systems, but do not capture nor manage variability at both levels. Some *Ad hoc* techniques are trying to handle either platform variability (as reconfigurable architectures [21, 20] [18]) or functional variability (as multiple scenarios [19, 25] or multi-modes systems [26, 17]). On the other hand,

approaches tackling both kinds of variability [12] are focusing on optimal platform selections to implement multiple functional variants at a lower cost, but they do not manage structural and behavioral properties (e.g. data sizes, memory capacities, graphic pipelines).

In this paper we propose an approach that extends these researches by supporting a complete modeling and assessment of structural, behavioral and variability properties of the targeted embedded systems by combining embedded system design engineering [22, 15, 16] and Product line engineering techniques [1, 8]. The proposed solution is model driven and i) captures high level variable data-flow and platform specifications following principles of separation of concern, ii) maps variable data-flow requirements into a description of the targeted variable hardware platform, so to infer the embedded system design space (i.e. all system implementations), iii) transforms the design space into a behavioral product line to reuse automated reasoning techniques (i.e. SAT solving, variability-aware model checking) to explore and assess the functional feasibility of all system design implementations in a single run. The framework also allows to remove invalid designs from the design space by constraining it.

This paper is an extended version from a paper published in the Variability and Software Product Line Engineering track of the SAC 2018 conference. In this extension, we detail a complete evaluation of the proposed approach with:

- complete implementation details with end-to-end sample materials,
- a qualitative evaluation on a real industrial use case in automotive systems, i.e., an instrument cluster product line,
- a quantitative evaluation on the scalability of the approach, using large generated datasets on both application and platform sides,
- a discussion on the threats to validity.

The remainder of the paper is organized as follows. Section 2 introduces the context and motivations illustrated by a running example. Section 3 presents the proposed framework, detailing each model and step. Section 4 details the qualitative and quantitative validation, and discusses threats to validity, while Section 5 concludes the paper.

2. MOTIVATIONS

Requirement gathering and validation of this research work have been realized in the context of an industrial collaboration with Visteon Electronics, a world class leader in automotive systems (e.g. instruments clusters, infotainment, connected vehicles). In the following we introduce one of the company case studies, extract a running example, determine requirements from them and discuss related work.

2.1 Case study

The case study is focusing on functional validation of some instrument clusters. By applying various data-flow image processing effects, such as blending, warping and scaling, an instrument cluster system improves driver experience with useful and high quality 2D/3D Human-Machine Interface (HMI). The embedded hardware platforms used to develop these systems are more and more highly configurable, but constrained in terms of architecture and capacities. Furthermore, multiple graphic data-flows variants can meet the client HMI requirements, but they also depend on the platform architectures and capacities. We consider this case study as representative of variability-intensive data-flow oriented systems. Different forms of variability, from high-level data-flows to low-level platforms lead to a huge number of possible system solutions, which feasibility is extremely complex and tedious to predetermine early in the development process.

2.2 Running Example

We now introduce a running example of a simplified instrument cluster. The data-flow on Fig. 1 represents different image flow processing that meet the HMI functional requirements. Images are processed by graphical tasks: image d_2 has two different possible resolutions (e.g. 800x480, 480x320) and will be processed by task C. Image d_1 can be either processed by task A or task B.

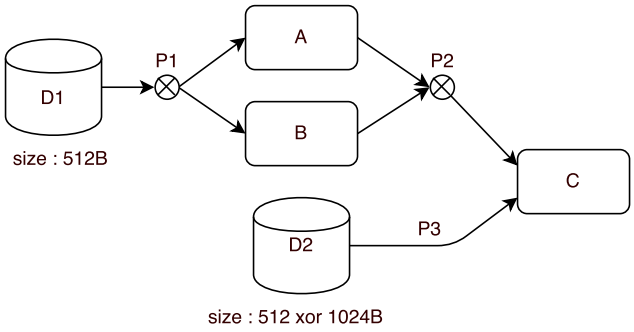


Figure 1: Functional specification

On the hardware side (cf. Fig. 2), the platform provides image processing capabilities through non programmable pipeline processors of DCU (Display Controller Unit) or GPU (Graphic Processing Unit) type, as well as data storage functionalities through RAM (Random Access Memory) and ROM (Read Only Memory). RAM and GPU are optional in the actual hardware products, so a system implementation may contain or not these components. Among variabilities in platforms, one can write data into and/or read data from RAM memory while only reading data is possible from ROM. Moreover, memory storage have limited and possibly variable (e.g. RAM) capacity. Contrary to a DCU, which renders directly processed images into display, a GPU needs to store its processing result into RAM. Graphical hardware processors are often designed as a multi-step pipeline, composed of several hardware implemented processing steps, and processor internal fifo memory buffers transferring data from one step to another. In our example, while a GPU can apply A followed by B processing on images in a single pass, a DCU can apply A, then followed by C.

Images and processing could be, respectively, stored and pro-

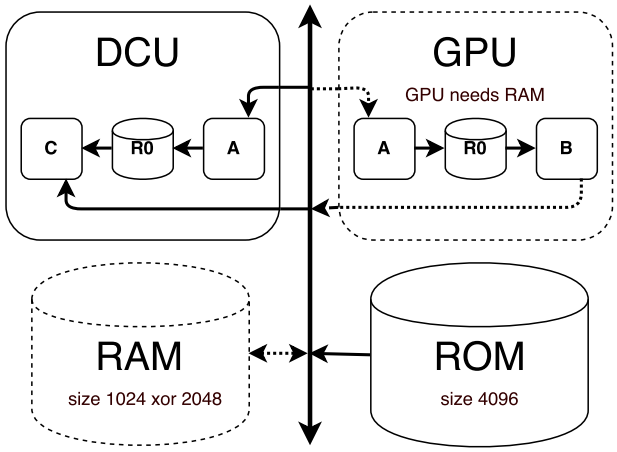


Figure 2: Platform specification

cessed by multiple components, while images can be stored on RAM or on ROM: task A could be processed by both GPU and DCU, but a data-flow variant containing B task can only be implemented on a system containing a GPU. Finally, data mapping on memory are constrained in terms of storage capacity.

Consequently, to be valid, a system implementation has to fulfill (i) structural constraints such as not violating maximum memory capacities, (ii) behavioral constraints such as using correctly processor pipelines and memories, and (iii) variability constraints such as component dependency and exclusion. In our example, the application data-flow has 4 variants while the Platform exposes 3 architecture variants. Even with this simplified case, this leads to 178 possible implementations, in which 58 satisfy constraints and could be developed by engineers.[1]

2.3 Related Work

In the context of our work, engineers must be assisted to assess the functional feasibility of the different potential embedded system solutions, with means to capture both the high-level functional requirements and the specifications of targeted variable platforms. Ideally, a solution should be able to capture structural, behavioral and variability properties of both functional and platform specifications at a fine-grained level, so to use these input models to automatically infer all possible embedded system implementation designs and assess the resulting consistent design space.

In the product line engineering, lots of approaches [1, 8] are capable to model variable transition-based systems such as safety-critical systems or protocols, and to validate, through variability-aware model-checking, temporal properties and behavioral aspects. However, given high-level data-flows and platform specifications, these approaches are not capable of automatically map data-flows on platforms in order to infer and assess the resulting design space. Assessing the different mapping manually is not feasible in practice, as the activity would be tedious and error-prone.

In embedded system design engineering, most of the approaches capture high-level application and platform specifications, and map an application on hardware platforms in order to find, by *design space exploration*, an optimized system implementation for a single functional specification on a single platform [22, 15]. Consequently, they do not capture nor manage variability at the application level and hardware platform variability is limited to component capacities (e.g. memory and bus size, processor frequency). Some other approaches try to handle some limited variability in functional specifications (e.g. optional task, alternative tasks, variable data) (as multiple scenarios [19, 25] or multi-modes systems [26, 17]), but they do not manage platform variability. Some others try to handle some limited variability in platforms (e.g. optional resource, resource dependency, memories sizes) with reconfigurable architectures [21, 20] [18]. To the best of our knowledge, none of these approaches handle variability in both application and platform sides so to assess the feasibility of our class of problem.

Interestingly, the recent approach of Graf et. al. [13, 14] manages some variability in both platform and functional specifications. On the platform variability side, resource components can be selected or not, while optional and mutually exclusive task groups are managed on the functional part. However, the approach is handling a coarse-grain form of requirements and cannot capture some of our specifications (e.g.data and memory sizes, as well as platform aspects such as processor pipelines or fifo buffers). Additionally, only structural validation of the system implementations is supported. Behavioral properties (e.g. data sizes, memory capacities, graphic pipelines) and behavioral constraints (e.g., absence of deadlock, reachability, liveness, safety etc.), which are fundamental in our case, cannot be checked.

3. PROPOSED FRAMEWORK

3.1 Overview

The proposed approach follows a model driven decomposition, based on the well-known robust Y-Chart pattern [2, 16], which separates application and platform into different concerns. This allows modular specification and reasoning about the different parts of the specified embedded systems. Given high-level variable dataflow and platform inputs that notably captures the variability of functional and platform specifications, the framework will i) map all implementation of each data-flow variants on each platform configuration, ii) generate a Featured Transition System *(FTS)* [8] from the system design space model, *i.e.* representing system implementations (cf. Fig. 3). This model consists in an extended form of automaton product line, which is then iii) checked in one run by a variability-aware model checker.

As shown on Fig. 3, our framework consists of three main models and two processes. We give here an overview while the following sections will detail and illustrate these elements.

Variable applications: a functional expert is in charge of capturing the functional requirements (cf. fig. 1) of the em-

[1]Finding the best solution among the remaining correct solutions is also an important problem, but out of the scope of this paper.

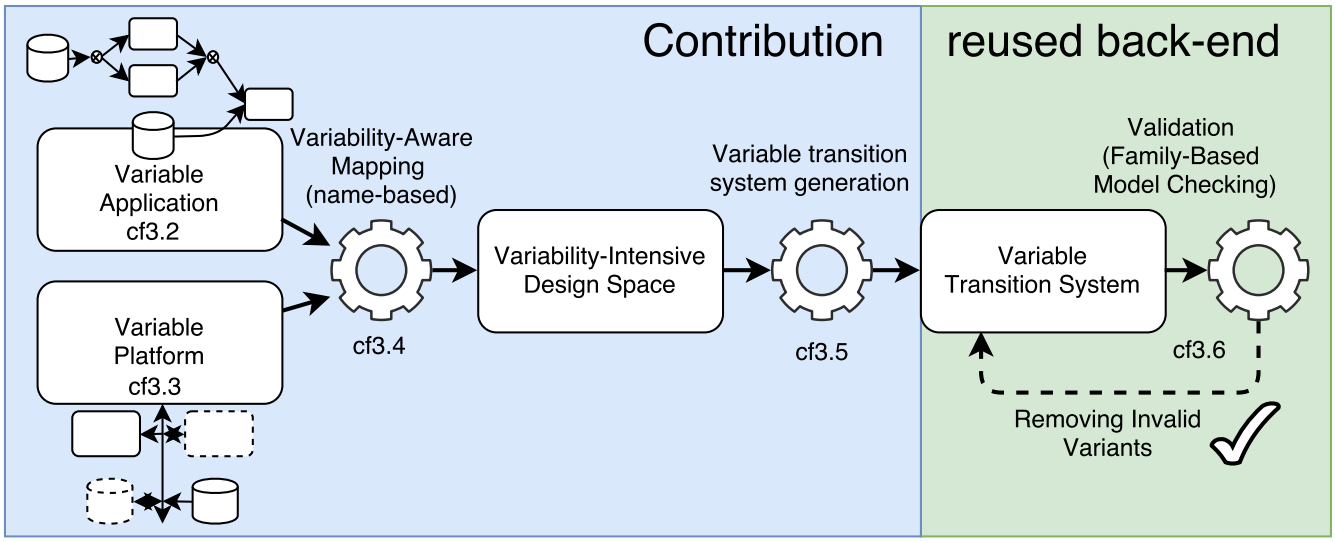


Figure 3: Framework Overview

bedded system through an extended data-flow (cf. sec. 3.2). 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).

Variable platforms: on this side, a platform expert is in charge of expressing the platform specification (cf. fig. 2) as a templated component based system (cf. sec. 3.3). This model contains a set of components connected with each others. Similarly to the application one, the platform model also captures the variability of the defined components.

Variability-aware mapping process: the mapping algorithm (cf. sec. 3.4) consumes the application and platform input models previously defined and generates the *Variability-Intensive Design Space, i.e.* representing system implementations. It is made of two steps: (i) it finds for each task data and data-path (cf. fig. 1) all the possible mappings on appropriate platform processors and storage (cf. fig. 2); basically this is done by matching the task names with the names of the hardware functions of processors; data-paths are mapped on reachable memory of appropriate processor hardware functions; (ii) as the design space contains all mapping possibilities, the algorithm prunes unfeasible mappings w.r.t. structural and variability constraints.

Design space as a behavioral product line: from the system design space model, a *Behavioral Product Line* representing all system implementations is generated (cf. sec. 3.5). This product line is represented as a featured automaton, so that we can reuse and adapt techniques that rely on variability-aware model-checking to validate the inferred systems. The basic idea is to transform a variable data-flow, a variable platform, and mappings to a data-flow automaton using, through a mapping automaton, a platform automaton to execute it. Valid executions of the application automaton should then respect generated properties representing end state reachability to ensure that the execution is correctly scheduled and executed onto the platform automaton.

Validation process: the validation process reuses automated reasoning techniques to assess structural and behavioral functional feasibility of the system variants represented by the behavioral product line (cf. sec. 3.6). The model checking is going to determine classic properties, such as safety, absence of deadlock, and our state reachability generated property, on all variants in one run. As a result, the validation solves and extracts valid variants respecting all structural, behavioral and variability constraints.

3.2 Applications as Variable Data-Flows

In our approach, a functional expert captures structure, behavior (data, task, data-path, etc.) and variability aspects (data size, alternative flows, etc.) of the functional requirements of the embedded system through an extended data-flow model. The extensions concern variability and data aspects of functional requirements, and in the following, we propose a formal data-flow model to do so.

Definition 1. *A variable data-flow graph is a tuple* $VDG = (T, D, Path, E, \zeta)$ *where:*

- T *is a set of tasks,*
- D *is a set of source data, and,* $\zeta : D \rightarrow \{s_0, ..., s_i \in \mathbb{N}^*\}$ *returns a set of alternative sizes of data,*

$$|\zeta(d \in D)| = \begin{cases} > 1, & \text{if } d \text{ has a variable size} \\ 1, & \text{if } d \text{ has not a variable size} \end{cases}$$

- $Path$ *is a set of data-paths by which producer and consumer (i.e. tasks and data) are connected.*
- $E \subseteq (T \cup D) \times Path \times T$ *is the set of edges representing flows processing between producers and consumers.*

The set of connected, input data-paths to a task $I(t)$*, output data-paths from a task or data* $O(v)$ *are denoted by:*

$$I(t) : \{p \in Path | (x, p, t) \in E\},$$

$$O(v \in T \cup D) : \{p \in Path | (v, p, x) \in E\}.$$

Similarly $I(p)$, input tasks to a output data-paths, and $O(p)$, output tasks from a input data-paths, are denoted by:

$$I(p) : \{prod \in T \cup D | (prod, p, x) \in E\},$$
$$O(p) : \{t \in T | (x, p, t) \in E\}.$$

Then:

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

A variable data-flow represents multiple data-flow variants. To implicitly represent all these variants in a single model, we follow the same approach as in variable workflows from [13], allowing data-paths to have multiple input and output tasks connected.

A data-path can be connected to multiple alternative, input tasks if $|I(p)| > 1$, and output tasks if $|O(p)| > 1$. But, If $|I(p)| = 1 \wedge |O(p)| = 1$, the data-path is connected to only one input and output task (i.e the data-path has no flow variability).

As data can have alternative sizes, we introduce the function ζ which returns the set of alternative sizes $S = \zeta(d)$, each datum has at least one size and if $|\zeta(d)| > 1$ the size of d is variable.

If the data-flow has no flow variability, $\forall p \in Path, |I(p)| + |O(p)| = 2$, and no data variability, $\forall d \in D, |\zeta(d)| = 1$, the data-flow is not variable.

The functional specification of the running example VDG_{re} is then represented as

$(T_{re} = \{a, b, c\}, D_{re} = \{d_1, d_2\}, Path_{re} = \{p_1, p_2, p_3\},$

$E_{re} = \{(d_1, p_1, a), (a, p_2, c), (d_1, p_1, b), (b, p_2, c), (d_2, p_3, c)\})$

with,

$\zeta(d_1) = 512, \zeta(d_2) = \{512, 1024\},$

$I(p_1) = d_1, I(p_2) = \{a, b\}, I(p_3) = d_2,$

$O(p_1) = \{a, b\}, O(p_2) = c, O(p_3) = c,$

$I(a) = I(b) = p_1, I(c) = \{p_2, p_3\},$

$O(a) = O(b) = p_2, O(c) = \emptyset, O(d_1) = p_1, O(d_2) = p_3.$

3.3 Platforms as Variable Resource Graphs

A variable platform specification is represented by a templated component based system (multi-pass processors, streaming processor, read-only memory, read-write memory, write-only memory, first-in-first-out buffers etc) where platform can have optional resource components and variability constraints on resources (dependency, incompatibility, etc.). To capture variability aspects of a platform specification, we propose a formal architecture model defined as follows.

DEFINITION 2. *A variable resource graph is a tuple $VRG = (Proc, S, C_s, \xi, \theta, \phi_{requires}, \phi_{excludes})$ where:*

- *$Proc = (F, B, C_b \subseteq (F \times B) \times (B \times F))$ is a processor composed of a set F of possible functions, B is a set of processor internal* first-in-first-out *buffers and C_b the connections between the different functions and buffers representing the processor pipeline.*
- *S is a set of memory storage, and $\xi : S \rightarrow \{c_0, ..., c_i \in \mathbb{N}^*\}$ returns a set of alternative capacities of storage $s \in S$,*

  $$|\xi(s)| = \begin{cases} > 1, \text{ if } s \text{ has a variable storage capacity} \\ 1, \text{ if } s \text{ has not a variable storage capacity} \end{cases}$$

- *$C_s \subseteq (S \times Proc) \cup (Proc \times S)$ is the set of connections between memory storage and processors,*
- *$R \subseteq Proc \cup S$ is the set of resource components (i.e. processors and memory storage),*
- *$\theta : R \rightarrow \mathbb{B}$ return true if a component (i.e. processor or memory storage) is optional,*
- *$\phi_{requires} : R \rightarrow R$ captures dependency between resource components, similarly $\phi_{excludes} : R \rightarrow R$ captures incompatibility.*

The set of, input memories to a processor function $I(f)$, output memories from a processor function $O(f)$ are denoted by:

$$\exists p = (F, B, C_b) \in Proc,$$
$$I(f \in F) : \{m \in S \cup B | (m, p) \in C_s \vee (m, f) \in C_b\},$$
$$O(f \in F) : \{m \in S \cup B | (p, m) \in C_s \vee (f, m) \in C_b\}.$$

As a variable platform represents multiple platform configurations, we also capture implicitly all these configurations 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.

If the platform has no component variability $\forall r \in R, \theta(r) = \perp$ and no variable memory storage, $\forall s \in S, |\xi(s)| = 1$, the platform is not variable.

The platform specification of the running example VG_{re} is then formalized as

$(Proc_{re} = \{DCU, GPU\}, S_{re} = \{RAM, ROM\},$

$C_{s_{re}} = \{(RAM, DCU), (ROM, DCU),$
$(RAM, GPU), (ROM, GPU), (GPU, RAM)\})$

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,

$\xi(ROM) = 4096, \xi(RAM) = \{1024, 2048\}$,

$\theta(GPU) = \theta(RAM) = \top, \theta(DCU) = \theta(ROM) = \bot$

$\phi_{requires}(GPU) = RAM, \phi_{requires}(RAM) = \emptyset$,

$I(a_{gpu}) = \{ROM, RAM\}, O(a_{gpu}) = \{r0_{gpu}, RAM\}$,

$I(c_{dcu}) = \{r0_{dcu}, ROM, RAM\}, O(c_{dcu}) = \emptyset$.

3.4 Variability-Aware Mapping Process

The mapping algorithm takes as inputs the variable data-flow and configurable platform models in order to find all embedded system implementations. We propose a mapping model to not only capture all implementations of a single data-flow into a single platform but to capture all data-flow variants implementations onto all platform configurations. Our variability-aware mapping model can be seen as a product line of traditional mapping models.

DEFINITION 3. *A variability-aware data-flow-oriented mapping $VM = (Tm, Dm, Em)$ where:*

- *$Tm \subseteq T \times F$ is the set of possible mappings of tasks on processors $\forall (t, f) \in Tm$, t can be mapped on processor function f because f can implement t,*
- *$Dm \subseteq D \times S$ is the set of mappings of data on memory storage,*
- *$Em \subseteq E \times (S \cup B)$ is the set of data-paths mapping on memory by which data are consumed/produced.*

DEFINITION 4. *The Variability-Aware Mapping function $M = VDG \times VRG \rightarrow VM$ sorts topologically the data-flow and finds appropriate mapping for each data, task and data-paths of the data-flow using resources of the resource graph.*

Basically, each valid mapping should respect consistency constraints such as that
(1) All tasks are mapped to, a least, one processor function:

$\forall t \in T, \exists (t, f) \in Tm$,
(2) All data are mapped to, at least, one memory storage:

$\forall d \in D, \exists (d, s) \in Dm$,
(3) All data-paths are mapped to, at least, one appropriate mapping. For data-path starting by an input datum, the storage on which the datum is mapped has to be reachable by the processor function on which the task consuming the datum is mapped.

$\forall e = (d \in D, p, t) \in E, \exists (e, s \in S) \in Em$,

$\exists (d, s) \in Dm, \exists (t, f) \in Tm, s \in I(f)$,
For data-path between tasks, the memory on which the output of the first task is mapped has to be reachable by the processor function on which the second task is mapped.

$\forall e = (t \in T, p, t') \in E, \exists (e, m) \in Em$,

$\exists (t, f) \in Tm, \exists (t', f') \in Tm, m \in O(f) \wedge m \in I(f')$
The mapping model of the running example VM_{re} is then formalized as

$(Tm_{re} = \{(a, a_{dcu}), (a, a_{gpu}), (b, b_{gpu}), (c, c_{dcu})\}$,

$Dm_{re} = \{(d_1, RAM), (d_1, ROM), (d_2, RAM), (d_2, ROM)\}$,

$Em_{re} = \{((d_1, p_1, a), RAM), ((d_1, p_1, a), ROM),$
$((d_1, p_1, b), RAM), ((d_1, p_1, b), ROM),$
$((a, p_2, c), r0_{dcu}), ((a, p_2, c), RAM), ((b, p_2, c), RAM),$
$((d_2, p_3, c), RAM), ((d_2, p_3, c), ROM))\})$

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

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

3.5 Design Space as a Behavioral Product Line

Automata and model-checking techniques have been widely used to model and validate real-time and embedded systems [4, 5]. Interestingly, the basic approach used is to schedule an application automaton using a platform automaton [10]. Unfortunately, these approaches are not design to manage any variability aspect of specifications.

Our framework relies on Featured-Transition-Systems (FTS) to represent and validate the design space. FTS has the strength to model explicitly the variability points structurally, through a Feature Diagram [3] *(FD)*, instead of modeling variability points behaviorally, by optional transition with possible constraints [24]. This eases the transformation to featured automaton and the removal of invalid implementations from it. In our approach, we also use LTL property to ensure that all valid execution paths of all system implementations reach the end state of all task of the data-flow.

DEFINITION 5. *A featured automaton is a tuple $FA = (Loc, Loc_0, I, Act \subseteq Aff \cup \phi \cup Com, trans, \chi, Ch, L, AP, d, \lambda)$ such that:*

- *Loc is a finite set of locations, $Loc_0 \in Loc$, is a set of initial states and $I \in Loc$, is a set of final states,*
- *Ch is a finite set of communication channels,*
- *χ is a finite set of variables,*
- *Act is a set of, Aff which is a finite set of variable affectations, ϕ which is a finite set of guards in a boolean expression form and Com, which is a set of communications $Com \subseteq \{c!m, c?m, c!?m | c \in Ch, m \in \chi\}$*
- *$trans \subseteq Loc \times Act \times Loc$ are state transitions,*
- *$d = (N \subseteq N_m \cup N_{opt} \cup N_{xor}, DE \subseteq N \times N, Tcl)$ is a Feature Diagram (FD), N is the set of mandatory, optional and alternatives features, DE represents relation between features, Tcl are constraints between features, $[\![d]\!]_{FD} \subseteq \mathcal{P}(N)$ is the set of valid product configurations,*
- *$\lambda : trans \rightarrow \mathbb{B}(N)$ is a total function that labels transitions with feature expressions.*
- *AP is a set of atomic proposition and $L : Loc \rightarrow AP$ labels transitions with atomic propositions.*

A transition $s \xrightarrow{\alpha} s'$ is possible for the set of product configurations $P \subseteq [\![\lambda(s \xrightarrow{\alpha} s')]\!]$ and if

$\forall g \in \alpha \cap \phi$, *g is satisfied,*
$\forall (c?m) \in \alpha \cap Com$, *wait for data event* $c!m$,
$\forall (c!?m_0) \in \alpha \cap Com$, *send data event* $c!m_0$ *but wait for data event* $c!m_1$ *with* $m_0 = m_1$.

DEFINITION 6. *A Linear Temporal Logic property (LTL) is a temporal expression of atomic proposition that all possible executions of system variants should satisfy as,* $[\![fa]\!]_{FA} \models \varphi$ *where*
$\varphi ::= a \in AP | \varphi \wedge \varphi | \diamond \varphi$. *Symbol* $\diamond$ *means that the property will become true at some point in the future.*

We now show how our design space is transformed to a FA. To simplify the transformation process, let us use the following functions:

$f : T \cup Path \cup R \rightarrow N, f_s : D \cup S \times \mathbb{N}^* \rightarrow N,$

$f_{to} : Path \rightarrow N, f_{from} : Path \rightarrow N,$

$f_{to} : Path \times T \rightarrow N, f_{from} : T \cup D \times Path \rightarrow N,$

$f_m : T \cup Path \rightarrow N, f_{tm} : T \times F \rightarrow N, f_{pm} : Path \times S \cup B \rightarrow N$, which transforms model elements to features.

For example, in our running example, the functions would be:

$f(a) = A, f_s(d_2, 1024) = D_2Size1024$

$f_{to}(p_1) = p_1\_To, f_{from}(p_{from}) = p_1\_From,$

$f_{to}(p_1, a) = P_1ToA, f_{from}(a, p_2) = P_2FromA,$

$f_m(a) = A_m, f_m(p_1) = P1_m,$

$f_{tm}(a, a_{dcu}) = AOnA_{dcu}, f_{pm}(p_1, ROM) = p_1OnROM$

$f(RAM) = RAM, f_s(RAM, 1024) = RAMSize1024$, ...

Similarly,

$c : T \cup D \cup Path \cup F \cup S \rightarrow Ch,$

$c_m : T \cup Path \rightarrow Ch$

transforms model elements to communication channels to interact with them at automaton level.

A first function $Gen_{FA} : VDG \rightarrow FA \times LTL$ transforms a variable data-flow graph into a FA and generates the LTL property in the following way.

(1.1) it transforms each datum d with variable size into a xor feature group (cf. fig.4(a)):

$\zeta(d) > 1 \implies \forall s \in \zeta(d), \exists (f(d) \in N_{xor}, f_s(d, s)) \in DE$

(1.2) it creates the automaton for each source datum $d \in D$ (cf. fig.4(b)), after setting the datum size, calling the mapping automaton (cf. fig. 6(a, b)) that will allocate the datum on the memory.

$\forall s \in \zeta(d), \exists \{t_0 = (s_0 \xrightarrow{size(in)=s} s_1)$, where,
$|\zeta(d)| > 1 \implies \lambda(t_0) = f_s(d, s),$
$s_1 \xrightarrow{\forall p \in O(d), c_m(p)!?in} s_2,\ s_2 \xrightarrow{\forall p \in O(d), c(p)!in} s_3\} \in trans$

(2.1) it transforms each variable data-path p in a xor feature group (cf. fig. 4(a)).

$|O(p)| > 1 \implies \forall o \in O(p),$
$\exists (f_{to}(p) \in N_{xor}, f_{to}(p, o)) \in DE$
$|I(p)| > 1 \implies \forall i \in I(p),$
$\exists (f_{from}(p) \in N_{xor}, f_{from}(i, p)) \in DE$

(3.1) it creates task/data-paths consistency constraints (cf. fig. 4(a)).

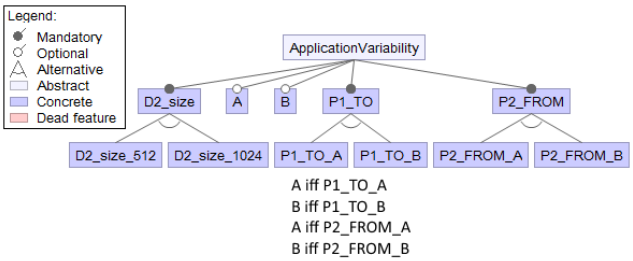
$\forall t \in T, \forall p \in |I(t)|, |O(p)| > 1 \implies$
$\exists (f(t) \iff f_{to}(p, t)) \in Tcl$
$\forall t \in T, \forall p \in |O(t)|, |I(p)| > 1,$
$\exists (f(t) \iff f_{from}(t, p)) \in Tcl$

(3.2) it creates for each task t the automaton (cf. fig. 4(c)) that will wait for data-paths allocation, then call the mapping automaton (cf. fig. 6(c)) to execute the task.

$\exists \{t_0 = (s_0 \xrightarrow{\forall p \in I(t), c(p)?in} s_1)$, where,
$f(t) \in N_{opt} \implies$
$\lambda(t_0) = f(t) \wedge \lambda((s_0 \rightarrow s_4) \in trans) = \neg f(t),$
$s_1 \xrightarrow{\forall p \in O(t), c_m(p)!?out} s_2\ , s_2 \xrightarrow{c_m(t)!?in,out} s_3,$
$s_3 \xrightarrow{\forall p \in O(t), c(p)!out} s_4 \in I,$
where, $L(s_4) = t_{end} \in AP\} \in trans$

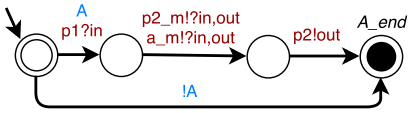
(3.3) it generates the LTL formula that checks that a valid execution must, at some point, satisfy atomic proposition of all data-flow task terminal states.

$\varphi = \diamond(\wedge_{s \in I, L(s) \neq \emptyset} L(s))$



(a) Application FD

(b) Datum d_1 FA



(c) Task A FA

Figure 4: Partial variable data-flow application FA

The second function $Gen_{FA} : VRG \rightarrow FA$ transforms a variable resource graph into a FA in the following way.

(1) it creates feature constraints on resource implementation (cf. fig. 5(a)).

$\forall r \in R, \theta(r) = \top \implies \exists f(r) \in N_{opt}$

$\forall r \in R, \forall r_{req} \in \phi_{requires}(r), \exists (f(r) \implies f(r_{req})) \in Tcl$

$\forall r \in R, \forall r_{exc} \in \phi_{excludes}(r), \exists (f(r) \implies \neg f(r_{exc})) \in Tcl$

(2.1) it creates for each storage s features representing storage alternative sizes.

$\forall c \in \xi(s),\ \exists (f(s) \in N_{xor}, f_s(s,c)) \in DE$

(2.2) it creates for each storage s an automaton that represents basic memory behavior (cf. fig. 5(b)), *cons* and *cap* are respectively the consumed size and the maximal capacity of the storage. Through channels, one can allocate memory, and if there is not enough memory, an error is raised.

$\forall c \in \xi(s), \exists \{t_0 = (s_0 \xrightarrow{cons=0} s_1)$, where,
$f(s) \in N_{opt} \implies$
$\lambda(t_0) = f(s) \wedge \lambda((s_0 \to s_4) \in trans) = \neg f(s)$,
$s_1 \xrightarrow{cap=c} s_2$, where, $\lambda(s_1 \xrightarrow{cap=c} s_2) = f(s,c)$,
$s_2 \xrightarrow{c(s)?in, cons+=size(in)} s_3,\ s_3 \xrightarrow{cons<size} s_2$,
$s_3 \xrightarrow{cons \geq size, error} s_4\} \in trans$

(3) it creates for each processor p an automaton that models basic graphic processor pipeline behavior (cf. fig. 5(c)). When a processor function is executed, the input and output are checked to verify that the pipeline is not misused.

$\forall p = (F, B, C_b) \in Proc, \forall f \in F,$
$\forall (s_i, p) \in C_s, \forall (p, s_o) \in W_s, \forall (b_i, f) \in C_b, \forall (f, b_o) \in C_b,$
$\exists \{t_0 = (s_0 \xrightarrow{c(f)?in,out} s_1)$, where,
$f(p) \in N_{opt} \implies$
$\lambda(t_0) = f(p) \wedge \lambda((s_0 \to s_4) \in trans) = \neg f(p)$,
$s_1 \xrightarrow{loc(in)=s_i \wedge \forall (b_i,f) \in R_b, b_i = free} s_2$,
$s_1 \xrightarrow{loc(in)=b_i \wedge b_i = in} s_2$,
$s_2 \xrightarrow{loc(out)=s_o \wedge \forall (f,b_o) \in W_b, b_o = free} s_3$,
$s_2 \xrightarrow{loc(out)=b_o \wedge b_o = free} s_3,\ s_3 \xrightarrow{c(f)!in,out} s_0\} \in trans$

A third function $Gen_{FA} : VM \to FA$ transforms a variability-aware dataflow-oriented mapping into a FA as follows.

(1.1) it creates features representing all possible task mappings on processor function (cf. fig. 6(a)).

$\forall (t,f) \in Tm, \exists (f_m(t) \in N_{xor}, f_{tm}(t,f)) \in DE$

(1.2) it creates for each task mapping the automaton that executes the processor function according to the mapping configuration (cf. fig. 6(c)).

$\forall t \in T, \forall (t,f) \in Tm,\ \exists \{t_0 = (s_0 \xrightarrow{c_m(t)?in,out} s_1)$, where,
$f_m(t) \in N_{opt} \implies$
$\lambda(t_0) = f_m(t) \wedge \lambda((s_0 \to s_3) \in trans) = \neg f_m(t)$
$t_1 = (s_1 \xrightarrow{c(f)!?in,out} s_2)$, where, $\lambda(t_1) = f_{tm}(t,f)$,
$s_2 \xrightarrow{c_m(t)!in,out} s_3\}, \in trans$

(2.1) Like 1.1, it creates features representing all possible data-path mappings on memory.

$\forall ((x,p,y),s) \in Em, \exists (f_m(p) \in N_{xor}, f_{pm}(p,s)) \in DE$

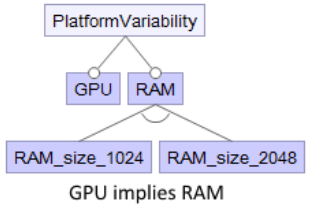
(2.2) Like 2.2, it creates for each data-path mapping the automaton that allocates memory (cf. fig.6(b)).

$\forall p \in Path, \forall ((x,p,y),s) \in Em,$
$\exists \{s_0 \xrightarrow{c_m(p)?out} s_1,\ t_0 = (s_1 \xrightarrow{c(s)!?out,loc(d)=s} s_2)$
where, $\lambda(t_0) = f_{pm}(p,s),\ s_2 \xrightarrow{c_m(p)!out} s_3\} \in trans$

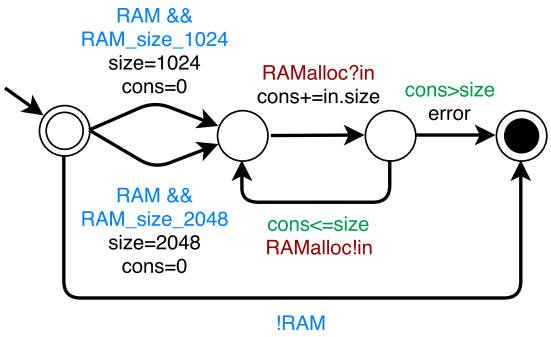
Finally the function $Gen_{FA} : VDS \to FA$, defined by:
$Gen_{FA}((vdg, vrg, vm))$:

$$Gen_{FA}(vdg) || Gen_{FA}(vrg) || Gen_{FA}(vm),$$

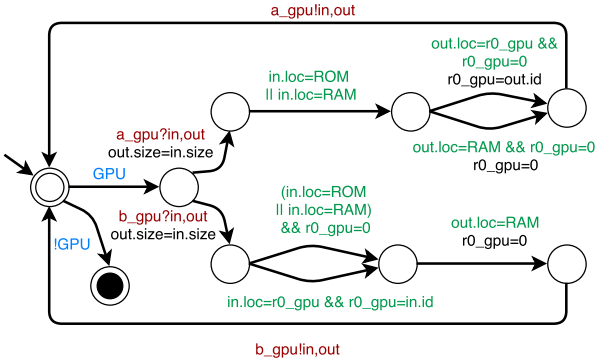
transforms our design space into a featured automaton.



(a) Platform FD



(b) RAM storage



(c) GPU processor

Figure 5: Partial variable platform FA

To preserve the consistency of the design space, variability constraints are inferred such as:

(1.1) Task features with variable data-path features are not implemented on all data-flow variants, then those features are made optional (cf. fig. 4(a)).

$$\forall t \in T, \\ \exists p_i \in I(t), |O(p_i)| > 1 \vee \exists p_o \in O(t), |I(p_o)| > 1 \\ \implies f(t) \in N_{opt}$$

(1.2) Variable task features have their mapping variable too; if a task feature is implemented its mapping must be implemented too, and vice-versa (cf. fig. 4(a) & 6(a)).

$$\forall t \in T, f(t) \in N_{opt} \implies \\ f_m(t) \in N_{opt} \wedge (f(t) \iff f_m(t)) \in Tcl$$

(2.1) If a task mapping feature is implemented on a processor function, the implemented input and output path mappings have to be reachable (cf. fig. 6(a)).

$$\forall (t, f) \in Tm, \forall p_i \in I(t), \forall p_o \in O(t), \\ \exists (f_{tm}(t, f) \iff \\ (\bigvee_{((x,p_i,t),m \in I(f)) \in Em} f_{pm}(p_i, m)) \wedge \\ (\bigvee_{((t,p_o,x),m' \in O(f)) \in Em} f_{pm}(p_o, m'))) \in Tcl$$

(3.1) If a task mapping feature using an optional processor is implemented, the processor must be implemented too.

$$\forall p = (F, x, y) \in Proc, f(p) \in N_{opt}, \forall (t, f \in F) \in Tm \\ \implies \exists (f_{tm}(t, f) \implies f(p)) \in Tcl$$

Similarly, if a data-path mapping feature is implemented on fifo buffer of optional processor (3.2) or optional memory storage (3.3), the resource have to be implemented.

(3.2) $\forall pu = (F, B, x) \in Proc, \forall ((y, p, z), b \in B) \in Em, \\ f(pu) \in N_{opt} \implies \exists (f_{pm}(p, b) \implies f(pu)) \in Tcl$

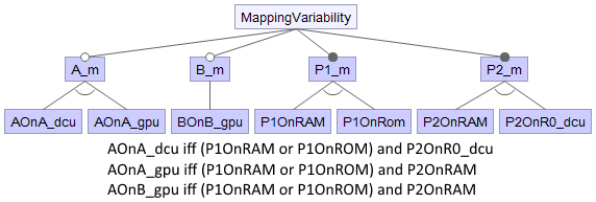
(3.3) $\forall ((x, p, y), s \in S) \in Em, f(s) \in N_{opt} \implies \\ \exists (f_{pm}(p, s) \implies f(s)) \in Tcl$

As an illustration, in our running example, the rules would be:
(1.2) $A \iff A_m$, $B \iff B_m$
(3.1) $BOnB_{gpu} \implies GPU$, $AOnA_{gpu} \implies GPU$
(3.2) $P2OnR0_{gpu} \implies GPU$
(3.3) $P1OnRAM \implies RAM$, $P2OnRAM \implies RAM$

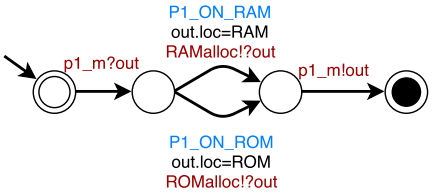
$P3OnRam \implies RAM$

3.6 Validation Process

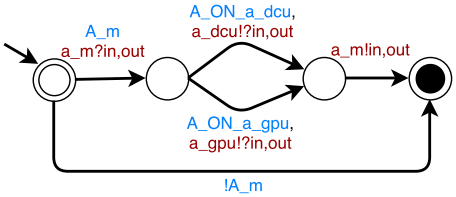
As our form of behavioral product lines is based on FTS [8], model checking techniques can be directly reused. In our implementation (cf. next section), we reuse the ProVeLines checker as a back-end for the validation process. The process consists in verifying all execution paths of all products $[\![fa]\!]_{FA}$ of the product line, in an efficient way by exploiting commonalities between different products. Theoretically, the more the products share common behavior and the more



(a) Mapping FD



(b) data-path Mapping p_1 FA



(c) Task Mapping A FA

Figure 6: Partial Variability-Aware Mapping FA

efficient should be the variability aware model checking in comparison of iterative model checking on individual systems [7]. Instead of exploring all executions for each system implementation, the model-checker explores an execution π once for all implementations P able to produce this specific execution:

$$P = \{p \in [\![d]\!]_{FD} | \pi \in [\![fa_{|p}]\!]_A\}.$$

As mentioned in the previous section, some system configurations may expose inconsistent behaviors (e.g., memory allocation error, violation of graphical pipeline constraints). These behaviors will abort the execution and the basic properties (e.g. safety, absence of deadlock, state reachability) will obviously not be satisfied. In our validation process, we are able to remove these configurations from the system by relying again on the back-end model checker [7]. It computes the set of bad product configurations, which we remove from the feature diagram of the product line by adding the appropriate cross-tree constraints.

4. VALIDATION

4.1 Implementation

4.1.1 Overview

The framework depicted in Fig. 3 has been entirely implemented in Java. It consists of 3 main modules: i) meta-models of variable application and configurable platform (cf. Fig.7 and 8) ii) mapping meta-model Fig.9 and algorithm (cf. listing 3) iii) generators that transform the design space

composed by all system sub-domains (application, platform, mapping) (cf. listing 4) into formal models of behavioral product line (cf. listings 5 and 6) in order to remove invalid products (cf. listing 8) reusing automated formal reasoning techniques (cf. listing 7).

The first module allows for specifying a variable data-flow oriented application (cf. listing 1) and a configurable platform (cf. listing 2) via fluent APIs. As a result, the running example inputs are captured in less that 30 lines of code. The second module calls our mapping algorithm (cf. listing 3) in order to infer, at the end, the resulting design space. The third and last module transforms the design space into a *Feature Model* in *TVL* (cf. listing 5) and a *Featured Automaton* in *fPromela* (cf. listing 6), capturing, respectively the structural variability and behavior of the design space.

We reuse the ProVeLines model-checker [9, 6], which consumes *TVL* and *fPromela* inputs to assess all system designs, in one run of variability-aware model checking. The resulting outputs, printed as a set of invalid sub-products lines (c.f listing 7), are directly used to constraint the design space to only obtain valid products (cf. listing 8).

4.1.2 Applications as Variable Data-Flows

Listing 1 illustrates how we capture the functional requirements (cf. fig. 1) of the embedded system through an extended data-flow Java API. The data-flow meta-model (cf. Fig. 7) contains the classic structure and behavior of the data-flow (data instanced at line 3,6, task at line 4,5,7, data-path at line 2), but also captures the variability in both structural properties (e.g., data size at line 6) and behavioral properties (e.g., alternative flows by allowing data-paths to have multiple input and output tasks connected at line 4,5).

Listing 1: Running Example Application

```
1  Application app = new Application("WarpWithWhat");
2  Path p1 = app.addPath("P1"); Path p2 = ...; Path p3
       = ...;
3  DataSource d1 =
       app.addDataSource("D1").addSize(512).connect("o",
       p1);
4  Task ta = app.addTask("ta", "A").connect(p1,
       "i").connect("o", p2);
5  Task tb = app.addTask("tb", "B").connect(p1,
       "i").connect("o", p2);
6  DataSource d2 =
       app.addDataSource("D2").addSizes(512,
       1024).connect("o", p3);
7  Task tc = app.addTask("tc", "C").connect(p2,
       "i0").connect(p3, "i1");
8  app.split(p1).to(ta).to(tb);
9  app.join(p2).from(ta).from(tb);
```

4.1.3 Platforms as Variable Resource Graphs

Listing 2 illustrates how we express the platform specification (cf. fig. 2) of the embedded system through a resource component based Java API. The platform meta-model (cf. Fig. 8) contains templated resource components such as multi-pass processors instanced at line 13 and streaming processor at line 6. Other elements in the template can be hardware functions instanced at line 7,10,14, read-only memory at line 2, read-write RAM memory at line 3, first-in-first-out buffers at line 8,14, relevant elements being connected with each others. In addition, a platform can have optional resource components (line 4,13) and variability dependency (line 15) on resources.

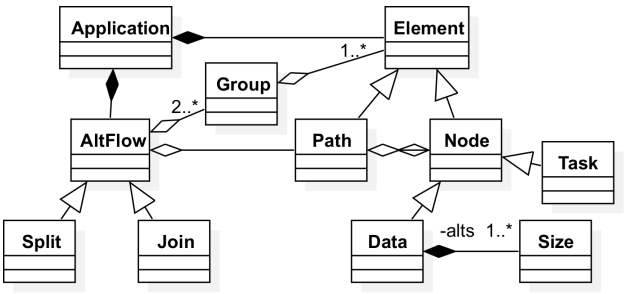


Figure 7: Application Metamodel

Listing 2: Running Example Platform

```
1  Platform plt = new Platform("Kepler");
2  Storage rom = plt.addStorage("ROM",
       Type.READ_ONLY).addCapacity(4096);
3  Storage ram = plt.addStorage("RAM",
       Type.READ_AND_WRITE).addCapacities(1024, 2048);
4  ram.setOptional();
5
6  Component dcu = plt.addComponent("DCU");
7  Processor a_dcu = dcu.addProcessor("a", "A");
8  Memory r0_dcu = dcu.addFIFOBuffer("RO");
9  a_dcu.connectToInputPort("i", ram,
       rom).connectToOutputPort("o", r0_dcu);
10 Processor c_dcu = dcu.addProcessor("c", "C");
11 c_dcu.connectToInputPort("i0", ram, rom,
       r0_dcu).connect("i1", ram, rom);
12
13 Component gpu =
       plt.addComponent("GPU").setOptional();
14 Processor a_gpu = ...;Memory r0_gpu = ...;Processor
       b_gpu = ...;
15 gpu.requires(ram);
```

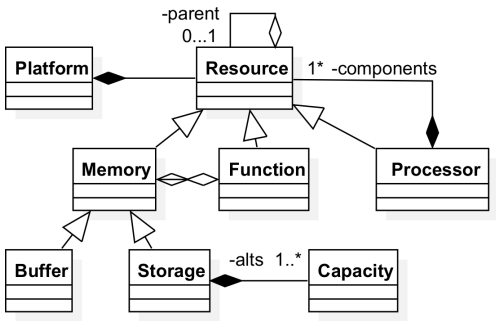


Figure 8: Platform Metamodel

4.1.4 Variability-Aware Mapping Process

The mapping algorithm (cf. listing 3) takes as inputs the variable data-flow and configurable platform Java models, and generates the *Variability-Aware Mapping Space*(cf. Fig. 9 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 constraints to ensure mapping space consistency (line 13).

Listing 3: 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     }
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 }
```

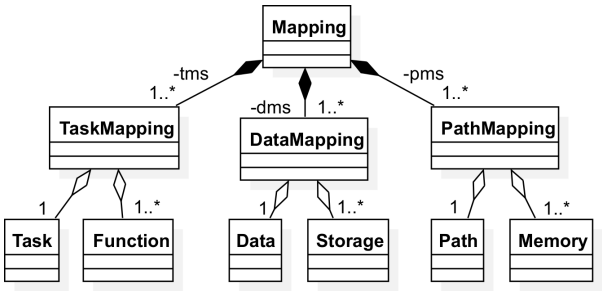


Figure 9: Mapping Metamodel

4.1.5 *Design Space as a Behavioral Product Line*

From the system design space model (cf.listing 4), which is the consistent composition of our 3 system sub-domains (i.e., application, mapping, platform), a *Behavioral Product Line* representing all system implementations is generated into a *Feature Automaton* (cf.listing 4). While the behavior of the whole product line is encoded in *fPromela* (cf. listing 6), a *Feature Model* in *TVL* encodes its structural variability (cf.listing 5).

Our framework relies on Featured-Transition-Systems (FTS) to formally reason on the structure and behavior of the design space. FTS has the strength to model explicitly the variability points structurally, through a Feature Diagram [3] *(FD)* in *TVL*(cf.listing 5).

Thus, a single variability model is able to capture data size at line 9-12, memory capacity at line 19-22, data mapping 28-31, task mapping 32-35, alternative flow 9-12,13,14 variabilities, resource optionality 18,24, resource dependency 45, mapping 41 and design space consistency constraints 39, 43. On the other hand, the behavior of the design space is captured by an executable network of featured automata in *fPromela* where state transitions are guarded by constraints on feature values.

To execute the variable application over the configurable platform, featured automata that capture behavior of data-flow processes, such as data node (lines 18-34) and task node (lines 36-52), call functions over platform resource featured automata, such as memory storage (line 2-16) and processor.

Each featured automaton may have variable properties such as capacity for memory storage process (lines 8-11), data size (lines 21-24) and data deployment location (lines 25-30) for data node process, or task deployment (line 42-47) for task process. In addition to properties, a featured automaton may be optional (i.e., behavior is executed if the element is present in the system design cf. lines 39-50).

According to the subset of design decisions to explore, variable properties are incrementally fixed. For example, for system designs where $D2$ with a size of 1024 is deployed on RAM with a capacity of 1024, we observe that after allocating $D2$ on RAM (line 26,6), RAM is full, and any other data allocation on RAM (e.g., $D1$ or data of $P2$) would lead to a memory violation (cf. listing 7 line 4).

Listing 4: Generate Running Example Formal Models from Design Space

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

Listing 5: Part of Running Example Design Space variability in TVL

```
1  root DesignSpaceVariability{
2     group allOf{
3        ApplicationVariability group allOf{
4           ...
5           P1_to group oneOf{
6              P1_to_TA,
7              P1_to_TB
8           },
9           D2_size group oneOf{
10             D2_size_512,
11             D2_size_1024
12          },
13          opt TA,
14          opt TB
15       }
16       group PlatformVariability{
17          ...
18          opt RAM group allOf{
19             RAM_size group allOf{
20                RAM_size_1024,
21                RAM_size_2048
22             }
23          },
24          opt GPU
25       }
26       group MappingVariability{
27          ...
28          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
   ...
   D2_On_RAM => RAM;
   ...
   GPU => RAM;
   ...
}
```

Listing 6: Part of Running Example Design Space behavior in fPromela

```
...
active proctype Storage_RAM(){ atomic{
   Data in;
   ...
   do
   :: RAMalloc?in ->
       cons = cons + in.size;
       if
       :: RAM_capacity_1024 -> size = 1024;
       :: RAM_capacity_2048 -> size = 2048;
       fi;
       assert(cons <= size);
       RAMalloc!in;
   od;
   ...
};}
...
active proctype Data_D2(){ atomic{
   Data out;
   ...
   if
   :: D2_size_512 -> out.size = 512;
   :: D2_size_1024 -> out.size = 1024;
   fi;
   if
   :: D2_On_RAM -> RAMalloc!out;
                   RAMalloc?eval(out);
   :: D2_On_ROM -> ROMalloc!out;
                   ROMalloc?eval(out);
   fi;
   ...
   P3!out;
   ...
};}
...
active proctype Task_TA(){ atomic{
   Data in, out;
   ...
   if
   :: TA -> P1?in
       ...
       if
       :: TA_On_GPU_a -> GPU_a!in, out;
           ...
       :: TA_On_DCU_a -> DCU_a!in, out;
           ...
       fi;
       ...
   :: else -> skip;
   fi;
   ...
};}
...
```

4.1.6 *Validation Process*

The generated formal models (*fPromela* and *TVL*) are checked through *ProVeLines* with specific command lines (cf.listing 7 line 1). It returns output containing non feasible subsets of products (line 4) that are used to invalid variants by constraining the design space variability space (cf. listing 8 line 7). More precisely, the model checking is going to verify products against inconsistent behaviors (e.g., memory allocation error, violation of graphical pipeline constraints) and more classic properties, such as safety, absence of deadlock and state reachability on all variants in one run. This single execution [7] makes also possible to remove all products (cf.listing 7 at line 4) leading to an invalid execution , so to improve and speed-up the verification process (see next Section).

Listing 7: Part of Running Exemple ProVeLines output

```
.\provelines -check -exhaustive -nt
    running_example.pml
...
assertion failled : (assert(cons <= size);) at
    line... for products:
   D2_size_1024 && D2_On_RAM && RAM_capacity_1024
       && D1_On_RAM && P1_On_RAM
...
```

Listing 8: Part of Valid DesignSpace 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);
   ...
}
```

4.2 Evaluation

4.2.1 *Industrial Use Case*

In order to validate our tooled approach on an industrial scale, we applied it to a real low-end market instrument cluster provided by Visteon, the automotive systems company we collaborate with.

The functional requirements of the cluster represents a variable data-flow with 3 source images processed by 8 tasks connected by 9 data-paths. Each source image has two different resolutions (i.e HD and LD) and two tasks sub flow sequences are alternative through a *xor* join/split data-path,

Table 1: Industrial use case results

Variability	Implementation variants	Platform variants	data-flow variants	States explored (re-explored)	Time (ms)	ms / variants	states / variants
NONE	78	0	0	2453 (331)	27	0.346	31.448
Data size	624	0	8	15254 (2406)	201	0.322	35.976
Platform	424	24	0	5673 (546)	65	0.153	13.379
Platform and data size	3392	24	8	37435 (4856)	602	0.177	11.036
data-path	150	0	2	4727 (981)	74	0.493	31.513
data-path and data size	1200	0	16	29066 (6994)	587	0,489	24,222
ALL	>4800	24	16	72704 (14652)	2361	-	-
Platform mult. mem	16408	40	0	134941 (4534)	4010	0.244	8.224
Platform mult. proc	2848	80	0	19625 (3224)	337	0.118	6.890
Pltf. mult. proc & mem	516608	160	0	1341999 (175304)	289721	0.560	2.597

resulting in 16 data-flow variants. The platform specification of the cluster is then represented by a variable hardware component system with 2 memories (a Video RAM and a ROM Flash) and 3 processors (two multi-pass GPU bitblitter and one streaming- based DCU). Each processor has a pipeline processing of 4 stages and 3 fifo buffers. In terms of platform variability, the 2 bitblitters and the VRAM are optional. Each memory has 2 different configurable sizes at manufacturing time. The number of platform configurations in the use case is then 24.

If each data-flow variant had one possible implementation on each platform configuration, the number of different cluster system implementations would be 384. In reality, some platform configurations do not provide the graphical functionalities required by some data-flow variants. Furthermore, due to multiple task implementation choices, data-flow variants have several thousand possible implementation alternatives onto a platform configuration. Setting the platform configuration to the higher end (i.e. selecting VRAM and all processors), one can find 72 and 78 possible implementations for two data-flow variants that take different xor data-path decisions. This is due to more pipelining opportunities in the second data-flow variant, even if there is more data-path mapping possibilities in the first one.

Table 1 shows time measurements of the complete toolchain while varying the different variability dimensions over the use case. In the first seven rows, we observe that the whole process is performing well with small to medium scale of variabilities. Data and memory size variability verifications are faster and require more state exploration than platform component and data-path variabilities. Component and data-path variabilities are also slower to check than data and memory size variabilities. It is likely to be due to the fact that contrary to size variability, hardware component and data-path variability are strongly impacting the implementation variability, and consequently the state space of the model checker.

We have also complemented this experimentation by taking a single structural data-flow from the industrial use case with a simulated larger platform, itself with multiple memories and processors. Results in the last three rows of Table 1 show that solving can scale to a large number of implementation variants. Even if the solving time is significant, we observe that the number of states explored to assess all the implementation variants is significantly low. This shows that behavioral commonalities between system variants are used to speed-up the verification process.

4.2.2 Scalability of System Variabilities

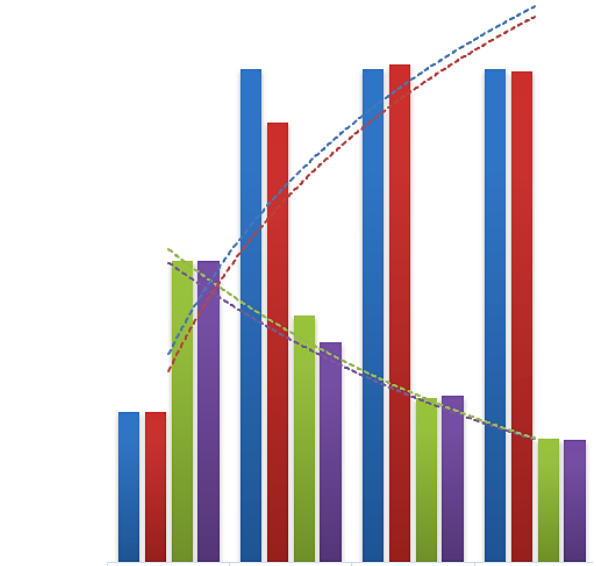
We now analyze the scalability of our solution against system variability by increasing the application, platform and mapping variability on simulated data. Fig. 10.a shows measurements of our toolchain for data size and memory capacities variability dimensions (called element variability) while Fig. 10.b is about mapping dimensions. Each variable system denotes his variability dimensions in the following format :

$$flow; size; resource; capacity; mapping$$

Where $flow$ represents the sum of flow variants of variable data-paths, $size$ the sum of alternatives data sizes, $resource$ the number of optional resources, $capacity$ the sum of alternatives memory capacities and $mapping$ the sum of alternatives mapping of application element onto platform resources.

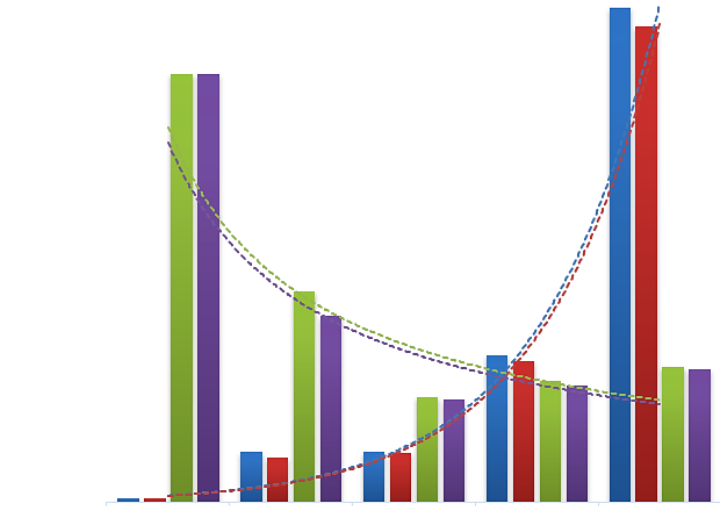
Fig. 10.a shows a system where data size and memory capacities variability dimensions have been progressively increased so that the first and last system counts, respectively, 384 and 6912 variants. We show relative time and (automaton) states metrics – in total and per variants – compared to the normalized system presenting the lowest variability. Thus, for element variability dimensions (data size and memory capacity), even if, obviously, the time and states number needed to verify the system increase according to variability dimensions, the verification time and states number needed by variant decrease.

For the scalability of the mapping variability dimension, we mainly increase progressively the mapping dimension over 5 systems, the lowest system containing 64 variants while the highest has 23328. We observe that the needed time and explored states number grow quickly. We think this is due to the intrinsic high complexity of both binding and scheduling [22], which leads to configuration space and state space explosion during checking. However, the verification speed-up by variant is still interesting for high variability system.



	0;2;0;0;4	0;2;0;4;4	0;3;0;4;4	0;4;0;4;4
states	1,00	3,27	3,28	3,28
time	1,00	2,92	3,30	3,26
states/variant	2,00	1,64	1,09	0,82
time/variant	2,00	1,46	1,10	0,82

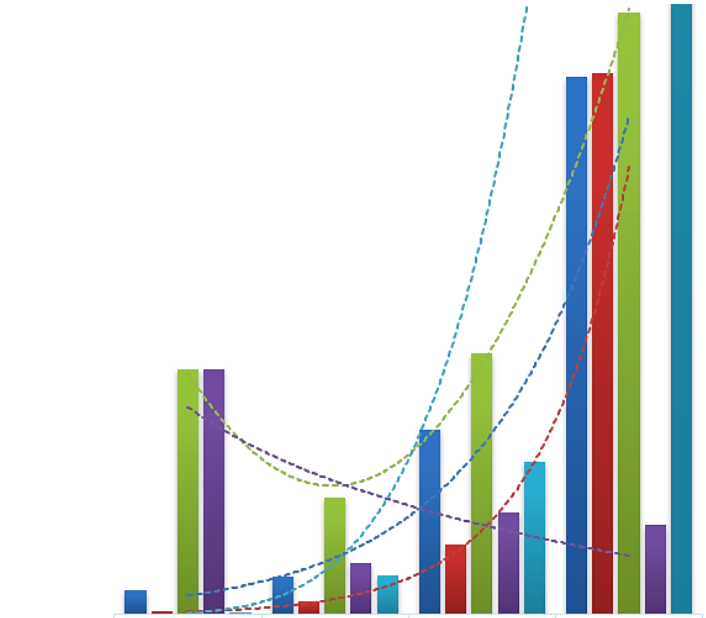
(a) Element Variability



	0;0;0;0;2	0;2;0;4;4	0;2;0;4;6	0;3;0;6;8	0;3;0;6;10
states	1,00	11,84	11,81	34,42	115,32
time	1,00	10,43	11,58	33,08	110,98
states/variant	100,00	49,35	24,60	28,33	31,64
time/variant	100,00	43,48	24,12	27,23	31,19

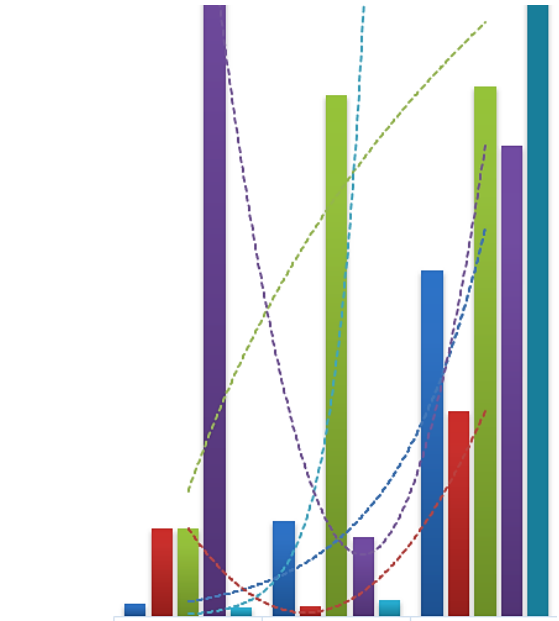
(b) Mapping Variability

Figure 10: Variability Analysis



	0;0;0;0;16	0;0;0;0;22	0;0;0;0;27	0;0;0;0;28
states	10,00	15,57	75,09	218,94
time	1,50	5,24	28,79	220,40
states/variant	100,00	47,65	106,61	244,89
time/variant	100,00	21,24	41,71	36,55
Complexity	1,00	16,12	62,18	1490,27

(a) Mapping Complexity



	4;0;0;3;12	4;0;0;2;12	4;0;0;3;14
states	1,50	10,82	39,43
time	10,00	1,17	23,33
states/variant	10,00	59,38	60,33
time/variant	100,00	9,01	53,55
Complexity	1,00	1,93	13018,36

(b) Flow Complexity

Figure 11: Complexity Analysis

4.2.3 Scalability of System Complexities

We analyze here the scalability of our solution against system complexity by growing the application and platform size. Fig. 11.a shows an example where we increased progressively the size and complexity of both application and platform sides while keeping the mapping variability at a common factor (i.e., %40).

The normalized system presenting the lowest complexity and variability. It contains 14 application nodes and paths mapped on a 10 resources platform with 48 variants, while the most complex system has 22 nodes, 13 resources, and 2880 variants. A global system *complexity* metric, taking into account the system size and it's number of variants, has been discussed with our industrial partner.

We observe that the verification time and states needed increase quickly according to the system complexity (application and platform sizes). Interestingly, as the needed states and time to explore per variant in more complex while the system is growing, the verification time per variant is still lower. This observation can also be made on Fig. 11.b, where we progressively increase the complexity of a system with flow variability.

4.3 Threats to Validity

The validation on a single case study can be considered as the major internal threat to validity. Nonetheless it was incrementally built by numerous meetings with different domain experts of the automotive systems company. We gathered specification and feedback from several applications and platforms case studies, and we finally chose the presented one as the most representative among them.

As for external threats, we identified the data sets as the main issue. While the data sets are large, they are still simulated. Our creation procedure has been built to mimic the structure and behavior of both the platforms and the applications, taking the real case studies as a basis to be expanded by generators. Still we do not have empirical evidence of their correspondence.

5. CONCLUSION AND FUTURE WORKS

A tremendous amount of variability can be observed in embedded systems, and especially in data-flow oriented ones, which are now systematically built from highly variable specifications and target diverse hardware platforms configurable at a very high level of detail. To handle the early functional assessment of all these possible configurations, we proposed in this paper a tooled approach that takes variable data-flow specifications and variable hardware platform models to map them together and transform them into a behavioral product line representing the potential design space. These models and toolchain allow to use automated reasoning techniques to explore and assess the functional feasibility of all represented variants in a single run, and invalid products can be removed by adding constraints to the product line.

We reported on the application of the proposed approach to a real-world industrial use case of automotive instrument cluster, giving hints on a potential good applicability. Our experimental validation with large simulated datasets also shows a good scalability of the prototype implementation for industrial-scale applications and platforms.

As future work, we first plan to facilitate the usage of the framework with domain specific languages for input models (specification and platform), and to conduct larger experiments with them on different and new case studies from our industrial partner. Interestingly, we think that some product lines optimization techniques [11, 23] could be applied to assess more variable and complex embedded systems.

We will also extend our variability-focused framework by taking into account quality attributes (e.g. cost, run-time etc.). The extension would then provide optimized product selection as a complement to the functional validation presented in this paper. We expect this more complete framework to be applicable in different contexts, being similar in the separation of application models being mapped onto component-based platforms.

6. ACKNOWLEDGMENTS

We thank Visteon Electronics and the *Association Nationale de la Recherche et de la Téchnologie* for continuously supporting this research. We thank also Emmanuel Roncoroni and Olivier Bantiche who brought industrial expertise in instrument cluster engineering and Maxime Cordy for his valuable support on the ProVeLines model-checker.

7. REFERENCES

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

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

[3] D. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, volume 3714, pages 7–20. Springer, 2005.

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

[5] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[6] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Model checking software product lines with snip. *Int. Journal on Software Tools for Technology Transfer (STTT)*, pages 1–24, 2012.

[7] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 321–330. ACM, 2011.

[8] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. 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, 2010.

[9] M. Cordy, A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. 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, 2013.

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

[11] A. S. Dimovski, A. S. Al-Sibahi, C. Brabrand, and A. Wąsowski. Efficient family-based model checking via variability abstractions. *International Journal on Software Tools for Technology Transfer*, 19(5):585–603, 2017.

[12] S. Graf, M. Glaß, J. Teich, and C. Lauer. 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, 2014.

[13] S. Graf, M. Glaß, D. Wintermann, J. Teich, and C. Lauer. 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, 2013.

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

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

[16] B. Kienhuis, E. Deprettere, K. Vissers, and P. Van Der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *Application-Specific Systems, Architectures and Processors, 1997. Proceedings., IEEE International Conference on*, pages 338–349. IEEE, 1997.

[17] G. Palermo, C. Silvano, and V. Zaccaria. 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, 2008.

[18] G. Palermo, C. Silvano, and V. Zaccaria. Variability-aware robust design space exploration of chip multiprocessor architectures. In *Design Automation Conference, ASP-DAC. Asia and South Pacific*, pages 323–328. IEEE, 2009.

[19] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele. 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, 2012.

[20] K. Sigdel, M. Thompson, A. D. Pimentel, C. Galuzzi, and K. Bertels. System-level runtime mapping exploration of reconfigurable architectures. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[21] V.-M. Sima and K. Bertels. 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, 2009.

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

[23] P. Temple, J. A. Galindo, M. Acher, and J.-M. Jézéquel. Using machine learning to infer constraints for product lines. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 209–218. ACM, 2016.

[24] M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. 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, 2016.

[25] P. Van Stralen and A. Pimentel. Scenario-based design space exploration of mpsocs. In *Computer Design (ICCD), 2010 IEEE International Conference on*, pages 305–312. IEEE, 2010.

[26] S. Wildermann, F. Reimann, J. Teich, and Z. Salcic. Operational mode exploration for reconfigurable systems with multiple applications. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1–8. IEEE, 2011.