Multifaceted Automated Analyses for Variability-Intensive Embedded Systems
Sami Lazreg, Maxime Cordy, Philippe Collet, Patrick Heymans, Sébastien Mosser

To cite this version:
Sami Lazreg, Maxime Cordy, Philippe Collet, Patrick Heymans, Sébastien Mosser. Multifaceted Automated Analyses for Variability-Intensive Embedded Systems. 41st ACM/IEEE International Conference on Software Engineering, May 2019, Montréal, Canada. hal-02061251

HAL Id: hal-02061251
https://hal.archives-ouvertes.fr/hal-02061251
Submitted on 8 Mar 2019

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.
Multifaceted Automated Analyses for Variability-Intensive Embedded Systems

Sami Lazreg
Visteon Electronics
Université Côte d’Azur
CNRS, I3S, France

Maxime Cordy*
SnT, University of Luxembourg
Université Côte d’Azur
Luxembourg

Philippe Collet
Université Côte d’Azur
CNRS, I3S, France

Patrick Heymans
University of Namur
Belgium

Sébastien Mosser
Université Côte d’Azur
CNRS, I3S, France

Abstract—Embedded systems, like those found in the automotive domain, must comply with stringent functional and non-functional requirements. To fulfill these requirements, engineers are confronted with a plethora of design alternatives both at the software and hardware level, out of which they must select the optimal solution wrt. possibly-antagonistic quality attributes (e.g. cost of manufacturing vs. speed of execution). We propose a model-driven framework to assist engineers in this choice. It captures high-level specifications of the system in the form of variable dataflows and configurable hardware platforms. A mapping algorithm then derives the design space, i.e. the set of compatible pairs of application and platform variants, and a verification algorithm then pinpoints the optimal system variants efficiently. The benefits of our approach are evaluated through a real-world case study from the automotive industry.

I. INTRODUCTION

In many embedded systems, requirements engineering and design activities are tightly intertwined and involve complex multi-criteria decision-making over various concerns. As an example, let us consider an entertainment feature in an automotive system. Specifying such a feature typically entails defining a set of functional and non-functional (aka. quality) requirements. Functional requirements define, for example, not only what content should be displayed through the Human Machine Interface (HMI), but also constraints imposed by the hardware, like not exceeding the available memory. Typical examples of non-functional requirements constrain manufacturing costs or execution time. A notoriously difficult problem is to establish whether such a set of requirements is feasible and what is the best design to implement it.

Design options are not only constrained by the requirements but also by the existing software and hardware architectures. In our case, an HMI-rendering automotive system consists of (1) a data processing application (i.e., data-flow oriented embedded software) and (2) a resource-constrained hardware platform (i.e., heterogeneous hardware components like non-programmable processors and data storage units). The application and the platform are, however, not completely fixed as they have variation points. There are three main sources of extensive variability. First, at the application level, multiple data-flow variants can achieve the functional requirements, differing in, e.g., the size of the flowing data chunks, the ordering of the operation tasks, or the choice between alternative, functionally-equivalent tasks. Second, there exists a diversity of configurable hardware platforms which can differ, e.g., in memory capacities and processing pipelines. Third, there are various ways of mapping and deploying a given application on a specific platform, e.g., choose a processor to perform a given task or select a memory unit to store a given data.

This threefold variability is typical in automotive and many other kinds of embedded systems [1]. Unfortunately, it leads to a high number of variants (1,548,288 in this particular case), each of which represents a specific system design alternative (or design for short), that is, a specific mapping of a specific application variant to a specific platform variant. Among these design alternatives, not all are able to realize the functional requirements to the same extent, and the same holds for the non-functional requirements. Given the sheer number of variants, a systematic consideration of all design alternatives is unfeasible for the software and system engineers whereas the high level of competition in industry puts a high pressure on them to deliver optimal solutions and do so timely [2]. Efficient automations therefore appear as a necessity.

Examples of questions the engineers need to quickly answer are: Can the specified HMI be properly rendered on platform X? Which feasible designs can be built with a budget of Y? Which feasible designs can execute in less than Z time? Which feasible designs, with a rendering quality higher than P and a manufacturing cost lower than Q, exhibit the fastest execution time? Which feasible designs reach the best tradeoff between rendering quality, manufacturing cost and execution time? Answering those questions not only requires a way to deal with the variability-induced combinatorial explosion (see previous paragraph), but also a way to reason simultaneously about different types of concerns: feasibility/satisfiability and optimality; functional and non-functional requirements; the structure and the behaviour of the system. Although significant progress was made in the recent years to automate reasoning on variability-intensive systems, existing solutions only address specific facets of the problem in isolation. As revealed by our experiments, partial solutions give suboptimal designs, while complete but non-variability-aware solutions do not scale. Hence the need for multifaceted, variability-aware analyses capable of answering all the above questions.

* This work was done while Maxime Cordy was part of the U. of Namur.
Without such solutions, engineers mostly resort to intuition and experience. Theoretical analyses can be made but are time-consuming and often turn out largely suboptimal, if not completely wrong. Simulators are sometimes provided by platform suppliers but analyzing all system variants requires simulations for all of them, which is unrealistic. Quick prototyping may occur when the platform is finally supplied but it is then too late to backtrack if the wrong platform was picked. In the end, current practices are deemed very unsatisfactory.

These observations were made by our partner Visteon Electronics, an international leader in automotive systems, and are corroborated by surveys such as [3]. They formed the motivation for the industry-academia partnership which led to this research effort. In this paper, we propose an approach that combines and extends existing research results. The industrial need is identified by observing the car manufacturers’ variable requirements for an entire range of cars. To be competitive, they are highly constrained on quality and cost. As such, one must know as early as possible in the development whether the expected HMI can be rendered properly on a candidate platform.

To achieve economies of scale, such systems are often built and sold to car manufacturers as product lines, which have to meet and optimize the manufacturers’ variable requirements for an entire range of cars. To be competitive, they are highly constrained on quality and cost. As such, one must know as early as possible in the development whether the expected HMI can be rendered properly on a candidate platform.

At industrial scale, the threefold variability introduced in Sec. I (i.e. from application, platform and mapping) prevents any product-based exhaustive feasibility checking, let alone exhaustive reasoning/optimization on quality attributes (e.g. cost, rendering quality, runtime). Yet, some separation of concerns is intrinsically possible as data-flows can be abstracted from the implementation by domain experts (i.e. rendering engineers). Platforms are already specified by hardware experts to organize competitive tendering on hardware providers.

Fig. 2 shows an example of data-flow specification with quality attributes. The different processing flows that meet the HMI functional requirements are captured by a variable data-flow. Images are processed by graphical tasks. Image D1 can be processed by tasks A or B. D2 has three different possible resolutions and is processed by task D. The images produced by A (or B) and D are then processed by task C, which delivers the final result. Task and image resolution impacts the HMI rendering quality. In our case, performing B instead of A significantly improves the rendering quality. Also, as the resolution of D2 is increased, the overall quality raises as well.

At the platform level (Fig. 3), image processing functions A, B, C, D are provided by a non-programmable Graphical Processing Unit (GPU) and an advanced Display Controller Unit (DCU). Within DCU, there are three functions (A, C and D) appear on an electronic visual display (see Fig. 1). By applying various data-flow image processing effects (e.g., 3D gauges, 3D view of the car), it improves the driving experience.
Manufacturing cost and rendering quality are quality attributes which depend only on the structure of the design (e.g. size of input data, choice between alternative tasks, components of the platform). Execution time, however, depends on both structure (e.g. processor frequency) and behaviour (e.g. scheduling of tasks and memory access operations).

Those requirements are not enough, though, as market competition forces engineers to deliver the best system to each specific customer. Among the variants, they must thus find those offering the best trade-off between the quality attributes. Achieving our goal thus requires solving the problem of efficiently identifying the variants that:

1) **Satisfy the FC requirements.** We further decompose this sub-problem in two challenges: (a) checking the FC requirements that depend only on the structure of the design (challenge FCS) and (b) checking FC requirements that depend also on its behaviour (FCB).

2) **Satisfy the NF requirements**, that is, checking: (a) those that depend only on the structure (NFS) and (b) those that also depend on the behaviour (NFB).

3) **Optimize the trade-off between the quality attributes (NFOB).** This challenge requires considering all quality attributes. Optimizing only those that depend on the structure (NFOS) is easier, but leads to suboptimal solutions (as revealed by our evaluations).

We must solve all those challenges to give engineers the means of making appropriate design decisions at an early stage.

III. STATE OF THE ART AND RELATED WORK

Many approaches were proposed to tackle parts of the above challenges. First, research in variability modeling is black-box oriented and focused on structural aspects. It attempts to assess [6], [7] or efficiently predict [8]–[13] non-functional properties of the whole product line, thus tackling both FCS and NFS. Extensions of these methods with multi-objective optimization [14]–[19] allow to find optimal variants wrt. structural quality attributes, thereby solving NFOS. However, those approaches lack reasoning support on the system behaviour and are thus unable to, e.g., search for optimal scheduled.

Behavioural analyses of variable systems were addressed by variability-aware model checking against functional requirements (challenge FCB) [20]–[23] or one particular non-functional aspect (e.g. real time [24]–[26], reliability [27], [28], income [19], quality of service [29]). Resource-optimal execution [30]–[33] and worst-case execution time [34] are still determined product by product. Also, contrary to our framework, these approaches are unable to automatically map variable workflow specifications on configurable platform descriptions in order to infer a system design space. Therefore, applying them obliges to manually model and assess all possible systems designs. They thus inefficiently tackle NFOB. Even when both levels are captured in software/hardware product lines with dependencies and constraints [35]–[40], the expressiveness is also insufficient, because either behaviour is not considered or only functional requirements are checked.
Some system design frameworks [4], [41]–[48], model, assess and optimize quality attributes (NFB and NFOB), but do not capture nor manage all variability dimensions. Specific techniques attempt to efficiently handle either platform variability [49]–[51], application variability [52]–[55] or deployment variability [56], but they are limited to one dimension at a time and cannot reason on the whole problem.

Other approaches [57]–[59] tackle both functional and platform variabilities by focusing on an optimal platform configuration for a multi-variant application. Contrarily to our work, they do not find a design (i.e., a mapping of a functional variant onto a platform configuration) that ensures an optimal execution. Our recent work [23] can capture and reason on variability at all three levels. It is, however, limited to functional requirements (challenges FCS and FCB).

IV. OVERVIEW OF THE FRAMEWORK

Our approach follows the model-driven Y-Chart process [4], [5] which explicitly separates application and platform. This allows for the modular specification of, and the reasoning on, the different parts of the system. As shown in Fig. 4, our framework involves several models and processes. On the left, the inputs are application and platform models such as those of Fig. 2 and Fig. 3. Domain experts are expected to model the application as an extended concurrent data-flow model with quality attributes. The model contains the classic structure and behaviour of the data-flow (data, task, edge) and its variability. Additionally, platform experts provide the platform specification as a templated concurrent component-based system, which also captures platform variability.

First process (detailed in Sec. V): from the input models the framework generates a variability-intensive design space that captures all valid deployment mappings of the variable application onto the configurable platform. A mapping basically allocates tasks to processors and data to memory components.

Second process (detailed in Sec. VI): from the design space, we generate representations that allow for reasoning on the structure and the behaviour of all designs. In addition to the input models, experts define the NF requirements as constraints and a cost function representing trade-offs between quality attributes. To relate NF requirements with the design space, we rely on feature models with quality attributes [10], [14] for the structural part, and on featured transition systems [22] with added weights [60] for the behavioural part.

Third process (detailed in Sec. VII): we reuse feature-model reasoning and apply our new model-checking algorithms to identify the designs that meet the requirements (FCS, FCB, NFS and NFB) and are optimal (NFOB). Additionally, we can provide the execution traces that optimize the quality attributes while satisfying the requirements. Such a trace shows not only the designs able to execute it but also the behaviour it exhibits (i.e., how the application tasks are executed and scheduled onto the platform), thereby helping the upcoming engineering of the designs.

V. APPLICATIONS, PLATFORMS AND MAPPING

To model the application, the platform and their quality attributes, we extend our former modelling framework [23] to support NF requirements, as it is limited to checking designs that satisfy the FC requirements. Thus, we define formal models for variable dataflows and configurable platforms.

Definition 1: A variable data-flow graph is a tuple \( VDG = (N, P, E, \Psi, \chi_d) \) where \( N = T \cup D \) is a set of nodes (\( T \) are tasks and \( D \) are data); \( P \) is a set of communication paths by which data flows between producers (tasks and data) and consumers (tasks); \( E \subseteq N \times P \times T \) is the set of flow processing between producers and consumers via available paths; \( \Psi \) is a set of attributes; \( \chi_d : N \rightarrow (\Psi \rightarrow \bigcup_{\psi \in \Psi} \nu(\psi)) \rightarrow \{true, false\} \) is a function that associates a node \( n \) to the set of values that (all or a subset of) the attributes in \( \Psi \) can take, where \( \nu(\psi) \) is the finite set of values that \( \psi \) can take. Intuitively, these graphs encode two forms of variability. The first consists of variations in the data flow, as we allow data paths to have multiple connected producers and consumers; this follows the same approach as in variable workflows [57]. The second lies in the alternative attribute values that a node can take. For example, consider again Fig. 2. We see that that datum \( D2 \) can have three size values. This corresponds to a design variation of the application (i.e. the size of the processed data). On the contrary, the quality of \( D2 \) represents the impact of data size on the overall quality of the system, which is also determined by the quality value of \( B \). Thus, the overall quality depends on (i) the size of \( D2 \) and (ii) whether \( A \) or \( B \) consumes \( D1 \). 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 the paper, without loss of generality: one can use instead other aggregation functions (e.g. average, maximum).

Our definition of \( \chi_d \) allows one to define cross-cutting constraints over the attribute values of a given node \( n \). For instance, the quality value of \( D2 \) is directly linked to its size property: a size of 256 leads to a quality of 0, 512 to 1, and 1024 to 3. We see that \( \chi_d(n) \) defines which valuations of the attributes are valid altogether. This flexible definition, akin to the notion of configuration of non-boolean parameters [61]–[65], can express that some attribute values are forbidden in \( n \), and that the value of given attributes in \( n \) restricts the values of the others.

Definition 2: A variable resource graph is a tuple \( VRG = (R, C, \Theta, \Psi, \chi_r) \) where \( R = P \cup S \) is a set of resources (\( P \) are processors and \( S \) are memory storage); \( C \subseteq (S \times P) \cup (P \times S) \) is a set of connections between processors and storage, where \((p, s) \in C \) (resp. \( s, p \in C \)) means that processor \( p \) writes from (resp. reads to) storage \( s \); \( \Theta : 2^R \rightarrow \{true, false\} \) encodes which subsets of resources constitute a valid platform configuration; \( \Psi \) is a set of attributes; \( \chi_r : R \rightarrow (\Psi \rightarrow \bigcup_{\psi \in \Psi} \nu(\psi)) \rightarrow \{true, false\} \) associates a resource \( r \) to the set of values that the attributes in \( \Psi \) can take.

The function \( \chi_r \) is defined similarly to its counterpart in variable data-flow graphs and offers the same benefits.
Another source of variability encoded by this model comes from the optional nature of resources, that is, two variants of the platform may differ by their constituent resources. Θ symbolically encodes which subsets of resources constitute a valid platform in the form of Boolean constraints.

The third and last part of our formal model is the mapping rules that must be satisfied to deploy any given application variant on any given platform variant. These rules are automatically generated from the variable data-flow graph and the variable resource graph. A mapping allocates each task to at least one processor function, each datum to at least one memory storage, and each path to at least one processor function, each datum to at least one memory storage, and each path to at least one buffer or memory storage. A valid mapping must ensure that the required data can be read and written by the hardware functions using them. For each path p, producer node i and consumer task o of p, there must exist a buffer or storage b associated to p, a function f_i associated to i, and a function f_o associated to o, such that f_i can write to b and f_o can read from b.

Altogether, the two graphs and their mapping rules constitute the *variability-intensive design space*, as they define the set of the design alternatives that can result from a valid deployment of a variant of the application onto a configuration of the platform.

**VI. ENABLING REASONING ON THE DESIGN SPACE**

From the variability-intensive design space, we generate intermediate representations to reason on the structure and the behaviour of all variants and on their quality attributes.

In product-line engineering [66], structural variability is commonly addressed by modelling *features* and their interdependencies as a *feature model* (FM) [67]. Since we consider NF requirements, we use an extension of FMs where features have quality attributes [10], [14], [68], which we name *Priced Feature Model* (PFM). Reasoning on the PFM allows for determining which design variants satisfy the requirements that depend on the structure of the design (FCS and NFS).

**Definition 3**: A PFM is a tuple \( pfm = (F, Q, Ψ_p, η, Ψ_r) \) where \( F \) is a set of Boolean features; \( Q \) is a set of positive real-valued quality attributes; \( Ψ_p \) is a set of cross-cutting constraints over \( F \) defining which subsets of features form a valid variant; \( η : F \times Q \rightarrow \mathbb{R}_0^+ \) is a function defining how each \( f \in F \) changes the value of each \( q \in Q \); \( Ψ_r \) is a set of constraints over \( Q \) defining what are the valid aggregated values for a given attribute \( q \). The semantics of a PFM is a partial function \( \langle pfm \rangle : 2^F \rightarrow \mathbb{R}_0^+ \) such that (i) \( \langle pfm \rangle \) is defined only for the valid variants: \( F' \in dom(\langle pfm \rangle) \Rightarrow F' \models Ψ_p \); (ii) the aggregated value of each attribute \( q \) satisfies the constraints: \( \forall q \in Q : \eta(F', q) \models Ψ_τ \); (iii) each attribute is associated with its aggregated value: \( \langle pfm \rangle(F', q) = \eta(F', q) \) where \( \eta(F', q) \) is the aggregated value of \( q \) in the variant represented by \( F' \subseteq F \).

An excerpt of PFM is shown in Fig.5. Each variation point that may occur in the application (e.g., consumer of \( P1 \) is \( A \) or \( B \)), platform (e.g., RAM is present or not) or mapping (e.g., \( D1 \) can be stored in ROM or RAM) gives rise to an optional feature in the PFM. The attributes representing design variations (e.g. the size of \( D2 \)) are transformed into alternative features, while those representing quality attributes (e.g. cost of manufacturing) become the PFM’s quality attributes. \( Ψ_τ \) results directly from the mapping rules (3a and 3b), the presence conditions \( Θ \) between resources (5), consistency rules requiring that alternative application node or optional resources used in the mapping become mandatory (1,2,4), and constraints over attribute values encoded in \( χ_d \) and \( χ_r \). \( Ψ_τ \) correspond to the NF requirements defined by the engineers. Finally, \( η \) is obtained from \( χ_d \) and \( χ_r \).

To check requirements that depend on behaviour, we need to systematically evaluate all executions of all design variants. A common approach to achieve this for a single design is to model-check an automaton modelling the system behaviour [69], [70], which results from the scheduling [71] of an application automaton on a platform automaton. These approaches, however, cannot handle variability and are thus limited to one system at a time. Accordingly, we propose to generate, from our variability-intensive design space, a *Features Weighted Automaton* (FWA) [60] – a recent formalism that combines featured transition systems [22] with priced automata [72]. Basically, FWA is a variability-aware, weighted extension of finite state machines where each transition \( t \) is annotated with a formula defining (i) which variants can execute \( t \); and (ii) conditional weight values that depend on the variant executing \( t \). This formula, named *weighted feature
expression, allows one to represent compactly the evolution of quality attributes as the execution of the variants progresses.

Definition 4: Let $F$ be a set of features and $\{\tau_1 \ldots \tau_n\}$ an ordered set of $n$ positive real-valued weights. Then a weighted feature expression is a (possibly partial) function $\gamma : 2^F \rightarrow \mathbb{R}_{\geq 0}$ that associates a variant $F' \subseteq F$ with a vector $w$ such that $w_i$ is the value of the $i$-th weight for $F'$.

Intuitively, the weighted feature expression $\gamma$ associated to a given transition $t$ is such that a variant $F'$ belongs to $\text{dom}(\gamma)$ if and only if $F'$ can execute $t$, and $\gamma(t)$ returns the value of $t$’s weights for $F'$. In our case, each weight corresponds to a distinct quality attribute and represents the value added to this attribute when $F'$ executes the transition $t$.

Definition 5: Let $T = \{\tau_1 \ldots \tau_n\}$ be a set of positive real-valued, quality attributes and $\text{pfm} = (F, Q, \Psi_p, \eta, \Psi_r)$ be a PFM, such that $Q \subseteq T$. A FWA over $T$ and $\text{pfm}$ is a tuple $\text{fwa} = (S, s_0, s_f, T, \gamma \rightarrow)$ where $S$ is a set of states, $s_0$ is the initial state, $s_f$ is the final state, $T \subseteq S \times S$ is the transition relation, and $\gamma \rightarrow : T \rightarrow 2^F \rightarrow \mathbb{R}_{\geq 0}$ associates each transition with a weighted feature expression.

A FWA defines, for each variant $F' \subseteq F$, the set of paths starting from $s_0$ to $s_f$ that $F'$ can execute, together with the weight vector associated to each path: $\forall F' \subseteq F: [[\text{fwa}]](F') = \{ (p, w) \in S^+ \times \mathbb{R}_{\geq 0}^n : p = s_0 \rightarrow \cdots \rightarrow s_k = s_f \text{ and such that (i) } p \text{ exists in } \text{fwa}: \forall j : 0 \leq j < k : (s_j, s_{j+1}) \in T; (ii) \text{ all transitions of } p \text{ are executable by } F'; F' \in \bigcap_{j=0}^{k-1} \text{dom}(\gamma \rightarrow(s_j, s_{j+1})); (iii) w \text{ is the sum over all associated weight vectors: } w = \sum_{j=0}^{k-1} \gamma \rightarrow(s_j, s_{j+1}) \}.$

Note that we assume all quality attribute values to be positive. Negative values are supported modulo transformation. In PFM, one can replace negative values on a feature by positive values on the alternative features. In FWA, one can replace negative values on a transition by positive values on the alternative transitions.

From the design space, we generate a network of FWAs, where each source datum, task, processor and memory storage corresponds to a distinct FWA whose states and transitions encode the different steps of their process. Data automata implement the process of sending the data onto a task automaton, which itself transmits the data to the automaton of the processor that provides the function whereon the task is mapped. The automaton of a processor models standard processing pipeline behaviour [23]. During this processing, the processor can store input and output data by transferring them to its dedicated storage automaton.

An excerpt of FWA modelling storage behaviour is shown in Fig. 6. It models memory read and write via input/output interfaces while checking that memory capacity is never exceeded. Weighted feature expressions occur in the transitions of the hardware resources automata (i.e. processors and memories) that correspond to processing instructions and memory accesses, as these operations impact the “execution time” quality attribute. In the excerpt, such an expression is shown in pink and models that the increase of time depends on RAM frequency. The automaton modelling all the designs results from the parallel composition of the individual automata, which can synchronize their transitions on specific synchronization actions (e.g. the transfer action in Fig. 6 sends data only when another FWA executes the corresponding action).

Our FWA differs from that of Fahrenberg et al. [60]: (i) ours is linked to a PFM to also check structural requirements; (ii) we support parallel composition and consider time as a special weight in that we execute fastest actions first and dynamically update the remaining time delay of other parallel actions to simulate time elapsing; (iii) Fahrenberg et al. [60] represents $\gamma \rightarrow$ as a partition of the set of variants because their verification algorithm – which is not tooled – relies on matrix representations, whereas ours relies on antichains [73] to scale wrt. the large state space incurred by our evaluation cases.
VII. ALL-IN-ONE VERIFICATION AND OPTIMIZATION

Our FWA formalism allows us to design a first algorithm to solve all challenges FCS, FCB, NFS and NFS for all variants at once. The key idea is to perform an exploration of the state space of the automaton in search of paths that can reach the final state while satisfying the FC and NF requirements.

The first step filters out variants that do not satisfy PFM constraints \(\Psi_p\) and \(\Psi_r\), thereby ensuring that the structure of the variants does not violate the requirements. Then we explore all paths starting from the initial state. 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 and for all variants and assert that these values satisfy the NF requirements. In the end, we obtain a set of paths going from the initial to the final state, together with, for each path \(p\), (a) the valid variants that can execute \(p\) and (b) the values of the quality attributes corresponding to \(p\) and each of those variants.

This first algorithm finds all variants satisfying the requirements. We have to find what are the optimal variants satisfying the requirements while providing the best values for the quality attributes. Since these attributes can be antagonistic (e.g., manufacturing costs can decrease to the detriment of rendering quality), this is assimilated to multi-objective optimization.

To drive our search for optimal, we define a cost function over the attributes: let \(\zeta(\tau_1, \ldots, \tau_n) = \theta_1 \times \tau_1 + \cdots + \theta_n \times \tau_n\) be our cost function, such that \(\theta_i \in \mathbb{R}^+\) is the coefficient associated to the attribute \(\tau_i\). Then, our objective is to discover the variant that minimizes \(\zeta\). This is achieved by modifying our exploration algorithm in order to (i) record the optimal variants and their associated attribute values, and (ii) stop exploring a path as soon as all quality attributes reach a worse (i.e. higher) value. This latter heuristics requires the cost function to be monotonic (i.e., there is no element in \(R\)) 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 attributes. This rule ensures that infinite cycles are avoided.

Algorithm 1 details this exploration procedure. It takes as input a FWA, a PFM and a cost function \(\zeta\). It iteratively computes \(R\), the reachability relation that associates each state \(s\) to the \(\gamma\) function encoding all variants that can reach \(s\) and their associated attribute values. At first (Line 1), \(R\) contains \(s_0\) together with \(\gamma_0\), such that \(\text{dom}(\gamma_0) = \text{dom}(\text{p}fm)\) and \(\gamma_0(F^i, \tau_i) = 0\) for any variant \(F^i\) and attribute \(\tau_i\). Then, we start the exploration from \(s_0\) (L3–4) and iterate over the states encountered successively (L5–20). At each iteration, we retrieve the state \(s\) reached last, together with its associated \(\gamma\) function (L6). Here, \(\gamma\) encodes the variants that can reach \(s\) and associates to each variant the values of its attribute values when following the path that led to \(s\). If all these variants yield a value for \(\zeta\) greater than the current optimum \(\zeta^*\) (L7), we do not pursue the exploration further from this state. Otherwise, we distinguish between the cases where \(s\) is \(s_f\) (L8–10) and where it is not (L11–18). In the first case, we assign \(\zeta^*\) to the minimal cost over all variants that can reach the final state. In the second case, we compute the set of successors of \((s, \gamma)\) (L12), given by \(\text{Post}(s, \gamma) = \{(s', \gamma')|(s, s') \in T \land \gamma' = \gamma \land \gamma' \rightarrow (s, s')\}\) where \(\text{dom}(\gamma_1 \otimes \gamma_2) = \text{dom}(\gamma_1) \cap \text{dom}(\gamma_2)\) and \(\forall F' \in \text{dom}(\gamma_1 \otimes \gamma_2) : (\gamma_1 \otimes \gamma_2)(F') = \gamma_1(F') + \gamma_2(F')\). This means that \(\gamma'\) is defined only for the variants that can reach \(s\) and execute the transition from \(s\) to \(s'\), and it sums the attribute values of each variant in \(\gamma\) with its attribute values on the transition \((s, s')\).

We add a successor if 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 attributes. This rule ensures that infinite cycles are avoided. To do so, we use the comparison operator \(\succeq\) over weighted feature expressions, defined as \(\gamma_1 \succeq \gamma_2 \equiv \forall F' \in \text{dom}(\gamma_1) : (\gamma_1)(F') \geq (\gamma_2)(F')\). 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 [24], [60], which we do not detail due to lack of space. Finally, we return the set of variants \(F^*\) that can reach \(s_f\) while minimizing \(\zeta\), together with the optimal cost (L21–22).

VIII. IMPLEMENTATION AND EVALUATION

A. Implementation

We implemented our framework as a toolchain combining a new Java tool with (extensions of) existing model checkers.
Our Java tool\(^1\) consists of two modules, each of which implements a process depicted in Fig. 4. The first one allows for specifying a variable data-flow graph and a variable resource graph via a fluent API. Then, it calls our mapping algorithm to generate the design space. Second, an automata generator takes as inputs the design space, NF requirements and the cost function to generate an FWA with its associated PFM. Two concrete syntaxes are used, as we can then invoke two independent model checkers to search for optima.

One is UPPAAL-CORA\(^2\), an established tool to carry out cost-optimal reachability analyses that we reuse as is with optimal settings. It takes as input a network of Linearly Priced Timed Automata (LPTA)\(^3\). LPTA can be regarded as FWA without variability, and as such can only encode the behaviour of the variants separately. Our automata generator actually transforms our design space into a network of LPTA in the UPPAAL-CORA format. It also generates 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\(^4\). Additionally we use SPLOT’s feature model reasoning library\(^5\) to restrict the configuration to valid products. 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\(^6\), which can check variability-intensive systems. We chose this tool because it was extended over the years, by both its original developers\(^7\) and others\(^8\),\(^9\),\(^10\), to solve multiple model-checking problems including real-time verification\(^11\). We fully implemented Alg. 1 in a new version of ProVeLines. To achieve this, we first extended ProVeLines’ input language – Promela\(^12\) – 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 format supported by ProVeLines, viz. TVL\(^13\), \(^14\), \(^15\).

**B. Qualitative Evaluation**

Our first evaluation assesses the usefulness of our approach for practitioners on the basis of Visteon’s instrument cluster system (see Sec. II). More precisely, our evaluation concerns an important module of the whole system. Yet, it remains a real-world case that was selected by our partner as representative in terms of size, complexity and variability.

With the assistance of an expert engineer, we reverse-modelled the variable application and platform of the existing system. Some technical simplifications were made as we aim at facilitating early design decisions: we do not consider data and parameters that only have minor impact on the system’s runtime; we abstract away from data content and consider data sizes as the most influential factor for runtime performance; we do not model mechanisms like internal cache replacement policies, AXI bus 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 types, four 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 generated mapping rules reduced this number to 1,878. By adding the NF requirements rendering quality $\geq 2$ and manufacturing cost $\leq 280$, we further diminish this number to 894. By checking the behaviour against the requirements (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.

We performed the behavioural analyses using both UPPAAL-CORA and ProVeLines, which provided the same results for all variants. This increases our confidence that the transformation of the design space into LPTA and Promela is consistent. Expert’s confirmation is also needed, though, as mistakes may originate from the input models themselves. The expert validated that the optima returned by our tools conform to the very best designs that the company could produce over the past years. The quality attributes’ values also corresponded to what is expected. Regarding execution time (cycles), there are slight differences due to hardware modelling simplifications; however, the numbers are close to reality (< 10% difference) and the relative orders between variants are preserved. In the end, the expert validated that our approach helps make optimal design decisions that will provide significant gains at all stages of development.

**C. Quantitative Evaluations**

The second part of our evaluation focuses on the efficiency and the scalability of our approach in terms of execution time. This is indeed essential, as the total number of variants can be high in real-world systems. In addition to the instrument cluster case study (numbered case #0), we constituted a dataset of realistic topologies that were generated based on our partner’s history. To do so, our model generator relies on multivariate Gaussian distributions whose parameters were settled on the basis of Visteon’s past systems. Thereby, it ensures that the characteristics of our generated data-flow and platform models are similar to real-world cases. Amongst 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 valid variants to check). We carried out three series of experiments

---

\(^1\)https://bitbucket.org/SamiLazreg/enlighter

\(^2\)https://bitbucket.org/maxcordy/provelines-cora

\(^3\)Linearily Priced Timed Automata

\(^4\)150% model approach

\(^5\)SPLOT’s feature model reasoning library

\(^6\)ProVeLines

\(^7\)Original developers

\(^8\)Others

\(^9\)19

\(^10\)29

\(^11\)Real-time verification

\(^12\)Promela

\(^13\)TVL

\(^14\)https://bitbucket.org/maxcordy/provelines-cora
presented hereunder. Table I provides the results, such that the results of the different series of experiments are separated by double borders. All benchmarks were run on a MacBook Pro 2014 with a 2.8 GHz Intel Core i7 processor and 16 GB of DDR3 RAM. We repeated each experiment ten times and computed the average, although random deviations were low.

**Product-based vs. family-based.** Our first experiments evaluate the efficiency of our method to verify that *all* variants satisfy the requirements, that is, we consider only the four challenges FCS, FCB, NFS and NFB. We compare the runtime of our family-based verification algorithm with an alternative, product-based one that checks each variant separately. Both algorithms are implemented in ProVeLines, which allows us to compare both approaches on an equal technological ground.

The results are presented on the left part of Table I. It depicts, for each approach and model, the time needed to check all variants as well as the total number of states that were 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, the family-based method outperforms the product-based one, reducing the verification time from 22.60 seconds to 3.33. The generated models allow us to observe that the benefits of the family-based method 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). On the contrary, 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 see 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, while family-based algorithms generally perform better as variants share more commonalities. This also explains the poor performance of our algorithm in case #3.

**Optimization in ProVeLines and UPPAAL-CORA.** Our second experiment compares the efficiency of the two model checkers to solve challenge NFOB, that is, we compare UPPAAL-CORA cost-optimal reachability algorithm against our Algorithm 1. The two tools present clear differences, notably in history (UPPAAL-CORA’s development started in the early 2000’s, while ProVeLines was released in 2013), in focus (cost-optimal reachability in continuous-time models vs. family-based model checking in discrete-time models) and in input syntax (LPTA vs. Promela). Still, we believe this comparison can highlight interesting research directions.

Results are given at the centre of Table I. Like any product-by-product approach, UPPAAL-CORA suffers from every increase in the number of variants, except for cases #4 and #6 where it luckily alleviates complexity with branch-and-bound optimizations. It systematically performs poorer than ProVeLines. Even worse, apart from cases #0–2, #4 and #5, UPPAAL-CORA consumes too much memory (> 16GB) and raises a fatal error. We assumed two reasons behind this. First, UPPAAL-CORA does not implement partial-order reduction [80] and thus considers all possible interleavings between LPTAs, including during their configuration. This creates an exponential blow-up as the number of LPTAs increases. Second, we use a model that encodes the behaviour of all variants 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. Process interleaving is therefore responsible. One way to circumvent this is to assign priorities over the automata. However, in most cases, this will cause to miss better scheduling opportunities and will yield suboptimal results. In spite of the disappointing results for UPPAAL-CORA, the fact that it outperforms product-based ProVeLines on case #0 tends to indicate that combining our family-based algorithm with UPPAAL-CORA’s efficient search heuristics can be a promising future work.

Through these experiments, we also observe that looking only for optima in ProVeLines, 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

<table>
<thead>
<tr>
<th>TABLE I: Results for the three quantitative evaluations. Times are in seconds.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Ins. Ct. (#0)</td>
</tr>
<tr>
<td>Gen. #1</td>
</tr>
<tr>
<td>Gen. #2</td>
</tr>
<tr>
<td>Gen. #3</td>
</tr>
<tr>
<td>Gen. #4</td>
</tr>
<tr>
<td>Gen. #5</td>
</tr>
<tr>
<td>Gen. #6</td>
</tr>
<tr>
<td>Gen. #7</td>
</tr>
<tr>
<td>Gen. #8</td>
</tr>
<tr>
<td>Gen. #9</td>
</tr>
<tr>
<td>Gen. #10</td>
</tr>
<tr>
<td>Gen. #11</td>
</tr>
</tbody>
</table>
product-based algorithm used in the first series of experiments. In this regard, our Algorithm 1 exacerbates the benefits of a family-based algorithm. For the models with the most variants (#9–11), Algorithm 1 is 5 to 128 times faster than the product-based method. Interestingly, it seems to be way less affected by the number of variants than the all-variant verifications.

Prioritization based on structural optima. The last part of our evaluation studies whether the structural optima (i.e. the solutions of NFOS) are sufficiently close to the overall optima (i.e. the solutions of NFOB). If that is the case, solving NFOS would yield an imperfect but relevant, cost-effective solution, as structural optimality can be resolved statically, that is, without requiring the exploration of a large state space. We use the ClaferMOO tool [14], [81] to compute the structural optima based on our PFMs. To complete the toolchain, we extended it to generate PFMs in ClaferMOO’s format, as well as our verifier module to retrieve the variants that are structural optima and only allow these variants by adding a constraint in the PFM. Then, we use ProVeLines in family-based mode to assess the time needed to discover which structural optima lead to the lowest cost function value, as well as the difference between this value and that of the overall optimum.

Results are given on the right side of Table I. The first two columns give the number of structural optima returned by ClaferMOO, as well as the time needed to run ProVeLines only on those structural optima. The last column gives the increase of cost function, in percentage, due to considering only structural optima instead of all variants. In cases #1–3, #5 and #7, the variants differ only by their behaviour and are thus all structural optima. For the other cases, we see that ProVeLines’ runtime is dramatically decreased as it has to check fewer variants. This, however, comes at the cost of a notable increase in cost function value (up to 96%), except for case #8 where it increases by less than 2%. We conclude that structural optima are still far from being overall optima, and thus are not sufficient. Yet, in some cases they can constitute viable solutions to get a quick answer or when the state space of the system is too large to be exhaustively explored.

D. Threats to Validity

Internal. The main threat to internal validity is the selection of the used case study. Although it was built from specification and feedback meetings with system and platform experts from our partner company, it is a single case. Still, Visteon chose it as being representative in terms of shape and complexity of the data flow, the platform and their variability.

External. Several threats to external validity exist. First, our scalability experiments have been conducted over large but simulated data sets. We nevertheless expect the creation procedures to respect the structure and behaviour of the potential applications and platforms of our industrial partner. More generally, we also expect the proposed approach to be applicable in other domains where data flows can be an appropriate modelling support, like stateless enterprise processes with micro-services. On the platform side, the component-based representation seems general enough to cover different forms of deployment architectures, but we do not have any evidence yet of this generalization possibility.

Second, satisfiability and optimality checking processes depend highly on the level of detail and on the quality of application and platform modelling. On the platform side, a design not detailed enough could lead to loss of relevance in the performance metrics, and could even produce false optimal results. From the feedback we got from our industrial partner, the level of detail is currently sufficient as it is at the level they use to specify the platform when selecting suppliers. Still, in other contexts, some details may be hidden due to intellectual property management. As for applications, they are described with data-flows that are well mastered by domain 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 good organization of existing software. More globally, as the engineers were able to understand proposed designs that result from our framework, machine learning technique fed by their feedback on good or bad solutions could help when details are missing. Introducing this in the framework remains an open issue.

IX. Conclusion

In many data-flow oriented embedded systems, three levels of variability significantly increase the complexity of the design space: variable high-level data-flows are deployed in many different ways over highly configurable component-based hardware platforms. We provided a modelling and reasoning framework that unifies state-of-the-art techniques on structural reasoning with a novel variability-aware model-checking algorithm, which evaluates the functional feasibility, the non-functional satisfiability and optimality on the whole design space. This design space comes as a mapping between two other models: one represents the variable applications with a data-flow model complemented with quality attributes, the other models platform variability through connected components, also with quality attributes. We showed how the design space is transformed into featured priced timed automata to enable different reasoning and optimization operations.

The application of the proposed approach to a medium-scale industrial case of an automotive instrument cluster demonstrates its end-to-end ability to check and optimize a complex design space. System experts revealed that, even on small data flow and platform models, the optimal designs were non-trivial to identify for them, giving us confidence on the relevance of the approach. Experiments on large simulated design spaces also show that the prototyped toolchain exhibits good scalability and outperforms non-variability-aware solutions.

We believe that the models used in the framework (data-flows and components) make the contribution potentially applicable in different contexts. Our future works will start with an extension of the framework to facilitate its usage, introducing domain-specific languages for input models. We also plan to conduct larger evaluations on different data flows coming from various product lines of our industry partners.