The Journal of Supercomputing (2023) 79:2180–2211
https://doi.org/10.1007/s11227-022-04741-8

Toward a context-driven deployment optimization for embedded systems: a product line approach

Abdelhakim Baouya[1] · Otmane Ait Mohamed[2] · Samir Ouchani[3]

Accepted: 21 July 2022 / Published online: 8 August 2022

Abstract
Producing a large family of resource-constrained multi-processing systems on chips (MPSoC) is challenging, and the existing techniques are generally geared toward a single product. When they are leveraged for a variety of products, they are expensive and complex. Further in the industry, a considerable lack of analysis support at the architectural level induces a strong dependency on the experiences and preferences of the designer. This paper proposes a formal foundation and analysis of MPSoC product lines based on a featured transition system (FTS) to express the variety of products. First, features diagrams are selected to model MPSoC product lines, which facilitate capturing its semantics as FTS. To this end, the probabilistic model checker verifies the resulting FTS that is decorated with tasks characteristics and processors' failure probability. The experimental results indicate that the formal approach offers quantitative results on the relevant product that optimizes resource usage when exploring the product family.

Keywords Product derivation · Model checking · Reliability · Product usage contexts · MPSoC

✉ Abdelhakim Baouya
abdelhakim.baouya@univ-grenoble-alpes.fr; abdelhakim.baouya@gmail.com

Otmane Ait Mohamed
otmane.aitmohamed@concordia.ca

Samir Ouchani
souchani@cesi.fr

[1] Université Grenoble Alpes, VERIMAG, Grenoble, France

[2] ECE Department, Concordia University, Montreal, Canada

[3] Ecole d'Ingénieur CESI, Aix-en-Provence, France

1 Introduction

Efficient utilization of computation components in Multi-Processing System-On-Chip (MPSoC) is still a primary issue. Design-Space Exploration (DSE) techniques [46] have been proposed to automatically explore the driven alternative architectures by various system quality attributes (i.e., communication traffic, power, energy), and reporting the near-optimal ones. This daunting process is handled at different levels of abstraction in the design flow [27], and its efficiency relies on the used technique to evaluate the design point at that abstraction level [20, 41]. *Architectural* DSE, referred to as deployment [3] independently to physical structures, usually focuses on the overall system design and its quality attributes (i.e., the number of processing units, allocation). In contrast, *micro-architecture* DSE focused on the internal component architecture space (i.e., bus arbiter policy, processor architecture). However, although the evaluation tools at a low level can perform a cycle-accurate analysis [5, 43], it is still challenging to exploit them due to (i) a large number of options and configuration parameters values and (ii) the prohibitively high total run-time.

Based on the Product Line (PL) [8] philosophy, Feature Diagrams (FDs) are a de facto standard where diverse platform variants are identified upfront, and a model of their differences and commonalities is created. Based on the hardware platform configuration provided by Meedeniya et al. [35], the feature diagram for the MPSoC PL shown in Fig. 1 depicts all possible MPSoC configurations (called "products"). Also, features selection for a particular product is not made arbitrarily. The product usage contexts often dictate features selection. The notion of context variability is introduced by [21] and [47] to identify the features selection context.

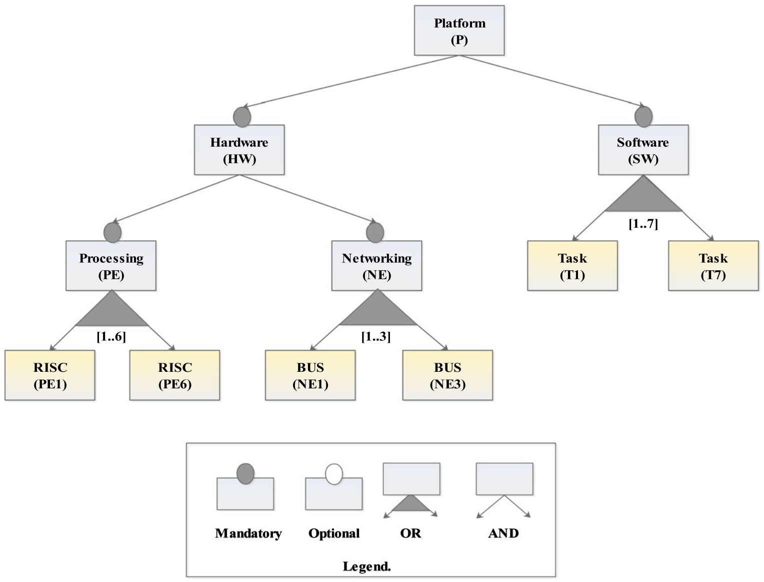


Fig. 1 System feature diagram of the MPSoC PL

In this paper, the MPSoC product line is divided into two kinds of lines: *system* and *context*. The former portrays the hardware platform features, whereas the latter describes the conditional assembly. The study shines a spotlight on the decision-making at the architectural level by using probabilistic model checking to produce a reliable product configuration that optimizes tasks assignment and prunes all possible design alternatives that do not meet given constraints.

1.1 Variability-intensive system

According to Van Gurp et al. [48], "`Variability`" is commonly understood as the ability of a software system or software artifacts (i.e., components, modules) to be adapted so that they fit a specific context. Building on the definition of variability, Variability-intensive System [34] is a system where variability and related challenges contribute essential influences to software analysis, design, implementation, and evolution. When designed and appropriately implemented, variability-intensive systems can significantly improve development costs, speed, and quality compared to developing and maintaining single products. The variability-intensive system includes product lines, configurable or customizable single systems [17], context-aware mobile applications [32], or OSGi[1] bundles.

1.2 Problem statement

By relying on the deployment studied in the state-of-the-art [26, 31, 38, 51]. All of them state the lack of deployment efficiency in an MPSOC. Sengupta et al. [44] and Conrady et al. [13] rely mainly on meta-heuristic approaches, especially Genetic algorithms, to deploy a processing element. Still, unfortunately, the process requires considerable processing time and memory capacities to come up with only an approximate solution. While other approaches [26, 31] focus more on the low-level view of MPSOC, however, they do not consider the different layers of MPSOC like drivers and other embedded software.

1.3 Current challenges

A *product* of the MPSoC product line is specified by features encompassing a subset of processing and networking elements. A literature search [49] revealed how to model products by means of transitions system (TS) as shown in Fig. 3a, b. Both TS are produced randomly and do not rely on some selection criteria, the user selects the physical unit (e.g., transition $(\mathtt{s} = 1) \xrightarrow{\mathtt{PE1}} (\mathtt{s} = 2)$ in Fig. 3a) modeled as states transition and then makes the deployment (e.g., transition $(\mathtt{s} = 5) \xrightarrow{\mathtt{DeployPE3}} (\mathtt{s} = 6)$ in Fig. 3a) modeled also as states transition. The satisfaction is based on the user preference. The first variant in Fig. 3a contains two processors and one bus, whereas the second one, the platform is constituted of three processors and two buses (i.e.,

[1] Open Services Gateway initiative.

we refer by $\mathtt{PR_i}$ to processors, $\mathtt{NE_j}$ to buses, and $\mathtt{T_i}$ to tasks). No logic is applied for selection criteria. Although the problem related to MPSoC variability-space exploration is addressed in several works Mendis et al. [36], Xie et al. [51], Malazgirt and Yurdakul [31], none has provided accurate means for checking variability-intensive MPSoC over temporal properties concerning its usage context.

1.4 Contributions

The paper relies on the formalism of the Featured Transition System (FTS) proposed by Classen et al. [11] that captures all products at a glance. The model is fed to a probabilistic model checker that provides critical insights on the optimal deployment, such as the number of allocated processing elements, while considering reliability as an objective. The model is enhanced with quality metrics related to the operational profile of product components, such as the processing speed and data sizes that are requested to be served by the processing elements.

Our approach is composed of two main phases as portrayed in Fig. 4—FTS construction and quantitative assessment—and validation phase. In the first phase, we combine the system, the context's feature diagram, the task's graph, and the components' reliability obtained from quality metrics [37] to build the FTS in the PRISM input language. The output of this phase is checked by STORM tool [14], a model checker that offers better performance than traditional symbolic model checkers. In particular, we identified a set of properties that can be expressed in a probabilistic fashion (i.e., in PCTL) to address the following question: "How reliable is the mapping of software tasks on an MPSoC platform." The validation phase considers tasks and hardware platform to perform simulation using the SCoPE tool that opens perspectives of integration with the TASTE tool [2]. A list of acronyms used in the rest of the article is given in Table 1. In a nutshell, we summarize the main contribution of our work.

- Providing the main concepts needed to understand MPSOC components and feature diagrams.
- Formalizing the deployment problem in MPSOC in an understandable and easy way.
- Presenting the theoretical foundation of PTS and FTS as well as describing their semantics in PRISM.
- Developing an approach that runs three phases (construction, verification, validation) to check the correct and precise deployment of the different components in MPSoC.
- Experimenting and validating our developed approach on a real and complex use case.

The paper is organized as follows. Section 2 reviews the preliminaries, and Sect. 3 introduces the needed concepts related to our reference architecture. Then, Sect. 4 develops our approach regarding tasks assignment and the FTS construction and verification. As an application, a case study from an automotive area in

Sect. 5 demonstrates the efficacy of the proposed approach. Sections 6 and 7 present an overview of the related work and concludes the paper, respectively.

2 Preliminaries

This section provides the required concepts forming the basics of our contribution.

2.1 Probabilistic transition system

Probabilistic Transition Systems (PTSs) [18] are a modeling formalism that extends classical Transition Systems (TSs) to exhibit probabilistic and nondeterministic features. Definition 1 formally illustrates a PTS where $\mathtt{Dist(S)}$ denotes the set of convex distributions over the set of states S and μ= [..., $\mathtt{s_i} \mapsto \mathtt{p_i}$, ...] is a distribution in $\mathtt{Dist(S)}$ that assigns a probability $\mu(\mathtt{s_i}) = \mathtt{p_i}$ to the state $\mathtt{s_i} \in \mathtt{S}$.

Definition 1 (Probabilistic Transition System) A Probabilistic transition system is a tuple $\mathtt{M} = \langle \bar{\mathtt{s}}, \mathtt{S}, \mathtt{L}, \Sigma , \delta\rangle$:

- $\bar{\mathtt{s}}$ is an initial state, such that $\bar{\mathtt{s}} \in \mathtt{S}$,
- $\mathtt{S}$ is a set of states,
- $\mathtt{L} : \mathtt{S} \to 2^{\mathtt{AP}}$ is a labeling function that assigns each state $\mathtt{s} \in \mathtt{S}$ to a set of atomic propositions taken from the set of atomic propositions ($\mathtt{AP}$),
- Σ is a finite set of actions,
- $\delta\colon \mathtt{S} \times \Sigma \to \mathtt{Dist(S)}$ is a probabilistic transition function assigning for each $\mathtt{s} \in \mathtt{S}$ and $\alpha \in \Sigma$ a probabilistic distribution $\mu \in \mathtt{Dist(S)}$.

For PTS's composition, this concept is modeled by the parallel composition as stipulated in Definition 2. During synchronization, each PTS resolves its probabilistic choice independently. For transitions, $\mathtt{s_1} \xrightarrow{\alpha} \mu_1$ and $\mathtt{s_2} \xrightarrow{\alpha} \mu_2$ that synchronize in α then the composed state $(\mathtt{s'_1}, \mathtt{s'_2})$ is reached from the state $(\mathtt{s_1}, \mathtt{s_2})$ with probability $(\mu_1(\mathtt{s'_1}) \times \mu_2(\mathtt{s'_2}))$. In the no synchronization case, a PTS takes a transition where the other remains in its current state with probability one.

Definition 2 (*Parallel composition*) The parallel composition of two PTSs: $\mathtt{M_1} = \langle \bar{\mathtt{s}}_1, \mathtt{S_1}, \mathtt{L_1}, \Sigma_1, \delta_1\rangle$ and $\mathtt{M_2} = \langle \bar{\mathtt{s}}_2, \mathtt{S_2}, \mathtt{L_2}, \Sigma_2, \delta_2\rangle$ is a PTS $\mathtt{M} = \langle (\bar{\mathtt{s}}_1, \bar{\mathtt{s}}_2), \mathtt{S_1} \times \mathtt{S_2}, \mathtt{L_{s_1}} \cup \mathtt{L_{s_2}}, \Sigma_1 \cup \Sigma_2, \delta\rangle$ where: $\delta\colon \mathtt{S_1} \times \mathtt{S_2}, \Sigma_1 \cup \Sigma_2$ is a set of transitions $(\mathtt{s_1}, \mathtt{s_2}) \xrightarrow{\alpha} \mu_1 \times \mu_2$ such that one of the following requirements is met.

1. $\mathtt{s_1} \xrightarrow{\alpha} \mu_1, \mathtt{s_2} \xrightarrow{\alpha} \mu_2$, and $\alpha \in \Sigma_1 \cap \Sigma_2$,
2. $\mathtt{s_1} \xrightarrow{\alpha} \mu_1, \mu_2 = [\mathtt{s_2} \mapsto 1]$, and $\alpha \in \Sigma_1 \setminus \Sigma_2$,
3. $\mu_1 = [\mathtt{s_1} \mapsto 1], \mathtt{s_2} \xrightarrow{\alpha} \mu_2$, and $\alpha \in \Sigma_2 \setminus \Sigma_1$.

Example 1 To illustrate the applicability of PTS to model dependability, we rely on the case study presented in [28]. The system comprises a processor (M) which reads and processes data from three sensors ($\mathtt{S}_1$, $\mathtt{S}_2$ and $\mathtt{S}_3$) and uses them to control two actuators ($\mathtt{A}_1$ and $\mathtt{A}_2$). A concrete example of such a system might be a gas boiler, where the sensors are thermostats and the actuators are valves. Any of the three sensors can fail with probability $\mathtt{p}$ expressed by transitions $(\mathtt{s} = 3) \rightarrow (\mathtt{s} = 2)$ and $(\mathtt{s} = 2) \rightarrow (\mathtt{s} = 1)$ in Fig. 5a, but they are used in triple modular redundancy: the processor can determine sufficient information to proceed provided two of the three are functional. If more than one becomes unavailable the system is shut down. In similar fashion, it is sufficient for only one of the two actuators to be working, but if this is not the case, the system is shut down. The processor can also fail (Fig. 5b). This can be either a permanent fault expressed by transition $(\mathtt{i} = 2) \rightarrow (\mathtt{i} = 0)$ with probability $\mathtt{p}_\mathtt{f}$ or a transient fault expressed by transition $(\mathtt{i} = 2) \rightarrow (\mathtt{i} = 1)$ with probability $\mathtt{p}_\lambda$. In the latter case, the situation can be rectified automatically by the processor rebooting itself as expressed by transition $(\mathtt{i} = 1) \rightarrow (\mathtt{i} = 2)$. In either case, the system is automatically shut down. The graphical representation of PTS associated with the sensor and processor behavior is portrayed in Fig. 5.

2.2 Property specification

The properties specification language PCTL associated with PTSs is expressed by the following BNF grammar:

$$\varphi ::= \mathtt{true} \mid \mathtt{ap} \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathtt{P}_{\bowtie \mathtt{p}}[\psi]$$
$$\psi ::= \mathtt{X}\varphi \mid \varphi_1 \cup^{\leq \mathtt{k}} \varphi_2 \mid \mathtt{F}^{\mathtt{t}}\varphi$$

Where "$\mathtt{ap}$" is an atomic proposition, $\mathtt{k} \in \mathbb{N}$, $\mathtt{p} \in]0,1[$ and $\bowtie \in \{<, \leq, >, \geq\}$. "$\wedge$" represents the conjunction operator and "$\neg$" is the negation operator. The probabilistic path operator $\mathtt{P}_{\bowtie \mathtt{p}}[\psi]$ provides the probability to satisfy a path formula ψ with the constraint $\bowtie \mathtt{p}$. "$\mathtt{X}$," "$\cup^{\mathtt{k}}$" and "$\cup$" are the next, the bounded until and the until temporal logic operators, respectively. Other operators can be derived such as:

- $\mathtt{false} \equiv \neg\mathtt{true}$,
- $\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2)$,
- $\varphi_1 \Rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$,
- $\varphi_1 \Leftrightarrow \varphi_2 \equiv \varphi_1 \Rightarrow \varphi_2 \wedge \varphi_2 \Rightarrow \varphi_1$,
- Future: $\mathtt{F}\varphi \equiv \mathtt{true} \cup \varphi$ or $\mathtt{F}^{\leq \mathtt{k}}\varphi \equiv \mathtt{true} \cup^{\leq \mathtt{k}} \varphi$ where $\mathtt{k} \geq 0$,
- Generally: $\mathtt{G}\varphi \equiv \neg(\mathtt{F}\neg\varphi)$ or $\mathtt{G}^{\leq \mathtt{k}}\varphi \equiv \neg(\mathtt{F}^{\leq \mathtt{k}}\neg\varphi)$ and $\mathtt{k} \geq 0$.
- $\mathtt{P}_{\geq \mathtt{p}}[\mathtt{G}\varphi] = \mathtt{P}_{\leq 1-\mathtt{p}}[\mathtt{F}\neg\varphi]$

Below, two requirements (queries) of the system presented in Fig. 5 are expressed in PCTL and illustrated in the natural language.

- $\mathtt{P}_{=?}[(\mathtt{s} < 2)\&\&(\mathtt{i} = 2)]$ "*The probability that the number of working sensors has dropped below 2 and the processor is functioning (and so can report the failure).*"

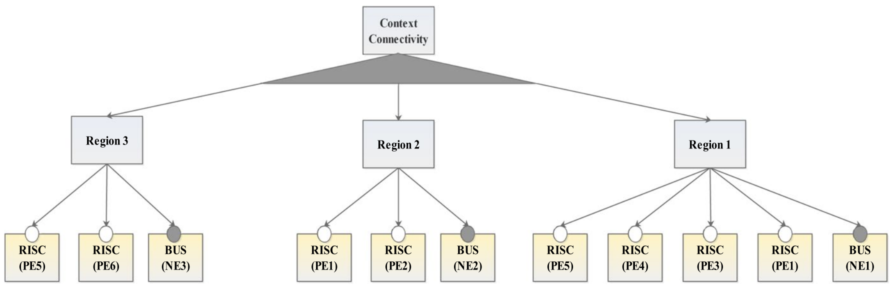
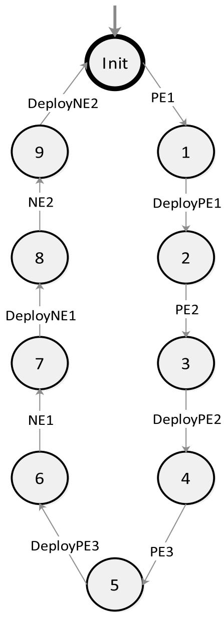
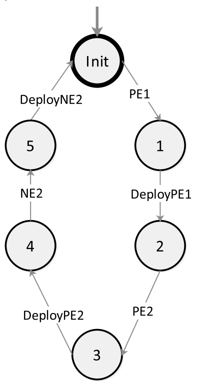


Fig. 2 Context feature diagram of the MPSoC PL

Fig. 3 TS products generated from Figs. 1 and 2



(a) TS of the Product a.



(b) TS of the Product b.

Table 1 A list of acronyms used in the article

MPSoC	Multi-processing systems on chips	PL	Product line
FD	Feature diagram	PCTL	Probabilistic computation tree logic
LTL	Linear-time temporal logic	FTS	Featured transition system
MDP	Markov decision process	PTS	Probabilistic transition system
RISC	Reduced instruction set computer	DSE	Design space exploration

- $\mathtt{P}_{=?}[(\mathtt{i} = 1) \cup (\mathtt{i} = 2)]$ "*The probability that the processor will be rebooted after a transient failure.*"

To specify a satisfaction relation of a PCTL formula in a state "s," a class of adversaries has been defined [18] to solve the nondeterministic choice in a PTS. Hence, a PCTL formula should be satisfied under all adversaries. The satisfaction relation of a PCTL formula is denoted by "$\models$" and defined as follows, where "s" is a state and "π" is a path (sequence of states). In this paper, the path "π" is obtained by a memoryless adversary [18].

- $\mathtt{s} \models \mathtt{true}$ is always satisfied,
- $\mathtt{s} \models \mathtt{ap} \Leftrightarrow \mathtt{ap} \in \mathtt{L}(\mathtt{s})$ and L is a labeling function,
- $\mathtt{s} \models \varphi_1 \wedge \varphi_2 \Leftrightarrow \mathtt{s} \models \varphi_1 \wedge \mathtt{s} \models \varphi_2$,
- $\mathtt{s} \models \neg\varphi \Leftrightarrow \mathtt{s} \not\models \varphi$,
- $\mathtt{s} \models \mathtt{P}_{\bowtie \mathtt{p}}[\psi] \Leftrightarrow \mathtt{P}(\{\pi | \pi \models \psi\}) \bowtie \mathtt{p}$ such that the probability of the path $\pi = \mathtt{s}_0 ... \mathtt{s}_\mathtt{n}$ is given by $\mathtt{P}(\pi) = \prod_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{p}(\mathtt{s}_\mathtt{i}, \mathtt{s}_{\mathtt{i}+1})$,
- $\pi \models \mathtt{X}\varphi \Leftrightarrow \pi(1) \models \varphi$ where $\pi(1)$ is the second state of π
- $\pi \models \varphi_1 \cup^{\leq \mathtt{k}} \varphi_2 \Leftrightarrow \exists \mathtt{i} \leq \mathtt{k} : \forall \mathtt{j} < \mathtt{i}, \pi(\mathtt{j}) \models \varphi_1 \wedge \pi(\mathtt{i}) \models \varphi_2$,
- $\pi \models \varphi_1 \cup \varphi_2 \Leftrightarrow \exists \mathtt{k} \geq 0 : \pi \models \varphi_1 \cup^{\leq \mathtt{k}} \varphi_2$.

2.3 Feature diagrams

Product Line (PL) engineering [45] is a method for expressing large-scale systems, including common and variable features. To express such configuration, FD is dedicated to express variability as depicted in Figs. 1 and 2. A FD has exactly one root (in the example `platform` and `context`). Features have a type: either they are "`mandatory`" (e.g., Feature `BUS` in Fig. 2), stating that they must be selected or "`optional`" (e.g., Feature `RISC` in Fig. 2), stating that may be selected during the derivation process. Multiple optional features are structured in groups and also have a type: an "`or`" group means that at least one of the group's features has to be selected, whereas an "`And`" group requires the selection of all features. A configuration of an FD is said to be valid if it does not contradict any of the constraints imposed by the context FD.

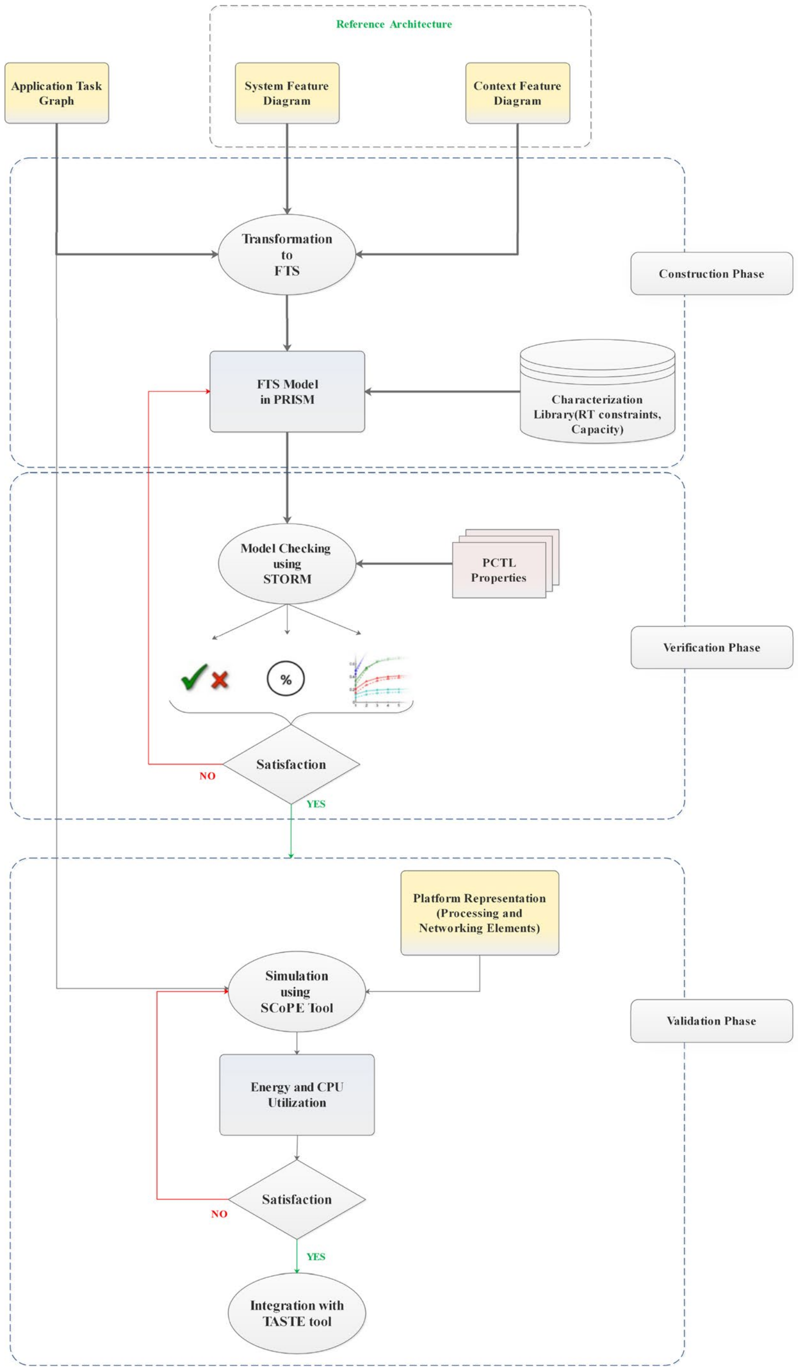
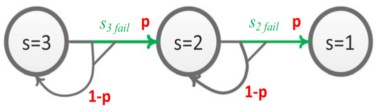
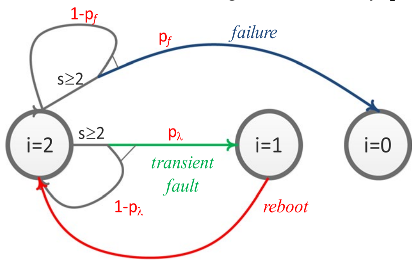


Fig. 4 Proposed methodology for context-driven deployment



(a) Sensor Behavior Failing with Probability `p` .



(b) Processor Behavior Failing with Probability $\mathtt{p_f}$ and Transient Failure Probability $\mathtt{p_\lambda}$.

Fig. 5 Overall system structure of sensors-processor in Example 1

2.4 Context feature diagrams

A system is defined as context-aware if it provides relevant services to the user depending on the context where the system is evolving. For instance, the adaptation feature of smartphones that changes the orientation of the screen (landscape/portrait mode) depends on the phone position. Designers of context-aware systems identify the relevant context features used to adapt the system behavior. Our approach relies on the topology (i.e., connectivity map) structure that describes the physical connections of hardware components. We use "`context`" variability to model those "context" features intend to be selected or not. The related FD is depicted in Fig. 2. For instance, depending on the current processing element, multiple processors could be selected and linked in the same region. If the processing element `PE5` is selected, then multiple successors are available such as: `PE3`, `PE4`, `PE6`, `PE3`. Also, if the `PE5` and `PE6` are selected, in this case one network element, `NE3` is selected.

However, contemporary product lines approaches do not distinguish context features independently from the system features. According to Capilla et al. [9], one approach is to model context by anchoring system features in one branch and context features in another, as in Hartmann and Trew [21], Ubayashi et al. [47]. However, this approach overloads the number of the relationship between both types of components. The second alternative is to label only those features that relate to context changes as in Mauro et al. [7, 15, 33]. The second alternative is the basis of our approach, distinguishing system features from context features.

3 `FTS` formalism

In this section, we recall the basic concepts and definitions that will be used throughout the paper.

Let $\mathtt{N}$ be a set of all features of a variability-intensive system. A specific set of features $\mathtt{p} \subseteq \mathtt{N}$ specifies an instance of the variability expressed in the system feature diagram or *product* in PL terminology. A variability system is then a set of products, i.e., a set of sets of features $\mathtt{px} \subseteq \mathscr{P}(\mathtt{N})$.

Definition 3 An FD $\mathtt{d}$ is a tuple $\langle \mathtt{N}, \mathtt{px} \rangle$, where $\mathtt{N} \subseteq \mathtt{F}$ is the set of features and $\mathtt{px} \subseteq \mathscr{P}(\mathtt{N})$ is the set of products; We also write $[[\mathtt{d}]]_{\mathtt{FD}}$.

In this paper, the behavior of the individual product is represented with probabilistic transition systems (PTS). A PTS is a directed graph where transitions are labeled with actions. As an example, the semantics of a subset of hardware platform derived from the FD in Fig. 1 is a combination of processing and networking elements (using short features names): $\{\{\mathtt{PE5}, \mathtt{NE3}, \mathtt{PE6}\}, \{\mathtt{PE5}, \mathtt{NE3}, \mathtt{PE6}\}, \{\mathtt{PE5}, \mathtt{NE1}, \mathtt{PE1}\}\}$.

A configurable product behavior of $\mathtt{p}$ is a non-empty infinite sequence $\sigma = \mathtt{s}_0\alpha_1\mathtt{s}_1\alpha_2$ with $\mathtt{s}_0 \in \mathtt{I}$ such that $\mathtt{s}_\mathtt{i} \xrightarrow{\alpha_{\mathtt{i}+1}} \mathtt{s}_{\mathtt{i}+1}$ for all $\mathtt{i} \geq 0$. A *path* is an execution from which the information about the transitions has been removed, i.e., , the path π for the execution σ is the sequence $\mathtt{s}_0\mathtt{s}_1 \ldots$. The *i*th state in a path π is denoted by $\pi_\mathtt{i}$, and the first state being π_0. The semantic of a TS, written $[[\mathtt{ts}]]_{\mathtt{TS}}$, is a set of paths.

Classen et al. [11] propose a Featured Transition System (FTS) to describe the behavior of all the products of a PL. Features labeling the FTS transitions belong to the product if and only if the transitions are part of the product behavior. For instance, the situation in which a transition is present if and only if both features $\mathtt{a}$ and $\mathtt{b}$ that are part of the product can be easily modeled with feature expression $\mathtt{a} \wedge \mathtt{b}$ Feature expressions are obtained from the context feature diagram. Formally, FTS can be described in terms of automata as:

Definition 4 An FTS is a tuple $\mathtt{fts} = \langle \bar{\mathtt{s}}, \mathtt{S}, \mathtt{L}, \Sigma, \delta, \mathtt{d}, \gamma \rangle$:

- $\langle \bar{\mathtt{s}}, \mathtt{S}, \mathtt{L}, \Sigma, \delta, \mathtt{d}, \gamma \rangle$ is a PTS,
- d is a feature model, and
- $\gamma : \delta \rightarrow (\{\bot, \top\}^{|\mathtt{N}|} \rightarrow \{\bot, \top\})$ is a Boolean function over a set of features labeling each transition with a feature expression.

The behavior of two products introduced in Figs.3a, b of the PL hardware platform in Fig. 1 can be represented with FTS as in Fig. 6. The feature expression (i.e., required features to enable the transition) of the transition is shown next to its actions label, separated by a slash. PE1 is selected when transition $((\mathtt{s} = 10) \rightarrow (\mathtt{s} = 11))$ is triggered if and only if the feature expression $(\mathtt{PE2} \wedge \mathtt{NE2}) \vee ((PE3 \vee PE4 \vee PE5) \wedge NE1)$ is stated true.

Feature expression configuration is obtained from the context feature diagram where $\mathtt{PE1}$ is part of $\mathtt{Region1}$ and $\mathtt{Region2}$. $\mathtt{NE1}$ is selected when transition $((\mathtt{s} = 17) \rightarrow (\mathtt{s} = 20))$ is triggered if and only if the feature expression $\mathtt{PE1} \vee \mathtt{PE3} \vee \mathtt{PE4} \vee \mathtt{PE5}$ is stated true. Feature expression construction is detailed in Sect. 4.4.

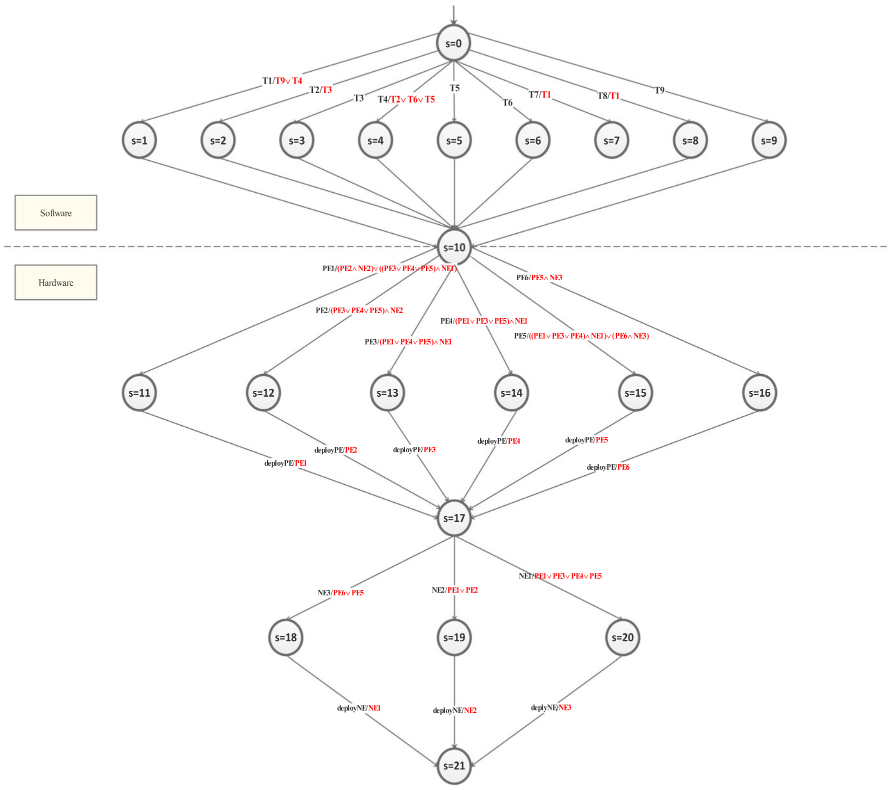


Fig. 6 FTS of software and hardware platform

In the case where processor PE5 is selected, two alternatives are available to select a networking element since the choice is non-deterministic. Two transitions $(\mathtt{s}=17) \rightarrow (\mathtt{s}=20)$ and $(\mathtt{s}=17) \rightarrow (\mathtt{s}=18)$ can be fired since the Boolean feature expressions leading to the selection of NE1: $\mathtt{PE1} \vee \mathtt{PE3} \vee \mathtt{PE4} \vee \mathtt{PE5}$ and NE3: $\mathtt{PE6} \vee \mathtt{PE5}$ are evaluated to true, respectively, except the expression leading to the selection of NE2: $\mathtt{PE1} \vee \mathtt{PE2}$ is evaluated to false.

Definition 5 (*Projection in FTS*) The projection of an FTS fts to a product $\mathtt{p} \in [[\mathtt{d}]]$ noted $\mathtt{fts}_{|\mathtt{p}}$ is a PTS pts'=$\langle \bar{\mathtt{s}}, \mathtt{S}, \mathtt{L}, \Sigma, \delta, \mathtt{d}, \gamma \rangle$ where $\delta' = \{\mathtt{t} \in \delta | \mathtt{p} \vDash \gamma(\mathtt{t})\}$.

The behavior of a particular product of the PL is deduced through a *projection*. So, If the feature expression of FTS transitions is not evaluated to be true, then these transitions are removed. Definition 5 portrays the projection of an FTS.

In the deployment plan of SPL in Fig. 1, e.g., a valid product in the transition system of Fig. 3b (i.e., projection) is derived randomly where the hardware platform contains a set of physical units (i.e., $\mathtt{PE1}, \mathtt{PE2}, \mathtt{NE2}$). This is not admitted by the FTS semantics according to which the platform requires the selection of one processor from $\{\mathtt{PE1}, \mathtt{PE2}, \mathtt{PE3}, \mathtt{PE4}, \mathtt{PE5}, \mathtt{PE6}\}$. The choice between the transitions $(\mathtt{s} = 10) \rightarrow (\mathtt{s} = 11)$, $(\mathtt{s} = 10) \rightarrow (\mathtt{s} = 14)$ could be non-deterministic in the first step of the derivation. Moreover, there are more alternatives during the derivation process depending on the presence of some physical units (e.g., $(\mathtt{s} = 10) \rightarrow (\mathtt{s} = 12)$ is selected only if the Boolean expression $\mathtt{PE3} \vee \mathtt{PE4} \vee \mathtt{PE5} \wedge \mathtt{NE2}$ is satisfied). Finally, one of the product projection corresponds to $\mathtt{fts}_{|\{\mathtt{T}_1,..,\mathtt{T}_n,\mathtt{PE1},\mathtt{NE2},\mathtt{PE2}\}}$.

The FTS represents all products configurations of the PL, and its semantics comprises the semantics of all feasible projections. Its formal definition is given below.

Definition 6 (*Semantics of FTS*)

$$[[\mathtt{fts}]]_{\mathrm{FTS}} = \bigcup_{\mathtt{c} \in [[\mathtt{d}]]_{\mathrm{FD}}} [[\mathtt{fts}_{|\mathtt{c}}]]_{\mathrm{PTS}}$$

FTS are meant to represent the configuration (i.e., behavior) of the myriad instances of the variability-intensive system. A product derivation is driven by the evaluation of feature expression in FTS. Meanwhile, we do not expect engineers to write this specification manually. So, context information is employed to enrich our FTS model. We need to provide an algorithm considering system and context feature model as inputs to build our FTS.

4 Correct modeling and sound analysis of FTS

In this section, we present FTS modeling using the PRISM language. We use the STORM model checker to accept PRISM source code as input language. Indeed, STORM model checker provides a range of different engines that pursue different approaches to reason on Markov models such as Sparse, Decision Diagram (DD), Hybrid (combine the Sparse and DD engines), etc. The innovation brought by Storm Model Checker is the exploration and abstraction-refinement-based engines [6]. The former is based on the idea of applying techniques from machine learning. On the fly, it tries to explore parts of the system that contribute most to the model checking result. The latter starts with a coarse over-approximation of the concrete model. This abstract model is then analyzed. Then, based on the obtained results, one of two things happens, either the result carries over to the concrete model and can return an answer, or refining the abstracted model. In the last case, the abstraction is performed continuously until reaching a particular answer.

4.1 PRISM language

To construct and analyze FTS with STORM [14], it must be specified in PRISM language. A description of the supported models is provided in [22]. Markov Decision Process (MDP) is selected because it captures probabilistic systems' behavior by supporting non-determinism and uncertainty. FTS requirements are specified by PCTL temporal logic to express all properties.

Generally, a probabilistic system "$\mathtt{S}$" that is described as a PRISM program "$\mathtt{P}$" that comprises a set of "$\mathtt{n}$" modules ($n > 0$), the state of each one is defined by the evaluation of a countable set of finite-ranging local variables. The global state of the system is the evaluation of the local variables ($\mathtt{V_l}$) and the global ones ($\mathtt{V_g}$) denoted by $\mathtt{V} = \mathtt{V_l} \cup \mathtt{V_g}$.

The behavior of each module is a set of guarded commands. Generally, a command takes the following form: $[\mathtt{a}]\mathtt{g} \rightarrow \mathtt{p_1} : \mathtt{u_1} + \cdots + \mathtt{p_m} : \mathtt{u_m}$, or, $[\mathtt{a}]\mathtt{g} \rightarrow \mathtt{u}$, which means, for the action "$\mathtt{a}$" if the guard "$\mathtt{g}$" is true, then, an update "$\mathtt{u_i}$" is enabled with a probability "$\mathtt{p_i}$". For the second case, for the action "$\mathtt{a}$" if the guard "$\mathtt{g}$" is true, then, the update "$\mathtt{u}$" is enabled. A guard is a logical proposition consisting of variables evaluation and propositional logic operators. The update "$\mathtt{u_i}$" is an evaluation of variables expressed as a conjunction of assignments: $(\mathtt{v}'_\mathtt{j} = \mathtt{val_j})\& \ldots \&(\mathtt{v}'_\mathtt{k} = \mathtt{val_k})$ where $\mathtt{v}'_\mathtt{j}$ are local variables and $\mathtt{val_i}$ are values evaluated via expressions denoted by "*eval*" that requires type consistency ($eval : \mathtt{V_l} \rightarrow \mathbb{N} \cup \{\mathtt{true}, \mathtt{false}\}$). The formal definition of a command is given in Definition 7.

Definition 7 (*PRISM command*) A PRISM command is a tuple $\mathtt{c} = \langle \mathtt{a}, \mathtt{g}, \mathtt{u} \rangle$, where:

- $\mathtt{a}$: is an action label,
- $\mathtt{g}$: is a predicate over $\mathtt{V}$,
- $\mathtt{u} : \{(\mathtt{p_i}, \mathtt{u_i}) | \; \mathtt{m} > 1, 0 < \mathtt{p} < 1, \sum_{\mathtt{i}=1}^{\mathtt{m}} \mathtt{p_i} = 1$ and $\mathtt{u_i} = \{(\mathtt{v}, \mathtt{eval(v)}) : \mathtt{v} \in \mathtt{V_l}\}\}$

A module that describes the behavior of a sub-part of a system can be considered as a set of commands. The variables of each module are declared and initialized locally. A module is formally defined in Definition 8.

Definition 8 (*PRISM module*) A PRISM module "$\mathtt{M}$" is a tuple $\mathtt{M} = \langle \mathtt{V_l}, \mathtt{I_l}, \mathtt{C} \rangle$, where:

- $\mathtt{V_l}$: is a finite set of local variables associated with the module M,
- $\mathtt{I_l}$: is the initial values of $\mathtt{V_l}$,
- $\mathtt{C} = \{\mathtt{c_i} : 0 \leq \mathtt{i} \leq \mathtt{k}\}$ is a finite set of commands that defines the behavior of the module M.

To describe the composition between modules, PRISM uses the following Communicating Sequential Processes (CSP) [23] operators.

- Synchronization: It is a parallel composition of modules. For two modules $\mathtt{M_1}$ and $\mathtt{M_2}$, their synchronization is denoted by $\mathtt{M_1}||\mathtt{M_2}$ and they can synchronize only on actions appearing in both $\mathtt{M_1}$ and $\mathtt{M_2}$.
- Interleaving: It is an asynchronous parallel composition of modules that are fully interleaved without synchronization. $\mathtt{M_1}$ interleaves with $\mathtt{M_2}$ is denoted by $\mathtt{M_1}|||\mathtt{M_2}$.

- Parallel Interface: It is a restricted parallel composition of modules. The modules synchronize only on shared actions. For example, let $\{\mathtt{a}, \mathtt{b}, \ldots\}$ be the set of shared actions between $\mathtt{M}_1$ and $\mathtt{M}_2$, the interface parallel composition of $\mathtt{M}_1$ and $\mathtt{M}_2$ in $\{\mathtt{a}, \mathtt{b}, \ldots\}$ is denoted by: $\mathtt{M}_1|[\mathtt{a}, \mathtt{b}, \ldots]|\mathtt{M}_2$.

As a result, Definition 9 stipulates formally a system containing $\mathtt{n}$ modules and combined by a CSP algebraic expression.

Definition 9 (*PRISM system*) A PRISM system is a tuple $\mathtt{P} = \langle \mathtt{V}, \mathtt{I}, \mathtt{exp}, \mathtt{M}_1, \ldots, \mathtt{M}_\mathtt{n}, \mathtt{CSPexp}\rangle$, where:

- $\mathtt{V} = \mathtt{V}_\mathtt{g} \bigcup_{\mathtt{i}=1}^{\mathtt{n}} \mathtt{V}_{\mathtt{li}}$: is a finite set of the union of global and local variables,
- $\mathtt{I} = \mathtt{I}_\mathtt{g} \bigcup_{\mathtt{i}=1}^{\mathtt{n}} \mathtt{I}_{\mathtt{li}}$: is a finite set of the initial values of global ($\mathtt{I}_\mathtt{g}$) and local ($\mathtt{I}_\mathtt{l}$) variables,
- exp is a set of global logic expressions,
- $\mathtt{M}_1, \ldots, \mathtt{M}_\mathtt{n}$ is a countable set of modules,
- CSPexp is a CSP algebraic expression.

Example 2 The PRISM model of the system described in Fig. 5 comprises three modules, one for the sensors, one for the actuators, and one for each processor. Lines 5–10 in listing 1 show the section of the PRISM language description which models the sensors. This constitutes a single module sensor with an integer variable $\mathtt{s}$ representing the number of sensors currently working. The module's behavior is described by one guarded command, which represents the failure of a single sensor. Its guard "$\mathtt{s} > 0$" states this can occur at any time, except when all sensors have already failed. The action $(\mathtt{s}' = \mathtt{s} - 1)$ simply decrements the counter of functioning sensors.

Listing 1: PRISM Code for the Sensor and Processor

```
 1  /* a sensor fails on average once a month */
 2  const double p;
 3  const double p_f;
 4  const double p_λ;
 5  module sensors
 6  /* number of sensors working */
 7      s : [0..3] init 3;
 8  /* failure of a single sensor */
 9      [] s > 0 → p : (s'=s-1) + 1-p: true;
10  endmodule
11
12  module proci
13  /* state: 2=ok, 1=transient fault, 0=failed*/
14      i : [0..2] init 2;
15  /* failure of processor*/
16      [] i >0 && s ≥ 2 → p_f : (i'=0)+ 1 − p_f : (i'=2);
17  /* transient fault */
18      [] i = 2 && s≥2 → p_λ : (i'=1) + 1 − p_λ : (i'=2);
19  /* reboot after transient fault */
20      [reboot] i = 1 → (i'=2);
21  endmodule
```

Lines 12–21 in listing 1 show a second module which is the PRISM language description of the input processor. The module has a single variable $\mathtt{i}$ with range $\{0, 1, 2\}$ which indicates which of the three possible states the processor is in, i.e., whether it is working, is recovering from a transient fault, or has failed. The three guarded commands in the module correspond, respectively, to the processor failing, suffering a transient fault, and rebooting. Two points of note are as follows. Firstly, the guards of these commands can refer to variables from other modules, as evidenced by the use of $\mathtt{s} \geq 2$. This is because the processor ceases to function once it has detected that less than two sensors are operational. Secondly, the last command contains an additional label reboot, placed between the square brackets at the start of the command. This is used for synchronizing actions between modules, i.e., allowing two or more modules to make transitions simultaneously.

4.2 FTS reachability

As reported in Sect. 1.4, the primary purpose of the proposed approach is to capture the FTS based on its system and context feature diagram and encode it into PRISM input language as PTS. Thus, the PRISM model checker performs a search in the state space of the FTS, and thus it needs an equal representation in PRISM language that is faithful to the FTS semantics. Therefore, the model checking algorithm has to keep track of the states and the products in which they are reachable.

The reachability relation $\mathtt{R}$ is the computed structure by the model checking algorithm as the FTS is explored. It is a set of couples $(\mathtt{s}, \mathtt{px})$ such that a state $\mathtt{s}$ is reachable by the products in $\mathtt{px}$.

Definition 10 A reachability relation of an FTS is a total function, $\mathtt{R} : \mathtt{S} \mapsto \mathtt{P}(\mathtt{N})$, so that $\forall \mathtt{s} \in \mathtt{S}, \mathtt{p} \in \mathtt{R}(\mathtt{s})$, $\mathtt{s}$ is reachable in $\mathtt{fts}_{|\mathtt{p}}$:$\exists \pi \in [[\mathtt{fts}_{|\mathtt{p}}]]_{\mathtt{TS}}$, $\mathtt{i} \in \mathrm{N}$, and $\mathtt{head}(\pi_{\mathtt{i}}) = \mathtt{s}$.

Computing $\mathtt{R}$ is efficiently handled by STORM while exploring the FTS. STORM model checker implements a lot of engines as the constructed models can be stored as binary decision diagrams (BDDs) and multi-terminal BDDs (MTBDDs). They have demonstrated to enable the verification for large hardware circuits. Moreover, encoding the reachable states in PRISM language by Boolean variables that are true if the reachable state is selected.

Given a state $\mathtt{s}$ reachable by-products in $\mathtt{px}$, a transition leaving $\mathtt{s}$, say $\mathtt{t} = \mathtt{s} \rightarrow \mathtt{s}'$, can be fired for all products if the selected feature belongs to the connectivity context of the required components, else, $\mathtt{s}'$ will only be reachable by a subset of $\mathtt{px}$. It is formalized in the following definition.

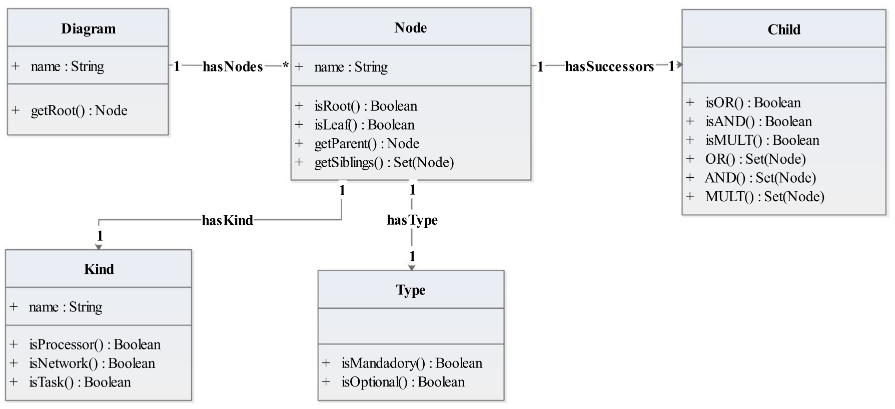


Fig. 7 Feature diagram metamodel

Definition 11 The successors of a state $\mathtt{s} \in \mathtt{S}$ for a product $\mathtt{px} \in \mathtt{P(N)}$ are given by:

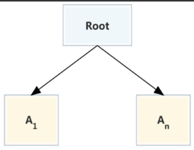
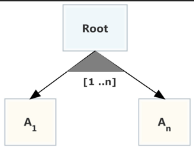
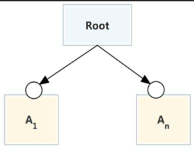
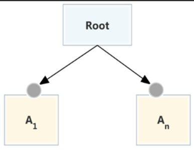
$$\mathtt{Post(s, px)} = \{(\mathtt{s}', \mathtt{px}') | \mathtt{s} \xrightarrow{\alpha} \mathtt{s}' \in \delta \wedge \mathtt{px}' = \mathtt{px} \cap [[\gamma(\mathtt{s} \rightarrow \mathtt{s}')]]\}$$

Let us illustrate this with the configuration platform given in Fig. 6. State $(\mathtt{s} = 10)$ is an initial state, and thus reachable by all products. From there, the transitions $(\mathtt{s} = 10) \xrightarrow{\mathtt{PE1}} (\mathtt{s} = 11)$ can be only fired by products in $[[(\mathtt{PE2} \wedge \mathtt{NE2}) \vee (\mathtt{PE3} \wedge \mathtt{PE4} \wedge \mathtt{PE5} \wedge \mathtt{NE1})]]$. The transition $(\mathtt{s} = 17) \xrightarrow{\mathtt{NE1}} (\mathtt{s} = 20)$ can be fired for all products in $[[(\mathtt{PE1} \vee \mathtt{PE2})]]$.

4.3 FTS platform construction

Figure 7 defines the metamodel that structures the construction of the FTS based on the feature diagram. The class `Diagram` located in the top left captures the initial node of the feature diagram through the function `getRoot()`. The class `Diagram` contains a set of nodes and it is specified by the relation `hasNodes()` with a cardinality $(1 \rightarrow *)$. The class `Node` is identified by its attribute `name` of type `String`. This class is endowed with four operations. `isRoot()` indicates if the `Node` is the root of the feature diagram, whereas the operation `isLeaf()` indicates if the Node has no successors. For example, the processing element `PE1` in Fig. 1 is a leaf node and the `Platform` is the root. The operation `getParent()` returns the predecessor of the currently visited node except the root in which the operation returns `NULL`. The operation `getsiblings()` returns the set of nodes that are located at the same level. If the node has successors, the relation `hasSuccessors()` identifies the kind of successors according to the concepts studied in Sect. 2 by the class `Child`. Three kinds of relationships are identified in the paper as a function: `OR()`, `AND()`, and `MULT()`.

Table 2 Mapping from feature diagram to PRISM formula

	Feature Diagram	PRISM Context Notation
Conjunction		$A_1 \& \ldots \& A_n$
Disjunction		$A_1 \| \ldots \| A_n$
Optional		$(!A_1 \| A_1) \& \ldots \& (!A_n \| A_n)$
Mandatory		$A_1 \& \ldots \& A_n$

Each operation returns a set of successors. Meanwhile, when the feature diagram is explored, Algorithm 3 has to identify the nature of that relationship through the operation `isOR`(), `isAND`() and `isMULT`(). Each of that operation returns a Boolean value. Moreover, the node could be `mandatory` or `optional`. The types of the successors are identified when the operation `isMULT`() returns true. For the sake of accuracy, three kinds of nodes are identified: `Processor`, `Network` or `Task` using the operations `isProcessors`(), `isNetwork`() and `isTask`(), respectively. The relative operations belong to the class `Kind`.

Listing 2: FTS Construction in PRISM Language

```
1  procedure FDToPRISM(fd: System, ctx:Context)
2
3  if (fd.hasNodes() is Not NULL) do
4      $explore$(fd.getRoot(), ctx)
5  endif
6
7  end
8
9  procedure explore(root: Note, ctx:Context)
10 var
11     i: Integer
12     j: Integer
13 if (root.hasSuccessors() is Not null) do
14     let s= fd.hasSuccessors()
15         if (s.isMULT()) do
16
17             for n in s.MULT() do
18                 if (n.isLeaf())do
19                   if(n.hasKind().isProcessor()) do
20                    let $c_i$ = findContext(n,ctx.getRoot())
21                    $writeFormula$ (“$ctx_i$”,$c_i$)
22                    $writeCommand$ (“[$PE_i$] $s_i$ & $ctx_i \rightarrow$ ($s_i' = \bot$) &($s_{i+1}' = \top$) & ($PE_i' = \top$)”)
23                    $writeCommand$ (“[deploy$PE_i$] $s_{i+1}$ & $PE_i \rightarrow$
24                                          (1-PR):$s_{i+1}$ +(PR):($s_{i+1}' = \bot$) &($s_j' = \top$) & ($PE_i' = \bot$);”)
25                   endif
26
27                   if(n.hasKind().isNetwork()) do
28                    let $c_j$ = findContext(n,ctx)
29                    $writeFormula$ (“$ctx_j$”,$c_j$)
30                    $writeCommand$ (“[$NE_j$] $s_j$ & $ctx_i \rightarrow$ ($s_j' = \bot$) &($s_{j+1}' = \top$) & ($NE_j' = \top$)”)
31                    $writeCommand$ (“[deploy$NE_j$] $s_{j+1}$ & $NE_j \rightarrow$
32                                              ($s_{j+1}' = \bot$) &($s_{last}' = \top$) & ($NE_j' = \bot$);”)
33                   endif
34                 return
35                 else
36                  $explore$(n, ctx)
37                 endif
38                 i = i +1
39             next
40
41         endif
42
43 endif
44 end
```

Based on the metamodel in Fig. 7, the task graph and the feature diagrams exploration process are modeled in Listing 2 and Listing 3. In Listing 2, the exploration starts by exploring the system feature diagram as depicted in lines 3–7. In lines 13–44, the algorithm verifies the type of successors. In line 15, the algorithm checks if the successors are multiples, then it deeply traverses the tree until the three leaves are located. If the kind of the feature is a processor (Line-19), then the algorithm returns the context of that leaf as PRISM formula ctx_i. The formula is defined by Table 2 as follows as described in line 22. The engine will check if the formula is satisfied to activate the next state. Line 23 portrays a probabilistic command where the successful execution depends on the quality metrics reported in the task graph (see Sect. 3). When the processor is selected, then the equivalent commands are depicted in lines 30–31. The algorithm explores recursively the feature diagram (line 36) such that the exploration terminates when the leaf is located.

The algorithm in Listing 3 is customized according to the context of the deployment. So, the objective is to find the tree leaves and their siblings to construct the formula. For instance, in lines 7–19, if the node successor's branch is a disjunction, the formula is generated following Table 2 (line 2). In contrast, if the successor branch is a conjunction, the formula is produced following Table 2 Line-1. However, if the siblings are optional or mandatory, as shown in Table 2line 1, the PRISM code is generated according to lines 19–32 in Listing 2.

Listing 3: FTS Feature Context Extraction

```
 1  procedure findContext(nc: Node, root: Node, key :Boolean)
 2  var
 3      ctx : String;
 4  if (root.hasSuccessors() Is Not NULL) then
 5      let s := root.hasSuccessors()
 6      if s.isOR() then
 7          for n in s.OR() do
 8              if (n.isLeaf() && n.name = nc.name) then
 9                  let vNode= n.name
10                  for sb in n.getSibilings() do
11                      vNode := vNode "|" findConext(nc,sb,true)
12                  Next
13                  return vNode
14              else if n.isLeaf() && key = true then return n.name
15                  else if n.isLeaf() && key = false then return ""
16                      else return findConext(nc,sb,false)
17              ctx := ctx & vNode
18          Next
19      else if s.isMULT() then
20              for n in s.MULT() do
21                  if (n.isLeaf() && n.name = nc.name) then
22                          let vNode:= n.name
23                          for sb in n.getSibilings() do
24                              vNode := vNode "&" findConext(nc,sb,true)
25                          Next
26                      return vNode
27                  else if n.isLeaf() && key = true then
28                            if(n.isMandatory()) then return n.name
29                            else if (n.isOptional()) then return ("!" & n.name & "|" & n.name)
30                       else if n.isLeaf() && key = false then return ""
31                            else return findConext(nc,sb,false)
32              ctx := ctx & vNode
33              Next
34      else
35              for n in s.AND() do
36                  if (n.isLeaf() && n.name = nc.name) then
37                      let vNode= n.name
38                      for sb in n.getSibilings() do
39                          vNode := vNode "&" findConext(nc,sb,true)
40                      Next
41                    return vNode
42                  else if n.isLeaf() && key = true then return n.name
43                       else if n.isLeaf() && key = false then return ""
44                            else return findConext(nc,sb,false)
45              ctx := ctx & vNode
46             Next
47  return ctx
48  end
```

4.4 FTS software construction

Following the same manner of the platform construction, the same metadata that structures the construction of the FTS based on the feature diagram is used. In Fig. 6, a top part is dedicated for task selection based on its predecessors. For instance, Task T1 is selected if the task T9 and T4 were selected and that, the transition $((s = 0) \rightarrow (s = 1))$ is enabled. The same process is executed for the rest of the tasks. It is represented by the PRISM command in Listing 4 (lines 1–16). Each task is declared as a Boolean variable and initialized to false (line 4–7) except the FTS node "$\bar{s}$" is initialized to true. In the beginning, the STORM engine selects randomly one task from a set of tasks, as mentioned in line 9. When the command is enabled, it sets the task variable to true. The context in which the task is selected depends on its tasks predecessor. For instance, a task is considered as an initial task means that its context formula is set to true (line 16).

Listing 4: A Portion of Task Selection in PRISM

```
1   module fts_software_hardware
2
3   /* tasks declaration */
4   T1 : bool init false;
5   ...
6   T9 : bool init false;
7
8   [T_i] s_i & ctx_i → (s'_i = ⊥) &(s'_{i+1} = ⊤) & (T'_i = ⊤);
9
10  endmodule
11
12  /* Defining the context in which the tasks are selected */
13  formula ctx_1 = T9 & T4;
14  ..
15  formula ctx_9 = true;
```

5 Evaluation

The developed tool[2] takes as input system and context feature diagrams related to the hardware platform and the tasks graph to build FTS in PRISM language. STORM model checker accepts PRISM language as input and PCTL properties to perform verification on automotive systems following steps presented in Fig. 4.

5.1 Construction phase

The application in Fig. 9 is mapped into the MPSoC system is the Automatic Braking System (ABS) [35]. The logical views of the subsystems are depicted in Fig. 8. The "ABSMainUnit" is the decision-making unit concerning the wheels braking

[2] Eclipse Modeling Tools: Tools and run-times for building model-based applications.

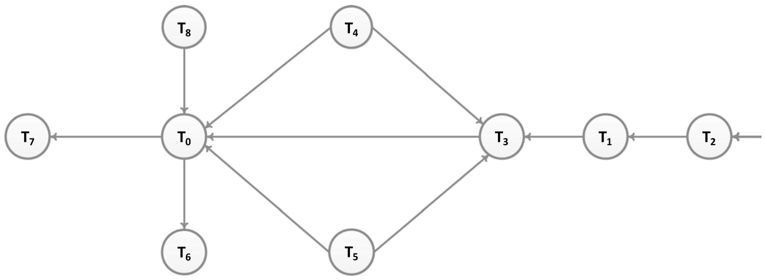


Fig. 8 Task graph

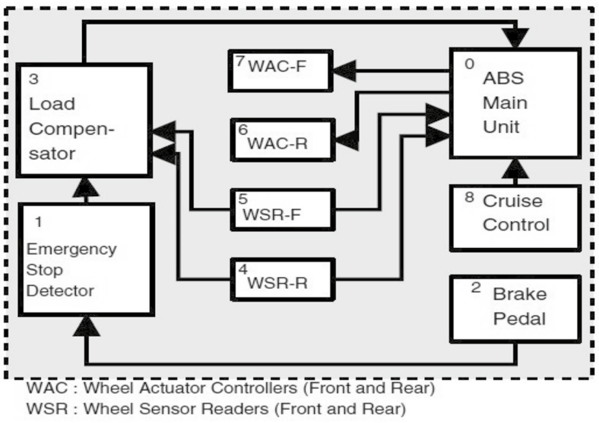

3 Load Compen-sator
7 WAC-F
6 WAC-R
0 ABS Main Unit
5 WSR-F
4 WSR-R
8 Cruise Control
1 Emergency Stop Detector
2 Brake Pedal
WAC : Wheel Actuator Controllers (Front and Rear)
WSR : Wheel Sensor Readers (Front and Rear)

Fig. 9 Automatic braking system

levels, while the "`LoadCompensatorunit`" computes adjustment factors from the wheel-load sensing inputs. Transceiver software components (4 to 7) are associated with each wheel and communicate with brake actuators and sensors. "`BrakePedal`" is the software component that reads from the dedicated sensor and sends data to the "`EmergencyStopDetection`" software module (Fig. 9).

Additional parameters are considered in the construction of the FTS model. i) software workload (wl): a computational load of a software component in executing a requested task, and ii) the processing speed (ps) expressed in MI (million instructions). Also, parameters are specified for the link between component T_i to T_j. Data size (ds) is the quantity of data exchanged between tasks T_i and T_j during one communication event and expressed in KB (kilobytes). To obtain a reliability estimation of the automotive architecture in focus, the FTS is extended with reliability's of the ABS system processing elements and communication links. So, the reliability of component PE_i can be computed as:

$$R_i = e^{-fr(d(T_i)) \times \frac{wl(T_i)}{ps(PE_i)}} \quad (1)$$

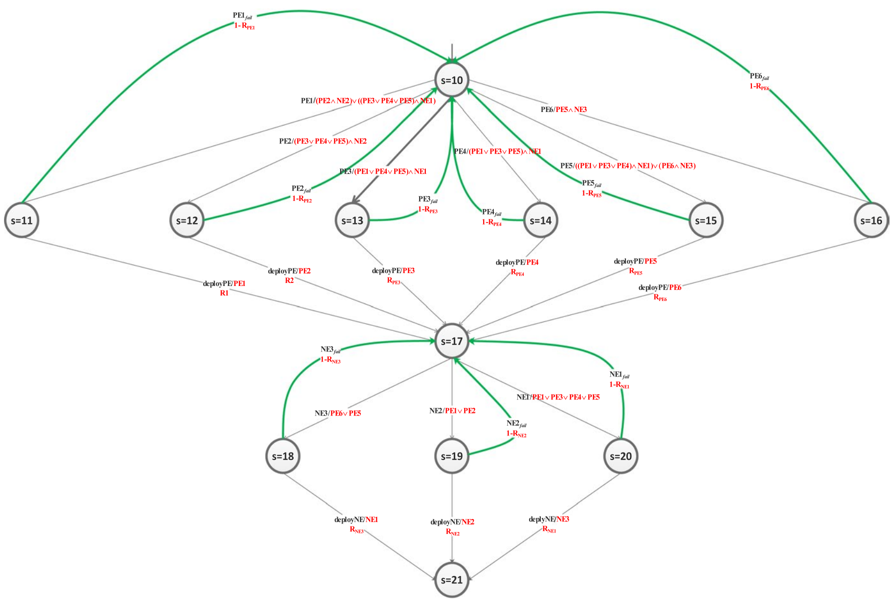


Fig. 10 A part of FTS in Fig. 6 with components reliability

Where $\mathtt{d(T_i)}$ denotes the processing elements $\mathtt{PE_i}$ selected for $\mathtt{T_i}$, and $\mathtt{fr}$ its failure rate. A similar computation can be employed for the reliability of communication elements which are characterized by failure rates ($\mathtt{fr}$) of hardware buses and the time taken for communication defined as a function of buses data rates $\mathtt{dr}$ and data sizes $\mathtt{ds}$ required for software communication:

$$\mathtt{R_i} = \mathtt{e}^{-\mathtt{fr(d(T_i),d(T_j))} \times \frac{\mathtt{ds(T_i,T_j)}}{\mathtt{dr(d(T_i),d(T_j))}}} \tag{2}$$

The resulting FTS is portrayed in Fig. 10, where processing and networking elements fail with probability $1 - \mathtt{R_{PEi}}$ and $1 - \mathtt{R_{NEi}}$, respectively. For instance, if $\mathtt{PE1}$ is selected then a failure may happen with probability $1 - \mathtt{R_{PE1}}$ when the transition $(\mathtt{s} = 11) \xrightarrow{\mathtt{PE1_{fail}}} (\mathtt{s} = 10)$ is triggered. A correct processor selection is modeled with transition $(\mathtt{s} = 11) \xrightarrow{\mathtt{deployPE1}} (\mathtt{s} = 17)$ with probability $\mathtt{R_{PE1}}$. Also, Networking elements behave as the same fashion as processing elements. For instance, if the networking element $\mathtt{NE2}$ is selected a failure may happen may happen with probability $1 - \mathtt{R_{NE2}}$ when the transition $(\mathtt{s} = 19) \xrightarrow{\mathtt{NE2_{fail}}} (\mathtt{s} = 17)$ is triggered. A correct bus selection is modeled with transition $(\mathtt{s} = 19) \xrightarrow{\mathtt{deployPE1}} (\mathtt{s} = 21)$ with probability $\mathtt{R_{NE2}}$.

5.2 Verification phase

To perform analysis, we first study the longest expected time to assure tasks deployment successfully on hardware platforms. STORM model checker provides quantitative

Table 3 Deployment probability

Processors	Min probability	Max probability
1	0.26	0.27
2	0.48	0.5
3	0.65	0.7
4	0.93	0.94
5	0.99	0.99
6	0.80	0.81

results of all possible architecture configurations, including worst and best-case scenarios. This is done using PCTL properties of the form:

$$\mathrm{Rmin} = ?[\texttt{F(\}\}success''})] \tag{3}$$

$$\mathrm{Rmax} = ?[\texttt{F(\}\}success''})] \tag{4}$$

which represent the minimum (best case) and maximum (worst case) expected time value, from the beginning until the completion of deployment. Thus, the model is enhanced with rewards associated with states or transitions.

For the MDP model, it associates to each state a value characterizing the expected time between any two deployments. Depending on the task characteristics, the minimum and the maximum expected time for task assignments is 39.8 and 41-time units, respectively. Moreover, we can determine that the application model is mapped to the maximum desired number of processing elements such that the PRISM label reference “success” is true (Listing 5 line 6). This label expresses that the application tasks were deployed. We augment our model with a reward structure that computes the number of processing elements during the assignment to respond to this property. The relative enhancement is described Listing 5 in lines 1–3.

The reward is assigned to transitions of a model labeled with actions PE_i. It is specified similarly to state rewards, within the **rewards**...**endrewards** construct.

To check the reliable deployment with optimal processors and buses utilization, the properties (5) and (6):

$$\mathrm{Pmin} = ?[\texttt{F} \leq 40\ (\texttt{\}\}success''}\ \&\ \texttt{\}\}PE''}\ \&\ \texttt{\}\}NE''})] \tag{5}$$

$$\mathrm{Pmax} = ?[\texttt{F} \leq 40\ (\texttt{\}\}success''}\ \&\ \texttt{\}\}PE''}\ \&\ \texttt{\}\}NE''})] \tag{6}$$

enable us to estimate the minimum and maximum probability that the deployment terminates within 40 time steps with a certain number of buses “NE” and processors “PE.”

Our assessments show that not all the deployment candidates are faithful to the requirement of ISO 26262 [25]. This standard targets reliability near 99.99%. Quantitative experiments depicted in Table 3 pinpoint the optimal derived MPSoC platform exploiting five processors, with a probability of 99.997% in the best and worst cases. The evaluation provides a clear view of the number of processors the designer may use to perform the deployment. The advantage of the verification is the ability to enrich our model with parameters that are not visible in the simulation, like reliability. Uncertainties need to be taken into account for system reliability modeling and assessment. The exponential distribution is used to model uncertainties with components failure rates. The approach cannot record the identity of the processors running the tasks due to the high complexity of the FTS model storage. However, a high or a low number of processors is not enough to judge the validity of the deployment. In some cases, the number of processors impacts the consumed energy, which leads to finding a trade-off between the number of processors and energy consumption.

Listing 5: Portion of the PRISM Code

```
const int P1;
const int P6;
const int N1;
const int N6;

rewards "PEi"
    [PEi] true : 1;
endrewards

rewards "NEj"
    [NEj] true : 1;
endrewards

rewards "time"
    true: 1;
endrewards

label "success" = task0 & task1 & task2 & task3 & task4 & task5 & task6 & task7 & task8;

label "PE" = PE1 = P1 & PE2 = P2 & PE3 = P3 & PE4 = P4 & PE5 = P5 & PE6 = P6;
label "NE" = NE1 = N1 & NE2 = N2 & NE3 = N3;
```

5.3 Validation phase

Although our proposed approach based on probabilistic model checking can estimate solutions that satisfy the high-level constraints and design objectives, the quality of such solutions can also be radically different at a low level. Many metrics can be used in these comparisons. So, we try to validate the relevant product based on simulation as in Fig. 4 to check the performance that is not visible at the FTS level (i.e., Fig. 6). Thus, we measure the quality of our solutions in terms of CPU usage and energy.

SCoPE tool [42] is a C++ library that implements the mechanic to perform MPSoC HW/SW co-simulation and also Network-on-Chip (NoC) analysis. Results may encompass energy, execution time, power, and several executed instructions. Moreover, these outputs may feed design-space exploration to select suitable processors for embedded systems.

The designer starts by defining the hardware infrastructure as it is represented in the feature diagram. The hardware primitives are described in the SCoPE C++

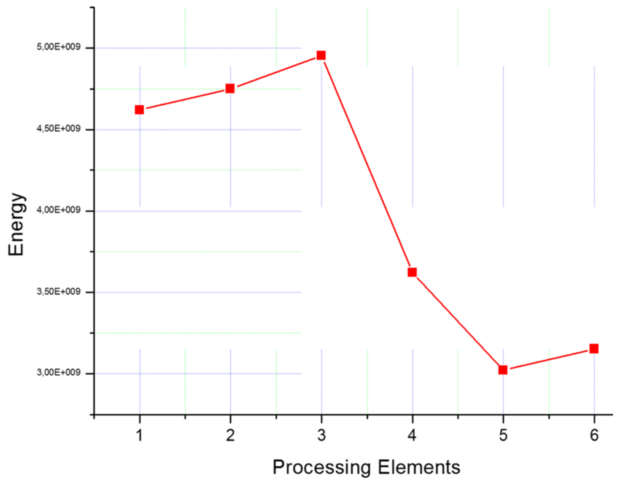


Fig. 11 Energy consumption

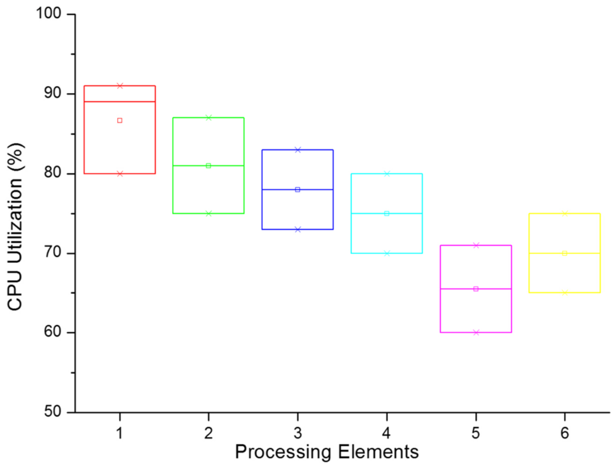


Fig. 12 The variation on CPU utilization

library. SCoPE hardware components are interconnected using standard TLM2 interfaces. When the simulation is done, and results are accepted, the engineers can embed the code and the platform architecture at the high-level representation of the TASTE tool [2, 12] using AADL [16]. A portion of the SCoPE platform description is presented in "Appendix".

According to the study, the obtained estimations are close to the high-level assessment based on model checking, such that the 80% usage of processors is never attained for the deployment in five and six processors. In contrast, the core energy required to execute the application is acceptable. The different combinations of products in the deployment schema in Fig. 3 are detailed in Figs. 11 and 12. The graphical representation in Fig. 12 portrays the minimum and the maximum CPU utilization for each processor's configuration. Also, Fig. 11 portrays the consumed energy while the application is running.

To examine the data variation on the graph, it can be seen that the platform with at least one processor and at most four processors results in high CPU utilization and energy. These observations are in line with the already obtained results in Paul et al. [40], Hoque et al. [24], Cinque et al. [10], such that the on-chip cache in the processor is becoming sensitive to failures induced by "soft error." Another important observation that does not appear in our experiments is that as the processing elements are used, the volume of exchanged data via buses increases, lengthening the actual execution time tasks. As observed, it does not influence our experiments.

6 Related work

The current work related to the application of PL is vast and varied, and we try to survey relatively close ones. The aspects covered in this section are modeling language and traditional DSE approaches.

6.1 Model representation

Ziadi et al. [52] propose a UML profile for variability with optional and alternatives stereotypes. This profile is used to model behavior with sequence diagrams, and no verification mechanisms are provided. In contrast, Ghezzi and Sharifloo [19] and Lanna et al. [29] proposed UML and feature diagrams for variability. The diagram used to model product line behavior is a sequence diagram with stereotypes that should correspond to the variability expressed in the feature diagram. The behavioral diagram is transformed into a probabilistic model such as DTMC. Moreover, the approach provides the mechanism to check quantitatively whether the derived products satisfy reliability properties expressed in PCTL.

Andrés et al. [1] provided a formal representation of an FD. They define an algebraic language, called SPLA, to describe Software Product Lines and use SAT-solver to check the satisfiability of an SPL. Varshosaz et al. [50] and Classen et al. [11] proposed featured transitions systems (FTSs), a compact mathematical package to express the behavior of all possible derivations that could occur. Then, the implemented model checking algorithms check all products and identify faulty ones against LTL properties.

These relevant papers do not give a meaning to context features that drive the selection of system features. Some articles, such as Mauro et al. [33], Hartmann and Trew [21] and Ubayashi et al. [47] defined approaches and frameworks that allow modeling customizable evolving context-aware PLs and offer much more expressivity for a product derivation and adaptation [4, 39].

6.2 Design space exploration

In literature, DSE methods depend on the particular abstraction level and design objectives. For example, at lower levels of abstraction, evaluations tools such as RTL simulators [30, 53] are capable of caring slow but cycle-accurate analysis. Meanwhile, at the higher level of abstraction, estimation techniques ranging from analytical models to system-level simulation such as [26, 31, 36, 38, 51] allow designers to obtain estimations of the final deployment candidate. Despite the design solution achievement with a relatively low number of simulations, a total run-time typically in orders of hours of each DSE experiment is still a common denominator. An extensive overview of existing work in the field of software deployment is provided [46].

Compared to the existing works, our paper extends the proposed work [11, 50] to model the tasks and the hardware elements derivations. Moreover, the paper addressed the problem of evaluating the deployment reliability regarding the hardware components' operational profile. In particular, we introduced a probabilistic model checking approach that can self-balance the number of architectural points to the intended percentiles of the values, which ultimately characterize the system reliability.

7 Conclusion

In this work, we presented a context-driven deployment methodology to analyze a real-time application mapped on an MPSoC system following the product line approach. The presented method relies on the probabilistic model checker called STORM as a basis for modeling and analyzing the real-time applications and MPSoC platforms. The application is characterized by a set of tasks where the platform is derived by varying the components of the hardware library. For an illustration purpose, we presented a real case study adapted from the automotive industry. Compared to the traditional techniques based on simulation, our proposed approach can estimate the reliability of the MPSoC system according to the platform topology constraints. Besides, the user can introduce more restrictions upon the parameterizable model for further resources management, such as power/energy and memory access.

In addition to traditional design objectives such as system performance, there is an increasing need for taking system security into account. Embedded systems are becoming more ubiquitous and interconnected, and they attract attackers. Security, however, cannot be quantified because it interferes with conventional system objectives, contrary to the earlier mentioned ones. Further, addressing this issue at the very early design stage required new techniques. Our future target is to make the software reconfigurable at run-time to accommodate dynamic tasks that require dynamic resource usage.

Appendix: SCoPE platform description

The description of the system to be simulated is done in the sc_main function. In this function, we described the HW platform, the SW infrastructure, and the application SW. The structure of a common SCoPE sc_main function is portrayed in Listing 6. For each OS model, it is required to indicate the number of processors that will be controlled by the OS (line 9). To load the application SW, it is necessary to load in the OS models and the name of the entry function of each application in line 14.

Listing 6: Portion of SCoPE Platform Description

```
 1
 2  #include <iostream>
 3  using namespace std;
 4  /* platform headers */
 5  #include "sc_scope.h"
 6
 7  int sc_main(int argc, char **argv){
 8
 9  int num_processors = 6;
10  /* Creating the SW infrastructure */
11  UC_rtos_class *rtos = new UC_rtos_class(num_processors,"arm926t");
12
13  /* Loading the application SW with a function name and its arguments*/
14  (*rtos)[0]->new_process(Brake_padle, NULL);
15
16  /* Creating the HW platform description and set the bus frequency (freq) */
17  UC_TLM_bus_class *bus = new UC_TLM_bus_class(sc_gen_unique_name("HAL"), freq);
18
19  /* Processor binding */
20  /* repeat for all the cpus ( (*rtos)[0], (*rtos)[1], (*rtos)[2],...)*/
21  (*rtos)[0]->bind(bus);
22
23  /* set the simulation time */
24  sc_start(500, SC_MS);
25
26
27  /* Simulation finished */
28  cout << "Main finish" << endl;
29  cout << "Simulated time: " << sc_time_stamp() << endl;
30
31  }
```

Funding The research leading to the presented results has been undertaken within the research profile **CPS4EU** (https://cps4eu.eu/)—Cyber-Physical Systems For Europe, funded by the European Union, grant number: 826276.

Data availability All data generated or analyzed during this study are included in this published article. A link to PRISM source code is available at: https://github.com/hakimuga/Towards-a-Context-driven-Deployment-Optimization-for-Embedded-Systems.

Declarations

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

Conflict of interest The authors declare that they have no conflict of interest.

Informed consent Informed consent was obtained from all individual participants included in the study.

References

1. Andrés C, Camacho C, Llana L (2013) A formal framework for software product lines. Inf Softw Technol 55(11):1925–1947
2. Baouya A, Mohamed OA, Bennouar D, Ouchani S (2019) Safety analysis of train control system based on model-driven design methodology. Comput Ind 105:1–16
3. Baouya A, Mohamed OA, Ouchani S, Bennouar D (2021) Reliability-driven automotive software deployment based on a parametrizable probabilistic model checking. Expert Syst Appl 174:114572. https://doi.org/10.1016/j.eswa.2021.114572
4. Bashari M, Bagheri E, Du W (2018) Self-adaptation of service compositions through product line reconfiguration. J Syst Softw 144:84–105
5. Bhat A, Samii S, Rajkumar R (2017) Practical task allocation for software fault-tolerance and its implementation in embedded automotive systems, pp 87–98
6. Brázdil T, Chatterjee K, Chmelík M, Forejt V, Křetínský J, Kwiatkowska M, Parker D, Ujma M (2014) Verification of Markov decision processes using learning algorithms. In: Cassez F, Raskin JF (eds) Automated Technology for Verification and Analysis. Springer, Cham, pp 98–114
7. Brugali D, Capilla R, Mirandola R, Trubiani C (2018) Model-based development of qos-aware reconfigurable autonomous robotic systems. In: 2018 Second IEEE International Conference on Robotic Computing (IRC), pp 129–136. https://doi.org/10.1109/IRC.2018.00027
8. Capilla R, Bosch J, Trinidad P, Ruiz-Cortés A, Hinchey M (2014) An overview of dynamic software product line architectures and techniques: observations from research and industry. J Syst Softw 91:3–23
9. Capilla R, Ortiz Óscar, Hinchey M (2014) Context variability for context-aware systems. Computer 47(2):85–87. https://doi.org/10.1109/MC.2014.33
10. Cinque M, Cotroneo D, Della Corte R, Pecchia A (2019) A framework for on-line timing error detection in software systems. Future Gen Comput Syst 90:521–538
11. Classen A, Cordy M, Schobbens PY, Heymans P, Legay A, Raskin JF (2013) Featured transition systems: foundations for verifying variability-intensive systems and their application to ltl model checking. IEEE Trans Softw Eng 39(8):1069–1089
12. Conquet E, Perrotin M, Dissaux P, Tsiodras T, Hugues J (2010) The taste toolset: turning human designed heterogeneous systems into computer built homogeneous software. In: European Congress on Embedded Real-Time Software (ERTS 2010), Toulouse, France
13. Conrady S, Kreddig A, Manuel M, Doan NAV, Stechele W (2021) Model-based design space exploration for fpga-based image processing applications employing parameterizable approximations. Microprocess Microsyst 87:104386. https://doi.org/10.1016/j.micpro.2021.104386
14. Dehnert C, Junges S, Katoen JP, Volk M (2017) A storm is coming: A modern probabilistic model checker. Computer aided verification. Springer, Berlin, pp 592–600
15. de Sousa Santos I, de Jesus Souza ML, Carvalho MLL, Oliveira TA, de Almeida ES, de Castro Andrade RM (2017) Dynamically adaptable software is all about modeling contextual variability and avoiding failures. IEEE Softw 34(6):72–77. https://doi.org/10.1109/MS.2017.4121205
16. Feiler PH (2010) Model-based validation of safety-critical embedded systems. In: 2010 IEEE Aerospace Conference, pp 1–10. https://doi.org/10.1109/AERO.2010.5446809
17. Ferreira F, Vale G, Diniz JP, Figueiredo E (2021) Evaluating t-wise testing strategies in a community-wide dataset of configurable software systems. J Syst Softw 179:110990. https://doi.org/10.1016/j.jss.2021.110990
18. Forejt V, Kwiatkowska M, Norman G, Parker D (2011) Automated verification techniques for probabilistic systems. In: Bernardo M, Issarny V (eds) Formal Methods for Eternal Networked Software Systems (SFM'11), LNCS, vol 6659. Springer, pp 53–113
19. Ghezzi C, Sharifloo AM (2013) Model-based verification of quantitative non-functional properties for software product lines. Inf Softw Technol 55(3):508–524; special issue on software reuse and product lines

20. Gries M (2004) Methods for evaluating and covering the design space during early design development. Integration 38(2):131–183. https://doi.org/10.1016/j.vlsi.2004.06.001
21. Hartmann H, Trew T (2008) Using feature diagrams with context variability to model multiple product lines for software supply chains. In: 2008 12th International Software Product Line Conference, pp 12–21
22. Hensel C (2018) STORM model checker. http://www.stormchecker.org. Accessed 21 Dec 2018
23. Hoare CAR (1978) Communicating sequential processes. Commun ACM 21(8):666–677. https://doi.org/10.1145/359576.359585
24. Hoque KA, Mohamed OA, Savaria Y, Thibeault C (2014) Probabilistic model checking based dal analysis to optimize a combined tmr-blind-scrubbing mitigation technique for fpga-based aerospace applications. In: 2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE), pp 175–184. https://doi.org/10.1109/MEMCOD.2014.6961856
25. International Organization for Standardization (ISO) (2013) Road vehicles—Functional safety—Part 9: Automotive safety integrity level (ASIL)-oriented and safety-oriented analyses. https://www.iso.org/fr/search.html?q=26262. Accessed 19 Jan 2019
26. Jiang W, Sha EHM, Chen X, Yang L, Zhou L, Zhuge Q (2017) Optimal functional-unit assignment for heterogeneous systems under timing constraint. IEEE Trans Parallel Distrib Syst 28(9):2567–2580
27. Keutzer K, Newton AR, Rabaey JM, Sangiovanni-Vincentelli A (2000) System-level design: orthogonalization of concerns and platform-based design. IEEE Trans Comput Aided Des Integr Circuits Syst 19(12):1523–1543. https://doi.org/10.1109/43.898830
28. Kwiatkowska M, Norman G, Parker D (2006) Controller dependability analysis by probabilistic model checking. Control Eng Pract 15(11):1427–1434
29. Lanna A, Castro T, Alves V, Rodrigues G, Schobbens PY, Apel S (2018) Feature-family-based reliability analysis of software product lines. Inf Softw Technol 94:59–81
30. Mahapatra A, Schafer BC (2018) Veriintel2c: abstracting rtl to c to maximize high-level synthesis design space exploration. Integration
31. Malazgirt GA, Yurdakul A (2016) Prenaut: design space exploration for embedded symmetric multiprocessing with various on-chip architectures. J Syst Archit 6:66
32. Marinho FG, Andrade RM, Werner C, Viana W, Maia ME, Rocha LS, Teixeira E, Filho JBF, Dantas VL, Lima F, Aguiar S (2013) Mobiline: a nested software product line for the domain of mobile and context-aware applications. Sci Comput Program 78(12):2381–2398. https://doi.org/10.1016/j.scico.2012.04.009. special Section on International Software Product Line Conference 2010 and Fundamentals of Software Engineering (selected papers of FSEN 2011)
33. Mauro J, Nieke M, Seidl C, Yu IC (2018) Context-aware reconfiguration in evolving software product lines. Sci Comput Program 163:139–159
34. Maxim B, Mistrík I, Galster M (2019) Software engineering for variability intensive systems: foundations and applications
35. Meedeniya I, Aleti A, Grunske L (2012) Architecture-driven reliability optimization with uncertain model parameters. J Syst Softw 85(10):2340–2355
36. Mendis H, Indrusiak LS, Audsley NC (2015) Bio-inspired distributed task remapping for multiple video stream decoding on homogeneous nocs. In: 2015 13th IEEE Symposium on Embedded Systems For Real-Time Multimedia (ESTIMedia), pp 1–10
37. Mühlbauer F, Schröder L, Schölzel M (2018) Handling of transient and permanent faults in dynamically scheduled super-scalar processors. Microelectron Reliab 80:176–183
38. Ouni B, Mhedbi I, Trabelsi C, Atitallah RB, Belleudy C (2017) Multi-level energy/power-aware design methodology for mpsoc. J Parallel Distrib Comput 100(C):203–215
39. Pascual GG, Lopez-Herrejon RE, Pinto M, Fuentes L, Egyed A (2015) Applying multiobjective evolutionary algorithms to dynamic software product lines for reconfiguring mobile applications. J Syst Softw 103:392–411
40. Paul S, Cai F, Zhang X, Bhunia S (2011) Reliability-driven ecc allocation for multiple bit error resilience in processor cache. IEEE Trans Comput 60(1):20–34. https://doi.org/10.1109/TC.2010.203
41. Pimentel AD (2017) Exploring exploration: a tutorial introduction to embedded systems design space exploration. IEEE Des Test 34(1):77–90. https://doi.org/10.1109/MDAT.2016.2626445
42. Posadas H, Villar E, Ragot D, Martinez M (2010) Early modeling of linux-based rtos platforms in a systemc time-approximate co-simulation environment. In: 2010 13th IEEE International

Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, pp 238–244. https://doi.org/10.1109/ISORC.2010.18

43. Qiu X, Ali S, Yue T, Zhang L (2017) Reliability-redundancy-location allocation with maximum reliability and minimum cost using search techniques. Inf Softw Technol 82(Supplement C):36–54
44. Sengupta A, Sedaghat R, Sarkar P (2012) A multi structure genetic algorithm for integrated design space exploration of scheduling and allocation in high level synthesis for dsp kernels. Swarm Evol Comput 7:35–46. https://doi.org/10.1016/j.swevo.2012.06.003
45. Siavvas MG, Chatzidimitriou KC, Symeonidis AL (2017) Qatch—an adaptive framework for software product quality assessment. Expert Syst Appl 86:350–366
46. Singh AK, Dziurzanski P, Mendis HR, Indrusiak LS (2017) A survey and comparative study of hard and soft real-time dynamic resource allocation strategies for multi-/many-core systems. ACM Comput Surv 50(2):1–40
47. Ubayashi N, Nakajima S, Hirayama M (2010) Context-dependent product line practice for constructing reliable embedded systems. Software Product Lines: Going Beyond. Springer, Berlin, pp 1–15
48. Van Gurp J, Bosch J, Svahnberg M (2001) On the notion of variability in software product lines. In: Proceedings Working IEEE/IFIP Conference on Software Architecture, pp 45–54. https://doi.org/10.1109/WICSA.2001.948406
49. Varshosaz M, Mousavi MR (2019) Comparative expressiveness of product line calculus of communicating systems and 1-selecting modal transition systems. In: Catania B, Královič R, Nawrocki J, Pighizzini G (eds) SOFSEM 2019: Theory and Practice of Computer Science. Springer, Cham, pp 490–503
50. Varshosaz M, Beohar H, Mousavi MR (2018) Basic behavioral models for software product lines: revisited. Sci Comput Program 168:171–185
51. Xie G, Chen Y, Liu Y, Wei Y, Li R, Li K (2017) Resource consumption cost minimization of reliable parallel applications on heterogeneous embedded systems. IEEE Trans Ind Inform 13(4):1629–1640
52. Ziadi T, Hélouët L, Jézéquel JM (2004) Towards a uml profile for software product lines. In: van der Linden FJ (ed) Software Product-Family Engineering. Springer, Heidelberg, pp 129–139
53. Zoni D, Cremona L, Fornaciari W (2018) Powerprobe: run-time power modeling through automatic rtl instrumentation. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), pp 743–748