Variability Management of Reliability Models in Software Product Lines: an Expressiveness and Scalability Analysis

Vinicius Nunes  
Departamento de Ciência da Computação - CiC  
Universidade de Brasília  
Brasília, Brasil  
Email: viniciusuriel@gmail.com

Paula Fernandes  
Departamento de Ciência da Computação - CiC  
Universidade de Brasília  
Brasília, Brasil  
Email: paulag6@gmail.com

Vander Alves  
Departamento de Ciência da Computação - CiC  
Universidade de Brasília  
Brasília, Brasil  
Email: valves@unb.br

Genaína Rodrigues  
Departamento de Ciência da Computação - CiC  
Universidade de Brasília  
Brasília, Brasil  
Email: genaina@cic.unb.br

Abstract—Some domains, specially those of critical systems, require dependable software. Ensuring dependability is not a trivial problem. Model checking can be used to estimate the reliability of a software through models that represent the behavior of the system. Through these models it is possible to estimate and measure quantitatively properties such as reliability. In the context of Software Product Lines (SPL), we need to check an entire family of systems. It is not feasible to build a model for each configuration of a SPL as the number of models required can be very large. Some contributions directly address this issue proposing techniques specifically tailored for SPL. Particularly, the technique of parametric model-checking allows the use of a single model to obtain properties values from different configurations through an arithmetic formula. However, even an arithmetic formula may not be easy to evaluate. If the number of operands is large enough the cost of evaluation of this formula could also be large. Current techniques may impose limitations over the variability and/or system architecture. To the best of our knowledge, to handle variability on model checking is still an open problem. This work is an investigation of the whole process of obtaining a parametric arithmetic formula for a SPL. Knowing this process and the factors that directly affect the growth of the formula, we are able to develop new techniques to deal with parametric model-checking in SPL with less restrictions.

I. INTRODUCTION

Ensuring software dependability, i.e., ensuring that a software has adequate levels of availability, reliability, security, integrity and maintainability is an issue especially important when it comes to critical systems, since a failure in these systems can lead to disastrous consequences. In particular, the reliability, continuity of correct software operation, is a fundamental property in this context [1].

Model-checking is a technique that can be used to verify non-functional properties such as reliability. Through software documentation artifacts as input, for example UML diagrams, it is possible to build models that estimates the software reliability [2].

Software dependability should be evaluated early in the development cycle, preferably in the design phase, since the cost of maintenance and evolution of a software in late stages of the development cycle can be expensive or unfeasible [3]. Through this analysis we can identify the most critical components and more appropriate design practices to mitigate the chance of a software failure and thus, increase its reliability [2].

This problem becomes more challenging when it comes to Software Product Lines (SPL) [4]. In a SPL each product is a different piece of software although it has some common artifacts in its structure. The reliability estimate of each product using ordinary techniques in each product separately may lead to a large amount of work since the number of products grows exponentially with the number of features of SPL and for each such configuration a reliability related model would have to be built.

Some contributions address this problem directly [5], [6], [7]. The strategy of these works consists of building a single model representing all SPL products. This can be done by using parametric model checking. Through this technique, we can obtain an arithmetic formula whose evaluation represents the numerical result of a property verification in the model. The parameterization allows us to represent variability of SPL in a single model, through different valuations of the parameters it is possible to represent different products [7], [8].

However, the current approaches may impose restrictions over SPL expressiveness, i.e., restrictions over variability and/or architecture of the SPL. These restrictions range from assumptions over the mapping among features and artifacts to limitations over the variability, such as deal only with Alternative features. Indeed, this problem lacks a more comprehensive and scalable approach.

Different strategies can be used to model variability and these strategies affects directly the size of the final formula. This size must be limited in such way that its evaluation be scalable, since an explosion in the number of operands of the formula can turn its evaluation impractical under restricted conditions.

We present an analytical study of the process of conversion
of a parametric model for an arithmetic formula and an approach to deal with the expressiveness issue emphasizing decisions that impact the size of the final formula and consequently the cost of further evaluation. Through this study, we are able to develop new parametric modeling techniques to handle variability efficiently with less restrictions to expressiveness. Our main contributions are twofold:

• **Expressiveness:** We show some strategies to improve expressiveness and how it can be used to handle optional features.

• **Scalability analysis:** A complete analysis of parametric model checking process applied to SPL. We discuss the formula size and present some practical implications of evaluating large formulas.

In Section II we further detail the problem and introduce some model checking concepts needed to better understand the following sections. In Section III we present a running example that will be used though the paper, in Section IV we present an comprehensive modeling approach and detail the parametric model checking. Section V we highlight the main aspects that impacts the formula size from a analytical and practical view point and show how we can extend current approaches to balance scalability and expressiveness. Section VI further discuss related works presented along the analysis. Finally, Section VII presents the conclusion.

II. BACKGROUND

Software dependability evaluation is an important issue, especially when it comes to critical systems. Estimating software reliability early in the development cycle allows us to make important decisions in design phase. By performing sensitivity analysis of the components, it is possible to identify which components are most critical in the software quantitatively.

This section presents the steps of model checking one product and a SPL introducing related concepts and tools.

A. Model checking one product

Model checking can be done before the development using the behavior models to build a model (step 1). These models, used as inputs to model checking tools, are able to verify properties such as reliability (step 2). Fig 1 illustrates the steps of this process.

![Fig. 1. Model checking process](image)

Model checking can be done by probabilistic model checking tools like PRISM (Step 2 of Fig 1). The PRISM tool use Markov chains to check properties like reliability in a model. Markov chains plays a key role in this work since the analysis is performed upon the theory used in this tools and not in its actual implementation. Markov chain is a probabilistic theory where the result of an experiment is influenced by the results of previous experiments. The chain is used to represent the dependency between the experiments and it is made of states and transitions. Each transition is labeled with probability value in such a way that the sum of the values of all probabilities of the outgoing transitions of a given state is equal to 100%. The chain starts in a certain state and makes transitions from one state to another according to the probability of transitions. The transition from one state to another is called step. At each step the transition probability of reaching a next state is independent of the probabilities of the prior transitions [9].

![Fig. 2. A Markov chain example](image)

These states and transitions can also be represented as a directed graph whose transitions are labeled with probabilities. Fig 2 shows an example of a Markov chain graph. The states which are not possible to leave, are called absorbing states. In Fig 2, we have two absorbing states: q4 and q3. These states are considered final states of the chain and with them we can check probabilities over the chain such as:

- What is the probability of eventually reaching q3? (unbounded time)
- What is the probability of reaching q4 within two steps? (bounded time)

Note that the above queries were classified in bounded and unbounded time. Queries with bounded time are used when the number of steps made should be limited by the chain in such a way that only those transitions which lead to the desired state are within the limit of steps considered. On the other hand, queries with unbounded time consider all transitions that somehow lead to the desired state.

Markov Chains can be further classified in discrete (DTMC) or continuous time (CTMC). CTMCs are stochastic models where the transitions are made at a certain rate instead of by probability [8]. The analysis presented here, like other works, uses DTMC models [7], [2].

PRISM tool specifies the PRISM language: a state-based language derived from the Reactive Modules formalism and uses temporal logic such as the Probabilistic Computational Tree Logic (PCTL) to build the Markov chain and check properties in the model [10], [11], [12].

With this language we model processes, which in PRISM are called modules. A model in PRISM is composed of a number of modules. Each module has a set of finite-ranged variables, which define the possible states of that module. The final model is the synthesis of all modules through parallel composition. Each module is composed of a set of guarded...
commands. For example, a DTMC command in PRISM takes the form:

```
Listing 1. PRISM Command
[ action ] <guard> --> <expression> : <variable update>;
```

The guard is a predicate over all variables in the model, and once it is satisfied, the module will make a transition with a certain probability, expressed by `expression`, to the update state. A command can have several pairs of `<expression> : <variable update>` representing transitions leaving the current state, in that case each pair is separated by a `+` symbol. Each expression may involve several rational constants and result in a rational number. The sum of all expressions in a single command is a rational number $p$ where $0 \leq p \leq 1$ that represents 0% and 100% probability respectively. The action can be used in order to tag a command that is synchronized with other commands in the same or in a different module. When there is no action label the command will run asynchronously.

PRISM performs model checking determining the quantitative value of each specified property and whether the model satisfies it. In our examples, we use PCTL to query the probability of reaching the final success states in order to estimate its reliability.

### B. Model checking SPL

Applying the same process in SPL is not feasible since all steps would have to be done for each configuration.

When it comes to product lines, it is desirable to build a single model capable of checking the reliability of all products of the SPL. However, this implies that we have to handle variability in the model. Such variability could be handled in the model artifact reducing the effort to obtain a model for each configuration. However, we still need to make a model checking for each configuration. Thus, the techniques that address this issue deal with variability in the model itself so as to allow the same model to verify properties of different configurations. This can be done by using parametric model checking. With parameters in the model we can change its semantic (by changing the parameters values) in such a way that it can represent different configurations. Fig 3 shows an overview of this process in SPL. Note that the process is the same, but the inputs and outputs changed considerably. In particular, we can highlight the feature model as the input and the arithmetic formula as the final output. These activities are conducted by SPL domain engineer. This formula contains the parameters used in the model to represent variability and its evaluation give us the final numeric value of reliability for each SPL configuration. SPL application engineer uses this formula to calculate the reliability of a specific configuration.

Variability in parametric model could also be handled in different ways. Parametric models consist of state machines; each transition is labeled by a probability parameter or constant value. We could, for example, handle variabilities by labeling some transitions with parameters whose evaluation with different values changes the semantics of the model.

We could also handle variability by adding special transitions labeled with parameters to skip some states according to its evaluation. We can go further and limit the valuation of the parameters to some range of valid values to better control the model behavior. These are just some examples of what can be done with a parametric model in order to handle variability inside the model.

Whatever your choice is, it will have direct impact on the final size of the arithmetic formula. Indeed some contributions have already highlighted this point. Some authors have already warned that the excessive use of parameters in the model can lead tools to actually not make the model checking and just output a formula representing all the computation [8], [7]. In our analysis, we show that wrong choices in modeling strategy can lead models to output large formulas.

These decisions are made in Step 1, presented in Fig 3 and this step can be manual, automated or semi-automated but Step 2 is virtually automated only (Although it can be manual, it would not be reasonable). This work details the PARAM tool process of obtaining the arithmetic formula from a parametric model (Step 2) emphasizing the modeling decisions and relating it to its actual impact on the formula size in a quantitative way. Knowing in advance the impact of the decisions, we can develop techniques more comprehensive with respect to types of variabilities and that generates a formula with an expected size.

PARAM is a tool for probabilistic parametric model checking. Similarly to PRISM, it deals with models based on Markov chains (CTMC, DTMC). This tool uses a variant of the PRISM language in which the main difference is the definition of the keyword `param`. This keyword is used to indicate that the value of a given variable is not constant and will not be available during model parsing.

We call PARAM model one that uses this variant of the PRISM language. The PARAM tool takes as input a PARAM model and a PCTL expression and produces as output a arithmetical formula with the parameters defined in the model. Through the evaluation of these parameters it is possible to obtain the values that answer the queries in the given PCTL [13].

In a PARAM model, the `expression` in a command (see Listing 1) may also have parameters. These expressions are polynomials whose evaluation (through parameters valuation) is the transition probability $p$ restricted to the same value range of PRISM transition: $0 \leq p \leq 1$ [14]. This characteristic is relevant and will be revisited later in this analysis.
PARAM synthesizes a finite automaton, extracts its corresponding regular expression and finally converts the regular expression to an arithmetic formula.

III. RUNNING EXAMPLE

To better illustrate the concepts presented throughout this paper we introduce an example of a SPL and a possible parametric model representing it.

Fig 4 presents a feature model excerpt of a system of vital signs monitoring. This excerpt is sufficient to illustrate the ideas presented in this work. This system consists of a central core and optionally invokes a component for monitoring via EKG (electrocardiograph) sensor or SPO2 (Saturation of Peripheral Oxygen) sensor (non exclusively). These components are mapped to the EKG and SPO2 features respectively.

Fig. 4. Vital Signal Monitoring Feature Model

This feature model has four possible configurations, one with just EKG selected, other with just SPO2 selected, other with both selected and one with just the root feature selected. We decide to use Optional features in the example due to its expressiveness. Note that, the same feature model of Fig 4 could be restricted by OR or Alternatives features, but these types of restriction would lead to a particular case of the example with less configurations.

Fig. 5. Configuration \{MONITORING,EKG,SPO2\}.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Artifacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONITORING</td>
<td>System Core</td>
</tr>
<tr>
<td>EKG</td>
<td>Component that handles EKG data</td>
</tr>
<tr>
<td>SPO2</td>
<td>Component that handles SPO2 data</td>
</tr>
</tbody>
</table>

TABLE I Configuration Knowledge

Fig 5 presents a sequence diagram illustrating the execution of the configuration \{MONITORING,EKG, SPO2\}. With the selected features of the given configuration and the CK (configuration knowledge, mapping between artifacts and features) we are able to build a system with three components: CORE, EKG and SPO2 [15]. Note that the correspondence between components and features are just a particularity of the given example. Table I describes the CK.

Other configurations have an analogous sequence diagram differing only be removing one component or another.

IV. ADDRESSING EXPRESSIVENESS

This section presents an approach to model variabilities in a PARAM model (Step 1 of Fig 3) and describes the process to obtain the arithmetic formula (Step 2 of Fig 3).

We summarize the main steps of this technique in order to allow the understanding of the analysis presented in section V. The goal is not to present algorithms but the problem, through which we can highlight some interesting characteristics such as the growth rate of the formula. Fig 6 presents an overview of the steps in the conversion process. First, a parametric model is parsed into a finite automaton(Step 2.1), then the automaton is reduced according to the restrictions imposed by the PCTL expression given as input (Step 2.2), from this automaton we obtain the corresponding regular expression (Step 2.3), which finally is converted into an arithmetic formula (Step 2.4). This conversion process take two inputs: PARAM model and PCTL expression.

Steps 2.1 and 2.2 are described in Section IV-A, Steps 2.3 and 2.4 are described in Section IV-B.

To better explain the process, the example of Section III will be expanded with a corresponding parametric model. Remember that, as mentioned in Section II, there are many different ways to handle variability in the PARAM model.

We present a novel approach that deals with Optional features. Other types of variabilities (OR, Alternatives, Mandatory) can be transformed to Optional features with cross tree constraints, thus they are just restrictions
over Optional features [16]. Indeed, we use this extended parametric modeling approach because the current one does not support Optional features and would limit the presented analysis [7].

The process to generate the formula is the same whatever is the model used as input.

The modeling strategy used to model the example of Section III is guided by the following rules:

1) Each software component is mapped to one PRISM module.
2) The transitions of the sequence diagram points to the component that will execute.
3) Each non mandatory feature of the feature model has a correspondent parameter whose valid values 1 or 0.
4) The variability is handled by bypass command that can skip the commands related to the deselected feature using the correspondent parameter.
5) Each step has a chance of failure associated with the executing component.

Rule 1 is a convenience rule since the example, for simplicity, has a correspondence between features and components. Rule 2 establishes a relationship between the sequence diagram and the PARAM model. Rule 3 ensures that only features that may have a correspondent parameter. Rule 4 defines how variability is handled and it is further discussed later. Rule 5 defines the approach used to calculate the SPL reliability since each software step has a associated chance of failure.

Listing 2. PARAM model

```
dtmc
param int fSPO2;
param int fEKG;
const double rCORE = 0.999;
const double rSPO2 = 0.995;
const double rEKG = 0.997;

module Core
s0 : [0..8] init 0;
| [ ] x0 = 1.0 -> (fSPO2 rCORE) : (s0' = 1) + (1 - fSPO2) : (s0' = 3) + (fSPO2 rEKG) : (s0' = 7);
| [SPO2] s0 = 1 -> (s0' = 2);
| [return_SPO2] s0 = 2 -> (s0' = 3);
| [fEKG_rCORE] s0 = 3 -> (fEKG rCORE) : (s0' = 4) + (1 - fEKG) : (s0' = 6) + (fEKG rCORE) : (s0' = 7);
| [EKG] s0 = 4 -> (s0' = 5);
| [return_EKG] s0 = 5 -> (s0' = 6);
| [success] s0 = 6 -> (s0' = 6) // END SUCCESS
| [FAIL] s0 = 7 -> (s0' = 7) // END FAIL
endmodule

module SPO2
s1 : [0..2] init 0;
| [SPO2] s1 = 0 -> rSPO2 : (s1' = 1) + (1 - rSPO2) : (s1' = 2);
| [return_SPO2] s1 = 1 -> (s1' = 1);
| [FAIL_SPO2] s1 = 2 -> (s1' = 2);
endmodule

module EKG
s2 : [0..2] init 0;
| [EKG] s2 = 0 -> rEKG : (s2' = 1) + (1 - rEKG) : (s2' = 2);
| [return_EKG] s2 = 1 -> (s2' = 1);
| [FAIL_EKG] s2 = 2 -> (s2' = 2);
endmodule
```

Constants, declared with reserved word const, prefixed with the letter r are the estimated reliability of one execution of a component. These constants are suffixed with the name of its correspondent components. The complementary of these values, \((1 - rCORE)\) for example, represents its chance of failure. Remember that these values represent probabilities, so the complementary is related with the 100% total.

Variables \(s0, s1\) and \(s2\) represent the state of its enclosing modules in the PARAM model. Changes on these variables represent a change in the overall state of the model.

Rules 3 and 4 are specific tailored to handle variability. It is done by inserting a bypass command before a command that synchronizes the execution with a module that is mapped to a feature. The bypass command has three transitions: One to the corresponding feature, other skipping to the first command after the commands related to the feature and one feature representing the chance of failure. The model in Listing 2) has two bypass commands: one for the SPO2 feature (lines 10-12) and another to the EKG feature (lines 15-17). Both of them have the three discussed transitions. In the SPO2 bypass command we use the parameter fSPO2 whose the valid values are limited to 0 and 1 (Rule 3) to select between the transition of line 11 and the pair of complementary transitions in lines 10 and 12. Note that the transition of line 11 is mutually exclusive with the pair of transitions in lines 10 and 12 since if the fSPO2 is valued with 1 it disables the transition of line 11 by associating 0% of probability to it and if it is value with 0 it simultaneously disables the transitions of lines 10 and 12 by associating 0% of probability to them and enables the transition of line 11 by associating 100% of probability to it. The pair of transitions of lines 10 and 12 represents the ordinary transitions of the model while the transition of line 11 is used to bypass the feature SPO2 skipping the commands related to that feature. The rationale is analogous to the feature EKG whose the bypass command is at line 15. In Section V we highlight characteristics that are inherent to probabilistic parametric model checking and the PARAM tool.

A. From model to DFA

The PARAM language is based on a formalism of concurrent components that allows us to represent synchronous and asynchronous components in a modular fashion. It provides abstractions over state machines which enables the use of high level concepts like modules and variables [17]. The transitions on these state machines can be labeled with probabilities. These state machines labeled with probabilities are modeled as Markov chains.

First, PARAM parses the model and build the correspondent Markov chain (Step 2.1 in Fig 6). The generated Markov chain follows the definition of a deterministic finite automaton (DFA) [18]:

\[ A = (Q, \sum, \delta, q_0, F) \]

- \(Q\) is the set of states.
- \(\sum\) is the set of input symbols, or alphabet.
- \(\delta\) is the transition function \((\delta : Q \times \sum \rightarrow Q)\)
- \(q_0\) is the initial state.
• $F$ is the set of final states.

Where $Q$ is the set of states of the state machine, $\sum$ is the set composed by all expressions that labels the transitions of the PARAM model, $\delta$ defines the transitions between states, $q_0$ is the initial state of the model and $F$ is composed by all absorbing states of the PARAM model.

The conversion process is mainly based on the following rules [17]:

• A state is a valuation of all variables in the model. Each different valuation represents a different state.
• A guard synchronization means that one or more variable value changes simultaneously. That is, a single change of state changes one or more variable values.

Note that, at this point, each transition label is just one token, one symbol on the alphabet $\sum$, even if it is a complex expression involving constants and parameters. This expressions can not be operated with other expressions while it is a token on the DFA. In order to clearly state this, we use the variable substitution presented in Table II, where we detail the token that will be used as replacement, the simplified expression replaced and lines on Listing 2 where the expressions appears.

<table>
<thead>
<tr>
<th>Token</th>
<th>Expression</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(ISPO2*0.001)</td>
<td>10</td>
</tr>
<tr>
<td>b</td>
<td>(ISPO2)</td>
<td>11</td>
</tr>
<tr>
<td>c</td>
<td>(ISPO2*0.001)</td>
<td>12</td>
</tr>
<tr>
<td>d</td>
<td>(IEKG*0.999)</td>
<td>15</td>
</tr>
<tr>
<td>e</td>
<td>(1-IEKG)</td>
<td>16</td>
</tr>
<tr>
<td>f</td>
<td>(IEKG*0.001)</td>
<td>17</td>
</tr>
<tr>
<td>g</td>
<td>0.995</td>
<td>26</td>
</tr>
<tr>
<td>h</td>
<td>0.005</td>
<td>27</td>
</tr>
<tr>
<td>i</td>
<td>0.997</td>
<td>34</td>
</tr>
<tr>
<td>j</td>
<td>0.003</td>
<td>35</td>
</tr>
</tbody>
</table>

**TABLE II** Variable Substitution

This variable substitution will be revisited in the sections ahead. Fig 7 illustrates the DFA obtained from the model of the Listing 2. Each state is labeled by a tuple ($s_0, s_1, s_2$) which represents the valuations of the variables $s_0$, $s_1$ and $s_2$ of the model.

Step 2.2 on Fig 6 consists of parsing the PCTL expression and eliminated from the DFA the states that are never visited in any path from initial state until some final state defined in the PCTL expression.

We want to calculate the reliability of all products in the SPL. So, we want to obtain the probability of the different configurations reaching the final state of success. As stated in Listing 2 line 20 represents the final state of success for all configurations. Thus, we want to calculate the probability of reaching some state where variable $s_0$ of the model has the value of 6. The following PCTL expression represents this query:

$$P = ? \{ true \ U \ s_0 = 6 \}$$  \hspace{1cm} (1)

This PCTL is used to determine the final states of success and the paths that lead to these states. With these paths we can reduce the DFA by removing the states that can not reach any final state. A path here is defined as in graph theory [19].

In the example of Fig 7 the final states are:

$$F = \{(6, 0, 0), (6, 0, 1), (6, 1, 0), (6, 1, 1)\}$$   \hspace{1cm} (2)

Any state that can not reach some of these states can be removed from the DFA. The states above can be removed from the DFA presented in Fig 7:

$$\{(2, 2, 0), (5, 0, 2), (5, 1, 2), (7, 0, 0), (7, 1, 0)\}$$   \hspace{1cm} (3)

This reduction concludes Step 2.2 in Fig 6. To compute the value queried by the PCTL we need to identify every path from the initial state to some final state of the reduced DFA. Each path is composed by a sequence of transitions and its corresponding labels. The value of these labels are multiplied to obtain the probability of reaching the final state from the initial state through that path. Adding the probabilities of all paths gives us the final probability value queried by the PCTL [20]. Section IV-B details the process of obtaining this value through the corresponding regular expression of the reduced DFA.

**B. From DFA to Formula**

Step 2.3 in Fig 6 consists of obtaining the corresponding regular expression from the reduced DFA. This regular expression is used to compute the final arithmetic formula as proposed by [21]. Regular expressions define the same class of languages that a DFA. DFA has a corresponding regular expression and vice versa [18].

The state elimination algorithm can be used to convert from a DFA to a regular expression. We used the JFLAP tool to model the DFA and compute its regular expression [22]. The regular expression of the reduced DFA is:

$$bel \ast +agl1l1 \ast +agldi11 * +bdii11 *$$   \hspace{1cm} (4)

Where the ' \ast ' is the Kleene star and ' + ' is the union operator, and the concatenation operator (implicit) is defined by two consecutive tokens.
This regular expression is converted into an arithmetic formula using the following recursive definitions [8]:

1) \( \text{val}(P^0) = \frac{p}{q} \)
2) \( \text{val}(x) = x, x \in \sum \)
3) \( \text{val}(r + s) = \text{val}(r) + \text{val}(s) \)
4) \( \text{val}(rs) = \text{val}(r) \cdot \text{val}(s) \)
5) \( \text{val}(r^*) = \frac{1}{1 - \text{val}(r)} \)

Where \( p \) and \( q \) are rational numbers, \( r, s \) are tokens and \( x \) are variables.

Note that the rule 5 is defined only when \( 0 \leq r < 1 \), if \( r = 1 \) then \( \text{val}(r^*) = 1 \). The rules should be applied in the order of precedence of the definition above. Applying these rules we obtain the following arithmetic formula to our guiding example:

\[
b \cdot e + a \cdot g \cdot e + a \cdot g \cdot d \cdot i + b \cdot d \cdot i
\]

Replacing back the variables as defined in Table II we have:

\[
(1 - fSPO2) \cdot (1 - fEKG) + (fSPO2 \cdot 0.999) \\
\times 0.995 \cdot (1 - fEKG) + (fSPO2 \cdot 0.999) \cdot 0.995 \\
\times (fEKG \cdot 0.999) \cdot 0.997 + (1 - fSPO2) \\
\times (fEKG \cdot 0.999) \cdot 0.997
\]

Equation 6 is the same formula generated by PARAM to the model in Listing 2 differing only on some simplifications. The final formula generated by PARAM tool is then:

\[
(4792403 \cdot fSPO2 \cdot fEKG - 1199000000 \cdot fSPO2 \\
- 799400000 \cdot fEKG + 20000000000)/(20000000000)
\]

This concludes Step 2.4 of Fig 6 and complete the entire process of converting a parametric model representing SPL configurations into an arithmetic formula.

According to the strategy used to model the SPL, the formula has two different parameters: \( fSPO2 \) and \( fEKG \). These parameters can be valued with \( 0 \) or \( 1 \) representing the deselection and selection of a feature respectively. The parameter \( fSPO2 \) is used to change the selection of the \( SPO2 \) feature and the parameter \( fEKG \) is used to change the selection of \( EKG \). Section V discusses some aspects of this process and highlights the main aspects related to the formula size.

V. SCALABILITY ANALYSIS

The formula size is strongly related to the DFA. We make an analytical assessment the labels of the DFA and how they impacts the formula size, the size of the regular expression and how it impacts the formula size and we relate these discussions with the modeling strategies.

Then, we show results from a simulation of our example expanded with more features to give a motivation of how quickly the formula size can grow with the number of features. We also discuss some practical implications of large formulas to a implementation in the context of a research project.

To analyze the size of the formula we need to define how that size will be measured. We measure the size of the formula according to the number of operands. Although this measure does not consider the costs of evaluating the different operations, it provides a dimension on the cost of evaluation that can be rounded up (case there are only multiplication, division and power) or down (case there are only additions and subtractions), and provides a notion about the cost of parsing the formula since a large formula has a large number of characters and bytes. Using this metric, the formulas on Section IV-B have the following values: Formula 5 has size 12, Formula 6 has size 20 and Formula 7 has size 9. All the formulas referenced in this Section refer to the formulas presented on Section IV-B.

A. Analytical Assessment

We analyze some aspects that directly impact the size of the formula. First we discuss the labels of the commands in a PARAM model and how they can make the formula grow, secondly we show the correspondence between the regular expression and modeling strategies.

In our example, the formula obtained by the regular expression has twelve operands and the final formula obtained by the PARAM tool has only nine. This is possible because some tokens were substituted by an expression with more than one operand, some of them constants which can be added to a single constant. A worse scenario would be if the tokens were expressions with additions and subtractions of parameters, since the parameters can not be operated. For example, suppose that we replace the token ‘b’ by ‘\( x \cdot y \)’ and ‘e’ by ‘\( z - w \)’ in Formula 5 and apply the distributive property then we would have:

\[
x \cdot z - x \cdot w + y \cdot z - y \cdot w + a \cdot g \cdot z - a \cdot g \cdot w \\
+ a \cdot g \cdot d \cdot i + x \cdot d \cdot i + y \cdot d \cdot i
\]

With this substitution we increase the size of the formula to 24.

Consider Formula 5. It has six different tokens (a,b,d,e,g,i), they come from the variable substitution. Remember that the variable substitution was a strategy to enforce the concept of token during the process. Suppose that each token is substituted by a different parameter. In that case, whatever operators has the formula, it could not have less than six operands, since the parameters can not be operated. The number of parameters affects directly the size of formula since if we have no parameter the formula could be always simplified to a single numerical value, and in the other hand if we have only parameters the formula could not be easily simplified.

Each token in the regular expression is a replacement of a expression of the Table II. These expressions come from the transitions of the PARAM model. Remember that each command in a PARAM model is composed of one or more transition and each transition is labeled by a polynomial that represents its probability. Polynomials are composed by terms and terms are define as a multiplication of a constant and zero or more variables [14]. In our case, these variables are the parameters of the PARAM model. A polynomial can be expressed in a product of sums or in a sum of products.

In our example, we label some transitions with a sum of terms, for example \( (1 - r_{\text{CORE}}) \). Sequential transitions
generates multiplication, this result come from the item 4 of the recursive definition presented in Section IV-B. Thus, sequential transitions labeled by sum of terms generates products of sums. The PARAM tool applies the distributive property trying to simplify the final formula and generates a sum of products. If we have more constants than parameters this heuristic will result in smaller number of operands since the rational constants can be added. As showed in previous examples these products grow in the numbers of terms, and consequently in the number of operands, as we increase the number of different parameters.

The structure of the final formula is obtained from the regular expression correspondent to the DFA. The regular expression obtained by converting the DFA through state elimination algorithm can differ in size and format depending on the order that the states are eliminated [23]. The regular expression can not be minimized efficiently [24]. These results make difficult to estimate the regular expression size.

However, we can use an upper bound of the size of the corresponding regular expression. Although the upper bound represents the worse case scenario, it provides the relation between the attributes of the original DFA and the size of its corresponding regular expression.

The size of the regular expression can be defined in many ways, but the number of alphabetic symbols is considered the most useful measure [25]. This measure is equivalent to the measure we defined to the size of the formula. By alphabetic symbol we mean any symbol that belongs to $\sum$ (Section IV-A). With this measure the size of the regular expression has the following upper bound:

$$|Q| \times |\sum| \times 4^{|Q|}$$ (9)

Where the $|Q|$ is the number of states and $|\sum|$ is the size of the alphabet. This upper bound shows that the number of states impacts exponentially in the worst case and the size of the alphabet affects linearly the size of the regular expression.

Note that the alphabet is composed by tokens, i.e. each expression in Table II is one different alphabet element. The number of parameters will affect the size of the formula since it increases the size of the alphabet. The parameters can increase even more the size of the alphabet since it can be used in expressions that will be considered different tokens.

In order to mitigate the size of the final formula we must observe the points discussed previously. The enumeration below summarizes then and directly relates them to the model:

1) The number of states in the model should be small.
2) The number of parameters in a expression that labels a transition should be small.
3) The number of parameters itself should be small.

Note that the Item 1 is the most critical variable, it affects directly and exponentially the size of the formula. Item 2 can be critical as the number of states sequential transitions labeled by polynomials increase. Item 3 is the less critical variable with linear impact, but the way the parameter is used in the model can be critical, for example, involving them on expressions.

The rules used to build the PARAM model of the example uses bypass commands to skip some transitions. This strategy conflicts with the Item 1 since the additional command to handle variability insert a unnecessary state, from the view point of system documentation, for each point in the code that interacts with components related with features. This strategy also conflicts with the Item 2 since in each of these points we include an transition labeled with a expression involving parameters (1–fSPO2, for example). These solutions should be avoided.

Ghezzi et al. handle variability by using the same parameter to select the feature and represents its reliability, and the same parameter can be used to represent more than one feature. A single parameter is used to select among several features belonging the same variation point, thus, only one parameter per variation point is needed. But this approach is restricted to Alternatives features. Variation points with Alternatives features only have a particular characteristic: this kind of variation point is always filled with one, and only one, feature in a given configuration [15]. Using this characteristic the model is built in such a way that a component can represent several Alternatives features (there is a assumption that each feature maps to one software component). The feature selection is defined by valuating the variation point parameter with the corresponding reliability value of a feature component. This modeling strategy is efficient considering the three items that impact on the formula size presented in this section.

This modeling strategy can be extended to cover variation points with Or and Optional features, but we need first introduce some concepts related to variation points [15]:

- Variation points can be classified as singular if it allows at most one feature related to that variation point in a SPL configuration
- Variation points can also be classified as non singular if it allows more than one feature related to that variation point to be included in a SPL configuration

We need also classify a variation points $VP$ with relation to its size:

- $size(VP)$: maximum number of features that can be selected to bound the variation point of a feature model.

All singular variation points can be handled similarly as an alternative variation point. If the variation point can be empty in a configuration, i.e., no feature is selected to fill it, we use a reliability value of 1. Although the PARAM model will have a module for a feature that is not selected, its valuation with 1 will make it neutral to the formula. Also, a singular variation point can be handled with only one variable.

Non singular variation points can be handled in straightforward way. We need a number of variables and modules in the PARAM model equal to its size. With this we can deal with configurations that selects all features of a variation points and any other combination by replacing the reliability value of non selected features of a configuration by 1 similarly to singular variation points.
Although this approach overcomes the feature model variability limitations it retains some restrictions such as the correspondence among features and components. Also, this approach does not consider the internal behavior of a component that is mapped to a feature.

As future work, we intend to propose a modeling approach aiming to overcome architectural and CK assumptions and variability restrictions. These approaches fits within Step 1 of Fig 3.

This analysis shows that the strategies involving bypass commands and/or expressions will lead to a growth of the formula size and should be avoided. We also propose some guidelines that can be used to extend existing approaches.

B. Practical Assessment

In this section we show from the practical view-point how the formula can grow with the number of features in the example of Section IV and exemplify obstacles faced when we try to evaluate large formulas at runtime.

This simulation shows the rate of growth of the formula in the number of operands. As will be discussed later the modeling strategy of our example can lead to large formulas.

This simulation consists of gradually increase the variability in the feature model of Fig 4 by introducing new features. Fig 8 illustrates the feature model growth, in the figure we have added a new feature ACCELEROMETER. The SENSOR_N represents the nth feature addition. The new features added are also Optional in order to retain the expressiveness as discussed in Section III.

We gradually add new features representing new sensors to our vital signal monitoring system.

![Fig. 8. Feature model extension](image)

Our simulation is composed of several iterations. At each iteration we introduce a new feature to feature model, a new correspondent component to sequence diagram and its correspondent module in PARAM model. Fig 9 presents sequence diagram similarly to the feature model where the component SENSOR_N represents the nth component correspondent to the nth feature added.

Fig 10 shows a graph between the number of features and the size of the final arithmetic formula in number of operands of each iteration. This graphic illustrates how fast the size of the formula can grow with the number of features. The modeling rules used are strongly related with this growth rate. This growth rate can be even more severe in a complex model.

Applying the method presented in Section IV, we generated a formula with 259,470 operators. Because of Java Virtual Machine limitation of the amount of code per method (65536 bytes), we had to split the formula in several methods and call them sequentially to calculate our reliability value [26].

We also tested to parser in runtime the formula using existing libraries 2. However, parsing extensive formulas is a highly cost process.

VI. RELATED WORK

Some contributions address directly the problem of model checking SPL, but all of them have some sort of restriction over variability and/or software architecture.

1Ambient Assisted Living Product Lines - CNPq Project  
2Libraries such as http://code.google.com/p/parity/ and http://projects.congrace.de/exp4j/
Ghezzi et al. propose the use of parametric model checking to verify properties of a software product line, their approach has limitations over the SPL architecture such as each feature must be restricted to one module and, probably the most critical restriction, the feature model must have only alternative features [7], [27]. On the other hand, such modeling approach is very efficient according to our analysis since it does not create any additional state neither uses expressions involving parameters.

Classen et al. proposed a state machine model where the behavior of the system (transitions) is annotated with the related feature [5], [6]. This work presents a model specifically tailored for SPL and handles the concept of feature natively. This approach creates a strongly coupling between the feature model and the architecture system in the model, since the flow of the system should be related with features it makes difficult to represent tangling and scattering. In addition, it does not allow a behavior (transition) on the model to be related with more than one feature. Further model checking over these models does not provide the flexibility of an arithmetic formula of the parametric model checking approach.

VII. CONCLUSION

Estimating software reliability is an important task, specially, for critical systems. This becomes harder when it comes to SPL. Although some contributions address this issue directly, they present some limitations as discussed in Section VI.

This work addresses the problem of making parametric model checking over SPL. We showed a general process to model checking a SPL and presented a detailed view of the parametric model checking of the PARAM tool, from the SPL documentation to the arithmetic formula. We presented an approach to deal with expressiveness limitations and highlighted the scalability implications. This work does not present algorithms and optimization implemented by the PARAM tool, we focus on the theoretical view of the problem.

The main gap of this process is Step 1 in Fig 3. Current approaches have limitations over the software architecture and/or variability. The size of the formula can grow quickly with the number of features if modeling approach does not consider aspects that impact its size. By exposing the concepts and trade-offs involved in the parametric model checking, we can develop new approaches to verify properties, such as reliability, in SPL with parametric model checking. Using the results of this analysis these approaches could make less restrictions on the variability addressed by making trade-offs between expressiveness and the formula size aware of what decisions are more critical to parametric model checking.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for invaluable feedback. This work has been partially supported by CNPq, under Edital MCT/CNPq 14/2009 - Universal - Faixa A, grant number 482481/2009-9.

REFERENCES