Fast Parametric Model Checking through Model Fragmentation

***Abstract*—Parametric model checking (PMC) computes algebraic formulae that express key non-functional properties of a system (reliability, performance, etc.) as rational functions of the system and environment parameters. In software engineering, PMC formulae can be used during design, e.g., to analyse the sensitivity of different system architectures to parametric variability, or to find optimal system configurations. They can also be used at runtime, e.g., to check if non-functional requirements are still satisfied after environmental changes, or to select new configurations after such changes. However, current PMC techniques do not scale well to systems with complex behaviour and more than a few parameters. Our paper introduces a fast PMC (fPMC) approach that overcomes this limitation, extending the applicability of PMC to a broader class of systems than previously possible. To this end, fPMC partitions the Markov models that PMC operates with into *fragments* whose reachability properties are analysed independently, and obtains PMC reachability formulae by combining the results of these fragment analyses. To demonstrate the effectiveness of fPMC, we show how our fPMC tool can analyse three systems (taken from the research literature, and belonging to different application domains) with which current PMC techniques and tools struggle.**

***Index Terms*—Parametric model checking, discrete-time Markov chains, non-functional properties**

I. Introduction

Parametric model checking (PMC) [1]–[4] is a formal technique for analysing the reliability and performance of systems with stochastic behaviour. The underlying concepts are very simple. Consider a foreign exchange trading (FX) system that obtains up-to-date currency exchange rates from two external web services, by first invoking one of the services, and only invoking the other service if the first service is busy (i.e., if its invocation times out). If the probabilities that the two services are busy are p_1 and p_2, then the system will successfully obtain the exchange rates with probability $p_{\mathsf{succ}} = p_1 + (1 - p_1)p2$.

Expressing the non-functional properties of software systems as *closed-form formulae*[1] like this has numerous benefits. At design time, the PMC formulae can be evaluated very efficiently to compare architectures associated with different parameter values, e.g., in software product lines [5], [6]. At runtime, they can be used to efficiently re-verify the satisfaction of non-functional requirements after environmental parameter changes [7], and to select new optimal values for the configuration parameters [8], [9]. They also enable the analysis of the non-functional property sensitivity to variations in the system parameters [10], and the computation of confidence intervals for the analysed non-functional properties [11], [12].

[1]i.e., mathematical expressions containing constants, variables and simple binary operations ($+$, $-$, $\times$, $/$, etc.) that can be computed in constant time

For instance, if the FX system from our earlier example must operate with $p_{\mathsf{succ}} \geq 0.99$ and its developers know (e.g., from service-level agreements) that $p_1 \geq 0.95$, they can compute the minimum acceptable success probability for the second web service as $p_2 = (0.99 - 0.95)/(1 - 0.95) = 0.8$. As another illustration, if unit testing is used to establish $[0.8, 0.9]$ as a .95 confidence interval for both p_1 and p_2 from our FX example, then a $.95^2 \approx .9$ confidence interval for p_{succ} is given by $[\min_{p_1,p_2\in[0.8,0.9]} p_{\mathsf{succ}}, \max_{p_1,p_2\in[0.8,0.9]} p_{\mathsf{succ}}] = [0.96, 0.99]$.

Of course, for non-trivial systems neither the PMC calculations nor the formulae they produce are this simple. In fact, they both become so complex that, despite significant advances in recent years, current PMC methods [1]–[3] and model checkers [13]–[15] do not scale well (i.e., time out, run out of memory, or yield formulae that are too large to evaluate) for many realistic systems with more than a few parameters.

Our paper introduces a fast PMC (fPMC) technique that builds on these advances, enabling the efficient PMC of reachability properties for parametric discrete-time Markov chains (pDTMCs) of systems with more complex behaviour and larger numbers of parameters than previously possible. fPMC *extends* and *automates* a recently proposed theoretical framework for efficient parametric model checking [4].

As shown in Fig. 1, fPMC partitions a pDTMC under analysis into *fragments* (i.e., subsets of model states and transitions that satisfy well-defined rules detailed later in the paper) accompanied by fragment-level reachability properties, and derives an *abstract model* for the analysed pDTMC by replacing each of its fragments with a single state. The fPMC fragments and abstract model are pDTMCs in their own right, and can be analysed individually (using standard PMC techniques) to produce a set of PMC formulae whose combination provides a *closed-form analytical model* for the system-level property of interest.

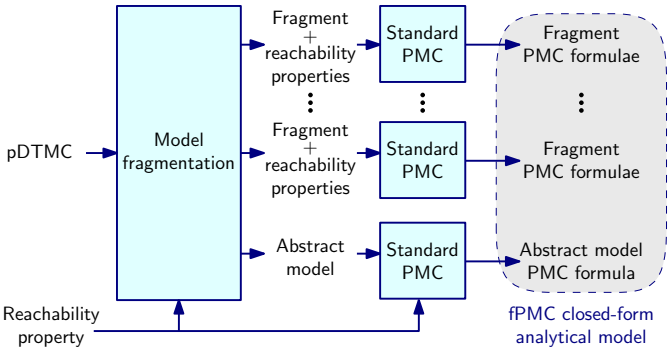


Fig. 1: Fast parametric model checking process

The main contributions of our paper are:

1) The first *model fragmentation* algorithm for the automated partition of pDTMCs into fragments with the characteristics defined by the theoretical framework from [4], which shows how model fragments can be exploited, but does *not* provide any model fragmentation technique, so its applicability is significantly limited (as discussed in Section II).
2) New theoretical results that allow the formation of model fragments from subsets of pDTMC states that do not (initially) meet the definition of a fragment. The new results define valid pDTMC structural modifications that enable the formation of model fragments of the right size for their "standard" PMC analysis to be feasible.
3) A prototype fPMC tool that implements our model fragmentation algorithm and new theoretical results. The fPMC tool invokes the Storm model checker [15] for the PMC of model fragments, fully automating the synthesis of closed-form analytical models for the analysed reachability properties.
4) An extensive evaluation that demonstrates the fPMC effectiveness at analysing pDTMC reachability properties for systems from three application domains.

We organised the rest of the paper as follows. Section II provides the required background on parametric model checking. Section III introduces a running example that we use to illustrate the use of our fPMC technique, which are detailed in Section IV. Sections V and VI describe our fPMC implementation and evaluation, respectively. Finally, Section VII discusses related work, and Section VIII summarises our results and suggests directions for future research.

II. Preliminaries

A. Parametric Model Checking

Parametric model checking (PMC) is a formal technique for the symbolic analysis of Markov chains whose transition probabilities are specified as rational functions over a set of continuous variables [1]–[3]. Formally, a parametric discrete-time Markov chain is a tuple $D = (S, s_0, \mathbf{P}, L)$, where: S is a finite set of states; $s_0 \in S$ is the initial state; $\mathbf{P} : S \times S \to [0, 1]$ is a transition probability matrix such that for any states $s, s' \in S$, $\mathbf{P}(s, s')$ gives the probability of transitioning from s to s', and, for any $s \in S$, $\sum_{s' \in S}, \mathbf{P}(s, s') = 1$; $L : S \to 2^{AP}$ is a labelling function that maps every state $s \in S$ to elements of a set of atomic propositions AP that hold in that state. A state $s \in S$ is *absorbing* if $\mathbf{P}(s, s) = 1$ and $\mathbf{P}(s, s') = 0$ for all $s \neq s'$, and a *transient* state otherwise.

When used for software performance and reliability engineering, the pDTMC states map to relevant configurations of the modelled system; and pDTMC transitions capture possible transitions between those states, corresponding to the feasible changes between their associated configurations. PMC is supported by the probabilistic model checkers PARAM [13], PRISM [14] and Storm [15]. These tools compute *closed-form formulae* for pDTMC properties specified in probabilistic computation tree logic (PCTL) [16]–[18] and defined (as in probabilistic model checking [19]–[23]) by the grammar:

$$\begin{aligned} \Phi &::= true \mid a \mid \Phi \wedge \Phi \mid \neg\Phi \mid \mathcal{P}_{=?}[\Psi] \\ \Psi &::= X\Phi \mid \Phi \,\mathrm{U}\, \Phi \mid \Phi \,\mathrm{U}^{\leq k}\, \Phi \end{aligned} \tag{1}$$

where Φ is a *state formula* and Ψ a *path formula*, $k \in \mathbb{N}_{>0}$ is a timestep bound and $\alpha \in AP$ is an atomic proposition.

The PCTL semantics is defined using a satisfaction relation $\models$ over the states S. Given a state s of a Markov chain D, $s \models \Phi$ means "Φ holds in state s", and we have: always $s \models true$; $s \models a$ iff $a \in L(s)$; $s \models \neg\Phi$ iff $\neg(s \models \Phi)$; and $s \models \Phi_1 \wedge \Phi_2$ iff $s \models \Phi_1$ and $s \models \Phi_2$. The *time-bounded until formula* $\Phi_1 \,\mathrm{U}^{\leq k}\, \Phi_2$ holds for a path iff Φ_1 holds in the first $i < k$ path states and Φ_2 holds in the $(i + 1)$-th path state; and the *unbounded until formula* $\Phi_1 \,\mathrm{U}\, \Phi_2$ removes the bound k from the time-bounded until formula. The *next formula* $X\Phi$ holds if Φ is satisfied in the next state. The state formula $\mathcal{P}_{=?}[\Psi]$ specifies the probability that paths starting at a chosen state s satisfy a path property Ψ. *Reachability properties* $\mathcal{P}_{=?}[\mathsf{true}\,\mathrm{U}\,\Phi]$ are equivalently written as $\mathcal{P}_{=?}[\mathsf{F}\,\Phi]$ or $\mathcal{P}_{=?}[\mathsf{F}\,R]$, where $R \subseteq S$ is the set of states in which Φ holds. For a full description of the PCTL semantics, see [16], [17].

B. Parametric Model Checking Using Model Fragments

We summarise the concept of a pDTMC fragment and its exploitation by the PMC approach introduced in [4], where the results from this section are taken from. As shown in Fig. 2, a fragment of a pDTMC $D = (S, s_0, \mathbf{P}, L)$ is a tuple

$$F = (Z, z_0, Z_{\mathsf{out}}), \tag{2}$$

where: $Z \subset S$ is a subset of transient MC states; z_0 is the (only) *input state* of F, i.e., $\{z_0\} = \{z \in Z \mid \exists s \in S \backslash Z \,.\, \mathbf{P}(s, z) > 0\}$; $Z_{\mathsf{out}} = \{z \in Z \mid \exists s \in S \setminus Z \,.\, \mathbf{P}(z, s) > 0\}$ is the non-empty set of *output states* of F, and all outgoing transitions from the output states are to states outside Z, i.e., $\mathbf{P}(z, z') = 0$ for all $(z, z') \in Z_{\mathsf{out}} \times Z$.

Fragments are not strongly connected components (SCCs) of the *graph induced by the pDTMC*[2] states and transitions. For instance, the "inner states" region of the fragment from Fig. 2 can contain one or more SCCs, and an outgoing transition from an output state in Z_{out} can reach a state s comprising an outgoing transition back to the input state z_0.

The theoretical framework from [4] shows that, if a fragment F of a pDTMC D is identified, then the "monolithic", one-step PMC of reachability, unbounded until and (not covered

[2]i.e., the directed graph comprising a vertex for each pDMTC state and an edge between each pair of vertices that correspond to pDTMC states between which a transition is possible

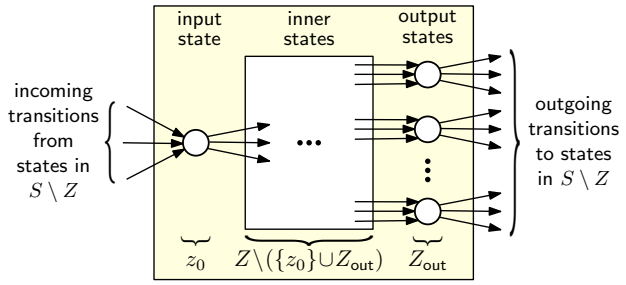


Fig. 2: Fragment of a pDTMC model

in our paper) reward properties [24] of D can be replaced by the following equivalent steps:

1) Compute PMC formulae for the probabilities of reaching all the output states of fragment F starting from its input state z_0; this amounts to the PMC of $\#Z_{\mathsf{out}}$ reachability properties of a pDTMC built from F removing the incoming transitions of z_0 and by replacing the outgoing transitions of each output state in Z_{out} with a "self-loop" transition of probability 1.
2) Build an *abstract pDTMC model* D' from D by replacing the states in Z and their internal transitions with a single state z' whose incoming transitions (and transition probabilities) are those of z_0. Additionally, state z' has outgoing transitions to every state that one or more states from Z_{out} have outgoing transitions to in D; and the probabilities of these transitions can be expressed in terms of the reachability properties computed in step 1. Due to space constraints, we do not provide the expressions for these transition probabilities in the paper; the interested reader can find them in [4].
3) Compute the PMC formula for the original property under analysis, for the abstract model from step 2.
4) Combine the PMC from step 1 and the PMC formula from step 3 into a system of equations.

The system of equations from step 4 provides a closed-form analytical model for the analysed property. This analytical model is equivalent to the PMC formula obtained by analysing the original pDTMC in one step.

Because the PMC analyses from steps 1 and 3 are performed on models that are smaller and simpler than D, this four-step PMC approach is often faster, produces much simpler closed-form expressions, or succeeds for more complex models than monolithic PMC can handle. However, no method for partitioning pDTMCs into fragments is proposed in [4]. Instead, the approach is only exploited for specific types of component-based systems (i.e., service-based systems and multi-tier architectures) for which: (i) pDTMCs can be obtained by combining manually pre-built pDTMC models of their components; and (ii) the component pDTMCs form fragments in the system pDTMC. This limits the applicability of the approach (to specific types of systems for which new pDTMCs are assembled following this recipe), requires additional effort and expertise, and can be error prone due to the manual steps involved.

III. Running Example

We illustrate our fPMC approach using a six-operation service-based system from the area of foreign exchange trading (FX) introduced in [25]. The FX system implements the workflow shown in Fig. 3 and described briefly below.

FX description. A trader can use FX in two execution modes. In the *expert* mode, FX runs a loop that analyses market activity, identifies patterns that satisfy the trader's objectives, and automatically carries out trades. Thus, the *Market watch* operation extracts real-time exchange rates (bid/ask price) of selected currency pairs. This data is used by a *Technical analysis* operation that evaluates the current trading conditions, predicts future price movement, and decides if the trader's objectives are: (i) "met" (causing the invocation of an *Order* service to carry out a trade); (ii) "not met" (resulting in a new *Market watch* invocation); or (iii) an error occured (triggering an *Alarm* operation to notify the trader about discrepancies/opportunities not covered by the trading objectives). In the *normal* mode, FX assesses the economic outlook of a country using a *Fundamental analysis* operation that collects, analyses and evaluates information (e.g., news reports, economic data and political events), and provides an assessment on the country's currency. If satisfied with this assessment, the trader can use the *Order* operation to sell/buy currency; then, a *Notification* operation confirms the completion of the trade.

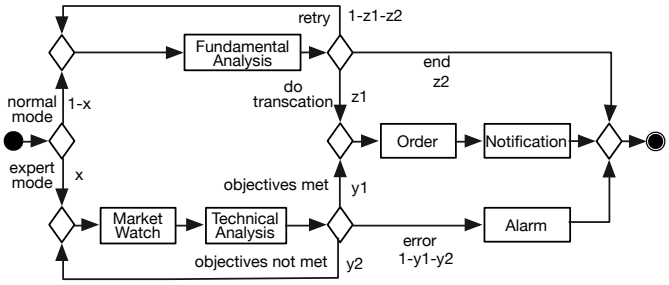


Fig. 3: FX service-based system, where x, $y1$, $y2$, $z1$, $z2$ are the (unknown) probabilities of different execution paths

Given its business-critical nature, assume that the software architect aims at designing an FX system with high reliability. Thus, for the i-th operation FX uses two functionally-equivalent service implementations and adopts a *sequential execution strategy with retry (SEQ_R)* based on which the two services per operation are invoked in order. If the first service fails, it is re-invoked with probability r_{i1}, whereas the operation is attempted using the second service with probability $1-r_{i1}$. If the execution of the second service fails, it is retried with probability r_{i2}; otherwise, with probability $1 - r_{i2}$, the entire system execution fails. Finally, assume that the selected service implementations per FX operation is based on the analysis results of the system-level probability of successfully completing the handling of a request.

FX pDTMC. Fig. 4 shows the pDTMC of the FX system specified in the modelling language of the PRISM model checker [14]. The model comprises a *WorkflowFX* module modelling the FX workflow (lines 9–52), and the parameters associated with the operational profile (line 2) and with the service implementations for each FX operation (lines 3–8); p_{ij} and r_{ij} signify the probability of successful execution and the probability of retrying the i-th operation using the j-th service implementation, respectively. Within the *WorkflowFX* module, the local variable s (line 10) models the state of the FX system, while op_i (line 12–13) and *retry* (line 11) encode the employed service implementation per operation and the retry status of a service implementation, respectively. Following the selection of the expert FX mode (line 15), the *Market watch* operation with retry and two service implementations is executed (lines 16–20). The first service implementation

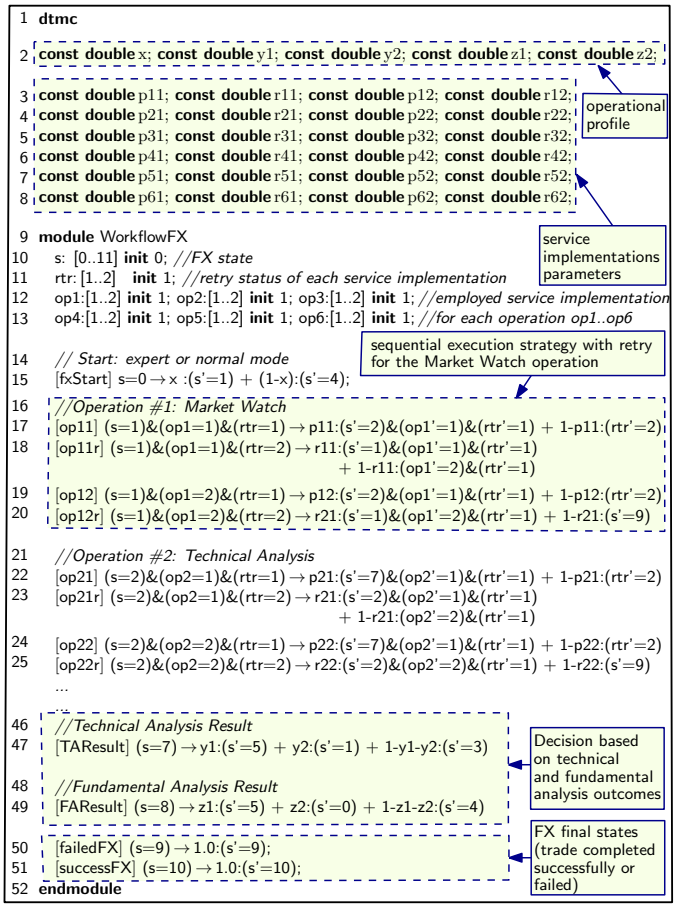


Fig. 4: pDTMC model of the FX system.

succeeds with probability p_{11} and FX moves to the *Technical analysis* operation, fails with probability $1 - p_{11}$ and retries with probability r_{11} (lines 17–18); otherwise, the second service implementation is executed and succeeds or is re-invoked with probabilities p_{12} and r_{12}, respectively. If both service implementations fail, the FX execution terminates (line 50). The other FX operations function similarly (lines 21–45) but we omit their details due to space constraints. The pDTMC model of the FX system produces the directed graph shown in Fig. 5, which comprises 29 states and 58 transitions and with the initial, failed and successful states coloured in blue, red and green, respectively.

Given this pDTMC model, we used PRISM [14] and Storm [15] to obtain the closed-form PMC formula for the probability of successfully handing a request (i.e., to reach the succeed state) encoded in PCTL as $\mathcal{P}_{=?}[\mathsf{F}\ s = 10]$. Despite the small number of states and transitions, neither model checker was able to compute the formula within an hour (on a computer with the specification from Section VI-B). We explain next how fPMC supports the computation of those formulae through automated model fragmentation.

IV. fPMC Approach

A. Markov Chain Fragmentation Algorithm

The fPMC partition of a pDTMC into fragments is performed by function FRAGMENTATION from Algorithm 1. This

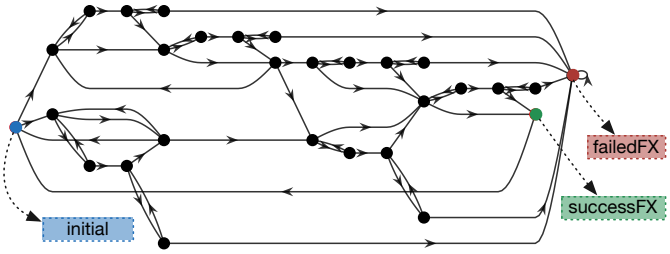


Fig. 5: The directed graph induced by the FX pDTMC from Fig. 4, showing the initial, failedFX and successFX states.

function takes three arguments:

- the graph $\mathcal{G}(V, E)$ induced by the pDTMC;
- the set of states R whose reachability $\mathcal{P}_{=?}[\mathsf{F}\ R]$ is being analysed;
- a *threshold* parameter $\alpha \geq 1$ whose role is described below,

and returns a set of pDTMC fragments FS, i.e., a set of tuples with the structure from Equation (2). As not all pDTMC states can be included into a fragment that follows the definition from Section II-B (e.g., states whose only outgoing transition is a self-loop transition do not satisfy the constraints associated with any type of fragment state), we extend this definition to include single-state fragments $F = (\{s\}, s, \{s\})$ for any pDTMC state s.

The algorithm starts by placing into FS single-state fragments for each state in the reachability set R (line 2). We preserve the elements of R as single-state fragments so they end up as states in the fPMC abstract model (see Fig. 1 and the description from Section II-B), so the reachability property $\mathcal{P}_{=?}[\mathsf{F}\ R]$ (which refers to these states) can be analysed for the abstract model. Next, the algorithm generates additional fragments in each iteration of the for loop from lines 3–25 as follows. First, a node z_0 not yet included in any fragment is selected (line 3) and inserted into the fragment state set Z, while the fragment output set Z_{out} is initialised to the empty set (line 4). A stack T is then populated with the states reached by outgoing transitions from z_0 by invoking (in line 6) the function TRAVERSE from Algorithm 2. Each state w from this stack is processed by the while loop from lines 7–19, ending up in Z_{out} if it satisfies the constraints associated with output fragment states (lines 9 and 10). When w does not satisfy these constraints, two options are possible (lines 12–16):

- If Z has accumulated fewer states than the threshold α, the graph traversal function TRAVERSE is invoked again to add to the stack the predecessor and successor vertices of w that are not already in the fragment (line 13).
- Otherwise, a restructuring of the graph (see Section IV-B) is invoked (line 15) to "force" w into becoming an output state, and thus to enable the formation of a fragment.

In this way, the threshold α provides a soft upper bound for the fragment size. When this bound is reached, the model restructuring techniques detailed in Section IV-B are used to force the formation of a valid fragment. The fragment candidate (Z, z_0, Z_{out}) finalised by the while loop is then

Algorithm 1 pDTMC model fragmentation

```
1: function FRAGMENTATION(G(V, E), R, α)
2:     FS ← {({r}, r, {r}) | r ∈ R}
3:     for all z_0 ∈ V \ R do
4:         Z ← {z_0}, Z_OUT ← {}
5:         T ← EMPTYSTACK()
6:         TRAVERSE(G, z_0, T, FS, R, Z, true)                    ▷ Alg. 2
7:         while ¬EMPTY(T) do
8:             w ← T.POP()
9:             if {i | (i,w)∈E} ⊆ Z ∧ {o | (w,o)∈E} ⊆ V\Z then
10:                Z_OUT ← Z_OUT ∪ {w}
11:            else
12:                if #Z < α then
13:                    TRAVERSE(G, w, T, FS, R, Z, false)          ▷Alg. 2
14:                else
15:                    RESTRUCTURE(G, w)                           ▷Section IV-B
16:                end if
17:            end if
18:            Z ← Z ∪ {w}
19:        end while
20:        if ¬VALIDFRAGMENT((Z, z_0, Z_OUT)) then
21:            Z ← {z_0}, Z_OUT ← {z_0}
22:        end if
23:        FS ← FS ∪ {(Z, z_0, Z_OUT)}
24:        R ← R ∪ Z
25:    end for
26:    return FS                                                  ▷ set of fragments
27: end function
```

Algorithm 2 Traversal of pDTMC induced graph

```
1: function TRAVERSE(G(V, E), w, T, FS, R, Z, isInput)
2:     if ¬isInput then                                  ▷ w is not the fragment's input
3:         I ← {i | i ∉ Z ∧ (i, w) ∈ E}
4:         if I ⊆ V \ R then
5:             T.PUSH(I)
6:         else
7:             FS ← FS ∪ {({w}, w, {w})}
8:             R ← R ∪ {w}
9:             return
10:        end if
11:    end if
12:    O ← {o | o ∉ Z ∧ (w, o) ∈ E}
13:    if O ⊈ R then
14:        for all o ∈ O do
15:            if o ∉ R then
16:                T.PUSH(o)
17:            else
18:                RESTRUCTURE(G, w)                          ▷ Section IV-B
19:            end if
20:        end for
21:    end if
22: end function
```

"downgraded" to a single-state fragment if it does not meet the definition from Section II-B (lines 20–22), after which it is added to the fragment set FS (line 23). Additionally, the states Z of this fragment are added to the set R of states already assigned to fragments (line 24), ensuring that the loop starting in line 3 does not reuse them to form other fragments.

Finally, the TRAVERSE function from Algorithm 2 takes a vertex w and examines its incoming edges (if w is not the initial fragment state z_0, lines 2–11), and its outgoing edges (at all times, lines 12–21). Vertices connected to w by these incoming and outgoing edges and not already in the set of fragment states Z are collected into the input vertex set I (line 3) and output vertex set O (line 12), respectively. The former are then added to the stack T if none of them belongs to an existing fragment (lines 4 and 5). Otherwise, the non-input vertex w has incoming edge(s) from another fragment, violating the condition that incoming transitions to inner and output fragment states are only allowed from states within the same fragment; hence, w becomes a single state fragment and the graph traversal is terminated (lines 6–9). Lastly, if the output set O has at least a vertex not in other fragments (line 13), growing the fragment under construction with w's successors may be feasible, and thus the for loop in lines 14–21 places the O vertices not in other fragments on the stack T (line 16) and attempts to restructure the graph to allow fragment formation if these vertices are already part of other fragments (line 18).

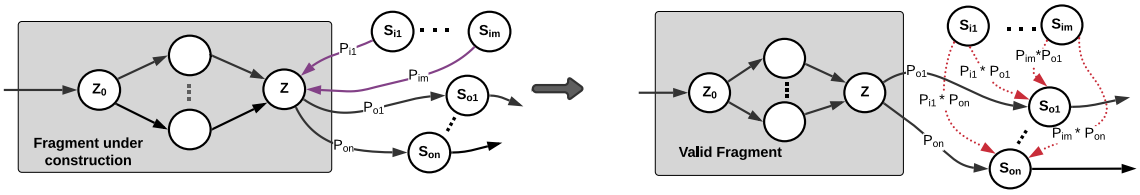
Due to space constraints, we do not provide a full analysis of Algorithms 1 and 2, but we note that they are guaranteed to produce a set of valid fragments, comprising: (i) the degenerate, single-state fragments created in lines 2 and 21 of FRAGMENTATION and in line 7 of TRAVERSE; (ii) the fragments that "pass" the validation from line 20 of FRAGMENTATION, which we assume correct. Furthermore, both algorithms terminate. TRAVERSE terminates because each of its statements (including the assembly of the vertex sets I and O, and its only for loop) operate with a finite number of vertices. FRAGMENTATION terminates because: (i) each iteration of its for loop adds at least one vertex (i.e., z_0) to R in line 24 until $V \setminus R = \{\}$ in line 3 (since V is a finite set) and the loop terminates; (ii) its while loop terminates since it iterates over the elements of stack T that can only contain one instance of vertices from the finite set V and is therefore finite too; and (iii) RESTRUCTURE invocations can only add a finite number of vertices to V, as we show in the next section.

B. Model Restructuring to Aid Fragment Formation

While the model fragmentation from §IV-A is guaranteed to produce valid fragments, the success of fPMC also depends on these fragments being neither too small nor too large. Small fragments may yield a large abstract model that may be unfeasible to analyse using standard PMC (Fig. 1); however, in our experience, the pDTMCs whose PMC we want to speed up comprise many loops (cf. Fig 5) that preclude the formation of only small fragments. In contrast, fragments that are too large for standard PMC are more likely to be obtained. As such, fPMC uses the threshold α to decide when to attempt to force the formation of a fragment in line 15 from Algorithm 1, and further attempts to enable fragment formation in line 18 from Algorithm 2, in the two scenarios below.

1) When a state z of a fragment under construction has both outgoing transitions to inner fragment states and $n \geq 1$ outgoing transitions (of probabilities p_1, p_2, …, p_n) to states s_1, s_2, … s_n not in the fragment (Fig 6a, left). In this case, the addition of one auxiliary state z'_i for each outgoing transition to a state s_i, $i = 1, 2, \ldots, n$ (Fig 6a, right) enables the

(a) Auxiliary state insertion to force formation of fragment containing the new states among its output states



(b) Transition replacement to force creation of output fragment state

Fig. 6: Model restructuring techniques supporting fragment formation

formation of a fragment that includes the auxiliary states as output states. Each auxiliary state z'_i has an incoming transition of probability p_i from state z, and an outgoing transition of probability 1 going to state s_i.[3]

2) When a non-input state z of a fragment under construction has $m \geq 1$ incoming transitions (of probabilities p_{i1}, p_{i2}, …, p_{im}) from states s_{i1}, s_{i2}, $\ldots s_{im}$ outside the fragment, and $n \geq 1$ outgoing transitions (of probabilities p_{o1}, p_{o2}, …, p_{on}) to states s_{o1}, s_{o2}, $\ldots s_{on}$ not in the fragment (Fig 6b, left). In this case, each of the m transitions from a state s_{ij} to z, $j = 1, 2, \ldots, m$, can be replaced by n transitions from state s_{ij} to states s_{o1}, s_{o2}, $\ldots s_{on}$ (Fig 6b, right), where the probability of transition from s_{ij} to s_{ok}, $k = 1, 2, \ldots, n$, is set to $p_{ij}p_{ok}$.

Theorem 1. *Applying the model restructuring techniques from Fig. 6 to a pDTMC does not affect its reachability properties.*

Proof. Consider the sets Π and Π' of all paths (i.e., sequences of possible states and state transitions) that satisfy a reachability property P of a pDTMC $D = (S, s_0, \mathbf{P}, L)$ before and after model restructuring is applied to change it to $D' = (S', s_0, \mathbf{P}', L)$, respectively. According to the semantics of PCTL [16], [17], we need to show that $\Pr_{s_0}(\Pi) = \Pr'_{s_0}(\Pi')$, where $\Pr_{s_0}$ is a probability measure defined over all paths $\pi = s_0 s_1 s_2 \ldots s_n$ starting in the initial state s_0 of pDTMC D such that $\Pr_{s_0}(\pi) = \prod_{i=0}^{n-1} \mathbf{P}(s_i, s_{i+1})$, and $\Pr'_{s_0}$ is a similarly defined probability measure for D'. We focus on the paths that differ between Π and Π', and show that $\Pr_{s_0}(\Pi \setminus \Pi') = \Pr'_{s_0}(\Pi' \setminus \Pi)$ for each technique in turn.

1) A path from $\Pi \setminus \Pi'$ has the form $\pi = s_0\omega_1 z s_i \omega_2$ for some $i \in \{1, 2, \ldots, n\}$ and subpaths ω_1, ω_2, with ω_2 ending in one of the states from the reachability state set. For any such path π, there is a corresponding path $\pi' = s_0\omega_1 z z'_i s_i \omega_2 \in \Pi' \setminus \Pi$, *and the other way around*. We have $\Pr_{s_0}(\pi) = \Pr_{s_0}(s_0\omega_1)\mathbf{P}(z, s_i)\Pr_{s_0}(s_i\omega_2) = \Pr_{s_0}(s_0\omega_1)p_i\Pr_{s_0}(s_i\omega_2) = \Pr'_{s_0}(s_0\omega_1)\ \mathbf{P}'(z, z'_i)\mathbf{P}'(z'_i, s_i)\ \Pr'_{s_0}(s_i\omega_2) = \Pr'_{s_0}(\pi')$, so the theorem holds for the first restructuring technique.

[3]Using a single auxiliary state z' obtained by combining states z'_1, z'_2, …, z'_n is also possible, but leads to more complex expressions for the new transition probabilities, so we did not opt for it in the current version of fPMC.

2) A path from $\Pi \setminus \Pi'$ has the form $\pi = s_0\omega_1 s_{oj} z s_{ik}\omega_2$, with $j \in \{1, 2, \ldots, m\}$, $k \in \{1, 2, \ldots, n\}$, and subpath ω_2 ending in one of the states from the reachability state set. Path π has a corresponding path $\pi' = s_0\omega_1 s_{oj} s_{ik}\omega_2 \in \Pi' \setminus \Pi$, *and the other way around*, and it is straightforward to show that $\Pr_{s_0}(\pi) = \Pr'_{s_0}(\pi')$ since $\mathbf{P}(s_{ij}, z)\mathbf{P}(z, s_{ok}) = \mathbf{P}'(s_{ij}, s_{ok})$, which completes the proof. □

From the two model restructuring techniques, only the first increases the number of pDTMC states – with as many auxiliary states as there are ougoing transitions from state z to states outside the fragment under construction. Because (i) the pDTMC has a finite number of states, each with a finite number of outgoing transitions; and (ii) the auxiliary states do not require the application of the first technique (as they have a single outgoing transition), it follows that the maximum number of auxiliary states that fPMC may create is also finite.

C. fPMC *Application to the Running Example*

Applied to the pDTMC and reachability property for the FX system from our running example (which current parametric model checkers cannot analyse, cf. §III), our fPMC tool (run for $\alpha = 6$) generated the model fragmentation from Fig. 7. The five single-state and eight multi-state fragments (13 fragments in total) comprise 49 states and 83 transitions compared to the 29 states and 58 transitions of the original pDTMC (Fig. 5), with the additional states and transitions due to the model restructuring techniques from §IV-B. The end-to-end computation of a closed-form analytical model for the FX success probability took fPMC 4.43s, and the algebraic formulae of this model contain 1,456 arithmetic operations and took 0.002s to evaluate in Matlab. As for all fPMC experiments presented in the paper, we carefully checked the correctness of our PMC formulae by ensuring that their evaluation for randomly generated combinations of parameter values produced the same numerical results (subject to negligible rounding errors) as those obtained by running the probabilistic model checkers PRISM [14] and Storm [15] to analyse the non-parametric Markov chains obtained by replacing the pDTMC parameters with these random values.

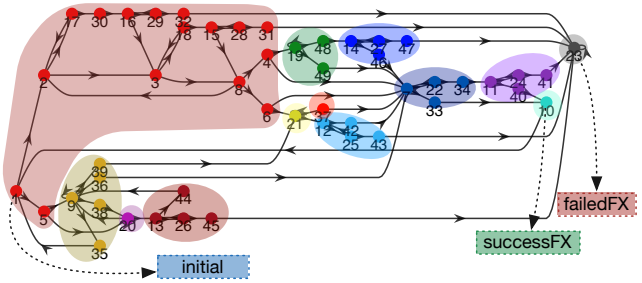


Fig. 7: Fragmentation of the pDTMC model from the running example (with the 13 fragments depicted in different colours)

V. Implementation

To ease the evaluation and adoption of fPMC, we developed a prototype Java tool that implements Algorithms 1 and 2 from §IV. The tool invokes the model checkers PRISM [14] and Storm [15] to obtain the pDTMC transition probability matrix, and to compute the PMC formulae for the pDTMC fragments and abstract model, respectively. The open-source fPMC prototype, the full experimental results summarised next, additional information about fPMC and the case studies used for its evaluation are available at https://bit.ly/3l0ygdi.

VI. Evaluation

A. Research Questions

RQ1 (Effectiveness): Can model fragmentation improve PMC efficiency and extend its applicability? We assess if fPMC can speed up PMC compared to PRISM [14] and Storm [15], and whether our approach enables the analysis of models that these probabilistic model checkers cannot handle.

RQ2 (Scalability): How does the number of parameters in a pDTMC model affect the ability of fPMC to fragment the model and compute closed-form formulae? Since the number of pDTMC model parameters is a factor influencing the performance of current PMC solutions, we investigate how fPMC performs as the number of model parameters increases.

RQ3 (Configurability): What is the impact of the hyperparameter α on fPMC? We examine how different threshold α values, i.e., the only fPMC hyperparameter (cf. Algorithm 1), affect fPMC in terms of the number of operations in the computed closed-form analytical model and execution time.

B. Experimental Setup

We evaluated fPMC for three software systems and processes taken from related research [4], [5], [25]–[27] and belonging to different application domains. We selected these systems because (i) their Markov models include hyperparameters that can be tuned to devise pDTMCs with various numbers of states S and transition probability matrices $\mathbf{P}$; and (ii) without changing the pDTMC structure, several non-trivial transition probabilities in $\mathbf{P}$ can become parameters, thus increasing the number of parameters within the model. Due to space constraints, we only provide brief descriptions of these systems; their full details are available from [4], [5], [25]–[27].

FX System. We have presented the high-level overview of the FX system in Section III and the pDTMC corresponding to the sequential execution strategy with retry (SEQ_R) in Fig. 4. We also consider the strategies below for executing between 1–5 functionally-equivalent service implementations per operation:

SEQ: The services are invoked in order, stopping after the first successful invocation or after the last service fails.

PAR: All services are invoked in parallel (i.e., simultaneously), and the operation uses the result returned by the first service terminating successfully.

PROB: A probabilistic selection is made among the available services with the total probability being equal to 1.

PROB_R: Similar to PROB, but if the selected service fails, it is retried with a given probability (as in SEQ_R).

PL System. This product line system (PL) [5], [27] models the process in various vending machines that enable a user to insert coins and select a beverage, based on which the vending machine delivers the beverage and, if needed, gives changes. The features of this system comprises the beverage type (soda, tea, or both), payment mode (cash or free) and taste preference (e.g., add lemon, sugar) enabling the derivation of vending machines, and accordingly the definition of pDTMCs, whose structures contain between 4 and 22 features.

COM Process. We consider a communication (COM) process [26] among n identical individuals, inspired by the way that honeybees emit an alarm pheromone to recruit workers and protect their colonies from intruders. Due to the self-destructive defence behaviour in social insects, the recruited workers die after completing their defence actions. Hence, a balance between efficient defence and preservation of a critical workers mass can be found. The induced pDTMC is a stochastic population model with n parameters.

We compare fPMC against the leading PMC model checkers PRISM (version 4.6) and Storm (version 1.5.1), both with their default settings. For a fair comparison, we ensured that both PRISM and Storm can process at least the simpler pDTMCs of these systems. We run each experiment using fPMC, PRISM and Storm, and observe (i) the time required to compute the PMC formulae, with a 60-minute timeout (the lower the execution time the better); and (ii) the number of arithmetic operations in the devised formulae (formulae with fewer arithmetic operations can be evaluated faster).

All experiments were performed on a MacBook Pro (early 2015) with 2.7GHz dual Core Intel i5 processor and 8GB RAM. The source code, the full Markov models and reachability properties, and all experimental results are publicly available at https://bit.ly/3l0ygdi.

C. Results & Discussion

RQ1 (Effectiveness). Table I shows the execution time, the number of arithmetic operations of the computed closed-form formulae and the number of fragments devised by fPMC using FX system variants with different execution strategies and number of functionally-equivalent service implementations per operation. The size of the derived pDTMCs ranges from 11 states and 22 transitions (SEQ with 1 service implementation) to 208 states and 399 transitions (PAR with 5 services).

TABLE I: Parametric model checking for the FX system showing the execution time, the number of arithmetic operations of the closed-form formulae and the number of fragments devised by fPMC.

STG	#SRV	#S	#T	Time (s)				#arithmetic operations				#fragments	
				fPMC	fPMC (α=∞)	Storm	PRISM	fPMC	fPMC (α=∞)	Storm	PRISM	fPMC	fPMC ($\alpha=\infty$)
SEQ	1	11	22	2.588	2.814	0.017	0.459	228	228	157	160	4	4
	2	17	34	2.684	3.448	2.31	51.46	479	479	35488	34074	6	6
	3	23	46	3.563	5.212	T	T	1374	1374	–	–	7	7
	4	29	58	4.366	5.195	T	T	3304	3304	–	–	9	9
	5	35	70	7.973	7.770	T	T	6481	6481	–	–	10	10
PAR	2	40	36	3.717	3.834	1.743	1.321	909	909	35519	32134	18	18
	3	64	111	5.369	6.751	T	T	7952	7952	–	–	26	26
	4	112	207	14.274	101.949	T	T	6062	68574	–	–	27	41
	5	208	399	171.903	T	T	T	24502	–	–	–	43	–
PROB	2	23	46	3.385	4.303	0.377	8.352	743	743	6517	20991	8	8
	3	29	64	4.533	5.618	3.735	T	1970	1970	47812	–	10	10
	4	35	82	6.253	7.322	34.836	T	4136	4136	196985	–	12	12
	5	41	100	10.694	13.016	619.505	T	7508	7508	593426	–	14	14
SEQ_R	2	29	58	5.340	216.792	T	T	3068	124854	–	–	9	3
	3	41	82	232.811	T	T	T	37317	–	–	–	12	–
	4	53	106	173.624	T	T	T	131336	–	–	–	30	–
	5	65	130	1561.751	T	T	T	385893	–	–	–	36	–
PROB_R	2	29	58	4.790	34.037	54.822	T	1735	27251	84449	–	15	3
	3	35	75	15.261	T	T	T	4832	–	–	–	19	–
	4	41	93	57.187	T	T	T	10437	–	–	–	23	–
	5	47	111	179.623	T	T	T	19306	–	–	–	27	–

Notation – STG: strategy adopted for the functionally-equivalent service implementations per FX operation; #SRV: available service implementations per operation; #S: pDTMC states; #T: pDTMC probabilistic transitions; fPMC: fPMC with $\alpha=15$ except from SEQ_R #SRV=4,5 where $\alpha=3$; fPMC ($\alpha=\infty$): fPMC variant where the invocation of the RESTRUCTURE function in Algorithm 1 is disabled; T: timeout–no result returned in 60min.

TABLE II: Parametric model checking for four variants of the PL system (with 4,16,18 and 22 features) showing the execution time and the number of arithmetic operations (#OP) of the computed closed-form formulae for different percentages of pDTMC model parameters. For the pDTMC with 4,16 and 18 features α=10, and for the pDTMC with 22 features α=5.

pDTMC	Model checker		Percentage of parameters in the pDTMC										
			P01	P10	P20	P30	P40	P50	P60	P70	P80	P90	P100
4 Features #S=92 #T=167 #params=75	fPMC	Time (s)	40.382	44.838	89.045	58.278	54.101	65.933	73.454	102.109	102.013	103.623	103.063
		#OP	38681	46975	50794	54903	55402	67449	75144	94483	94514	94757	92295
	Storm	Time (s)	0.002	0.01	0.221	3.195	32.000	1674.020	T	T	T	T	T
		#OP	3	772	6560	24048	39805	286349	800693	–	–	–	–
	PRISM	Time (s)	0.215	0.249	0.321	1.476	5.088	105.520	T	T	T	T	T
		#OP	1	29	578	5738	42062	475809	–	–	–	–	–
16 Features #S=110 #T=193 #params=83	fPMC	Time (s)	18.335	53.179	52.260	20.162	20.077	23.309	22.547	22.783	23.634	23.778	23.790
		#OP	58052	82776	81293	86648	86684	107475	107166	107339	107734	107751	107769
	Storm	Time (s)	0.001	0.361	15.709	23.489	31.624	164.624	164.941	T	T	T	T
		#OP	3	772	6560	24048	39805	286349	800693	–	–	–	–
	PRISM	Time (s)	0.353	14.917	T	T	T	T	T	T	T	T	T
		#OP	1	606	–	–	–	–	–	–	–	–	–
18 Features #S=104 #T=183 #params=79	fPMC	Time (s)	13.677	13.828	13.719	15.286	14.294	14.431	14.368	14.844	17.464	17.347	17.471
		#OP	25703	27643	27643	31499	31672	35523	35528	39400	70816	76303	76388
	Storm	Time (s)	0.02	0.043	0.116	0.554	18.773	911.039	T	T	T	T	T
		#OP	4	104	206	13101	75459	1179040	–	–	–	–	–
	PRISM	Time (s)	0.343	0.963	1.084	2.499	41.665	86.148	188.956	T	T	T	T
		#OP	4	82	177	10458	82654	1218965	3909823	–	–	–	–
22 Features #S=115 #T=198 #params=83	fPMC	Time (s)	24.268	27.804	46.97	59.16	49.217	42.943	48.598	50.338	49.306	50.015	66.467
		#OP	7731	7902	10431	210456	210464	210464	210473	222304	222304	260538	260465
	Storm	Time (s)	0.002	0.035	3.53	42.334	793.458	T	T	T	T	T	T
		#OP	4	104	206	13101	75459	1179040	–	–	–	–	–
	PRISM	Time (s)	0.548	1.821	53.239	T	T	T	T	T	T	T	T
		#OP	4	59	1225	–	–	–	–	–	–	–	–

Notation – #S: pDTMC states; #T: pDTMC probabilistic transitions; #params: total number of pDTMC parameters; $P01$,...,$P100$: percentage of pDTMC parameters maintained – the others are set to random values $\in [0,1]$ preserving the pDTMC structure; T: timeout–no result returned within 60 minutes.

For all pDTMCs of FX variants, fPMC succeeded in computing all closed-form formulae within the 60 minutes timeframe (fPMC column) taking $2.5s$ for the simplest models, and just under 240s for most models with the exception of SEQ_R with 5 services for which fPMC took $\approx 1560s$. In contrast, Storm computed the formulae for eight of 21 pDTMCs ($\approx 38\%$) with the majority being the simplest models across all execution strategies, except from PROB variants for which Storm produced all formulae. Finally, PRISM computed the PMC formulae for four of the 21 pDTMCs ($\approx 20\%$), which again were the pDTMCs with the fewest states and transitions.

Similar results were obtained for the four analysed PL system variants (with 4, 16, 18 and 22 features) whose results are shown in Table II. Considering column P100, for instance, the most difficult case in which the transition probability matrix $\mathbf{P}$ comprises only parameters (i.e., no transition is set to a constant value), fPMC returned the closed-form formulae in less than $\approx 104s$ in the worst case. Neither Storm nor PRISM

TABLE III: Parametric model checking of reachability properties $P1$,...,$P21$ from [26] for the COM process with $n=20$ individuals (parameters) and $\alpha=50$, showing the execution time and the number of arithmetic operations of the derived formulae. The pDTMC has 234 states and 444 transitions.

P#	Time (s)			#arithmetic operations (M: Megabytes)			
	fPMC	Storm	PRISM	fPMC	fPMC (abstract model)	Storm	PRISM
1	8.894	0.000	0.747	98568	1	2	20
2	8.848	0.001	0.648	98567	15	6	780
3	9.295	0.004	0.847	98791	224	1352	11724
4	9.239	0.030	1.241	1M	1561	16837	97814
5	9.146	0.427	3.002	105314	6747	122879	544375
6	9.025	0.461	4.867	118846	20279	610454	2M
7	8.340	1.768	OM	143612	45045	2M	–
8	9.353	15.065	OM	175280	76713	5M	–
9	8.898	18.870	OM	201098	102531	12M	–
10	8.802	94.334	OM	207676	109109	101M	–
11	9.162	73.880	OM	191660	93093	150M	–
12	8.976	316.619	OM	162203	63636	–	–
13	9.160	130.994	OM	133069	34502	167M	–
14	8.876	39.222	OM	113092	14525	130M	–
15	9.247	22.488	OM	103173	4606	82M	–
16	9.223	8.401	9.490	99646	1079	9M	–
17	9.077	3.853	3.936	98763	196	3M	4M
18	8.911	0.587	1.952	98595	28	1M	1M
19	9.353	0.128	0.928	98568	1	263799	300712
20	8.653	0.019	0.638	98568	1	38832	41439
21	9.124	0.004	0.411	98568	1	2861	2870

succeeded in any of the four PL system variants for $P100$.

While fPMC clearly outperforms both Storm and PRISM when handling complex models with several parameters, we noticed an interesting behaviour with simpler models (for FX and PL) and simpler properties (for COM). In particular, for the simplest FX models with SEQ, PAR and PROB strategies and 1–2 services (cf. Table I) mostly Storm, and rarely PRISM, computed the formulae faster than fPMC. This behaviour occurs for PL system models up to $P30$ in Table II (30% of the total pDTMC parameters were maintained) and properties $P1$–$P7$, $P16$–$P22$ for the COM process (Table III). Since both Storm and PRISM represent internally the induced pDTMCs with advanced data structures (e.g., sparse matrices, binary decision diagrams) and use sophisticated reachability analysis algorithms, this behaviour is logical for models and reachability properties that can exploit these features. As we have demonstrated, however, these features alone cannot handle reachability properties for pDTMCs with characteristics similar to those of the most complex models in Tables I–III. This observation indicates the potential of a hybrid probabilistic model checker that uses fPMC and Storm/PRISM interchangeably based on the pDTMC structure and reachability property.

Considering the complexity of the derived closed-form formulae as a factor of the number of involved arithmetic operations, we observe that for all systems and all variants, the complexity increases as the model complexity (states and transitions) increases. For instance, for the FX system with SEQ_R strategy (the motivating example) the arithmetic operations for the fPMC-computed formulae increase from 3068 to $\approx 386K$ as the number of functionally-equivalent services increases from two to five. In general, the fPMC-produced formulae contain fewer operations than those produced by Storm or PRISM. This observation is evident in Table III, where the closed-form analytical models derived from fPMC include $98K$–$208K$ operations for all properties, whereas, in many cases, the Storm- or PRISM-produced formulae are very large (several Megabytes (M) - up to 167M) and could not be analysed using our experimental machine.

These findings clearly indicate that fPMC, underpinned by the model fragmentation algorithm, significantly speeds up the parametric model checking of reachability properties by several orders of magnitude, produces closed-form analytical models of modest complexity, and enables the analysis of pDTMCs that leading probabilistic model checkers cannot handle. Also, the use of systems from different application domains, whose induced pDTMCs have distinct characteristics in terms of model structure and complexity, provides evidence supporting the general applicability of fPMC. This conclusion is reinforced by the fact that the fragmentation is completely automated and does not require any domain knowledge or human intervention (cf. Algorithms 1 and 2).

RQ2 (Scalability). We answer this research question by analysing how increasing the number of parameters in a pDTMC (i.e., the number of transition probabilities specified as rational functions), affects the execution time and complexity of formulae computed by fPMC, Storm and PRISM. We used the four PL system variants and varied the number of parameters in these models as a percentage of the total model parameters, setting the remaining parameters to randomly selected constant values that preserve the pDTMC structure. For all pDTMCs, we started from 1% (P01), continuing in increments of 10% (P10), 20% (P20) etc. until all transition probabilities are given by rational functions (P100).

Table II shows the PMC results for the four PL variants. For all PMC approaches, as expected, increasing the number of pDTMC model parameters incurs a corresponding increase in the execution time and formulae complexity. Both Storm and PRISM, however, exhibit exponential growth in these two metrics until P60, and fail to produce any formula from P70 thereafter. In fact, some of the formulae comprise as many as 3.99M and 1.17M arithmetic operations for Storm and PRISM, respectively. In contrast, fPMC terminates successfully in all cases, consuming at most 104s, indicating that fPMC is barely influenced by the increase in model parameters. This insight is also supported by the result that the fPMC-computed closed-form analytical models contain at most 261K operations.

Considering these results, we have strong empirical evidence indicating that the higher the percentage of transition probabilities specified as rational functions over the total number of transition probabilities, the more apparent is the performance gap favouring fPMC over Storm and PRISM. Accordingly, fPMC can support the derivation of closed-form analytical models for reachability properties whose associated pDTMCs specify as much as 100% transition probabilities as rational functions in the evaluated pDTMCs.

RQ3 (Configurability). Columns fPMC and fPMC ($\alpha=\infty$) in Table I compare the fPMC performance with the soft upper

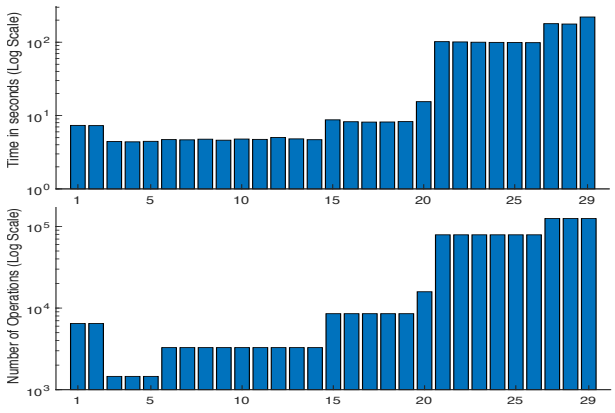


Fig. 8: Impact analysis of the threshold $\alpha \in \{1,..,29\}$ for the FX system from our running example (Section III).

bound for the fragment size enabled and disabled, respectively. Clearly, using α benefits fPMC, reduces its execution time and produces analytical models with a smaller number of operations. Fig. 8 shows in log scale the execution time and number of operations for the closed-form analytical models for threshold $\alpha \in \{1,..,29\}$ for the FX system from our running example. The general pattern indicates that larger α values, contribute to longer execution times and larger formulae, thus negatively influencing the fPMC performance. Since for larger α the model restructuring techniques are invoked less often, the computed formulae are unsurprisingly more complex. However, we observe a 'sweet spot' for $\alpha \in \{3,..,14\}$ signifying that fragments of those sizes produce the most effective pDTMC fragmentation. These findings illustrate that enhancing fPMC with mechanisms enabling the automated selection of α values (e.g., based on the pDTMC structure) can extend further its applicability; this is left for future work.

D. Threats to Validity

We limit **construct validity threats** that may originate from simplifications and assumptions made when establishing the experimental methodology using the pDTMCs and reachability properties from three publicly-available software systems and processes, also used in related research [4], [5], [25]–[27].

We mitigate **internal validity threats** that could introduce bias when identifying cause-effect relationships in our experiments by designing independent research questions and evaluating the effectiveness, scalability and configurability (sensitivity) of fPMC. To further reduce the risk of biased results, we compared fPMC against the latest versions of the leading probabilistic model checkers Storm [15] and PRISM [14] that were available when we conducted the evaluation. We also performed experiments using multiple variants of the studied software systems and process, and evaluated the correctness of all the fPMC-computed formulae by adopting the approach described in Section IV-C. Finally, we enable replication and verification of our findings by making all experimental results publicly available online at https://bit.ly/3l0ygdi.

We limit **external validity threats** that could affect the generalisability of our findings by evaluating fPMC using pDTMCS of systems and processes from three different application domains (i.e., service-based systems [28], [29], software product lines [30], [31], communication processes [32], [33]). Furthermore, we carried out experiments to check that fPMC can work with pDTMCs containing up to 83 parameters, incurring modest increase in execution time with the computed closed-form analytical models containing significantly fewer arithmetic operations than those produced by the model checkers PRISM and Storm. We reduce further the risk that fPMC might be difficult to use in practice by developing an fPMC prototype tool that requires the same domain expertise for specifying pDTMCs as for using PRISM and Storm. However, additional experiments are needed to confirm that fPMC can analyse pDTMCs modelling other software systems and processes than those employed in our evaluation.

VII. Related Work

Introduced by Daws [1] less than two decades ago, parametric model checking has been significantly advanced through the use of rational PMC formulae characteristics such as symmetry and cancellation properties [2], and of polynomial factorisation and strongly connected component decomposition [3]. Both the former advances (implemented by the model checkers PARAM [13] and PRISM [14]) and the latter (implemented by Storm [15]) have greatly improved the scalability of PMC, but are complementary to our fPMC approach, which leverages these advances in its standard PMC step (Fig. 1).

The recent PMC theoretical framework from [4], which fPMC extends and automates, is also complementary to our work. As detailed in §II-B, this approach requires manual assembly of the analysed pDTMC by an expert, which is time consuming, error prone, and only feasible for specific classes of component-based systems. To the best of our knowledge, fPMC is the first approach that can tackle the PMC of parametric Markov models of the complexity and with the number of parameters illustrated in §VI.

VIII. Conclusion

We presented, implemented and evaluated fPMC, an automated model fragmentation technique that enables the fast parametric model checking of reachability properties for systems with complex stochastic behaviour and large numbers of parameters. fPMC complements, leverages and extends the applicability of current PMC techniques and tools, so that closed-form analytical models can be computed for additional classes of systems. Given the usefulness of such analytical models in software engineering and other disciplines, we plan to continue to develop fPMC. Our next steps in this endeavour will focus on extending the types of properties handled by fPMC with unbounded until (§II-A) and reward [24] PCTL properties, on developing additional model restructuring techniques to support the fPMC model fragmentation, and on applying model fragmentation repeatedly so that large fragments and abstract models are further partitioned into smaller models, making PMC applicable to even larger systems.

REFERENCES

[1] C. Daws, "Symbolic and parametric model checking of discrete-time Markov chains," in *First International Conference on Theoretical Aspects of Computing (ICTAC)*, 2005, pp. 280–294.

[2] E. M. Hahn, H. Hermanns, and L. Zhang, "Probabilistic reachability for parametric Markov models," *International Journal on Software Tools for Technology Transfer*, vol. 13, no. 1, pp. 3–19, 2011.

[3] N. Jansen, F. Corzilius, M. Volk, R. Wimmer, E. Ábrahám, J.-P. Katoen, and B. Becker, "Accelerating parametric probabilistic verification," in *11th International Conference on Quantitative Evaluation of Systems (QEST)*, 2014, pp. 404–420.

[4] R. Calinescu, C. A. Paterson, and K. Johnson, "Efficient parametric model checking using domain knowledge," *IEEE Transactions on Software Engineering*, vol. PP, pp. 1–1, 2019.

[5] C. Ghezzi and A. M. Sharifloo, "Model-based verification of quantitative non-functional properties for software product lines," *Information and Software Technology*, vol. 55, no. 3, pp. 508–524, 2013.

[6] ——, "Verifying non-functional properties of software product lines: Towards an efficient approach using parametric model checking," in *15th International Conference on Software Product Lines SPLC*, 2011, pp. 170–174.

[7] G. Su, D. S. Rosenblum, and G. Tamburrelli, "Reliability of run-time quality-of-service evaluation using parametric model checking," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 73–84. [Online]. Available: https://doi.org/10.1145/2884781.2884814

[8] A. Filieri, C. Ghezzi, and G. Tamburrelli, "Run-time efficient probabilistic model checking," in *33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 341–350.

[9] A. Filieri and G. Tamburrelli, "Probabilistic verification at runtime for self-adaptive systems." *Assurances for Self-Adaptive Systems*, vol. 7740, pp. 30–59, 2013.

[10] A. Filieri, G. Tamburrelli, and C. Ghezzi, "Supporting self-adaptation via quantitative verification and sensitivity analysis at run time," *IEEE Transactions on Software Engineering*, vol. 42, no. 1, pp. 75–99, 2016.

[11] R. Calinescu, C. Ghezzi, K. Johnson, M. Pezzé, Y. Rafiq, and G. Tamburrelli, "Formal verification with confidence intervals to establish quality of service properties of software systems," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 107–125, 2016.

[12] R. Calinescu, K. Johnson, and C. Paterson, "FACT: A probabilistic model checker for formal verification with confidence intervals," in *22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2016, pp. 540–546.

[13] E. M. Hahn, H. Hermanns, B. Wachter, and L. Zhang, "PARAM: A model checker for parametric Markov models," in *22nd International Conference on Computer Aided Verification (CAV)*, 2010, pp. 660–664.

[14] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *23rd International Conference on Computer Aided Verification (CAV)*, 2011, pp. 585–591.

[15] C. Dehnert, S. Junges, J.-P. Katoen, and M. Volk, "A storm is coming: A modern probabilistic model checker," in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds. Cham: Springer International Publishing, 2017, pp. 592–600.

[16] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal Aspects of Computing*, vol. 6, no. 5, pp. 512–535, Sep. 1994.

[17] A. Bianco and L. de Alfaro, "Model checking of probabilistic and nondeterministic systems," in *Foundations of Software Technology and Theoretical Computer Science*, ser. LNCS, P. S. Thiagarajan, Ed., vol. 1026. Springer Berlin Heidelberg, 1995, pp. 499–513.

[18] F. Ciesinski and M. Größer, "On probabilistic computation tree logic," in *Validation of Stochastic Systems - A Guide to Current Research*, ser. LNCS, C. Baier *et al.*, Eds., vol. 2925. Springer, 2004, pp. 147–188.

[19] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[20] J. Katoen, "Advances in probabilistic model checking," in *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, ser. Lecture Notes in Computer Science, G. Barthe and M. V. Hermenegildo, Eds., vol. 5944. Springer, 2010, p. 25. [Online]. Available: https://doi.org/10.1007/978-3-642-11319-2_5

[21] M. Kwiatkowska, G. Norman, and D. Parker, "Stochastic model checking," in *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, ser. LNCS (Tutorial Volume), M. Bernardo and J. Hillston, Eds., vol. 4486. Springer, 2007, pp. 220–270.

[22] ——, *Probabilistic Model Checking: Advances and Applications*. Cham: Springer International Publishing, 2018, pp. 73–121. [Online]. Available: https://doi.org/10.1007/978-3-319-57685-5_3

[23] ——, "Advances and challenges of probabilistic model checking," in *Proc. 48th Annual Allerton Conference on Communication, Control and Computing*. IEEE Press, 2010, pp. 1691–1698.

[24] S. Andova, H. Hermanns, and J.-P. Katoen, "Discrete-time rewards model-checked," in *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer, 2003, pp. 88–104.

[25] S. Gerasimou, G. Tamburrelli, and R. Calinescu, "Search-based synthesis of probabilistic models for quality-of-service software engineering," in *30th Intl. Conf. on Automated Software Engineering (ASE'15)*, 2015, pp. 319–330.

[26] M. Hajnal, M. Nouvian, D. Šafránek, and T. Petrov, "Data-informed parameter synthesis for population markov chains," in *International Workshop on Hybrid Systems Biology*. Springer, 2019, pp. 147–164.

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

[28] D. Ameller, M. Galster, P. Avgeriou, and X. Franch, "A survey on quality attributes in service-based systems," *Software quality journal*, vol. 24, no. 2, pp. 271–299, 2016.

[29] C. Sun, R. Rossing, M. Sinnema, P. Bulanov, and M. Aiello, "Modeling and managing the variability of web service-based systems," *Journal of Systems and Software*, vol. 83, no. 3, pp. 502–516, 2010.

[30] P. Clements and L. Northrop, *Software product lines*. Addison-Wesley Boston, 2002.

[31] S. Apel, D. Batory, C. Kästner, and G. Saake, "Software product lines," in *Feature-Oriented Software Product Lines*. Springer, 2013, pp. 3–15.

[32] F. Dressler and O. B. Akan, "A survey on bio-inspired networking," *Computer Networks*, vol. 54, no. 6, pp. 881–900, 2010.

[33] ——, "Bio-inspired networking: from theory to practice," *IEEE Communications Magazine*, vol. 48, no. 11, pp. 176–183, 2010.

SUPPLEMENTARY MATERIAL TO FAST PARAMETRIC MODEL CHECKING THROUGH MODEL FRAGMENTATION

Theorem 2. *Function* FRAGMENTATION *returns a valid fragmentation of the graph $\mathcal{G}(V, E)$ as restructured by* TRAVERSE.

Proof. We prove this result by showing that: (a) the set FS assembled by FRAGMENTATION comprises only valid fragments; (b) the fragments from FS are disjoint (i.e., no vertex from V is included in more than one fragment); and (c) the function terminates and returns a set of fragments FS that includes all the vertices from V.

To prove part (a), we note that new tuples are added to FS in three lines from Algorithms 1 and 2. In line 2 of Algorithm 1 and line 7 of Algorithm 2, FS is augmented with tuples that represent degenerate, single-state fragments according to the extended definition of a fragment from the first paragraph of Section IV-A. Finally, in line 23 of Algorithm 1, FS is augmented with a tuple that either passes the fragment-validity check from line 20, or is reduced to a degenerate, single-state fragment in line 21 before it is included in FS. As such, any tuple inserted into FS is a valid fragment.

To prove part (b), we note that the vertices from the reachability set R provided as an argument to FRAGMENTATION are each placed into a degenerate, single-state fragment in line 2 of Algorithm 1, and that the vertex set Z of any fragment $(Z, z_0, Z_{\mathsf{OUT}})$ added to FS comes from $V \setminus R$, where R is updated (in line 24 of Algorithm 1, and in line 8 of Algorithm 2) to include all the vertices of new fragments included in FS. To see that the vertices of all new fragments come from $V \setminus R$, observe that vertex z_0 added to Z in line 4 of the algorithm comes directly from $V \setminus R$; and vertex w added to Z in line 18 (or included into FS as the only vertex of a degenerate fragment in line 7 of Algorithm 2) comes from the stack T, which can only acquire vertices from $V \setminus R$ (as enforced by the if statements before lines 5 and 16 from Algorithm 2). Therefore, the fragments from FS are disjoint.

To prove part (c), we note that both algorithms terminate. TRAVERSE terminates because each of its statements (including the assembly of the vertex sets I and O, and its for loop) operate with a finite number of vertices. FRAGMENTATION terminates because: (i) each iteration of its for loop adds at least one vertex (i.e., z_0) to R in line 24 until $V \setminus R = \{\}$ in line 3 (since V is a finite set) and the loop terminates with all vertices from V included in fragments from FS; (ii) its while loop terminates since it iterates over the elements of stack T that can only contain one instance of vertices from the finite set V and is therefore finite; and (iii) RESTRUCTURE invocations can only add a finite number of vertices to V, as shown in Sect. IV-B. Thus, FRAGMENTATION terminates, returning a fragment set FS that includes all the vertices from V. □

Theorem 3. *The function* FRAGMENTATION *requires at most $\mathsf{O}(n^2d^4)$ steps, where n is the number of vertices in V and $d = \max_{v \in V}\{indegree(v), outdegree(v)\}$.*

Proof. The function RESTRUCTURING requires at most $\mathsf{O}(d^2)$ steps to process up to $outdegree(z) - 1$ outgoing edges of a vertex z (Fig. 6a), or up to $(indegree(z) - 1)outdegree(z)$ pairs of incoming-outgoing edges of z (Fig. 6b). Additionally, the model restructuring technique from Fig. 6a may add up to $outdegree(z) - 1$ new vertices for each vertex z in V, yielding a restructured graph with nd vertices in the worst-case scenario.

The function TRAVERSE requires at most $\mathsf{O}(d^3)$ steps, as assembling the set I and processing it in lines 2–11 requires the inspection of the $indegree(w)$ incoming edges of w, assembling the set O requires $outdegree(w)$ steps, and processing it involves up to $outdegree(w)$ executions of the $\mathsf{O}(d^2)$-step RESTRUCTURING.

In the worst-case scenario where the fragmentation produces only single-vertex fragments, the execution of FRAGMENTATION requires the execution of its for loop from lines 3–25 for each of the n vertices of the original graph (with any additional vertices created by restructuring included into the same fragment as the vertex that led to their creation, cf. Fig. 6a). Each iteration of this loop executes: (i) TRAVERSE in line 6 in $\mathsf{O}(d^3)$ steps; (ii) the while loop from lines 7–19, which has at most nd iterations (one for each vertex from the restructured graph) that may each invoke the $\mathsf{O}(d^3)$-step TRAVERSE or the $\mathsf{O}(d^2)$-step RESTRUCTURING; and (iii) the fragment validity check from line 20 which requires no more than $\mathsf{O}(nd)$ operations. Thus, each iteration of the for loop from lines 3–25 is completed in no more than $\mathsf{O}(nd^4)$ steps (due to the while loop from lines 7–19), and the entire FRAGMENTATION requires $\mathsf{O}(n^2d^4)$ steps in the worst-case scenario. □

Note that the coefficients associated with n and d from the big-O notation in Theorem 3 are typically well below 1. For instance, in our experiments the for loop from FRAGMENTATION was only executed for a small fraction of the vertices in V (since many fragments with multiple vertices are typically produced), and RESTRUCTURING was only executed sparingly. Furthermore, it is worth noting that pDTMCs typically sparsely connected graphs, and therefore d is relatively small.