
Article:
Fang, Xinwei orcid.org/0000-0003-3630-2249, Calinescu, Radu orcid.org/0000-0002-2678-9260, Gerasimou, Simos orcid.org/0000-0002-2706-5272 et al. (1 more author) (2023) Fast Parametric Model Checking with Applications to Software Performability Analysis. IEEE Transactions on Software Engineering. pp. 4707-4730. ISSN 0098-5589

https://doi.org/10.1109/TSE.2023.3313645

Fast Parametric Model Checking with Applications to Software Performability Analysis

Xinwei Fang*, Radu Calinescu*, Simos Gerasimou, and Faisal Alhwikem

Abstract—We present an efficient parametric model checking technique for the analysis of software *performability*, i.e., of the performance and dependability properties of software systems. The new parametric model checking (pMC) technique works by using a heuristic to automatically decompose a parametric discrete-time Markov chain (pDTMC) model of the software system under verification into fragments that can be analysed independently, yielding results that are then combined to establish the required software performability properties. Our fast parametric model checking (fPMC) technique enables the formal analysis of software systems modelled by pDTMCs that are too complex to be handled by existing pMC methods. Furthermore, for many pDTMCs that state-of-the-art parametric model checkers can analyse, fPMC produces solutions (i.e., algebraic formulae) that are simpler and much faster to evaluate. We show experimentally that adding fPMC to the existing repertoire of pMC methods improves the efficiency of parametric model checking significantly, and extends its applicability to software systems with more complex behaviour than currently possible.

Index Terms—parametric model checking; software performability; nonfunctional software properties; Markov models

1 Introduction

Most software operates in environments characterised by workloads, usage profiles, failures and available resources that are stochastic in nature [17], [23], [43]. As such, probabilistic models such as Markov chains [11], [30], [34], [50], queueing networks [8], [31] and stochastic Petri nets [9], [47], [51] have long been used to analyse the *performability* (i.e., the performance, dependability and other nonfunctional properties) of software.

In this paper, we focus on the analysis of software performability using *parametric discrete-time Markov chains* (pDTMCs), i.e., Markov chains that have transition probabilities and/or that are augmented with rewards specified as rational functions over the parameters of the analysed system. The technique used to analyse these stochastic models is called *parametric model checking* (pMC) [18], [22], [39], [44]. Given a pDTMC model of a software system, and a set of nonfunctional system properties specified in probabilistic temporal logic, pMC computes algebraic formulae for these properties. The concept is straightforward. As a simple example, consider a web server (Figure 1) that handles two types of requests, and suppose that requests belong to these types with probabilities p and $1-p$. If the mean times to handle the two types of request are t_1 and t_2, respectively, then the expected (i.e., mean) time for handling a request is computed by pMC as $pt_1 + (1-p)t_2$.

The algebraic formulae produced by pMC have many important applications in software engineering. They can be used to analyse the sensitivity of nonfunctional software properties to parametric variability [30], to identify optimal system configurations for software product lines [35], [36], [46], and to establish confidence intervals for the analysed nonfunctional properties [2], [14], [15]. Furthermore, pMC formulae precomputed prior to deployment (when some of the system parameters are unknown) can be evaluated at runtime (when the parameter values are determined through monitoring), to verify if the nonfunctional requirements of a system are still satisfied after environmental changes [49], [55]. Last but not least, self-adaptive systems can use these formulae to efficiently select new configurations when requirements are violated after such changes [28], [54].

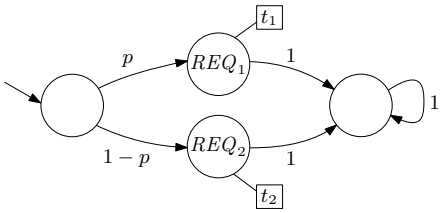


Fig. 1: pDTMC model of a web server that handles requests of type REQ_1 (received with probability p) and REQ_2 (received with probability $1-p$) with expected times t_1 and t_2, respectively.

Despite these benefits, pMC is seldom used in practice due to its limited scalability. While theoretical advances over the past decade [39], [44] and their implementation in state-of-the-art model checkers [24], [38], [45] have alleviated this limitation, existing pMC approaches are often unable to analyse pDTMCs with large numbers of parameters.

Our paper presents a fast parametric model checking (fPMC) technique that extends the applicability of pMC to software systems with considerably more complex behaviour and with much larger sets of parameters than cur-

* *The first two authors contributed equally to the article.*
- *X. Fang, R. Calinescu and S. Gerasimou are with the Department of Computer Science at the University of York, UK*
- *F. Alhwikem is with the Computer Science Department, College of Computer, Qassim University, Saudi Arabia*

rently possible. fPMC is a compositional analysis technique that uses well-defined rules (described later in the paper) to partition, in a heuristic manner, the graph induced by the pDTMC under analysis[1] into subgraphs called *fragments*. The fPMC fragments define small pDTMCs that are analysed individually to generate pMC subexpressions. Finally, the overall pMC result is obtained by combining these subexpressions with an expression produced by analysing an *abstract* model created by replacing each fragment from the original pDTMC with a single state.

fPMC fragments are not strongly connected components (SCCs) of the analysed pDTMC. They can typically be assembled to ensure that each fragment is small enough for its individual pMC analysis to be feasible, and large enough to avoid the creation of so many fragments that the abstract pDTMC becomes difficult to analyse. This flexibility yields both fragments that include only a part of an SCC and fragments that include multiple SCCs, and explains why fPMC can efficiently analyse many pDTMCs not handled by the SCC-partitioning pMC approach from [44], as shown in our experimental evaluation from Section 6.

fPMC builds on recent research that laid the groundwork for the use of pDTMC fragments to speed up parametric model checking [18]. However, that research provides no algorithm for the partition of pDTMCs into fragments. The main contributions of our paper are:

1) The fPMC theoretical foundation comprising algorithms **(a)** for pDTMC fragmentation, and **(b)** for pDTMC *restructuring*, to aid the formation of suitably sized fragments.
2) A new parametric model checking tool that **(a)** employs a simple heuristic to determine whether the analysis of a pDTMC requires fragmentation, and **(b)** performs the analysis of the pDTMC by using our fPMC fragmentation and restructuring algorithms if fragmentation is required, or by invoking the model checker Storm [24] otherwise.
3) An extensive evaluation of the fPMC theoretical foundation and tool for 62 variants of three pDTMC models and a wide range of performability software properties taken from the research literature.

A preliminary fPMC version that only supports the analysis of reachability probabilistic temporal logic formulae over pDTMCs was introduced in [27]. This paper extends the theoretical foundation from [27] with:

1) Support for the pMC of unbounded until formulae, which correspond to the analysis of software properties such as the probability of successful termination without intermediate errors or timeouts. To ascertain the usefulness of this new contribution, we analysed the frequency with which unbounded until formulae are used in leading-venue software engineering research papers from the repository assembled in [2], which includes all research papers that use discrete-time Markov chains and were published in CORE2020[2]
2) Support for the pMC of reachability reward formulae—a significant improvement because using pMC to analyse nonfunctional properties related to the performance, cost, utility and resource usage of software systems requires the specification of these properties as reward formulae over pDTMCs. Such properties of software systems are analysed using reward formulae in 50% of the research papers from the publication repository from [2].
3) Formal correctness proofs for the fPMC fragmentation algorithm and pDTMC restructuring methods, in contrast to the experimental evaluation from our previous work [27].
4) A formal complexity analysis of the end-to-end fPMC fragmentation technique, which aids in understanding the scalability of fPMC.

The three types of temporal logic formulae currently supported by fPMC allow the analysis of a wide range of software properties, including many properties that could not be handled by our preliminary fPMC version from [27]. For instance, out of all software properties analysed by the research papers from the repository in [2], 41% are reachability properties, 18% are unbounded until properties, and 23% are reachability reward properties—giving an 82% fPMC coverage. Thus, by adding support for unbounded until properties and reachability reward properties, we doubled the coverage of our algorithm. Additionally, we considerably extended and improved the validation of fPMC by evaluating it for a much broader range of models and properties (Section 6). Finally, we augmented the fPMC tool support with the heuristic for determining if the pDTMC under analysis requires fragmentation (Section 6.2.1).

The remainder of the paper is structured as follows. Section 2 provides formal definitions and explanations of the techniques used in this work. Section 3 describes a software system we use to motivate the need for fPMC and to illustrate its application. The fPMC algorithms and their proofs are presented in Section 4, followed by the implementation details in Sections 5. We then evaluate fPMC in Section 6, and discuss threats to validity in Section 7. Finally, Section 8 compares fPMC to related work, and Section 9 provides a brief summary and discusses directions for future work.

2 Preliminaries

Parametric model checking [18], [22], [39], [44] is a mathematically based technique for the analysis of pDTMC properties expressed in *probabilistic computation tree logic* (PCTL) [10], [19], [42] extended with *rewards* [4]. This section provides formal definitions for each of these concepts.

2.1 Discrete-time Markov Chains

Discrete-time Markov chains (DTMCs) are finite state-transition models used to analyse the stochastic behaviour

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

2. https://www.core.edu.au/conference-portal rank A* software engineering journals and conferences between 2016–2020. This analysis (whose detailed results are provided on our project's website [1]) found that 37% of these research papers use unbounded until formulae to examine properties of software systems.

of real-world systems. They comprise states that correspond to relevant configurations of the system under analysis, and transitions that model the changes that can occur between these configurations.

Definition 1. *A (non-parametric) discrete-time Markov chain is a tuple*

$$D = (S, s_0, \mathbf{P}, L), \tag{1}$$

where: (i) S is a finite set of states; (ii) $s_0 \in S$ is the initial state; (iii) $\mathbf{P} : S \times S \to [0, 1]$ is a transition probability matrix such that, for any states $s, s' \in S$, $\mathbf{P}(s, s')$ represents the probability that the Markov chain transitions from s to s', and, for any $s \in S$, $\sum_{s' \in S} \mathbf{P}(s, s') = 1$; and (iv) $L : S \to 2^{AP}$ 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.

Given a discrete-time Markov chain (1), a state $s \in S$ is an *absorbing state* if $\mathbf{P}(s, s) = 1$ and $\mathbf{P}(s, s') = 0$ for all $s \neq s'$, and a *transient state* otherwise. A *path* π over a DTMC D is a (possibly infinite) sequence of states from S, such that for any consecutive states s and s' from π, $\mathbf{P}(s, s') > 0$. The i-th state on the path π, $i \geq 1$, is denoted $\pi(i)$. For any state s, $Paths^D(s)$ represents the set of all infinite paths over D that start with state s.

To enlarge the spectrum of nonfunctional properties that can be analysed using DTMCs, these models are often augmented with *reward functions*.

Definition 2. *A reward function over a DTMC* (1) *is a function*

$$rwd : S \to \mathbb{R}_{\geq 0} \tag{2}$$

that associates a non-negative quantity (i.e., a reward*) with each Markov chain state.*

Finally, parametric DTMCs are used when a reward-augmented Markov chain contains probabilities or rewards that are unknown or that correspond to adjustable parameters of the system under analysis. These probabilities and rewards are specified as rational functions over a set of continuous variables that correspond to parameters of the modelled system and its environment.

Definition 3. *A rational function $q_V : \mathbb{R}^N \to \mathbb{R}$ over a finite set of $N \geq 1$ real-valued continuous variables $V = \{v_1, v_2, \ldots, v_N\}$ is a function that can be defined as*

$$q_V(v_1, v_2, \ldots, v_N) = \frac{P_V(v_1, v_2, \ldots, v_N)}{R_V(v_1, v_2, \ldots, v_N)}, \tag{3}$$

where P_V and R_V are polynomials over the variables from V.

Definition 4. *A reward-augmented parametric discrete-time Markov chain (pDTMC) over a set of continuous variables V is a Markov chain* (1) *with transition probability matrix $\mathbf{P} : S \times S \to Q_V$ and reward functions $rwd : S \to Q_V$, where Q_V denotes the set of all rational functions over V.*[3]

3. We note that, by requiring that a parametric discrete-time Markov chain is a Markov chain (1), Definition 4 implicitly requires that, for any allowed parameter valuation, each element of the transition probability matrix $\mathbf{P}$ is a valid probability, and the sum of the outgoing probabilities in each pDTMC state sums up to 1.

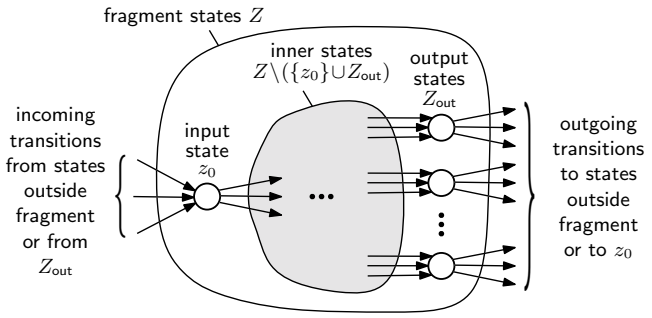


Fig. 2: pDTMC fragment $F = (Z, z_0, Z_{\text{out}})$

2.2 Probabilistic Computation Tree Logic

The properties of (non-parametric and parametric) discrete-event Markov chains are formally specified in reward-extended PCTL.

Definition 5. *A PCTL* state formula Φ, path formula Ψ, *and* reward state formula Φ_{R} *over an atomic proposition set AP are defined by the grammar:*

$$\begin{aligned} \Phi &::= true \mid a \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathcal{P}_{=?}[\Psi] \\ \Psi &::= \mathrm{X}\Phi \mid \Phi \ \mathrm{U} \ \Phi \mid \Phi \ \mathrm{U}^{\leq k} \ \Phi \\ \Phi^{:}_{\mathsf{R}} &:= \mathcal{R}^{rwd}_{=?}[\mathrm{I}^{=k}] \mid \mathcal{R}^{rwd}_{=?}[\mathrm{C}^{\leq k}] \mid \mathcal{R}^{rwd}_{=?}[\mathrm{F}\ \Phi] \mid \mathcal{R}^{rwd}_{=?}[\mathrm{S}] \end{aligned} \tag{4}$$

where $a \in AP$ is an atomic proposition, $k \in \mathbb{N}_{>0}$ is a timestep bound, and rwd is a reward structure (2).

The PCTL semantics is defined using a satisfaction relation $\models$ over the states $s \in S$ and paths $\pi \in Paths^D(s)$ of a Markov chain (1). Thus, $s \models \Phi$ means "Φ holds in state s", $\pi \models \Psi$ means "Ψ holds for path π", and we have: $s \models true$ for all states $s \in S$; $s \models a$ 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 *next formula* $X\Phi$ holds for a path π if $\pi(2) \models \Phi$. The *time-bounded until formula* $\Phi_1 \ \mathrm{U}^{\leq k} \ \Phi_2$ holds for a path π iff $\pi(i) \models \Phi_2$ for some $i \leq k$ and $\pi(j) \models \Phi_1$ for all $j = 1, 2, \ldots, i-1$; and the *unbounded until formula* $\Phi_1 \ \mathrm{U} \ \Phi_2$ removes the bound k from the time-bounded until formula.

The quantitative state formula $\mathcal{P}_{=?}[\Psi]$ specifies the probability that paths from $Paths^D(s)$ satisfy the path property Ψ. *Reachability properties* $\mathcal{P}_{=?}[\mathrm{true} \ \mathrm{U} \ \Phi]$ are equivalently written as $\mathcal{P}_{=?}[\mathrm{F}\ \Phi]$ or $\mathcal{P}_{=?}[\mathrm{F}\ R]$, where $R \subseteq S$ is the set of states in which Φ holds.

Finally, the reward formulae specify the expected values for: the *instantaneous reward* at timestep k ($\mathcal{R}^{rwd}_{=?}[\mathrm{I}^{=k}]$); the *cumulative reward* up to timestep k ($\mathcal{R}^{rwd}_{=?}[\mathrm{C}^{\leq k}]$); the *reachability reward* cumulated until reaching a state that satisfies a property Φ ($\mathcal{R}^{rwd}_{=?}[\mathrm{F}\ \Phi]$, or $\mathcal{R}^{rwd}_{=?}[\mathrm{F}\ R]$ if $R \subseteq S$ is the set of states in which Φ holds); and the *steady-state reward* in the long run ($\mathcal{R}^{rwd}_{=?}[\mathrm{S}]$). A complete description of the PCTL semantics is available in [4], [10], [42].

2.3 Parametric Model Checking through Fragmentation

Our fPMC technique builds on recently introduced theoretical results on the use of pDTMC *fragmentation* to speed up parametric model checking [18]. These results, which explain how pDTMC fragments can be exploited—but not how they could be obtained—are summarised below. We start by introducing the concept of a pDTMC fragment.

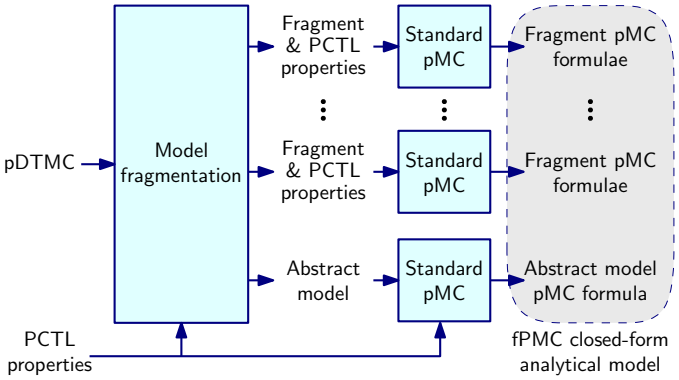


Fig. 3: Parametric model checking through fragmentation

Definition 6. *A fragment of a pDTMC* $(S, s_0, \boldsymbol{P}, L)$ *is a tuple*

$$F = (Z, z_0, Z_{\text{out}}), \tag{5}$$

where (Figure 2):

- $Z \subseteq S$ *is a subset of transient pDTMC states or* $Z = \{z_0\}$*;*
- z_0 *is the (only)* input state *of* F*, i.e.,* $\{z_0\} = \{z \in Z \mid \exists s \in S \setminus Z \,.\, \mathbf{P}(s, z) > 0\}$*;*
- $Z_{\text{out}} = \{z \in Z \mid (\exists s \in (S \setminus Z) \cup \{z_0\} \,.\, \mathbf{P}(z, s) > 0) \wedge (\forall z' \in Z \setminus \{z_0\} \,.\, \mathbf{P}(z, z') = 0)\}$ *is the non-empty set of* output states *of* F*.*

This definition is much less restrictive than that of a strongly connected component. In particular, any pDTMC state z (including any absorbing state) forms a one-state, *degenerate fragment* $F = (\{z\}, z, \{z\})$. Furthermore, the "inner" states $Z \setminus (\{z_0\} \cup Z_{\text{out}})$ of a fragment can include one or several SCCs. Finally, an SCC can be split into multiple fragments, because paths that start from the output states of a fragment and reach its input state (either directly or through intermediate states outside the fragment) are permitted.

Given a fragment $F = (Z, z_0, Z_{\text{out}})$ of a pDTMC D augmented with a reward function rwd, the pMC of reachability, unbounded until, and reachability reward properties of D can be carried out compositionally by using the following four-step process introduced in [18] and illustrated in Figure 3:

1) Use standard pMC to obtain algebraic formulae for:
   i) the probabilities $prob_z$ of reaching each of the output states $z \in Z_{\text{out}}$ of F from the input fragment state z_0;
   ii) the cumulative reward rwd_{out} associated with reaching the output state set Z_{out} from z_0.
2) Assemble an *abstract pDTMC model* $D' = (S', s'_0, P')$ augmented with a reward function rwd', where:
   i) $S' = (S \setminus Z) \cup \{z'\}$, i.e., the states from Z are replaced with a single, abstract state z';
   ii) $s'_0 = s_0$ if $z_0 \neq s_0$, and $s'_0 = z'$ otherwise;
   iii) the incoming transitions and transition probabilities of z' are inherited from z_0;
   iv) z' has outgoing transitions to each 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—details about the calculation of these probabilities are provided in [18];
   v) the new reward function is given by $rwd'(s) = rwd(s)$ for all $s \in S' \setminus \{z'\}$, and $rwd'(z') = rwd_{\text{out}}$.
3) Compute the pMC formula for the original property under analysis, for the abstract model from step 2.
4) Combine the pMC formulae from step 1 and the pMC formula from step 3 into a system of expressions.

The system of expressions 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, and we will refer to it as fPMC or fPMC-computed algebraic formulae in the rest of the paper.

The pMC carried out in steps 1 and 3 uses models that are simpler and smaller than the original model D. As such, this four-step approach is often faster, produces much smaller algebraic formulae, and enables the analysis of models that are larger and more complex than those supported by previous pMC methods.

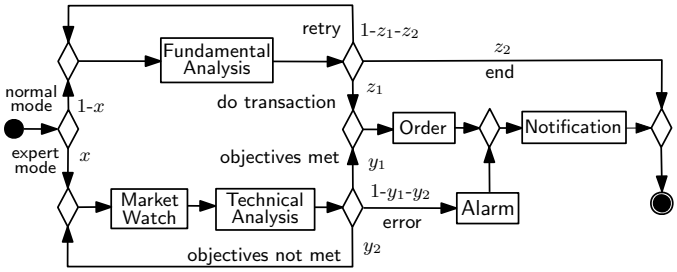


Fig. 4: FX service-based system, where x, $y1$, $y2$, $z1$, $z2$ are the (unknown) probabilities of different execution paths, i.e., the *operational profile* of the system

3 Motivating Example

In this section, we introduce a software system that will be used to illustrate the use of our fPMC approach throughout the paper. Taken from [33], [34], this is a six-operation service-based system performing trading in the foreign exchange (FX) market. The workflow of the FX system is shown in Fig. 4 and described briefly below.

FX workflow. The FX system has two execution modes that a trader can choose from: an *expert* mode and a *normal* mode.

In the *expert* mode, the *Market watch* operation extracts real-time exchange rates (i.e., bid/ask prices) of selected currency pairs, and this information is then passed to the *Technical analysis* operation for further analysis, such as evaluating the current trading conditions, predicting future price movement, and deciding which actions to take. Three actions can be taken: carrying out a trade by calling the *Order* operation; performing the *Market watch* operation again, e.g., on different or additional currency pairs; and reporting an error that triggers an *Alarm* operation. The *Order* and *Alarm* operations are each followed by a user *Notification* operation, and the end of the workflow.

In the *normal* mode, the system uses a *Fundamental analysis* operation to evaluate the economic outlook of a country

```
1   dtmc
2
3   // Operational profile parameters
4   const double x;const double y1;const double y2;const double z1;const double z2;
5   // Operation implementation parameters
6   const double p11; const double r11; const double p12; const double r12;
7   const double p21; const double r21; const double p22; const double r22;
8   const double p31; const double r31; const double p32; const double r32;
9   const double p41; const double r41; const double p42; const double r42;
10  const double p51; const double r51; const double p52; const double r52;
11  const double p61; const double r61; const double p62; const double r62;
12  // Reward parameters
13  const double t11; const double t12;
14  const double t21; const double t22;
15  const double t31; const double t32;
16  const double t41; const double t42;
17  const double t51; const double t52;
18  const double t61; const double t62;
19
20  module WorkflowFX
21    // FX states
22    s : [0..11] init 0;
23    // Retry status of each service implementation
24    rtry : [1..2] init 1;
25    // Employed service for operation op1...op6
26    op1 : [1..2] init 1; op2 : [1..2] init 1; op3 : [1..2] init 1;
27    op4 : [1..2] init 1; op5 : [1..2] init 1; op6 : [1..2] init 1;
28
29    // Start: expert mode or normal mode
30    [fxStart] s=0 → x:(s'=1) + (1-x):(s'=4);
31
32    // Operation #1 : Sequential execution strategy with retry for the Market Watch
33    [op11] (s=1)&(rtry=1)&(op1=1) → p11:(s'=2) + (1-p11):(rtry'=2);
34    [op11r] (s=1)&(rtry=2)&(op1=1) → r11:(rtry'=1) + (1-r11):(op1'=2)&(rtry'=1);
35    [op12] (s=1)&(rtry=1)&(op1=2) → p12:(s'=2) + (1-p12):(rtry'=2);
36    [op12r] (s=1)&(rtry=2)&(op1=2) → r12:(rtry'=1) + (1-r12):(s'=9);
      . . .
46    // Operation #3 : Sequential execution strategy with retry for the Alarm
47    [op31] (s=3)&(rtry=1)&(op3=1) → p31:(s'=6) + (1-p31):(rtry'=2);
48    [op31r] (s=3)&(rtry=2)&(op3=1) → r31:(rtry'=1) + (1-r31):(op3'=2)&(rtry'=1);
49    [op32] (s=3)&(rtry=1)&(op3=2) → p32:(s'=6) + (1-p32):(rtry'=2);
50    [op32r] (s=3)&(rtry=2)&(op3=2) → r32:(rtry'=1) + (1-r32):(s'=9);
      . . .
76    // Technical analysis result
77    [TAResult] s=7 → y1:(s'=5) + y2:(s'=1) + (1-y1-y2):(s'=3);
78    // Fundamental analysis result
79    [FAResult] s=8 → z1:(s'=5) + z2:(s'=9) + (1-z1-z2):(s'=4);
80    // FX final states (workflow completed successfully or failed)
81    [failedFX] s=9 → 1:(s'=9);
82    [successFX] s=10 → 1:(s'=10);
83  endmodule
84
85  // Time taken by services as state reward
86  rewards "time"
87    (s=1)&(op1=1)&(rtry=1) : t11;
88    (s=1)&(op1=2)&(rtry=1) : t12;
89    (s=2)&(op2=1)&(rtry=1) : t21;
90    (s=2)&(op2=2)&(rtry=1) : t22;
91    (s=3)&(op3=1)&(rtry=1) : t31;
92    (s=3)&(op3=2)&(rtry=1) : t32;
93    (s=4)&(op4=1)&(rtry=1) : t41;
94    (s=4)&(op4=2)&(rtry=1) : t42;
95    (s=5)&(op5=1)&(rtry=1) : t51;
96    (s=5)&(op5=2)&(rtry=1) : t52;
97    (s=6)&(op6=1)&(rtry=1) : t61;
98    (s=6)&(op6=2)&(rtry=1) : t62;
99  endrewards
```

Fig. 5: pDTMC model of the FX system

and decides whether to: call the *Order* operation to trade the currency of that country; re-do the *Fundamental analysis* operation; or end the execution of the workflow.

Given its business-critical nature, the underlying software architecture of the FX system needs to be highly reliable. To avoid single points of failure, each FX operation is implemented by two functionally-equivalent services, and each service is invoked in order using a *sequential execution strategy with retry* (SEQ_R). For the i-th operation, if the first service fails, it is re-invoked with probability r_{i1} or, with probability $1 - r_{i1}$, the second service is invoked. If the second service also fails, it is re-invoked with probability r_{i2}, or the operation is abandoned with probability $1 - r_{i2}$, leading to the failure of the entire workflow execution.

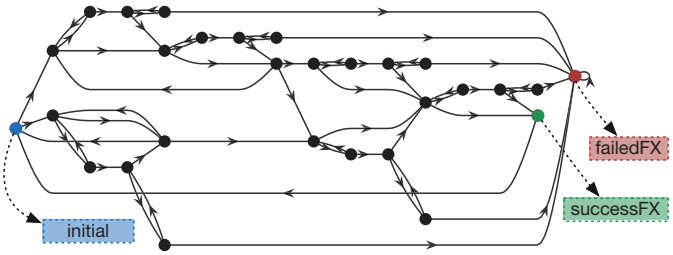


Fig. 6: Graph representation of the FX pDTMC; the model comprises 29 states and 58 transitions

FX pDTMC. Fig. 5 shows the pDTMC model of the FX workflow, specified in the modelling language of PRISM probabilistic model checker [45]. Lines 3–18 define the model parameters associated with **(a)** the FX operational profile (line 4), with **(b)** the implementations of the FX operations (lines 6–11), and with **(c)** the mean execution times of each service used by these implementations (lines 13–18). The parameters p_{ij}, r_{ij} and t_{ij} represent the probability of successful execution, the probability of retrying and the mean execution time, respectively, for the i-th operation using the j-th service implementation, where $i \in \{1, 2, \ldots, 6\}$ and $j \in \{1, 2\}$. The use of parameters to model the system aspects from lines 3–18 is justified because:

1) the operational profile of a system is often unknown when its model is developed;
2) modelling the system configuration by means of a set of parameters allows the exploration and better selection of suitable system configurations;
3) the execution times of individual services are not available until the system is deployed and executed (and may change over time).

Inside the *WorkflowFX* module, we use the local variable *state* (line 22) to model the operations in the FX system, and use *retry* (line 24) and op_i (line 26–27) to encode the retry status of a service implementation and the employed service implementation for each operation, respectively. The selection of the *expert* or *normal* mode is decided in line 30, and the execution of the FX operations is modelled in lines 33–75. Due to space constraints, we only show the modelling of the *Market Watch* operation (line 33–36) and the *Alarm* operation (lines 46–50); all the other FX operations are modelled similarly. For both operations, the invocation of the first service succeeds with probability p_{i1} and FX moves to the next operation, fails with probability $1 - p_{i1}$ and is retried with probability r_{i1} (lines 33 and 34, and lines 47 and 48, respectively); otherwise, the second service is executed and succeeds, or is re-invoked with probabilities p_{i2} and r_{i2} in lines 35 and 36, and lines 49 and 50, respectively. If both service implementations fail, the FX workflow execution terminates with a failure in line 81.

Fig. 6 shows the directed graph induced by the FX pDTMC, with the initial and final states of the FX workflow highlighted in colour. For this pDTMC model, we assume that we are interested in analysing the three non-functional properties from Table 1. We note that despite the relatively small number of states and transitions from the FX pDTMC model, the leading parametric model checkers PRISM [45]

TABLE 1: Non-functional properties for the FX system from the motivating example

ID	Property type	Informal description	Property specified in PCTL
P1	Reachability	probability that FX workflow completes successfully	$\mathcal{P}_{=?}[\mathrm{F}\ \mathsf{successFX}]$
P2	Reachability reward	expected workflow execution time	$\mathcal{R}^{time}_{=?}[\mathrm{F}\ \mathsf{failedFX} \vee \mathsf{successFX}]$
P3	Unbounded until	probability that FX workflow completes successfully without triggering an alarm	$\mathcal{P}_{=?}[\ !\,\mathsf{Alarm}\ \mathrm{U}\ \mathsf{successFX}]$

and Storm [24] could not return a closed-form pMC formula for any of these properties within an hour when run on the MacBook Pro computer we used in all our experiments (please see Section 6.1 for a detailed specification of this computer). In the next section, we explain how fPMC can successfully analyse those three properties via automated model fragmentation.

4 fPMC Theoretical Foundation

We present the model fragmentation algorithm that underpins the fPMC analysis of reachability, unbounded until and reachability reward pDTMC properties in Section 4.1. A pDTMC model restructuring algorithm that aids the formation of fragments is discussed in Section 4.2. Section 4.3 provides formal proofs for the correctness the fPMC algorithms. Finally, Section 4.4 illustrates the application of fPMC to the pDTMC model and PCTL properties from our motivating example from Section 3.

4.1 Markov Chain Fragmentation Algorithm

fPMC partitions a pDTMC into fragments that can be analysed individually by current parametric model checkers. This partition is carried out by the function FRAGMENTATION from Algorithm 1, supported by the auxiliary functions from Algorithms 2, 3 and 4. The function FRAGMENTATION takes four arguments:

1) the analysed pDTMC $D(S, s_0, \mathbf{P}, L)$;
2) the analysed PCTL formula ϕ, which can be a reachability property $\mathcal{P}_{=?}[\mathrm{F}\,\Phi]$, an unbounded until property $\mathcal{P}_{=?}[\Phi_1\,\mathrm{U}\,\Phi_2]$, or a reachability reward property $\mathcal{R}^{rwd}_{=?}[\mathrm{F}\,\Phi]$, where the inner state formulae Φ, Φ_1 and Φ_2 cannot contain the probabilistic operator $\mathcal{P}$;[4]
3) a reward function rwd over D, which is only relevant if ϕ is a reachability reward property (we assume that $rwd(s) = 0$ for all $s \in S$ otherwise);
4) a *fragmentation threshold* $\alpha \in \mathbb{N}_{>0}$, whose role is described later in this section.

Given these arguments, the function returns (line 29):

1) a restructured version of the original pDTMC, where the restructuring (described later in this section) aids the formation of fragments;
2) a revised version of the reward function rwd that matches the restructured pDTMC;
3) a set of fragments FS that satisfy Definition 6, with each state of the restructured pDTMC either included in a *regular, multi-state fragment* or organised into a *one-state (degenerate) fragment*.

Algorithm 1 pDTMC model fragmentation

1: **function** FRAGMENTATION($D(S, s_0, \mathbf{P}, L), \phi, rwd, \alpha$)
2: $\quad V \leftarrow \begin{cases} \{s \in S \mid s \models \Phi_1 \vee s \models \Phi_2\}, & \text{if } \phi = \mathcal{P}_{=?}[\Phi_1\,\mathrm{U}\,\Phi_2] \\ \{s \in S \mid s \models \Phi\}, & \text{if } \phi = \mathcal{P}_{=?}[\mathrm{F}\,\Phi] \vee \phi = \mathcal{R}^{rwd}_{=?}[\mathrm{F}\,\Phi] \end{cases}$
3: $\quad FS \leftarrow \{(\{s\}, s, \{s\}) \mid s \in V\}$
4: $\quad$ **for all** $z_0 \in S \setminus V$ **do**
5: $\quad\quad Z \leftarrow \{z_0\}$
6: $\quad\quad Z_{\mathsf{OUT}}, Z' \leftarrow \{\}, \{\}$
7: $\quad\quad T \leftarrow$ EMPTYSTACK()
8: $\quad\quad$ EXPAND($D(S, s_0, \mathbf{P}, L), rwd, FS, V, T, Z, z_0, \mathsf{true}$)
9: $\quad\quad$ **while** $\neg$EMPTY(T) **do**
10: $\quad\quad\quad z \leftarrow T$.POP()
11: $\quad\quad\quad$ **if** $pred(z) \subseteq Z \wedge succ(z) \subseteq S \setminus Z$ **then**
12: $\quad\quad\quad\quad Z_{\mathsf{OUT}} \leftarrow Z_{\mathsf{OUT}} \cup \{z\}$
13: $\quad\quad\quad$ **else**
14: $\quad\quad\quad\quad$ **if** $\#Z < \alpha$ **then**
15: $\quad\quad\quad\quad\quad$ EXPAND($D(S, s_0, \mathbf{P}, L), rwd, FS, V, T, Z, Z', z, \mathsf{false}$)
16: $\quad\quad\quad\quad$ **else**
17: $\quad\quad\quad\quad\quad$ TERMINATE($D(S, s_0, \mathbf{P}, L), \phi, rwd, FS, V, T, Z, Z', Z_{\mathsf{OUT}}, z$)
18: $\quad\quad\quad\quad$ **end if**
19: $\quad\quad\quad$ **end if**
20: $\quad\quad\quad Z \leftarrow Z \cup \{z\}$
21: $\quad\quad$ **end while**
22: $\quad\quad$ **if** $\neg$VALIDFRAGMENT($(Z, z_0, Z_{\mathsf{OUT}})$) **then**
23: $\quad\quad\quad Z \leftarrow \{z_0\}, Z_{\mathsf{OUT}} \leftarrow \{z_0\}$
24: $\quad\quad\quad$ REMOVENEWSTATES($D(S, s_0, \mathbf{P}, L), Z'$)
25: $\quad\quad$ **end if**
26: $\quad\quad FS \leftarrow FS \cup \{(Z, z_0, Z_{\mathsf{OUT}})\}$
27: $\quad\quad V \leftarrow V \cup Z$
28: $\quad$ **end for**
29: $\quad$ **return** $D(S, s_0, \mathbf{P}, L)$, rwd, FS
30: **end function**

The function starts by placing the pDTMC states that satisfy Φ_1 and Φ_2 (if the analysed property is an unbounded until formula $\mathcal{P}_{=?}[\Phi_1\,\mathrm{U}\,\Phi_2]$) or Φ (if the analysed property is a reachability formula $\mathcal{P}_{=?}[\mathrm{F}\,\Phi]$ or a reachability reward formula $\mathcal{R}^{rwd}_{=?}[\mathrm{F}\,\Phi]$) into a set of "visited" states V (line 2). Each state from V is then used to assemble a one-state fragment that is placed into the fragment set FS (line 3). The states from V are preserved as one-state fragments so that they can appear in the fPMC abstract model (see the description from Section 2.3). This allows the direct analysis of the PCTL property ϕ, which refers to these states from the abstract model.

Next, additional fragments $(Z, z_0, Z_{\mathsf{OUT}})$ are generated in each iteration of the for loop from lines 4–28 as follows. First, a node z_0 not yet included in any fragment is selected (line 4) and inserted into the fragment state set Z (line 5), while the fragment output set Z_{out} and a set Z' of new states that may be created during the assembly of the current fragment are initialised to the empty set (line 6). An empty stack, T, is created in line 7, and then populated with the states reached by the outgoing transitions from

4. Like other parametric model checking methods [22], [39], [44], fPMC only supports non-nested probabilistic properties.

Algorithm 2 Traversal of pDTMC-induced graph

```
1: function EXPAND(D(S,s0,P,L), rwd,FS,V,T, Z, Z', z, inputState)
2:   if ¬inputState then                    ▷ z is not the fragment's input state
3:     I ← pred(z) ∩ (S\Z)
4:     if I ∩ V = {} then
5:       T.PUSH(I)
6:     else
7:       FS ← FS ∪ {({z}, z, {z})}
8:       V ← V ∪ {z}
9:       return
10:    end if
11:  end if
12:  O ← succ(z) ∩ (S\Z)
13:  if O ⊈ V then
14:    T.PUSH(O\V)
15:    if V\O ≠ {} then
16:      O' ← RESTRUCTURESTATE(D(S, s0, P, L), rwd, Z, Z', z)
17:      T.PUSH(O')
18:    end if
19:  end if
20: end function
```

z_0 through invoking (in line 8) the function EXPAND from Algorithm 2. Each state z from this stack, T, is processed by the while loop from lines 9–21, ending up in Z_{OUT} if it satisfies the constraints associated with output fragment states (lines 11 and 12, where $pred(z) = \{i \in S \mid \mathbf{P}(i,z) \neq 0\}$ and $succ(z) = \{o \in S \mid \mathbf{P}(z,o) \neq 0\}$ denote the sets of predecessor and successor states of state z, respectively). When z does not satisfy these constraints, two options are possible (lines 13–19):

- If Z has accumulated fewer states than the threshold α, the graph traversal function EXPAND is invoked again to add to the stack the predecessor and successor vertices of z that are not already in the fragment (line 15).
- Otherwise, the function for the early fragment formation TERMINATE from Algorithm 3 is invoked to "force" z into becoming an output state (by restructuring the pDTMC) whenever that is possible (line 17).

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 4.2 are used to force the formation of a valid fragment if possible. Irrespective of the way in which z is processed in lines 11–19, it becomes part of the fragment being constructed, and therefore it is added to the fragment state set Z in line 20.

The fragment candidate (Z, z_0, Z_{OUT}) assembled by the while loop from lines 9–21 is validated in line 22. If the candidate does not satisfy the constraints from Definition 6, the fragment is "downgraded" by using its input state, z_0, to form a degenerate, one-state fragment (line 23) and any new states created through pDTMC restructuring are removed (in line 24) because they were not used. We note that this step is included mainly for purposes of the termination proof (see Theorem 1 in Section 4.3) and does not affect the fPMC output formula. After validation (and, if necessary, degradation or restructuring), the new fragment is added to the fragment set FS (line 26), and its states are added to the set of "visited" states V already assigned to fragments (line 27), ensuring that they are not re-used by the loop starting in line 4.

The growing of a fragment in Algorithm 1 is carried out by the function EXPAND from Algorithm 2. Given a state z, EXPAND examines:

- its incoming transitions in lines 2–11 (only if z is not the input state z_0 of the fragment under construction, i.e., if $inputState$ is false in line 2);
- its outgoing transitions in lines 12–19.

The states that are directly connected to z via the examined incoming and outgoing transitions, and that are not already in the set of fragment states Z, are collected into an input state set I (line 3) and an output state set O (line 12), respectively. These two sets of states are processed as follows.

Firstly, the states from I are added to the stack T (line 5) if none of them belongs to an existing fragment (line 4). Otherwise, z is organised into a one-state fragment and the traversal of the pDTMC-induced graph is terminated (lines 7–9) because, with an incoming transition from a state belonging to an already assembled fragment, z cannot be an inner or output state of the fragment under construction. Note that creating this one-state fragment halfway through assembling another fragment may impact the construction of the other fragment. If this is the case, then the issue will be detected and dealt with by the validation process from Algorithm 1 (line 22).

Secondly, if the output set O has at least one state not belonging to other fragments (line 13), growing the fragment under construction with the "successors" of z may be feasible. As such, the function:

- places the states from O that do not belong to other fragments onto the stack T (line 14);
- if O contains states belonging to previously constructed fragments (line 15), it attempts to continue to grow the fragment under construction by using the function RESTRUCTURESTATE from Algorithm 4 to extend the pDTMC with auxiliary states O' that allow z to become an inner fragment state (lines 16 and 17).

4.2 Early Termination of Fragment Construction and Model Restructuring to Aid Fragment Formation

As we will show in Section 4.3, the function FRAGMENTATION from Algorithm 1 is guaranteed to partition a pDTMC into a set of valid fragments. However, the success of fPMC also depends on these fragments being of an appropriate size. If a fragment is too large, existing pMC techniques (which fPMC uses for the fragment analysis, see Figure 3) will be unable to handle it. Conversely, partitioning a pDMTC into a very large number of small fragments may yield an abstract model whose analysis is unfeasible.

Based on our experience (see Section 6), pDTMCs that model complex systems often comprise many loops (e.g., see the pDTMC from Figure 6), whichs favour the formation of fragments that may be too large for existing pMC techniques to analyse. To address this issue, FRAGMENTATION uses the threshold α to decide when to force the formation of a fragment, preventing it from growing too large (line 14 from Algorithm 1). This early termination of the fragment formation is accomplished by the function TERMINATE. This function is supplied (in line 17) with complete information about the fragmentation process so far and, importantly, with a state z that does not satisfy the condition from line 11 and therefore cannot be an output state for the fragment

Algorithm 3 Early termination of fragment formation

1: **function** TERMINATE($D(S,s_0,\mathbf{P},L), \phi, rwd, FS, V, T, Z, Z', Z_{\mathsf{OUT}}, z$)
2: | **if** $pred(z) \cap (S \setminus Z) \neq \emptyset \wedge succ(z) \cap (Z \setminus \{z_0\}) = \emptyset$
3: | | $\wedge\ (\phi \neq \mathcal{R}^{rwd}_{=?}[\mathrm{F}\ \Phi] \vee rwd(z) = 0)$ **then**
4: | | RESTRUCTURETRANS($D(S, s_0, \mathbf{P}, L), Z, z$)
5: | | $Z_{\mathsf{OUT}} \leftarrow Z_{\mathsf{OUT}} \cup \{z\}$
6: | **else if** $pred(z) \cap (S \setminus Z) = \emptyset \wedge succ(z) \cap (Z \setminus \{z_0\}) \neq \emptyset$ **then**
7: | | $O \leftarrow$ RESTRUCTURESTATE($D(S, s_0, \mathbf{P}, L), rwd, Z, Z', z$)
8: | | $Z \leftarrow Z \cup O$, $Z_{\mathsf{OUT}} \leftarrow Z_{\mathsf{OUT}} \cup O$
9: | **else**
10: | | EXPAND($D(S, s_0, \mathbf{P}, L), rwd, FS, V, T, Z, z$, false)
11: | **end if**
12: **end function**

Algorithm 4 pDTMC restructuring

1: **function** RESTRUCTURETRANS($D(S, s_0, \mathbf{P}, L), Z, z$)
2: | $I \leftarrow pred(z) \cap (S \setminus Z)$
3: | $O \leftarrow succ(z)$
4: | **for all** $i \in I$ **do**
5: | | $\mathbf{P}(i, z) \leftarrow 0$
6: | | **for all** $o \in O$ **do**
7: | | | $\mathbf{P}(i, o) \leftarrow \mathbf{P}(i, z) \cdot \mathbf{P}(z, o)$
8: | | **end for**
9: | **end for**
10: **end function**

11: **function** RESTRUCTURESTATE($D(S, s_0, \mathbf{P}, L), rwd, Z, Z', z$)
12: | $O \leftarrow succ(z) \cap (S \setminus (Z \setminus \{z_0\}))$
13: | $NewStates \leftarrow \{\}$
14: | **for all** $o \in O$ **do**
15: | | $z' \leftarrow$ NEWSTATE()
16: | | $S \leftarrow S \cup \{z'\}$
17: | | **for all** $s \in S$ **do**
18: | | | $\mathbf{P}(s, z') \leftarrow 0$, $\mathbf{P}(z', s) \leftarrow 0$
19: | | **end for**
20: | | $\mathbf{P}(z, z') \leftarrow \mathbf{P}(z, o)$, $\mathbf{P}(z, o) \leftarrow 0$, $\mathbf{P}(z', o) \leftarrow 1$
21: | | $L(z') \leftarrow \{\}$
22: | | $rwd(z') \leftarrow 0$
23: | | $NewStates \leftarrow NewStates \cup \{z'\}$
24: | **end for**
25: | $Z' \leftarrow Z' \cup NewStates$
26: | **return** $NewStates$
27: **end function**

28: **function** REMOVENEWSTATES($D(S, s_0, \mathbf{P}, L), Z'$)
29: | **for all** $z' \in Z'$ **do**
30: | | $\{i\} \leftarrow pred(z')$, $\{o\} \leftarrow succ(z')$
31: | | $\mathbf{P}(i, o) \leftarrow \mathbf{P}(i, z')$
32: | **end for**
33: | $S \leftarrow S \setminus Z'$
34: **end function**

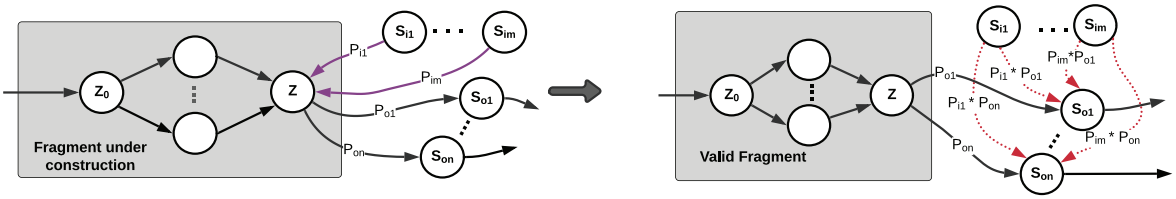
under construction. The role of TERMINATE is to modify the pDTMC states and/or transitions such that: (i) z meets the condition for being an output fragment state in the restructured pDTMC; (ii) the modifications do not affect the pMC result. This restructuring is possible in one of the following two scenarios, which are handled in lines 2–5 and 6–8 of TERMINATE, respectively. If neither scenario applies, TERMINATE cannot support the early termination of the fragment construction, and therefore needs to invoke the function EXPAND, which will continue to grow the fragment (line 10). As such, the threshold α only provides a soft upper bound for the size of an fPMC fragment.

Scenario 1. The state z has $m \geq 1$ incoming transitions (of probabilities p_{i1}, p_{i2}, …, p_{im}) from states s_{i1}, s_{i2}, …, s_{im} outside the fragment, and all its $n \geq 1$ outgoing transitions (of probabilities p_{o1}, p_{o2}, …, p_{on}) to states s_{o1}, s_{o2}, …, s_{on} outside the fragment or to the state z_0 of the fragment under construction (line 2 from Algorithm 3 and Figure 7a–left); additionally (for reasons explained in Theorem 2 in Section 4.3) either the analysed property ϕ is not a reachability reward property, or z is a zero-reward state. In this scenario, we replace the transition between each state s_{ij}, $1 \leq j \leq m$, and z with transitions of probabilities $p_{ij}p_{o1}$, $p_{ij}p_{o2}$, …, $p_{ij}p_{on}$, between s_{ij} and the states s_{o1}, s_{o2}, …, s_{on}, respectively. This modification of the pDTMC structure is shown in Figure 7a–right, and is carried out by function RESTRUCTURETRANS from Algorithm 4. To perform the restructuring, the function first assembles the sets of states $I = \{s_{i1}, s_{i2}, \ldots, s_{im}\}$ (line 2) and $O = \{s_{o1}, s_{o2}, \ldots, s_{on}\}$ (line 3), and then iterates through state pairs $(i, o) \in I \times O$, removing the transition from state i to state z (line 5) and inserting a transition from state i to state o. As a result of this modification of the original pDTMC, state z meets the condition for becoming an output state of the fragment under construction, and will be placed into the set of outputs states Z_{OUT} in line 5 from Algorithm 3.

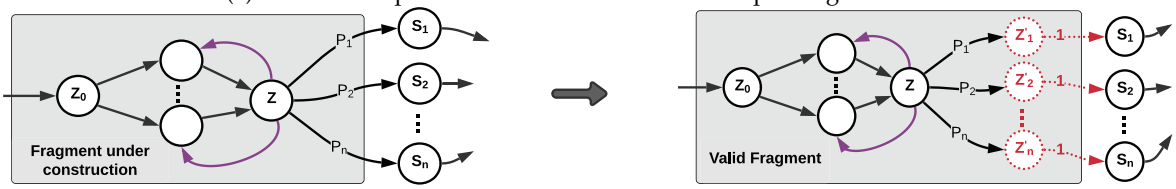
Scenario 2. The state z has no incoming transitions from outside the fragment under construction, but has outgoing transitions to one or more states inside the fragment in addition to $n \geq 1$ outgoing transitions (of probabilities p_1, p_2, …, p_n) to states s_1, s_2, … s_n outside of the fragment or to the state z_0 of the fragment under construction (Figure 7b–left). In this scenario, we augment the pDTMC with states z'_1, z'_2, …, z'_n, and we replace each transition between states z and s_j, $1 \leq j \leq n$ with a transition of probability p_j between z and z'_j *and* a transition of probability 1 between z'_j and s_j. This change supports the formation of a fragment whose output state set includes the auxiliary states z'_1, z'_2, …, z'_n (Figure 7b–right), and is performed by the function RESTRUCTURESTATE from Algorithm 4. This function assembles a set O comprising the states s_1, s_2, … s_n in line 12 and creates the set of $NewStates$ z'_1, z'_2, …, z'_n in the for loop from lines 14–24. After it is created in line 15, each new state is added to the state set S in line 16, has its incoming and outgoing transition probabilities initialised in lines 17–20, is associated with an empty label set and with a zero reward in lines 21 and 22, respectively, and is added to the set of $NewStates$ in line 23. This $NewStates$ set is added to the overall set of new states Z' in line 25 and then returned in line 26, so that the function TERMINATE can add the new states to both the set Z of fragment states and the set Z_{OUT} of output fragment states (line 8 from Algorithm 3).

The pDTMC restructuring through the creation of new states by RESTRUCTURESTATE may not always lead to the assembly of a valid fragment. When this is the case, the function REMOVENEWSTATES from Algorithm 4 is invoked in line 24 from Algorithm 1 to remove these unnecessary new states. This removal involves first restoring the transitions from the left-hand side of the transformation from Figure 7b (which is carried out by the for loop from lines 29– 32 of REMOVENEWSTATES), and then removing the states from the pDTMC (line 33).

Before providing correctness proofs for the fPMC fragmentation in the next section, we note that the function RESTRUCTURESTATE is also used to support the growing of the fragment under construction in the function EXPAND



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



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

Fig. 7: Model restructuring techniques supporting fragment formation

from Algorithm 2. This use occurs when EXPAND processes (in lines 16 and 17) a state z that has outgoing transitions to states belonging to previously constructed fragments.

4.3 Correctness of the fPMC Fragmentation

We start by showing that the pDTMC fragmentation produced by fPMC is valid, noting that the way in which the fragments of a pDTMC can be used to speed up parametric model checking is summarised in Section 2.3 and proven correct in our previous work [18]. We also note that the correctness proof is made easy by the fact that the fPMC fragmentation method is a heuristic, and therefore we only need to show that it returns a correct set of fragments for the (restructured) pDTMC model; nothing is proven about the quality of the fragmentation, which is evaluated experimentally later in the paper. Before providing this result, we show that the pDTMC restructuring carried out within each iteration of the for loop from lines 4– 28 of Algorithm 1 does not result in any new states being present in $S \setminus V$ at the beginning of the next loop iteration.

Lemma 1. *The set $S \setminus V$ used to select an input fragment state z_0 in line 4 of Algorithm 1 contains only elements from the initial pDTMC D passed as an argument to the function* FRAGMENTATION.

Proof. To prove the lemma, we show that no new state created by function RESTRUCTURESTATE during an iteration of the for loop is left in $S \setminus V$ by the end of that iteration. RESTRUCTURESTATE may be called in two parts of the fPMC fragmentation: in line 16 from Algorithm 2, and in line 7 from Algorithm 3. After any instance of the former call, the new states created by RESTRUCTURESTATE are added to the stack T (in line 17 of Algorithm 2) and then moved, one by one, to the fragment state set Z in line 20 of the while loop from lines 9– 21 of Algorithm 1. After any instance of the latter call, the new states created by RESTRUCTURESTATE are immediately added to the fragment state set Z in line 8 of Algorithm 3. As such, all newly created states will eventually end up both in Z and, due to the operations in lines 16 and 25 of Algorithm 4, also in S and Z'.

What happens to these new states after the while loop from lines 9– 21 from Algorithm 1 depends on whether the fragment (Z, z_0, Z_{OUT}) under construction is found to be valid or not in line 22 of Algorithm 1:

- If (Z, z_0, Z_{OUT}) is a valid fragment, it is added to the set FS in line 26 and the set of visited states V is extended with the states from Z (which include all the newly created states) in line 27. As a result, the new states end up in both S and V when the for loop iteration finishes, and therefore $S \setminus V$ will contain no such state.
- Otherwise, the newly created states are removed from S when function REMOVENEWSTATES is called in line 24 from Algorithm 1 (see also line 33 from Algorithm 4), and therefore $S \setminus V$ will again contain no new state by the time the for loop iteration finishes. We note that calling REMOVENEWSTATES restores the pDTMC D to its variant from the beginning of current iteration of the for loop from lines 4–28 of Algorithm 1 by using the state set Z', which contains all new states created during this iteration because Z' starts empty in line 6 of Algorithm 4 and is only modified to include the new states created by the function RESTRUCTURESTATE in line 25 of this function. To achieve the pDTMC restoration, REMOVENEWSTATES exploits the fact that each new state $z'_i \in Z'$ only has one incoming transition of probability p_i from a state z and one outgoing transition of probability 1 to a state s_i, where z and s_i are the states between which z'_i was "inserted" into D (see Figure 7b–right and line 20 from Algorithm 4). As such, REMOVENEWSTATES replaces these two transitions with a direct transition (of probability p_i) from state z to state s_i in lines 30 and 31 (thus restoring the pDTMC structure from Figure 7b–left), and then eliminates all new states from the pDTMC state set S in line 33.

□

Theorem 1. *Function* FRAGMENTATION *returns a valid fragmentation of the pDTMC $D(S, s_0, \boldsymbol{P}, L)$ as restructured by its auxiliary functions.*

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 state from S is included in more than one fragment); and

(c) the function terminates and returns a set of fragments FS that includes all the states from the state set S of the restructured pDTMC.

To prove part (a), we note that new fragments are added to FS in three lines from Algorithms 1 and 2. In line 3 of Algorithm 1 and in line 7 of Algorithm 2, FS is augmented with tuples that represent degenerate, one-state fragments according to the definition given in Section 2.3. Finally, in line 26 of Algorithm 1, FS is augmented with a tuple $(Z, z_0, Z_{\mathsf{OUT}})$ that either passes the fragment-validity check from line 22, or is reduced to a degenerate, one-state fragment in line 23 before it is included in FS. As such, any tuple inserted into FS is a valid fragment. We note that this part of the proof did not consider lines 9–21 from Algorithm 1 because, as stated earlier, these lines only influence the quality and not the correctness of the fragmentation.

To prove part (b), we note that the states from the set V assembled in line 2 of FRAGMENTATION are each placed into a degenerate, one-state fragment in line 3 of Algorithm 1, and that the state set Z of any fragment $(Z, z_0, Z_{\mathsf{OUT}})$ added to FS comes from $S \setminus V$, where V is updated (in line 27 of Algorithm 1, and in line 8 of Algorithm 2) to include all the states of new fragments included in FS. To see that the states of all new fragments come from $S \setminus V$, observe that:

(i) state z_0 added to Z in line 5 and line 23 of Algorithm 1 comes directly from $S \setminus V$ (line 4);

(ii) state z added to Z in line 20 of the same algorithm (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 $S \setminus V$ (as enforced by the if statements before lines 5 and 14 from Algorithm 2, and by the use of states newly created by function RESTRUCTURESTATE in line 17 from Algorithm 2);

(iii) the states added to Z in line 8 from Algorithm 3 are states newly created by function RESTRUCTURESTATE, which do not belong to any existing fragment.

Therefore, the fragments from FS are disjoint.

To prove part (c), we note that all the functions from Algorithms 1–4 terminate. RESTRUCTURETRANS and RESTRUCTURESTATE from Algorithm 4 terminate because each of their statements (including the assembly of the state sets I and O in RESTRUCTURETRANS and of the state set O in RESTRUCTURESTATE, and their for loops) operate with finite numbers of states. As such, EXPAND also terminates because it builds and operates with finite sets of states I and O, and invokes a function that terminates (i.e., RESTRUCTURESTATE). The function TERMINATE contains no loops and invokes one of three functions, each of which is guaranteed to terminate; therefore, TERMINATE is also guaranteed to terminate. Finally, FRAGMENTATION terminates because:

(i) each iteration of its for loop adds at least state z_0 from $S \setminus V$ to V in line 27, until $S \setminus V = \{\}$ in line 4 (since S is a finite set of states) and the loop terminates with all states from S included in fragments from FS;

(ii) its while loop terminates since it iterates over the elements of stack T, to which every state in the finite set S is added at most once;

(iii) according to Lemma 1, any new states that RESTRUCTURESTATE adds to the pDTMC are not present in $S \setminus V$ by the end of the FRAGMENTATION for-loop iteration in which they were created.

Thus, FRAGMENTATION terminates, returning a fragment set FS that includes all the states from S. □

Having demonstrated that the function FRAGMENTATION yields a valid fragmentation of the restructured version of the pDTMC received as its first argument, we will show next that using this restructured pDTMC instead of the original pDTMC to analyse the PCTL formula ϕ under verification does not change the pMC result.

Theorem 2. *Applying the model restructuring techniques from Algorithm 4 and Figure 7 to a pDTMC does not affect its reachability, unbounded until and reachability reward properties.*

Proof. We first show that the theorem holds for any reachability property $\phi = \mathcal{P}_{=?}[\mathrm{F}\,\Phi]$. To that end, we consider a generic pDTMC D, the pDTMC D' obtained by applying one of the model restructuring techniques from Figure 7 to D, and the sets of all paths Π over D and Π' over D' that satisfy ϕ. According to the semantics of PCTL, 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 $\pi = s_0 s_1 s_2 \dots s_n$ starting in the initial state s_0 of 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 from Figure 7 in turn:

- For the technique from Figure 7a, a path from $\Pi \setminus \Pi'$ has the form $\pi = s_0\omega_1 s_{ij} z s_{ok}\omega_2$, with $j \in \{1, 2, \dots, m\}$, $k \in \{1, 2, \dots, n\}$, and ω_1, ω_2 subpaths such that ω_2 ends in a state that satisfies Φ. Path π has a corresponding path $\pi' = s_0\omega_1 s_{ij} s_{ok}\omega_2 \in \Pi' \setminus \Pi$ (and the other way around) such that
$$\begin{aligned}\Pr_{s_0}(\pi) &= \Pr_{s_0}(s_0\omega_1 s_{ij})\mathbf{P}(s_{ij}, z)\mathbf{P}(z, s_{ok})\Pr_{s_{ok}}(s_{ok}\omega_2)\\ &= \Pr'_{s_0}(s_0\omega_1 s_{ij})\mathbf{P}'(s_{ij}, s_{ok})\Pr'_{s_{ok}}(s_{ok}\omega_2)\\ &= \Pr'_{s_0}(\pi')\end{aligned}$$
since the restructuring technique guarantees that $\mathbf{P}'(s_{ij}, s_{ok}) = \mathbf{P}(s_{ij}, z)\mathbf{P}(z, s_{ok})$.
- For the technique from Figure 7b, a path from $\Pi \setminus \Pi'$ has the form $\pi = s_0\omega_1 z s_i\omega_2$ for some $i \in \{1, 2, \dots, n\}$ and subpaths ω_1, ω_2, with ω_2 ending in a state that satisfies Φ. Path π has a corresponding path $\pi' = s_0\omega_1 z z'_i s_i\omega_2 \in \Pi' \setminus \Pi$ (and the other way around) such that
$$\begin{aligned}\Pr_{s_0}(\pi) &= \Pr_{s_0}(s_0\omega_1 z)\mathbf{P}(z, s_i)\Pr_{s_i}(s_i\omega_2)\\ &= \Pr'_{s_0}(s_0\omega_1 z)\mathbf{P}'(z, z'_i)\mathbf{P}'(z'_i, s_i)\Pr'_{s_i}(s_i\omega_2)\\ &= \Pr'_{s_0}(\pi')\end{aligned}$$
since the restructuring technique guarantees that $\mathbf{P}'(z, z'_i) = 1$ and $\mathbf{P}'(z'_i, s_i) = \mathbf{P}(z, s_i)$.

We showed that neither of the restructuring techniques from Figure 7 affects the value of the reachability property ϕ, and therefore the finite number of applications of these techniques within the fPMC fragmentation approach do not affect this value either.

To show that the theorem holds for a generic unbounded until property $\phi = \mathcal{P}_{=?}[\Phi_1 \,\mathrm{U}\, \Phi_2]$, we first note that the sets of paths Π over D and Π' over D' that satisfy ϕ are subsets of the path sets Π_{reach} and Π'_{reach} that satisfy the reachability property $\mathcal{P}_{=?}[\mathrm{F}\,\Phi_2]$ over D and D', respectively. We know from the first part of the proof that Π_{reach} and Π'_{reach} are equiprobable. Consider now a generic path $\pi \in \Pi_{\mathsf{reach}} \backslash \Pi$, i.e., a path that ends in a state that satisfies Φ_2, but without any intermediate state where Φ_1 is satisfied. We have two cases. If π is unaffected by the restructuring technique used to obtain the pDTMC D' from D, then $\pi \in \Pi'_{\mathsf{reach}} \backslash \Pi'$. Otherwise, the equiprobable path $\pi' \in \Pi_{\mathsf{reach}}$ constructed from π as in the first part of the theorem will not be in Π' because none of its intermediate states can satisfy Φ_2. Indeed, any such states that are identical to states from π do not satisfy Φ_1 because no intermediate state of π does, and any new states created by the pDTMC restructuring is labelled with an empty set of atomic propositions (in line 21 of Algorithm 4) and thus does not satisfy any PCTL formula. We showed that path sets Π and Π' are obtained by removing equiprobable paths from the equiprobable path sets Π_{reach} and Π'_{reach}. As such, Π and Π' are also equiprobable, and the theorem holds for unbounded until properties.

Finally, for a generic reachability reward property $\phi = \mathcal{R}^{rwd}_{=?}[\mathrm{F}\,\Phi]$, we note that the value of ϕ is given by a weighted sum of the probabilities of all DTMC paths that satisfy the associated reachability property $\mathcal{P}_{=?}[\mathrm{F}\,\Phi]$, where the weight associated with a path $\pi = s_0 s_1 s_2 \ldots$ is the cumulative reward $rwd(s_0)+rwd(s_1)+rwd(s_2)+\ldots$ for the states on the path. As shown in the first part of the theorem, the path formula $\mathrm{F}\,\Phi$ is satisfied by pairs of equiprobable paths π and π' over D and D', respectively. We will show that the paths in every such pair have the same cumulative reward. We have three cases. First, if π is unaffected by the restructuring technique used to obtain the pDTMC D' from D, then $\pi' = \pi$, and the two cumulative rewards are trivially equal. Second, when the restructuring from Figure 7a is used, a state z from path π is skipped on path π', but otherwise the two paths are identical. However, z is in this case a zero-reward state (cf. line 2 from Algorithm 3), so the two cumulative rewards are equal. Finally, when the restructuring from Figure 7b is used, path π' only differs from π through the inclusion of an auxiliary state z'_i. Since $rwd(z_i) = 0$ (cf. line 22 from Algorithm 4), the cumulative rewards for the two paths are again equal. As such, the theorem also holds for reachability reward properties. □

We have shown so far that fPMC produces valid pDTMC fragmentations, and that the model restructuring used during this fragmentation does not impact the pMC of reachability, unbounded until and reachability reward properties. The next result establishes the complexity of the fPMC fragmentation. To derive this result, we adopt the standard graph notation $indegree(s) = \#\{s' \in S \,|\, \mathbf{P}(s', s) \neq 0\}$, $outdegree(s) = \#\{s' \in S \,|\, \mathbf{P}(s, s') \neq 0\}$ and $degree(s) = \max\{indegree(s), outdegree(s)\}$ to denote the number of incoming transitions, the number of outgoing transitions, and the maximum between the two numbers, respectively, for a state s of a pDTMC.

Theorem 3. *The function* FRAGMENTATION *requires at most $\mathcal{O}(n^3 d)$ steps, where n represents the number of pDTMC states after model restructuring*[5] *and $d = \max_{s \in S} degree(s)$.*

Proof. The function RESTRUCTURETRANS (Algorithm 4) requires $\mathcal{O}(n)$ steps to assemble the state sets I and O, and at most $\mathcal{O}(d^2)$ steps to process up to $indegree(z) \cdot outdegree(z)$ pairs of incoming-outgoing transitions of a state z. As such, it has $\mathcal{O}(\max\{n, d^2\})$ overall complexity. Given a state z, the function RESTRUCTURESTATE requires $\mathcal{O}(n)$ steps to assemble the state set O, and then $\mathcal{O}(nd)$ steps to initialise $2n$ transition probabilities for each new state it creates, since one new state is created for each of the up to $outdegree(z)-1 < d$ outgoing transitions of z. Therefore, the overall complexity of RESTRUCTURESTATE is $\mathcal{O}(nd)$.

The function EXPAND requires at most $\mathcal{O}(nd)$ steps due to the invocation of RESTRUCTURESTATE, with its other operations (i.e., building the set I and placing it onto the stack T, and building the set O and placing $O \backslash V$ onto the stack T) performed in $\mathcal{O}(n)$ time.

The function TERMINATE requires $\mathcal{O}(n)$ time to evaluate the conditions whose values determine which of the functions RESTRUCTURETRANS, RESTRUCTURESTATE and EXPAND it needs to invoke. As such, the complexity of TERMINATE is given by the highest complexity among these three functions, i.e., $\mathcal{O}(nd)$ for both RESTRUCTURESTATE and EXPAND. Note that this complexity is higher than the $\mathcal{O}(\max\{n, d^2\})$ complexity of RESTRUCTURETRANS since $nd \geq n$, and (because $n \geq d$) $nd \geq d^2$.

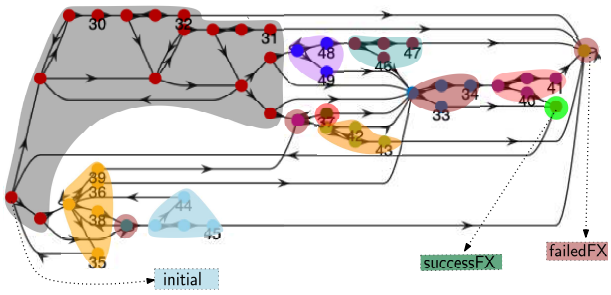
In the worst-case scenario where the fragmentation produces only one-state fragments, the execution of FRAGMENTATION requires the execution of its for loop from lines 4–28 for each of the $n_0 \leq n$ states of the initial pDTMC (with any new states created by RESTRUCTURESTATE included into the same fragment as the state that led to their creation, cf. Figure 7b). Each iteration of this loop executes:

i) EXPAND (line 8) in $\mathcal{O}(nd)$ steps;
ii) a while loop (lines 9–21) with at most n iterations (one for each pDTMC state) that may each invoke the $\mathcal{O}(nd)$-step EXPAND or the $\mathcal{O}(nd)$-step TERMINATE, yielding an $\mathcal{O}(n^2 d)$ complexity for the while loop;
iii) the fragment validity check from line 22, which requires no more than $\mathcal{O}(nd)$ operations.

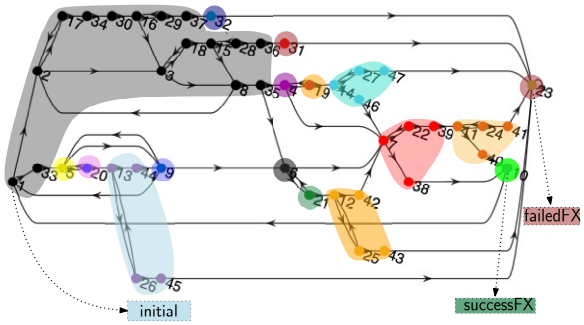
Thus, each iteration of the for loop from lines 4–28 is completed in no more than $\mathcal{O}(n^2 d)$ steps (due to the while loop from lines 9–21), and the entire FRAGMENTATION requires $\mathcal{O}(n^3 d)$ steps in the worst-case scenario. □

We note that the coefficients associated with n and d from the big-$\mathcal{O}$ notation in Theorem 3 are typically well below 1. For instance, in all our experiments, the for loop from FRAGMENTATION was only executed for a small fraction of the pDTMC states (because many fragments with multiple states are typically produced), and TERMINATE, RESTRUCTURETRANS and RESTRUCTURESTATE were only executed sparingly. Furthermore, it is worth noting that pDTMCs are typically sparsely connected graphs, and therefore d is relatively small.

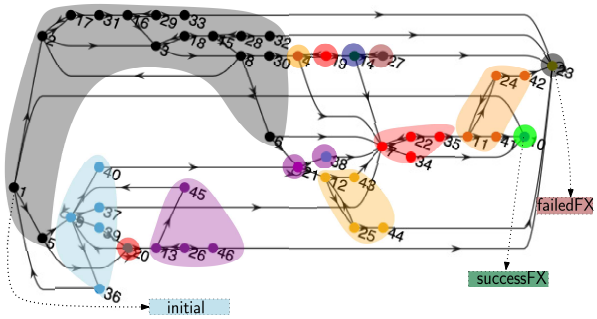
5. RESTRUCTURESTATE may add up to $outdegree(z) - 1$ new states for each of the n_0 states of the initial pDTMC (i.e., of the pDTMC that FRAGMENTATION receives as its first argument), yielding a restructured pDTMC with $n_0(d-1)$ vertices in the worst-case scenario.



(a) pDTMC fragmentation for the reachability property from the first row of Table 1: 13 fragments were obtained, including five one-state fragments.



(b) pDTMC fragmentation for the reachability reward property from the second row of Table 1 and the "time" reward function from Figure 5: 17 fragments were obtained, including 11 one-state fragments.



(c) pDTMC fragmentation for the unbounded until property from the last row of Table 1: 15 fragments were obtained, including nine one-state fragments.

Fig. 8: fPMC fragmentation of the pDTMC from Figure 5 for the PCTL properties from Table 1 and $\alpha = 5$, with different shading used to highlight each fragment.

4.4 *fPMC* Application to the Motivating Example

We illustrate the use of fPMC by using the pDTMC model and properties from our running example (cf. Figure 5 and Table 1). The leading parametric model checkers Storm and PRISM time out without producing results for any of these properties within 60 minutes of running on the computer with the specification provided in Section 6.1. The outcome of applying fPMC to this pDTMC and each of the three PCTL properties from Table 1 (with fragmentation threshold $\alpha = 5$) are summarised in Figure 8 and Table 2.

Figure 8 depicts the pDTMC fragments generated by fPMC, which are different for each property because the function FRAGMENTATION from Algorithm 1 starts by creating property-specific sets of one-state fragments in lines 2 and 3. Table 2 shows:

- the change in pDTMC size due to fPMC restructuring (an increase of 59%–69% in the number of states, and 31%–43% in the number of transitions compared to the initial pDTMC from Figure 6);
- the time taken by the end-to-end fPMC process from Figure 3 (under 8s for each property by running the tool presented in the next section on the computer with the specification provided in Section 6.1);
- the number of arithmetic operations from the algebraic formulae of the fPMC closed-form analytical model for each analysed property, and the time required to evaluate these algebraic formulae for a given parameter valuation by running MATLAB on the computer from Section 6.1 (up to 30ms for the evaluation of the 2020-operation formulae of property P2).

Given the large size of the fPMC algebraic formulae for the three FX properties, we do not include them in the paper; they are provided on our project website [1].

5 Implementation

We developed a parametric model checking tool that implements the fPMC algorithms presented in the previous section. This tool uses:

- the model checker PRISM, to identify the pDTMC states that satisfy Φ, Φ_1 and Φ_2 in line 2 of Algorithm 1, and to obtain the transition probability matrix $\mathbf{P}$ (used throughout the fPMC algorithms) from a pDTMC specified in the PRISM modelling language (e.g., see Figure 5);
- the model checker Storm, to apply standard pMC to each pDTMC fragment and to the abstract model from Figure 3.

We note that the current version of our tool selects the first pDTMC state not yet allocated to a fragment as the initial state z_0 of each new fragment from line 4 of Algorithm 1. While our experiments from Section 6 show that this selection yields good results, the associated pDTMC fragmentations may be suboptimal, and we plan to explore better approaches to selecting z_0 for future versions of the tool.

As pDTMCs with small numbers of parameters are already handled extremely efficiently by Storm, our tool only invokes the end-to-end fPMC approach for pDTMCs whose number of such parameters exceeds a user-configurable threshold $\beta \in \mathbb{N}_{>0}$. For pDTMCs with up to β model parameters, the tool invokes Storm directly, and the pMC is performed on the unfragmented model. According to our experimental results (presented in Figure 9), β values in the range 21..30 work well for most models.

6 Evaluation

6.1 Evaluation Methodology

We carried out extensive experiments to answer the research questions summarised below.

TABLE 2: Summary of fPMC application to the pDTMC and properties of the FX system in Section 3

		Restructured pDTMC†			fPMC closed-form analytical model (cf. Fig. 3)	
ID	**Property type**	**States**	**Transitions**	**fPMC time**	**Arithmetic operations**	**Evaluation time**
P1	Reachability	49	83	7s	1456	<1s
P2	Reachability reward	47	76	8s	2020	<1s
P3	Unbounded until	46	79	5s	1224	<1s

†starting from the initial FX pDTMC model with 29 states and 58 transitions in Figure 6

RQ1 (Efficiency): Does fPMC model fragmentation improve the efficiency of parametric model checking? We assess if our fPMC approach speeds up parametric model checking in comparison to PRISM [45] and Storm [24], and whether it can handle pDTMCs that cannot be analysed by the two leading model checkers (with a 60-minute timeout).

RQ2 (Result complexity): Does fPMC reduce the complexity of the closed-form formulae generated by parametric model checking? We assess whether the fPMC-computed algebraic formulae are simpler (in terms of number of arithmetic operations) than those computed by the leading model checkers, and whether they can be evaluated faster than those produced by Storm.

RQ3 (Configurability): How does the fragmentation threshold α affect the results of fPMC? We examine how different fragmentation threshold values affect fPMC in terms of the number of operations from the computed closed-form formulae, and execution time for producing those formulae.

Three pDTMC models with parameterization options that yield 62 different model variants are used in the evaluation. These pDTMCs model the behaviour of a service-based system (**FX system**), a software product line (**PL system**), and a middleware (**COM process**). The significant differences among the key characteristics of these models are summarised in Table 3. These three software systems and processes were chosen from distinct application domains and were sourced from leading software engineering venues [18], [20], [33], [36], [41] to mitigate the bias that might have been introduced if models assembled specifically for evaluating fPMC had been used. Additionally, these particular systems were selected because their Markov models contain:

(i) multiple transition probabilities that can be meaningfully specified as functions over a set of system parameters;
(ii) configuration variables that can be instantiated to obtain pDTMCs of different sizes (i.e., with a broad range of state and transition numbers).

All model variants available for the FX system and COM process were used in the evaluation. The PL model variants used in the evaluation were selected so as to include pDTMCs with a wide range of feature numbers, with a focus on variants with large numbers of features. More details about these models are provided below.

FX system. We introduced this system in Section 3, and its pDTMC corresponding to the sequential execution strategy with retry (SEQ_R) in Figures 5 and 6. For the fPMC evaluation in this section, we also considered the additional strategies below (applied to between one and five functionally equivalent service implementations per FX 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;
- PROB_R—similar to PROB, but if the selected service fails, it is retried with a given probability (as in SEQ_R).

PL system. We used a pDTMC model of a product line (PL) system taken from [20], [36]. This pDTMC models the software controller of a vending machine that dispenses a user-selected beverage and, if applicable, takes payment from and gives back change to the user.

The possible features of this system comprise: the beverage type (soda, tea, or both), the payment mode (cash or free), and the taste preference (e.g., add lemon or sugar). This variability enables the derivation of vending machines—and the specification of associated pDTMCs—with between four and 22 features.

COM process. We considered a communication (COM) process among $n \geq 2$ agents taken from [41], and inspired by the way in which honeybees emit an alarm pheromone to recruit workers and protect their colonies from intruders. Given the self-destructive defence behaviour in social insects (the recruited workers die after completing their defence actions), a balance between efficient defence and preservation of a critical mass of workers is required. The induced pDTMC is a stochastic population model with n parameters. The quantitative analysis of such stochastic models of multi-agent systems is often challenging because the dependencies among the agents within the population make the models complex.

Throughout the evaluation, fPMC is compared to the leading pMC model checkers PRISM (version 4.6) and Storm (version 1.5.1), both with their default settings. In total, 308 model variant/property combinations were analysed, of which 82 correspond to reachability properties, 165 correspond to reward properties, and 61 correspond to unbounded until properties. All experiments were performed on a MacBook Pro with 2.7GHz dual Core Intel i5 processor and 8GB RAM, using a timeout of 60 minutes. For a fair comparison, we ensured that both PRISM and Storm can

TABLE 3: Key characteristics of the systems and pDTMC models used for the fPMC evaluation

	FX system	**PL system**	**COM process**
Application domain	Financial	Vending machine controller	Communication protocol
System type	Service-based system	Software product line	Middleware
Number of model variants	21	40	1
Number of states	11–208	92–115	234
Number of transitions	22–399	167–198	444
Number of model parameters	11–71	10–198	20
Analysed properties	Reachability (FX:P1) Reward × 2 (FX:P2; FX:P4) Unbounded until (FX:P3)	Reachability (PL:P1) Unbounded until (PL:P2)	Reachability×21(COM:P1–P21) Reward (COM:P22)
Sample reachability property	$\mathcal{P}_{=?}[\mathrm{F}\ \mathsf{successFX}]$	$\mathcal{P}_{=?}[\mathrm{F}\ \mathsf{SUCCESS}]$	$\mathcal{P}_{=?}[\mathrm{F}\ \neg\mathsf{a0} \wedge \neg\mathsf{a1} \wedge \ldots \wedge \neg\mathsf{a19}]$
Sample reachability reward property	$\mathcal{R}^{time}_{=?}[\mathrm{F}\ \mathsf{successFX} \vee \mathsf{failFX}]$	–	$\mathcal{R}^{coin\_flips}_{=?}[\mathrm{F}\ \mathsf{b}]$
Sample unbounded until property	$\mathcal{P}_{=?}[\neg\mathsf{alarm}\ \mathrm{U}\ \mathsf{successFX}]$	$\mathcal{P}_{=?}[\neg\mathsf{Function1}\ \mathrm{U}\ \mathsf{SUCCESS}]$	–

successfully process at least the simplest pDTMC of each system. The following experimental data were collected:

1) the time required to compute the pMC formulae;
2) the number of arithmetic operations in the pMC formulae;[6]
3) the time required to evaluate the pMC formulae (in MATLAB) for a parameter valuation.

Software and data availability. The source code for our fPMC tool, as well as the models, properties and results for all the experiments presented in the paper, plus additional materials supporting the adoption of fPMC are freely available on our project's website [1].

Sanity check to verify the fPMC tool through testing. To examine the correctness of our fPMC tool, we evaluated the fPMC formulae produced for each analysed property using randomly generated combinations of parameter values, and confirmed that the resulting property value matched that produced by PRISM and Storm (subject to negligible rounding errors). For the purpose of this check, the PRISM and Storm results were obtained by running the probabilistic model checking on the non-parametric DTMC obtained by replacing the pDTMC parameters with the relevant combination of parameter values. While this is not a formal proof that the fPMC tool was implemented correctly, we note that even small alterations of the non-trivial formulae generated by the tool yield noticeable changes in the evaluation results, so this random testing strongly suggests that our tool implements the algorithms from Section 4 correctly.

6.2 Results and Discussion

6.2.1 RQ1 (Efficiency)

FX system. We used fPMC, PRISM and Storm to analyse pDTMC models corresponding to 21 variants of the FX system. The first of these system variants used a single service for each FX operation, and the remaining variants used one of the five execution strategies (i.e., SEQ, SEQ_R, PAR, PROB or PROB_R) and between two and five functionally equivalent services for each FX operation. For each of the 21 pDTMCs, four properties were analysed: the properties P1, P2 and P3 from Table 1, and an additional reachability reward property (property P4) used to establish the expected cost of executing the FX workflow.

The parametric model checking times for these experiments are presented in Table 4. These results show that fPMC successfully computed all pMC formulae well ahead of the 60-minute timeout for all four properties. It took fPMC just $2.9s$ to analyse properties P1 and P3 for the simplest pDTMC variant (ID 1), and under $600s$ for analysing each of the four properties for most of the other models. Only the analyses of properties P2 and P4 for the pDTMC variant with ID 17 required more time, i.e., 1305.4s and 1335.9s, respectively. In contrast, Storm only completed the analysis for 31 of the $21 \times 4 = 84$ model–property combinations before the 60-minute timeout. These combinations correspond to the simplest pDTMC variants (which Storm analysed slightly faster than fPMC) across all execution strategies except the PROB strategy. For this strategy, Storm produced pMC formulae for all model-property combinations, but with an execution time that increased very quickly over the fPMC time for the more complex PROB pDTMC variants with four and five functionally equivalent services per FX operation (i.e., the pDTMC variants with IDs 12 and 13 in the table). PRISM completed the analysis for even fewer model–property combinations: only 15 of the 84 pMC analyses returned results within 60 minutes. These results correspond again to the simplest pDTMC variants.

PL system. We used fPMC, PRISM and Storm to analyse pDTMC models corresponding to system configurations with four, 16, 18 and 22 software product line features, and with increasing numbers of parameters. To that end, we used different parameters for 10%, 20%, ..., 100% of the transition probabilities of the models for the four system configurations, obtaining $4 \times 10 = 40$ pDTMC variants. The reachability and unbounded until properties from Table 3 were analysed for each of these pDTMC variants, and the time taken by these analyses are reported in Table 5.

The results are similar to those obtained for the FX system. fPMC produced all pMC formulae successfully in between 13.3 and 93.1 seconds, while Storm and PRISM completed only 50% and 35% of the pMC analyses, respectively, before the 60-minute timeout. The two existing model checkers could analyse the pDTMCs with lower numbers of parameters, and performed their pMC faster

6. PRISM and Storm produce a single pMC formula per property, whereas fPMC yields a set of formulae per property (cf. Figure 3).

TABLE 4: Parametric model checking times (in seconds, '–' indicates a timeout) for the 21 FX pDTMC variants and their four properties (FX:P1 - Reachability, FX:P2 and FX:P4 - Reachability reward, FX:P3 - Unbounded until); STG, #SVC, #S and #T represent the strategy used to invoke multiple functionally equivalent services for an FX operation, the number of such services, and the numbers of pDTMC states and transitions, respectively.

pDTMC variant					FX:P1 pMC time			FX:P2 pMC time			FX:P3 pMC time			FX:P4 pMC time		
ID	STG	#SVC	#S	#T	fPMC	Storm	PRISM	fPMC	Storm	PRISM	fPMC	Storm	PRISM	fPMC	Storm	PRISM
1	–	1	11	22	3	<1	<1	4	<1	<1	3	<1	<1	4	<1	<1
2	SEQ	2	17	34	4	3	57	4	3	6	3	<1	18	5	3	6
3		3	23	46	4	–	–	7	–	–	4	1128	–	7	–	–
4		4	29	58	5	–	–	9	–	–	5	–	–	9	–	–
5		5	35	70	6	–	–	12	–	–	5	–	–	11	–	–
6	PAR	2	40	36	6	2	2	6	3	4	5	<1	2	6	2	3
7		3	64	111	7	–	–	8	–	–	6	–	–	9	–	–
8		4	112	207	14	–	–	22	–	–	15	–	–	33	–	–
9		5	208	399	31	–	–	45	–	–	27	–	–	78	–	–
10	PROB	2	23	46	3	<1	10	6	<1	–	5	<1	5	6	<1	–
11		3	29	64	5	3	–	9	5	–	5	1	–	8	5	–
12		4	35	82	7	27	–	12	46	–	7	10	–	11	49	–
13		5	65	130	8	593	–	17	611	–	8	153	–	16	580	–
14	SEQ_R	2	29	58	7	–	–	8	–	–	5	–	–	8	–	–
15		3	41	28	19	–	–	35	–	–	19	–	–	35	–	–
16		4	53	106	86	–	–	152	–	–	50	–	–	159	–	–
17		5	65	130	496	–	–	1305	–	–	331	–	–	1336	–	–
18	PROB_R	2	29	58	8	34	55	9	–	–	4	3	–	8	–	–
19		3	35	75	17	–	–	24	–	–	13	–	–	23	–	–
20		4	41	93	65	–	–	80	–	–	52	–	–	80	–	–
21		5	47	111	200	–	–	244	–	–	171	–	–	244	–	–

than fPMC for pDTMCs with the fewest parameters, but increasingly slower than fPMC for pDTMCs with more than approximately 40% of their transition probabilities specified as parameters.

COM process. As indicated in Table 3, a single pDTMC variant (with 234 states and 444 transitions) was analysed for the COM process. A number of 21 reachability properties and one reachability reward property (taken from [41]) were considered, and the times required to complete their parametric model checking are reported in Table 6.

Once more, fPMC was the only approach that successfully analysed all properties. Storm completed the analysis of reachability properties 1–13 much faster than fPMC, but took significantly longer to analyse the reachability properties 14–21, and timed out for the reachability reward property. PRISM completed the fewest analyses (12 out of 22) but produced the pMC formulae for these approximately 55% of the properties faster than fPMC.

Discussion. fPMC outperforms both Storm and PRISM in its ability to handle complex pDTMC with large numbers of parameters. In many of our pMC experiments with such models, fPMC completed its analysis within a few tens of seconds, while the other model checkers timed out after 3600 seconds. Thus, our approach often sped up the analysis of complex models by two or more orders of magnitude. Furthermore, the increase in the fPMC analysis time as the models became more complex was consistently much slower than the increase in the analysis time for the other model checkers for the FX and PL systems, and the fPMC pMC time was not affected much by the analysed property for the COM process.

For some of the simpler pDTMC variants (in the case of the FX and PL systems) or properties (in the case of the COM process), Storm and, only occasionally, PRISM completed the analysis faster than fPMC. These results are expected for the models and properties that Storm and PRISM can handle because the two leading model checkers use highly efficient internal representations (e.g., sparse matrices, binary decision diagrams) for DTMCs and sophisticated algorithms for their analysis. In contrast, fPMC needs to perform fragmentation before leveraging the same functionality (by using Storm) for the resulting fragments and the abstract model induced by these fragments (cf. Figure 3).

To exploit the capabilities of both fPMC (which can efficiently analyse complex pDTMCs that other tools cannot handle) and Storm (which can efficiently analyse simpler pDTMCs), our fPMC tool employs the user-configurable threshold β we mentioned in Section 5. For models with β or more parameters, the tool performs the analysis by using fPMC fragmentation, while for simple models with fewer than β parameters Storm is used directly to perform monolithic pMC. This simple heuristic represents a first step towards utilising the most suitable parametric model checking approach for the pDTMC under analysis. To support the selection of a suitable value for the threshold β, we compared the execution times of fPMC and Storm for pDTMC variants with different numbers of parameters from our evaluation. The result of this comparison is summarised by the histogram in Figure 9, which shows that most pDTMCs with up to 20 parameters were analysed faster by Storm, while almost all the pDTMCs with over 30 parameters were analysed faster by fPMC. The two tools were each able to analyse faster a subset of the pDTMCs with numbers of parameters in the range $21 \ldots 30$, though

TABLE 5: Parametric model checking times (in seconds, '–' indicates a timeout) for the 40 PL pDTMC variants and their two properties (PL:P1 and PL:P2); **#F** and **%PAR** represent the number of features in the model, and the percentage of parametric transitions, respectively.

pDTMC variant†			Reachability(PL:P1)			Unbounded until(PL:P2)		
ID	#F	%PAR	fPMC	Storm	PRISM	fPMC	Storm	PRISM
1	4	10	48	<1	<1	24	<1	<1
2		20	57	<1	<1	27	<1	<1
3		30	55	3	1	27	3	2
4		40	56	32	5	29	23	5
5		50	74	1671	106	37	712	57
6		60	82	–	–	42	–	–
7		70	87	–	–	52	–	–
8		80	92	–	–	50	–	–
9		90	93	–	–	52	–	–
10		100	93	–	–	51	–	–
11	16	10	47	<1	15	16	<1	5
12		20	50	16	–	16	6	–
13		30	20	23	–	17	3	–
14		40	20	32	–	16	4	–
15		50	23	165	–	17	16	–
16		60	22	165	–	18	92	–
17		70	23	–	–	18	–	–
18		80	24	–	–	19	–	–
19		90	26	–	–	19	–	–
20		100	24	–	–	19	–	–
21	18	10	14	<1	<1	14	<1	<1
22		20	14	<1	1	13	<1	<1
23		30	14	<1	2	13	<1	2
24		40	14	19	42	14	16	28
25		50	15	911	86	14	841	70
26		60	14	–	189	14	–	79
27		70	15	–	–	14	–	–
28		80	18	–	–	15	–	–
29		90	17	–	–	16	–	–
30		100	17	–	–	16	–	–
31	22	10	39	<1	2	38	<1	<1
32		20	46	4	53	41	3	23
33		30	42	42	–	42	39	–
34		40	42	793	–	41	569	–
35		50	42	–	–	42	–	–
36		60	43	–	–	42	–	–
37		70	43	–	–	43	–	–
38		80	43	–	–	44	–	–
39		90	48	–	–	49	–	–
40		100	49	–	–	49	–	–

†pDTMCs sizes: four-feature models = 92 states, 167 transitions
16-feature models = 110 states, 193 transitions
18-feature models = 104 states, 183 transitions
22-feature models = 115 states, 198 transitions

TABLE 6: Parametric model checking times (in seconds, '–' indicates a timeout) for the COM model and its 22 properties (COM:P1–P22)

Property		pMC time		
ID	Type	fPMC	Storm	PRISM
1	Reachability (COM:P1–P21)	13	<1	<1
2		12	<1	<1
3		12	<1	<1
4		12	<1	<1
5		13	<1	1
6		13	<1	<1
7		13	<1	3
8		13	<1	<1
9		13	<1	5
10		13	<1	2
11		12	4	4
12		12	8	9
13		12	2	–
14		12	15	–
15		12	19	–
16		13	94	–
17		12	74	–
18		12	317	–
19		12	131	–
20		11	39	–
21		12	22	–
22	Reward (COM:P22)	42	–	–

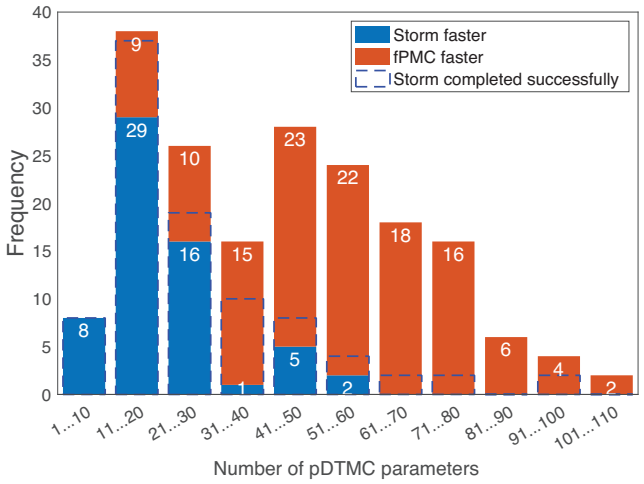


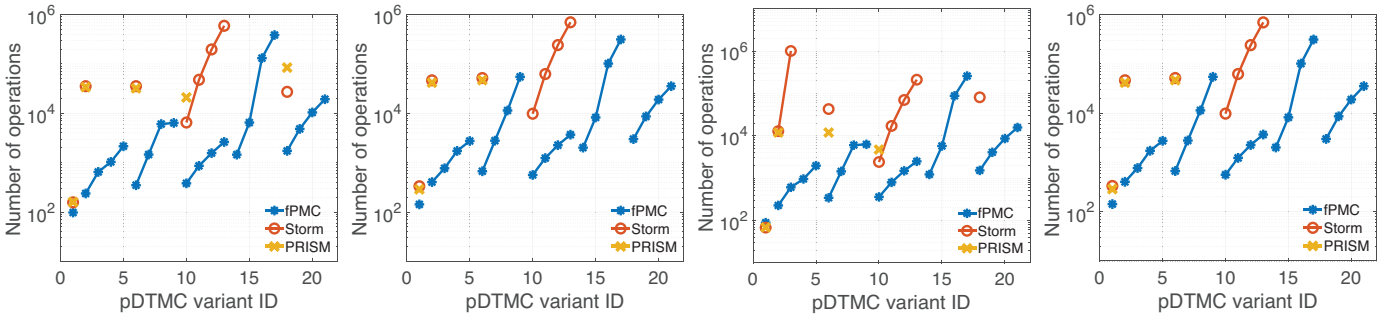
Fig. 9: pDTMCs analyses completed faster by Storm and by fPMC for models with different numbers of parameters. The dashed boxes show the total numbers of analyses completed by Storm within 60 minutes (fPMC completed all analyses).

Storm timed out before completing the analysis of several of these models. Further decomposing this set of pDTMCs into smaller ranges (e.g., between $21 \ldots 25$ and $26 \ldots 30$ parameters) does not yield a better separation into models handled faster by the two tools. As such, the comparison summarised in Figure 9 suggests that setting the threshold β to a value between 21 and 30 is likely to work well. While this rule of thumb may not always apply, we note that adopting it is never going to cause a problem: in borderline cases, one can easily analyse a pDTMC using both pMC tools. The exploration of more sophisticated heuristics that consider additional contributing factors to select the most appropriate model checking approach is an important area of future work.

6.2.2 *RQ2 (Result complexity)*

For each of our three case studies and experiments presented in the previous section, we compared the number of arithmetic operations from the pMC formulae generated by fPMC, Storm and PRISM, and the time required to evaluate the fPMC and Storm formulae in MATLAB on the computer with the specification from Section 6.1. We only considered Storm in the latter comparison because Storm completed significantly more pMC analyses than PRISM in our experiments (92 versus 55 out of a total of 186 analyses).

FX system. The sizes of the FX pMC formulae produced by fPMC, Storm and PRISM are shown in Figure 10. With one exception (for property P3 of the pDTMC variant with



(a) property FX:P1 (b) property FX:P2 (c) property FX:P3 (d) property FX:P4

Fig. 10: Number of operations in the pMC formulae for the FX pDTMC variants and properties from Table 4, with the values corresponding to the same service combination strategy (SEQ, PAR, etc.) joined by continuous lines to improve readability

TABLE 7: MATLAB evaluation time for the FX pMC formulae (in seconds)

pDTMC variant			P1		P2		P3		P4	
ID	STG	#SRV	fPMC	Storm	fPMC	Storm	fPMC	Storm	fPMC	Storm
1	–	1	<1	<1	<1	<1	<1	<1	<1	<1
2	SEQ	2	<1	4	<1	8	<1	8	<1	<1
6	PAR	2	<1	4	<1	9	<1	2	<1	<1
10	PROB	2	<1	<1	<1	<1	<1	<1	<1	<1
11		3	<1	5	<1	11	<1	11	<1	<1
12		4	<1	156	<1	272	<1	267	<1	13
13		5	<1	4245	<1	7507	<1	7499	<1	143
18	PROB_R	2	<1	17	<1	–†	<1	–†	<1	<1

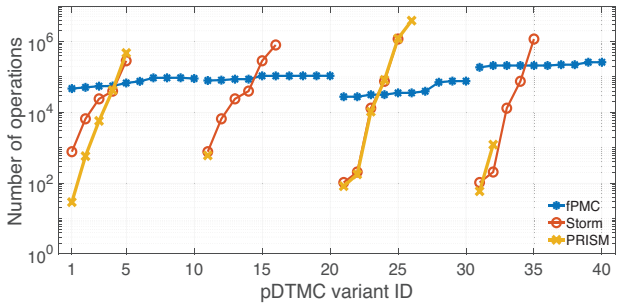
†pMC formula unavailable for evaluation due to Storm analysis timeout

ID 1), fPMC generated formulae with significantly fewer operations than the other model checkers. This difference increases quickly for larger and more complex models, with an extreme case (for property P1 of the pDTMC variant with ID 13) in which the formulae obtained by fPMC contain over 225 times fewer operations than the Storm pMC formula (i.e., 2629 versus 593426 operations).

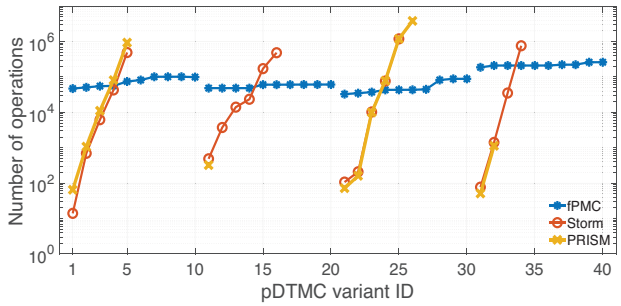
The MATLAB evaluation times for the pMC formulae produced by Storm and fPMC are reported (for the analyses completed by Storm) in Table 7. For the simplest models (e.g., pDTMC variants 1 and 10) all evaluations can be carried out within a few milliseconds. However, when the complexity of the model increases, the evaluation of the fPMC formulae is significantly faster than that of the Storm formulae. This is particularly noticeable for the pDTMC variants corresponding to the PROB service-combination strategy, e.g., the evaluation of the P2 property of pDTMC variant 13 took over 7500s when the Storm formula was used compared to only 16ms when the fPMC formulae were used. Even for pDTMC variants for which Storm completed the analysis faster than fPMC (e.g., those with IDs 2, 6, 10 and 11, cf. Table 4), the Storm pMC formulae are orders of magnitude larger than those computed by fPMC, and therefore they require much longer time to evaluate.

PL system. Figure 11 shows the pMC formula sizes generated by the three model checkers for the PL system. As expected, with a gradual increase of parametric transitions (i.e., the percentage of pDTMC transitions probabilities growing from 10% to 100%) in each model variant, larger algebraic formulae are obtained across all three model checkers, with exponential growth for the formulae computed by Storm and PRISM, neither of which can handle the pDTMC variants with over 60% of their transition probabilities specified as parameters.

For simpler pDTMC variants (i.e., those with up to 40% of their transitions specified as probabilities), the Storm and PRISM formulae have significantly fewer operators than the fPMC formulae. We investigated this unexpected result, and found it to be due to a large number of parameters in the fPMC abstract pDTMC model: one such parameter for each probability $prob_z$ of reaching an output state z of a fPMC fragment, as described in Section 2.3. As such, the fPMC abstract models generated from the pDTMC variants with between 10–40% parametric transition probabilities end up with more $prob_z$ parameters than the number of initial parameters from the PL pDTMC variants they are obtained from. However, for these pDTMCs variants, many of the fPMC fragments contain no or only a few PL parameters, and therefore multiple or even all $prob_z$ parameters for these fragments are in fact constant probabilities. We carried out separate experiments in which the constant values of such $prob_z$ parameters were used in the abstract fPMC model instead of these parameters, and the size of the resulting fPMC formulae became similar to that of the Storm and PRISM formulae for these pDTMC variants. Given these findings, we plan to include this simplification in the next version of our fPMC tool.



(a) property PL:P1



(b) property PL:P2

Fig. 11: Number of operations in the pMC formulae for the PL pDTMC variants and properties from Table 5, with the values corresponding to the system with the same number of features joined by continuous lines to improve readability

TABLE 8: MATLAB evaluation time for the PL pMC formulae (in seconds)

pDTMC variant			Reachability		Unbounded until	
ID	#F	%PAR	fPMC	Storm	fPMC	Storm
1	4	10	6	<1	6	<1
2		20	6	<1	6	<1
3		30	7	<1	7	<1
4		40	7	3	7	4
5		50	11	1636	10	1214
11	16	10	2	<1	2	<1
12		20	21	<1	6	<1
13		30	23	<1	6	<1
14		40	23	3	6	<1
15		50	33	216	9	68
16		60	34	4189	9	660
21	18	10	2	<1	2	<1
22		20	2	<1	2	<1
23		30	2	<1	3	<1
24		40	2	15	3	15
25		50	2	17114	3	16959
31	22	10	116	<1	106	<1
32		20	160	<1	142	<1
33		30	165	2	146	2
34		40	122	6050	153	5992

A similar trend can be observed in the MATLAB evaluation time for the PL pMC formulae (Table 8). For the formulae derived for the pDTMC variants with up to 30% parametric transitions, the Storm formulae can be evaluated within milliseconds, while the evaluation of the fPMC formulae takes seconds (for the pDTMC variants modelling PL instances with four and 18 features), tens of seconds (for pDTMC variants modelling PL instances with 16 features) or even more than 100s (or pDTMC variants modelling PL instances with 22 features). Again, this difference is due to fPMC operating with abstract models with unnecessarily many parameters, and can be resolved through the simplification explained earlier. For pDTMC variants with over 40% parametric transitions, the fPMC formulae are consistently and increasingly much faster to evaluate than those produced by Storm and PRISM, or these model checkers do not complete the pMC within 60 minutes.

COM process. As shown in Figure 12, the pMC formula sizes generated by fPMC, Storm and PRISM for the COM process follow a pattern similar to that obtained for the

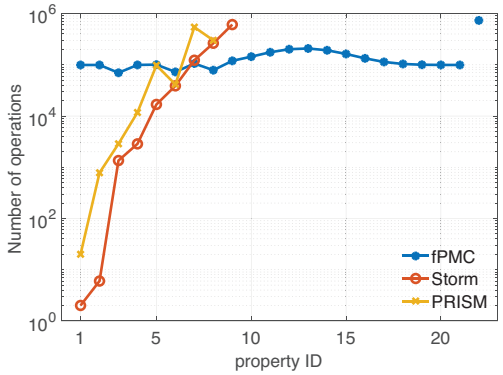


Fig. 12: Number of operations in the pMC formulae for the COM pDTMC and properties from Table 6; Storm and PRISM computed pMC formulae for several additional properties, but the multi-megabyte files required to store these extremely large formulae (which are available on our project website [1]) were difficult to process, so their numbers of operations are not provided in the diagram. The property ID ranges from COM:P1 to COM:P22.

PL system. Thus, for the simplest properties (i.e., those with IDs between 1 and 5), the sizes for the Storm and PRISM formulae are much smaller than those of the fPMC formulae; this is for the reason explained in our earlier discussion of the PL system results. However, as the analysed properties become more complex (i.e., because longer paths are required to reach the states from the reachability PCTL formulae), the number of operations from the Storm and PRISM formulae grows exponentially. In contrast, the size of the fPMC-computed formulae remains relatively stable with the increased complexity of the properties.

The MATLAB evaluation times for the Storm pMC formulae (Table 9) are again increasing exponentially from a few millisecond for properties 1–4 to tens of thousands of seconds for properties 9 and 10, with MATLAB unable to complete the evaluation before a seven-hour timeout for properties 11–21. MATLAB, however, managed to evaluate all formulae produced by fPMC within at most 42s.

Discussion. The complexity of the pMC formulae (i.e., their number of arithmetic operations) in our three case studies increased with the complexity of the analysed pDTMC and PCTL property. For the Storm and PRISM formulae, this

TABLE 9: MATLAB evaluation time for the COM process pMC formulae (in seconds)

Property ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
fPMC	29	33	15	35	35	15	33	17	19	34	35	38	41	42	36	35	35	42	36	33	30
Storm	<1	<1	<1	<1	1	2	156	330	24528	14042	*	*	*	*	*	*	*	*	*	*	*

*pMC formula failed to be evaluated in MATLAB using the experimental machine with a seven-hour timeout

increase was exponential for all case studies (cf. Figures 10, 11, and 12). In contrast, the complexity of fPMC formulae increased relatively little for the PL and COM case studies, and exponentially—but at a significantly slower rate than for the Storm and PRISM formulae—for the FX system. For low complexity PL and COM model-property combinations, Storm and PRISM computed pMC formulae with fewer operations than fPMC. This was an unexpected behaviour that we investigated and explained in the analysis of the PL experiments, and for which we proposed a fix (to be implemented in the next version of our fPMC tool).

The MATLAB evaluation time reflects the complexity of the pMC formulae generated by the three model checkers. The evaluation of the fPMC formulae took milliseconds for the FX system, up to 165s for the most complex PL model-property combination, and up to 42s for the COM process. In contrast, the evaluation time for the pMC formulae generated by Storm (for the subset of model-property combinations that the model checker could analyse within 3600s) increased very rapidly with the complexity of these formulae—from milliseconds for the simplest formulae to several hours for the complex ones. The much faster evaluation made possible by fPMC is highly beneficial, as it allows the parametric model checking of systems of far greater complexity than previously possible. Furthermore, self-adaptive systems that use parametric model checking (e.g., [28], [29]) can leverage the simpler fPMC formulae to perform their runtime decision-making much faster, and with significantly lower CPU overheads.

6.2.3 RQ3 (Configurability)

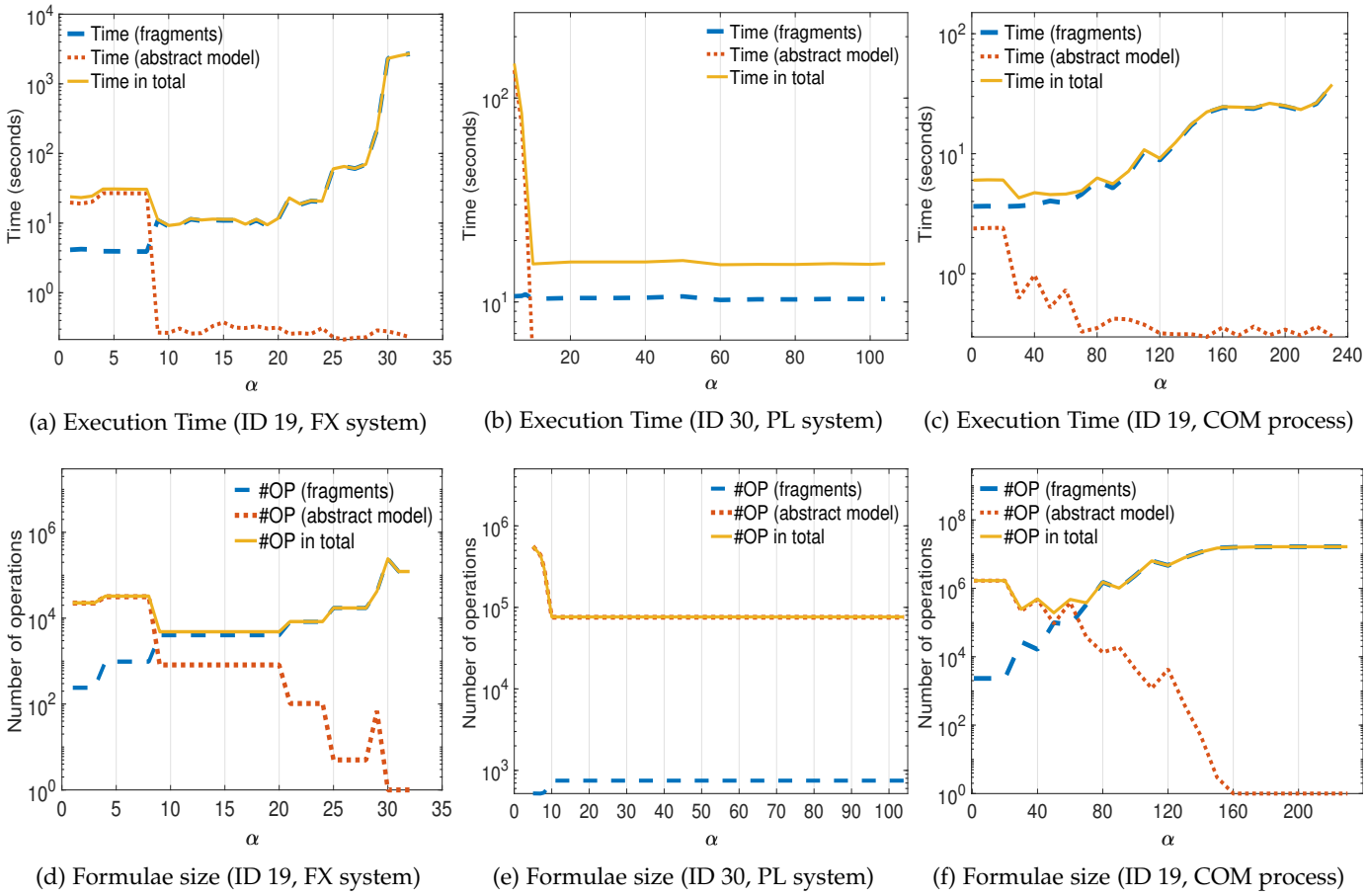
We evaluated the impact of varying the fPMC fragmentation threshold α on: (1) the fPMC execution time, and (2) the number of operations from the resulting algebraic formulae. To that end, we randomly selected one pDTMC model–reachability property combination from each of our three case studies, and we performed its fPMC analysis for all possible values of α (i.e., from 1 to the maximum number of states in the analysed pDTMC). The results of these experiments are summarised in Fig. 13.

FX system. Figures 13a and 13d show the results for the analysis of the reachability property of FX pDTMC variant 19, which has 35 states. fPMC completed the analysis successfully within 60 minutes for $1 \leq \alpha \leq 32$. For small α values ($\alpha < 10$), the time spent checking the abstract model was higher than that required to analyse the fragments. This is because smaller α values often lead to a larger abstract model by limiting the size of each fragment, and therefore increasing the number of fragments. For $\alpha \geq 10$, each fragment was allowed to grow larger, resulting in abstract models with fewer states. As a result, the time spent checking the abstract model decreases (and the time spent checking the fragments grows) with the increase of α. For $\alpha \geq 32$, fPMC could not complete the analysis of its first fragment within the 60-minute timeout. This indicates that a single fragment can become too large and complex (as a result of α being too large) to be handled by the model checker that fPMC uses for the analysis of individual fragments (i.e., Storm). The same pattern can be observed in the size of the obtained closed-form formulae: their numbers of operations increase in the fragments but decrease in the abstract model with the increase of α. A mid-range α value of between 10 and 20 minimises both the fPMC execution time and the complexity of the generated formulae.

PL system. Figures 13b and 13e show the results for the analysis of the reachability property of PL pDTMC variant 30. This pDTMC has 104 states, and fPMC completed its analysis within 60 minutes for all α values between 4 and 104. For $1 \leq \alpha \leq 3$, fPMC timed out in the analysis of the abstract model, which was too complex. For $\alpha > 3$ but still relatively small (i.e., $4 \leq \alpha \leq 8$), the time spent analysing the abstract model is higher than that required to analyse the fragments. As α increases further, a similar trend to that from Figure 13a can be observed: the abstract model analysis time starts to decrease, and the fragment analysis goes up slightly. However, increasing α beyond 10 has no noticeable impact on the analysis time and formula size. This is due to the fact that fPMC partitions this pDTMC into fragments of up to 10 states "naturally", meaning that the forced fragment termination from line 17 of Algorithm 1 is not exercised for $\alpha > 10$. As a consequence, the abstract model remains relatively complex even for large α values, and—irrespective of the value of α—the vast majority of the operations from the fPMC formulae for the PL reachability property come from the abstract model formula.

COM process. Figures 13c and 13f show the results for the analysis of reachability property 19 of the COM pDTMC model. This pDTMC has 234 states, and fPMC completed the analysis successfully for all possible values of α, i.e., $1 \leq \alpha \leq 234$. Similar to the FX and PL experiments, increasing α led to lower analysis times and fewer formula operations for the abstract model, but resulted in higher analysis times and more formula operations for the fragments. As for the FX system, an intermediate α value (of between 30–70 in this case) yields the lowest total analysis time and number of operations.

Discussion. The fragmentation threshold α serves as a soft upper bound for deciding whether fPMC should continue model traversal (adding further states to its stack, cf. line 14 of Algorithm 1) or invoke fragment termination by forcing the currently analysed state to become an output fragment state (cf. line 16 of Algorithm 1). The experiments summarised in Figure 13 suggest that the selection of optimal α values depends on many factors, including pDTMC structure and number of states. Trying all the possible values



(a) Execution Time (ID 19, FX system)
(b) Execution Time (ID 30, PL system)
(c) Execution Time (ID 19, COM process)
(d) Formulae size (ID 19, FX system)
(e) Formulae size (ID 30, PL system)
(f) Formulae size (ID 19, COM process)

Fig. 13: fPMC execution time for (a) the reachability property of FX pDTMC variant 19, (b) the reachability property of PL variant 30, and (c) reachability property 19 of the COM pDTMC; and fPMC formula complexity (i.e., number of operations) for the same pDTMC-property combinations of (d) the FX system, (e) the PL system, and (f) the COM process.

of α as what we did here is impractical. Thus, the determination of optimal α values (other than by exhaustive search) remains an open research question. Nevertheless, our experimental results suggest useful rules of thumb for the selection of suitable α values. First, adopting a default α value between 10–30 is likely to produce good results (by guiding fPMC to create abstract models with at least one order of magnitude fewer states than the original pDTMC). Second, increasing α may help if the time to analyse the abstract model is too high (or the formula produced by this analysis is too complex); conversely, decreasing α may help if the time to analyse one of the fragments is too high (or the formulae from the fragment analyses are too complex). Used in conjunction with hill climbing [53], the latter thumb rule could enable the optimisation of the threshold α, e.g., to ensure that fPMC yields formulae with the lowest number of operations possible.

7 Threats to Validity

Construct validity threats. These threats may be caused by over-simplifications and invalid assumptions made when devising the evaluation experiments. To avoid them, we carried out the evaluation of fPMC by using case studies based on two software systems and a communication process that are freely available from other published software engineering projects. Furthermore, the pDTMCs and properties evaluated in our paper were also used in related research [18], [20], [33], [36], [41].

Internal validity threats. To avoid these threats—which could have introduced bias in the identification of cause-effect relationships in our experiments—we evaluated fPMC by answering three independent research questions (cf. Section 6.1). To further mitigate the potential bias, the evaluation results were compared against the leading probabilistic model checkers Storm [24] and PRISM [45], and the correctness of each result produced by fPMC was individually checked using the approach described in Section 6.1. Finally, we published the source code and data from our experiments online [1], enabling other researchers to reproduce and verify our results.

External validity threats. These threats could affect the generalisability of our findings. As summarised in Table 3, we mitigated them in our evaluation by applying fPMC to three types of systems (i.e., service-based systems [3], [56], software product lines [5], [21], and communication processes [25], [26]) taken from different application domains. Moreover, we used multiple model variants and properties, allowing us to test our approach on a wide range of pDTMC structures and sizes, and thus to show that

fPMC provides consistently better performance in terms of faster computation time, fewer arithmetic operations in the derived algebraic formulae, and faster evaluation of these formulae than the model checkers PRISM and Storm for complex models. Finally, we eased the use of fPMC in practice by providing tool support for our approach. However, additional experiments are needed to confirm that fPMC is applicable to a wider range of pDTMC models and to other application domains.

8 Related Work

Parametric model checking was firstly introduced by Daws [22] less than two decades ago. The technique enables the analysis of a DTMC when some or all of its transition probabilities are specified as rational functions over the parameters of the modelled system. pMC produces an algebraic formula for each analysed property. Such formulae are used in the design and verification of software systems, e.g., to compare alternative system designs [35], [36], to dynamically evolve the configuration of self-adaptive software [16], [23], [28], [29], and for system parameter analysis and synthesis [12], [13], [30], [37], [52]. Despite this wide adoption, the computationally intensive nature of pMC limits the scalability and applicability of the multiple software engineering methods that rely on it.

To the best of our knowledge, the research aimed at improving the efficiency of pMC is limited to the approaches proposed in [7], [18], [32], [39], [44]. For the purpose of comparing these approaches to fPMC, we consider them organised into two classes: those which (like Daws' original pMC technique) operate on the entire pDMTC under analysis [7], [32], [39], and those which (like our fPMC approach) operate by partitioning this pDTMC into components that are then analysed individually [18], [44]. We term these classes of approaches standard pMC and compositional pMC, respectively.

Standard pMC approaches. The first class of pMC approaches are complementary to our work. Any of them can be used in conjunction with fPMC, to ensure that the individual probabilistic model checking of the fPMC fragments and fPMC abstract model is carried out efficiently. We summarise these approaches below.

In comparison to Daws' initial pMC technique [22], the technique proposed by Hahn et al. [39] yields a significant improvement in pMC performance. Both pMC approaches derive an algebraic formula for the probability of reaching a set of parametric Markov chain states specified in a PCTL path formula. However, instead of computing a regular expression by exploiting the pDTMC structure as in [22], the pMC technique from [39] produces a rational expression and leverages symmetry and "cancellation" properties of rational formulae to simplify this expression. The cancellation involves computing the greatest common divisor (GCD) of the denominator and numerator polynomials of the rational pMC formula, and using this GCD polynomial to simplify the formula. The model checkers PARAM [38] and PRISM [45] implement this technique.

According to our classification, Jansen et al. [44] propose a hybrid pMC approach, and we discuss its pMC "components" separately. The standard pMC component of [44] consists of sophisticated partial polynomial factorisations that support the efficient simplification of large pMC rational expressions. This pMC approach is implemented by the parametric model checker Storm [24].

More recently, Baier et al. [7] have introduced a new technique for obtaining simplified pMC formulae. This technique avoids the computationally expensive calculation of GCD polynomials (which the pMC approaches from [39], [44] rely on) by leveraging fraction-free Gaussian elimination, which is an existing efficient method for solving systems of parametric linear equations. This method is implemented in the Storm model checker.

A different type of approach to speeding up pMC is proposed by Gainer et al. [32]. This approach involves the stepwise elimination of the states of the analysed pDTMC, through a process that resembles the mapping of finite automata to regular expressions. The outcome of this elimination is a directed acyclic graph encoding of the pMC result instead of the usual rational formula produced by other pMC techniques. The authors' evaluation of the approach (implemented in their ePMC/IscasMC model checker [40]) shows that it can outperform the formula-based pMC engine used by PRISM by up to two orders of magnitude.

Compositional pMC approaches. These approaches operate in a similar way to fPMC. As such, we will compare each of them to our work.

The pMC approach devised by Jansen et al. [44] improves the efficiency of parametric model checking by decomposing the state transition graph induced by the pDTMC under analysis into strongly connected components (SCCs). pMC expressions are then computed independently for each SCC, and then combined to obtain the final pMC output in the form of a single rational formula over the parameters of the system modelled by the pDTMC. Because it is predetermined by the SCCs of the analysed pDTMC, this decomposition (which is implemented by the parametric model checker Storm) is very rigid. In particular, it may produce SCCs that are too large to be analysed efficiently, and that cannot be further decomposed. In contrast, the fPMC fragmentation of a pDTMC is much more flexible. The analysed pDTMC is partitioned into fragments whose size is guided by the fragmentation threshold α. These fragments can include one or more small SCCs. Most importantly, fPMC can split any SCCs that are too large to be analysed individually into multiple fragments. The experimental results from Section 6 provide ample evidence about the benefits of this flexible pDTMC partitioning. Furthermore, the fPMC fragmentation can, in theory, be applied repeatedly, e.g., to partition a large fragment into sub-fragments (although this still has to be confirmed experimentally).

The fPMC theoretical foundation comprises two complementary parts. The first part, which we introduced in [18], defines the method for using pDTMC fragments to speed up parametric model checking—but does not provide any method for partitioning a pDTMC into fragments. Therefore, the solution from [18] can only be applied when its users are able to exploit domain knowledge in order to manually specify the fragments of the analysed pDTMC. This limitation represents a significant barrier for the practical adoption of fragmentation-based pMC. The second part

of the fPMC theoretical foundation, which is introduced in this paper and complements our results from [18], removes this barrier by providing a tool-supported method for the automated fragmentation of pDTMCs.

9 Conclusion

We presented fPMC, a tool-supported technique for software performability analysis through compositional parametric model checking. fPMC supports the efficient analysis of reachability, unbounded until and reachability reward properties of parametric discrete-time Markov chains by automatically partitioning these models into fragments that can be analysed independently. The results of these analyses are then combined into a system of closed-form algebraic expressions that represent the solution of the initial parametric model checking problem.

To evaluate fPMC, we used it to analyse 28 PCTL properties of 62 pDTMC variants modelling three types of software systems (i.e., service-based systems, software product lines, and middleware) from different application domains. The experimental results show that fPMC can analyse pDTMCs with over 10–20 parameters much faster than previous pMC techniques, and—in many cases—when these techniques cannot complete their analyses within 60 minutes on a standard computer. Furthermore, our evaluation showed that the algebraic expressions generated by fPMC for such models comprise considerably fewer operations and are much faster to evaluate than those produced by previous pMC techniques.

In future work, we will explore several opportunities for extending the applicability and efficiency of fPMC. First, we will examine the possibility to apply fPMC fragmentation repeatedly, to partition pDTMC fragments that may be too large or too complex to analyse as a whole into sub-fragments. This opportunity, which is unique to our compositional pMC technique, has the potential to support the analysis of complex pDTMCs that cannot be handled by any existing model checkers.

Second, we aim to enhance the fPMC fragmentation algorithm with the ability to generate close-to-optimal fragments, i.e., fragments that: (i) are non-trivial in terms of structure, size and number of parameters; (ii) can be analysed efficiently; and (iii) produce pMC expressions of acceptable complexity. Options for obtaining such fragments include: **1)** adapting the fPMC fragmentation threshold to the characteristics of the fragment under construction; **2)** employing methods such as satisfiability modulo theories or mixed-integer linear programming to search for "good" fragments based on well-defined fragment constraints; **3)** implementing an improved policy for early termination of a fragment (e.g., based on the complexity of the fragment under construction rather its number of states); and **4)** using recent advances on partial exploration for Markov systems [6], [48] to instantiate and analyse the model for a set of parameter valuations, thereby guiding the fragmentation process with these results.

Last but not least, we plan to improve our fPMC tool. In particular, we will implement the simplification identified in Section 6.2.2. To that end, we will update the fPMC tool to ensure that abstract model parameters associated with constant-valued fragment reachability properties are replaced with the actual values of those properties. Additionally, we will explore options for selecting an effective pMC technique (e.g., standard or compositional) for the analysis of a given pDTMC, pDTMC fragment or pDTMC strongly connected component, paving the way for the development of a highly efficient hybrid parametric model checker that uses the available pMC techniques together.

Acknowledgements

This project has received funding from the UKRI project EP/V026747/1 'Trustworthy Autonomous Systems Node in Resilience', the ORCA-Hub PRF project 'COVE', and the Assuring Autonomy International Programme. The authors are grateful to the research groups who have developed the EPMC/IscasMC, PARAM, PRISM and Storm probabilistic and parameteric model checkers: our work would not have been possible without the significant theoretical advances and tools introduced by these research groups.

CRediT authorship contribution statement

Xinwei Fang: Conceptualization, Investigation, Methodology, Software, Validation, Visualization, Writing original draft. **Radu Calinescu:** Conceptualization, Formal Analysis, Funding Acquisition, Investigation, Methodology, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. **Simos Gerasimou:** Conceptualization, Funding Acquisition, Investigation, Methodology, Software, Supervision, Visualization, Writing – original draft. **Faisal Alhwikem:** Methodology, Software.

References

[1] "fPMC project website," 2023. [Online]. Available: https://www.cs.york.ac.uk/tasp/fPMC/

[2] N. Alasmari, R. Calinescu, C. Paterson, and R. Mirandola, "Quantitative verification with adaptive uncertainty reduction," *Journal of Systems and Software*, vol. 188, no. 111275, 2022.

[3] D. Ameller, M. Galster, P. Avgeriou, and X. Franch, "A survey on quality attributes in service-based systems," *Software quality journal*, vol. 24, pp. 271–299, 2016.

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

[5] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-oriented software product lines*. Springer, 2016.

[6] P. Ashok, Y. Butkova, H. Hermanns, and J. Křetínskỳ, "Continuous-time Markov decisions based on partial exploration," in *Automated Technology for Verification and Analysis: 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*. Springer, 2018, pp. 317–334.

[7] C. Baier, C. Hensel, L. Hutschenreiter, S. Junges, J. P. Katoen, and J. Klein, "Parametric Markov chains: PCTL complexity and fraction-free Gaussian elimination," *Information and Computation*, vol. 272, no. 104504, 2020.

[8] S. Balsamo, V. De Nitto Personè, and P. Inverardi, "A review on queueing network models with finite capacity queues for software architectures performance prediction," *Performance Evaluation*, vol. 51, pp. 269–288, 2003.

[9] S. Balsamo, P. G. Harrison, and A. Marin, "Methodological construction of product-form stochastic Petri nets for performance evaluation," *Journal of Systems and Software*, vol. 85, pp. 1520–1539, 2012.

[10] A. Bianco and L. de Alfaro, "Model checking of probabilistic and nondeterministic systems," in *Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*. Springer, 1995, pp. 499–513.

[11] R. Calinescu, "Emerging techniques for the engineering of self-adaptive high-integrity software," *Assurances for Self-Adaptive Systems*, pp. 297–310, 2013.

[12] R. Calinescu, M. Autili, J. Cámara, A. Di Marco, S. Gerasimou, P. Inverardi, A. Perucci, N. Jansen, J. P. Katoen, M. Kwiatkowska, O. J. Mengshoel, R. Spalazzese, and M. Tivoli, "Synthesis and verification of self-aware computing systems," *Self-Aware Computing Systems*, pp. 337–373, 2017.

[13] R. Calinescu, M. Češka, S. Gerasimou, M. Kwiatkowska, and N. Paoletti, "Efficient synthesis of robust models for stochastic systems," *Journal of Systems and Software*, vol. 143, pp. 140–158, 2018.

[14] 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.

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

[16] R. Calinescu and M. Kwiatkowska, "CADS*: Computer-aided development of self-* systems," in *International Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, 2009, pp. 421–424.

[17] R. Calinescu, R. Mirandola, D. Perez-Palacin, and D. Weyns, "Understanding uncertainty in self-adaptive systems," in *IEEE International Conference on Autonomic Computing and Self-Organizing Systems*, 2020, pp. 242–251.

[18] R. Calinescu, C. Paterson, and K. Johnson, "Efficient parametric model checking using domain knowledge," *IEEE Transactions on Software Engineering*, vol. 47, pp. 1114–1133, 2021.

[19] F. Ciesinski and M. Größer, "On probabilistic computation tree logic," *Validation of Stochastic Systems*, vol. 2925, pp. 147–188, 2004.

[20] 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 *International Conference on Software Engineering (ICSE)*, 2010, pp. 335–344.

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

[22] 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.

[23] R. De Lemos, D. Garlan, C. Ghezzi, H. Giese, J. Andersson, M. Litoiu, B. Schmerl, D. Weyns, L. Baresi, N. Bencomo *et al.*, "Software engineering for self-adaptive systems: Research challenges in the provision of assurances," *Software Engineering for Self-Adaptive Systems III. Assurances*, vol. 9640, pp. 3–30, 2018.

[24] C. Dehnert, S. Junges, J. P. Katoen, and M. Volk, "A storm is coming: A modern probabilistic model checker," in *International Conference on Computer Aided Verification (CAV)*, R. Majumdar and V. Kunčak, Eds., 2017, pp. 592–600.

[25] F. Dressler and O. B. Akan, "Bio-inspired networking: from theory to practice," *IEEE Communications Magazine*, vol. 48, pp. 176–183, 2010.

[26] ——, "A survey on bio-inspired networking," *Computer Networks*, vol. 54, pp. 881–900, 2010.

[27] X. Fang, R. Calinescu, S. Gerasimou, and F. Alhwikem, "Fast parametric model checking through model fragmentation," in *International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 835–846.

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

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

[30] 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, pp. 75–99, 2015.

[31] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi, "Enhanced modelling and solution of layered queueing networks," *IEEE Transactions on Software Engineering*, vol. 35, pp. 148–161, 2009.

[32] P. Gainer, E. M. Hahn, and S. Schewe, "Accelerated model checking of parametric markov chains," in *International Symposium on Automated Technology for Verification and Analysis (ATVA)*. Springer, 2018, pp. 300–316.

[33] S. Gerasimou, G. Tamburrelli, and R. Calinescu, "Search-based synthesis of probabilistic models for quality-of-service software engineering," in *International Conference on Automated Software Engineering (ASE)*, 2015, pp. 319–330.

[34] S. Gerasimou, R. Calinescu, and G. Tamburrelli, "Synthesis of probabilistic models for quality-of-service software engineering," *Automated Software Engineering*, vol. 25, pp. 785–831, 2018.

[35] C. Ghezzi and A. M. Sharifloo, "Verifying non-functional properties of software product lines: Towards an efficient approach using parametric model checking," in *International Software Product Line Conference (SPLC)*. IEEE, 2011, pp. 170–174.

[36] ——, "Model-based verification of quantitative non-functional properties for software product lines," *Information and Software Technology*, vol. 55, pp. 508–524, 2013.

[37] E. M. Hahn, T. Han, and L. Zhang, "Synthesis for PCTL in parametric markov decision processes," in *NASA formal methods symposium (NFM)*. Springer, 2011, pp. 146–161.

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

[39] 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.

[40] E. M. Hahn, Y. Li, S. Schewe, A. Turrini, and L. Zhang, "ISCASMC: A web-based probabilistic model checker," in *Formal Methods (FM)*, 2014, pp. 312–317.

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

[42] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal Aspects of Computing*, vol. 6, pp. 512–535, 1994.

[43] S. M. Hezavehi, D. Weyns, P. Avgeriou, R. Calinescu, R. Mirandola, and D. Perez-Palacin, "Uncertainty in self-adaptive systems: a research community perspective," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 15, pp. 1–36, 2021.

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

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

[46] A. Lanna, T. Castro, V. Alves, G. Rodrigues, P.-Y. Schobbens, and S. Apel, "Feature-family-based reliability analysis of software product lines," *Information and Software Technology*, vol. 94, pp. 59–81, 2018.

[47] C. Lindemann, "Performance modelling with deterministic and stochastic Petri nets," *Performance Evaluation Review*, vol. 26, no. 2, 1998.

[48] T. Meggendorfer and J. Křetínskỳ, "Of cores: a partial-exploration framework for Markov decision processes," *Logical Methods in Computer Science*, vol. 16, 2020.

[49] D. F. Mendonça, G. N. Rodrigues, R. Ali, V. Alves, and L. Baresi, "GODA: A goal-oriented requirements engineering framework for runtime dependability analysis," *Information and Software Technology*, vol. 80, pp. 245–264, 2016.

[50] C. Paterson and R. Calinescu, "Observation-enhanced QoS analysis of component-based systems," *IEEE Transactions on Software Engineering*, vol. 46, pp. 526–548, 2018.

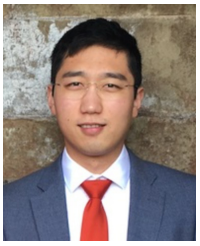
[51] D. Perez-Palacin and J. Merseguer, "Performance evaluation of self-reconfigurable service-oriented software with stochastic Petri nets," *Electronic Notes in Theoretical Computer Science*, vol. 261, pp. 181–201, 2010.

[52] T. Quatmann, C. Dehnert, N. Jansen, S. Junges, and J.-P. Katoen, "Parameter synthesis for Markov models: faster than ever," in *Automated Technology for Verification and Analysis: 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings 14*. Springer, 2016, pp. 50–67.

[53] B. Selman and C. P. Gomes, "Hill-climbing search," *Encyclopedia of cognitive science*, vol. 81, p. 82, 2006.

[54] G. F. Solano, R. D. Caldas, G. N. Rodrigues, T. Vogel, and P. Pelliccione, "Taming uncertainty in the assurance process of self-adaptive systems: a goal-oriented approach," in *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2019, pp. 89–99.

[55] G. Su, D. S. Rosenblum, and G. Tamburrelli, "Reliability of run-time quality-of-service evaluation using parametric model checking," in *International Conference on Software Engineering (ICSE)*, 2016, p. 73?84.
[56] 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, pp. 502–516, 2010.



Xinwei Fang is a lecturer in the Department of Computer Science at the University of York, UK. His research focuses on the design and development of trustworthy autonomous systems by understanding, detecting, and mitigating uncertainties that may arise at various stages of these systems through methods such as machine learning, model checking, and statistical analysis.

Radu Calinescu is Professor of Computer Science at the University of York, UK. His main research interests are in formal methods for self-adaptive software, probabilistic and parametric model checking, and sociotechnical resilience and safety of autonomous and AI systems. He is an active promoter of formal methods at runtime as a way to improve the resilience and trustworthiness of self-adaptive, autonomous and AI systems and processes.

Simos Gerasimou is Associate Professor at the Department of Computer Science at the University of York, UK. His research focuses on developing rigorous tool-supported approaches using model-based analysis, simulation and formal verification to support the engineering of trustworthy software for autonomous systems. His areas of expertise include safe AI, software engineering for AI and AI for software engineering, model-driven adaptive and autonomous systems, and assurances for autonomous systems.

Faisal Alwhiekm is an assistant professor at the Department of Computer Science, College of Computer, Qassim University, Buraydah, Saudi Arabia. He holds a PhD in Computer Science from the University of York, UK, and his main research focuses on V&V and using model-based analysis and design to create reliable and dependable software. His expertise includes model-driven engineering, self-adaptive systems, and formal software verification.