Towards a Fractionation-based Verification: Application on SysML Activity Diagrams

Samir Ouchani

Towards a Fractionation-based Verification: Application on SysML Activity Diagrams

Conference Paper · December 2018

DOI: 10.1145/3297280.3297480

1 author:

Samir Ouchani
Groupe Cesi
100 PUBLICATIONS **667** CITATIONS

SEE PROFILE

Towards a Fractionation-based Verification: Application on SysML Activity Diagrams

Samir Ouchani
LINEACT, Laboratoire d'Innovation Numérique, CESI Research
École d'Ingénieur en Informatique, CESI eXia
13545 Aix-en-Provence, France
souchani@cesi.fr

ABSTRACT

This work contributes to reduce the cost of the verification process of model-based systems, especially the ones designed as SysML activity diagrams, by relying on the concept of diagram calls defined in SysML standard. By exploiting the diagram calls concept, we propose a mechanism to transform a given diagram into its equivalent fractal (hierarchical) form. Further, we present an abstraction algorithm to reduce the size of the obtained SysML activity diagram by ignoring the irrelevant behaviors and by merging similar artifacts. To do verification, the abstracted diagram is transformed automatically into PRISM source code to check the system's requirements that are specified in the probabilistic temporal logic, PCTL. The soundness of the provided framework is presented for each step, and, its practical effectiveness is demonstrated on diagrams obtained by reversing the open source of OpenSAF middle-ware.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

Fractionation, SysML Activity Diagrams, Abstraction, Reduction, Probabilistic Verification.

ACM Reference Format:
Samir Ouchani. 2019. Towards a Fractionation-based Verification: Application on SysML Activity Diagrams. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3297280.3297480

1 INTRODUCTION

Motivations. In modern system development, various techniques have been demonstrated for modeling systems and software such as UML and SysML; also for verifying them to ensure their security, correctness, and safety before deployment, e.g. automatic verification, mechanization, and static/dynamic analysis. More particularly, qualitative model checking [1, 4] is the most widely used technique for the assessment of UML and SysML behavioral diagrams [12, 13].

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297480

Model checking is a automatic formal verification technique for finite state concurrent systems that checks the systems' requirements specified as temporal logic expressions. In addition to qualitative model checking, quantitative verification techniques based on probabilistic model checkers [1] have recently gained popularity by evaluating the satisfiability of a given property on systems that inherently exhibit probabilistic behavior. From a modeling perspective side, SysML activity diagrams are behavioral models that offer the possibility to call other diagrams of different kind such as state machine and sequence diagrams, and also they allow the specification of probabilistic behavior [11].

Background. Despite its wide use, model checking is generally a resource-intensive process that requires a large amount of memory and time processing. This is due to the fact that the systems' state space may grow exponentially with the size of the composed concurrent processes combined with the number of variables. To overcome this issue, various techniques have been explored [1, 4] for qualitative model checking and leveraged later to cover the probabilistic case. Among these techniques, several solutions aim at optimizing the employed model checking algorithms by introducing symbolic data structures based on binary decision diagrams, while others target to reduce the model as well as the requirement under verification.

Abstraction is one of the most relevant techniques addressing the state explosion problem [1, 4]. It can be seen as a mapping from a concrete model into a more abstract one that encapsulates the systems' behavior while being of a reduced size. The intuition behind this transformation is to be able to check a property against an abstract model, and, then to infer safely equivalent results on the concrete model. Abstraction [2] can be managed through, *merging* states and transitions where similar behaviors are encapsulated on a unified one, *abstracting* data where a set/range of values is represented by a single symbolic value. Abstraction can also be done via *restriction* by forbidding some behavior of the system; or by an *observer automaton* that restricts systems behaviors to those acceptable by an automaton that observes the system from outside. Among well-known deployed abstraction algorithms are: cone of influence, bi-simulation minimization, symmetry and partial order reductions [4]. Thence, it is of a major importance to reduce the complexity of a diagram as any approach translating the concrete UML/SysML diagrams into the model checker input language would be limited by the tool's abstraction mechanism, if exists. Moreover, abstracting the semantic model instead of the concrete diagram can be costly since the size of the semantic model is usually greater than that of the diagram itself. Consequently, this work explores the

second category by abstracting systems modeled in SysML activity diagrams prior verification.

Contributions. To overcome the verification and modeling limitations, we propose to reduce the cost of the verification process of a system, modeled as SysML activity diagrams, by fractioning the initial diagram to ease the use of other abstraction, reduction, and refinement operations. The overall framework developing the proposed solution is depicted in Fig. 1.

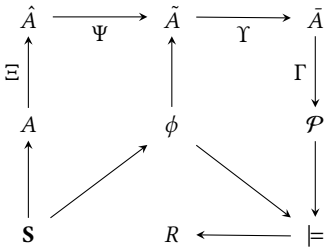

Figure 1: Fractionation-based Verification.

Starting from the specification of a given system **S**, the framework considers initially the SysML activity diagram A that models properly **S** and its requirement specification ϕ that is expressed as Probabilistic Computation Tree Logic (PCTL) to be guaranteed on A. Then, the function Ξ fractionates A to be $\hat{A}$ ($\hat{A}$ is the hierarchical representation of A) that is abstracted later using the function Ψ with respect to ϕ, and the result is a new diagram $\tilde{A}$. To reduce more $\tilde{A}$, the reduction function Υ is applied to return a new compacted diagram $\bar{A}$. Further, to verify the satisfiability of ϕ, the transformation function Γ translates $\bar{A}$ into $\mathcal{P}$, a PRISM source code. Finally, we demonstrate the practical application of the proposed framework using a case study that would be otherwise difficult to verify. Besides, we observe a significant reduction in the state space by an important rate, which makes probabilistic model checking helpful.

Outlines. The remainder of this paper is organized as follows. Section 2 discusses the related work and compares it with the proposed contribution. Section 3 presents the needed preliminaries for this work. Then, the proposed Fractionation-based framework is detailed in Section 4 and its implementation with the experimental results are shown in Section 5. Finally, Section 6 concludes this paper and provides hints on the possible future works.

2 RELATED WORK

In the literature, few works examine the abstraction and the analysis of SysML diagrams before verification while the majority rely on the implemented algorithms within the plugged verification tools. This work is compared to the existing initiatives that deal with the verification of SysML and UML diagrams, and probabilistic systems.

Ouchani et al. [13, 14] proposed an abstraction framework that reduces a SysML activity diagram with respect to a PCTL property through cone of influence. The obtained diagram is mapped to PRISM for verification. With respect to [13, 14] the current framework covers more SysML artifacts, and accelerate the verification process by fractioning diagrams which help to abstract easily the diagrams. Also, it is helpful for the refinement and the correction processes when needed. Further, this framework is applied on a concrete use case.

Westphal [17] proposed a reduction UML framework explicit to symmetric UML diagrams that are modeled with Rhapsody where the requirements are expressed as live sequence charts. Westphal's approach looks to reduce the charts of the requirement and to abstract data specified in the object reference model. The reduction exploits the strong assumption of symmetric diagrams by applying the symmetric reduction whereas data abstraction interprets data in a symbolic domain. Compared to our work, Westphal's approach [17] is applied only on symmetric models and the verification process when using Rhapsody is not well detailed.

Prashanth and Shet [15] proposed an abstraction technique specific to state chart diagrams. Initially, the approach computes a set of relevant events in the diagram related to a safety property expressed in LTL. The result is a new reduced diagram. However, the approach does not show the representation of the abstracted statechart either its soundness. Especially, the state space is changing for each safety property which means the specification of the diagrams is also changing accordingly in the used verification tool. Further, the approach does not cover the probabilistic behavior and the deterministic decisions.

Daoxi et al. [5] proposed to reduce a single UML state machine to preserve in the original model only the atomic propositions of a LTL property. Then, they translate the abstracted model into a Promela source code to be verified using Spin[1]. They adopt the same mechanism as [15] without showing the abstracted state machine. Further, both the abstraction and the translation mechanisms are not automatic. Compared to our proposed contribution, Daoxi et al. approach [5] is limited to a set of artifacts especially excluding the probabilistic decisions.

Del Mar Gallardo et al. [18] proposed an abstraction framework that abstracts data and events in state chart diagrams. Data abstraction segments the data domain by replacing data values with symbolic ones whereas the events abstraction customizes a selected set of events to be represented by a single event (called the mother state). In fact, the considered state chart diagram has initially a hierarchical form where the mother state encompasses already a set of events that makes the events abstraction as a natural step without automation. Further, the considered diagrams do not support probability, and the segmentation of the data domain is given to the users choice without guidance.

Holtzen et al. [9] generalize the abstraction of non-deterministic program for the probabilistic ones by explicitly quantifying the non-deterministic choices. They upgrade the main definitions and properties of the abstraction techniques to the probabilistic space. In general the main concepts can be leveraged for both domains, but the main difficulty is when abstracting a probabilistic choice, the authors do not show how to estimate the distributions and guards of a choice.

Brau et al. [3] combine heterogeneous modeling, abstraction and analysis techniques to be applied on an avionic case study. They use AADL[2] and CPAL[3] for modeling, the OCARINA toolchain to

[1] http://spinroot.com, Spin is a formal verification tool for multi-threaded software applications.
[2] The Architecture Analysis and Design Language (AADL) is an industry standard modeling language.
[3] https://www.designcps.com/, The Cyber-Physical Action Language (CPAL) is an implementation-oriented language meant to model and program Cyber-Physical Systems (CPS)

parse AADL models, and CPAL2X to extract data from CPAL source files. For analysis they use Alloy[4] that relies on SAT solver. The main contribution of [3] is applying the chain tool on a simple avionic case study, but without showing the way of specifying the assumptions and how they are guaranteed when using Alloy.

Kloetzer and Mahulea [10] abstract a robot environment by proposing a trajectory plan to monitor robots' motion in a dynamic environment following an exponential probabilistic function. Motions are built up on temporal logic by incorporating probabilistic assumptions on region and a robot which eases the abstraction, and to control better the movement in a region. The abstracted transition system of a robot movement has been defined without showing how it is produced based on the observed trajectory plan.

With respect to the studied works, the majority of them rely on the abstraction techniques built with the verification tools without proving the soundness the applied abstraction approach. Besides, our proposed framework looks prominent since it reduces the size of diagrams by a considerable rate and keeps a backward path for reconstruction when needed. Further, it develops an automatic abstraction and verification mechanism. Furthermore, this framework is scalable and can exploit the existing reduction techniques built within the verification tool, if needed.

3 PRELIMINARIES

We present in this section the main ingredients needed for the proposed framework, mainly, SysML activity diagrams and the PCTL temporal logic.

3.1 SysML Activity Diagrams

SysML activity diagrams are a graph-based representation to design system's behaviors at various levels of abstraction [8] where vertices are nodes that control flows in edges. Figure 2 illustrates the main graphical artifacts of SysML activity diagrams that can be decomposed into two categories of constructs: activity nodes and activity edges. Hence, in general the number of edges is always greater than the number of nodes.

The nodes are categorized into three types: activity invocation, object and control nodes. Activity invocation includes send and receive signals, action, and call behavior. Activity control nodes are initial, flow final, activity final, decision, merge, fork, join nodes, and the interrupt region. Activity edges are of two types: control flow and object flow. Control flow edges are used to show the execution path through the activity diagram and connect activity nodes. Object flow edges are used to show the flow of data between activity nodes. Concurrency and synchronization are modeled using forks and joins, whereas, branching is modeled using decision and merge nodes. While a decision node specifies a choice between different possible paths based on the evaluation of a guard condition (and/or a probability distribution), a fork node indicates the beginning of multiple parallel control threads. Moreover, a merge node specifies a point from where different incoming control paths follow the same path, whereas a join node allows multiple parallel control threads to synchronize and rejoin. Further, an interrupt region models the interruption when errors occur with a precise probability.

[4]http://alloytools.org, Alloy is a language for describing structures and a tool for exploring them.

To control the flow, the activation of a node depends only on the termination of its preceded node and the guard satisfaction of their related edge. Except for the initial node, which is activated when the main diagram is invoked.

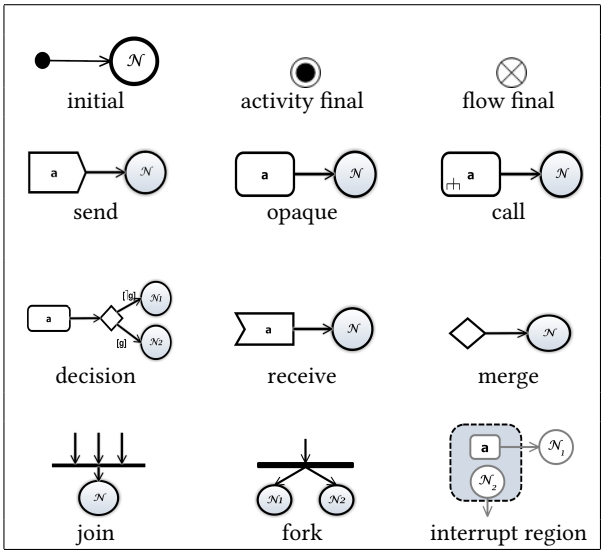

Figure 2: SysML Activity Diagram Artifacts.

When a call behavior action satisfies the input flow prerequisites, it consumes its input tokens and invokes the specified behavior. The values in the input tokens are made available to the invoked behavior as argument values. When the behavior's execution is terminated, tokens are offered on all outgoing control edges. Further, an activity diagram can invoke a new activity behavior in a decision node to evaluate guards or to measure a probability to take a decision. Furthermore, the behavior invocation supports both the synchronous and asynchronous calls. In the asynchronous case, the execution of the invoked behavior proceeds without any further dependency on the execution of the activity containing the invoking action. And in the synchronous case, the execution of the calling action is blocked until it receives a reply token from the invoked diagram.

We denote by a, a_i, a_t, and a_f opaque (atomic), initial, flow final, and activity final nodes, respectively; $\mathsf{a} \blacktriangleright v$ and $\mathsf{a} \blacktriangleleft v$ to send and receive a value v by the node a, $\mathsf{a} \uparrow A$ to call an activity A by a, $\mathsf{a}\diamond_{g,p}$ a guarded, probabilistic, or conditional probabilistic decision in a, $\mathsf{a}\blacklozenge$ a merge node, $\mathsf{a}|$ a join, $\mathsf{a}\|$ a fork, $\mathsf{a}\ddagger_p$ an interrupt region with a probability p, and $\rightarrowtail_g$ the activity edge with the guard g[5]. Definition 3.1 expresses formally SysML activity diagrams by assuming one initial node and one final node only for each diagram, and $Dist(\mathcal{N})$ denotes a convex probability distributions over the set of activity nodes $\mathcal{N}$.

Definition 3.1. An SysML activity diagram is a tuple $A = \langle \mathcal{N}, \mathcal{E}, \mathcal{G}, Grd, Prob \rangle$, where:

- $\mathcal{N}$ is a finite set of activity nodes such as a_i and a_f denote the initial and the final nodes, respectively;
- $\mathcal{E}$ is a finite set of activity edges,

[5]In this paper, we consider g a propositional logic formula that can be extended later.

- $\mathcal{G}$ is the set of guards,
- $Grd : \mathcal{E} \mapsto \mathcal{G}$ is a partial function that returns a guard for an edge, and
- $Prob : \mathcal{N} \mapsto Dist(\mathcal{N})$ is a partial probabilistic function that assigns for each node a convex discrete probability distribution $\mu \in Dist(\mathcal{N})$ over its output transitions.

3.2 PCTL Specification

We comment here what properties can be of relevance and how to express them in such a way that they can specify very well the requirements needed to check the functionality of the system under verification. A formalism that is able to express all the factors that diagrams describe, paths of actions, propositions on state variables, probabilities of occurrence of one or a sequence of actions can be PCTL [6]. Formulas ϕ in such a logic are generated by the following BNF grammar.

$$\phi \quad ::= \quad \top \mid ap \mid \phi \wedge \phi \mid \neg\phi \mid \mathrm{P}_{\bowtie p}[\psi]$$
$$\psi \quad ::= \quad \mathrm{X}\phi \mid \phi \mathrm{U} \phi \mid \phi \mathrm{U}^{\leq k} \phi$$

Here, $k \in \mathbb{N}$, $p \in [0, 1]$, and $\bowtie \in \{<, \leq, >, \geq\}$. A state formula can be "ap", an atomic proposition, or any propositional logic expression built from "ap". $\mathrm{P}_{\bowtie p}[\psi]$, called *probabilistic path predicate*, returns true if the probability to satisfy the *path formula* ψ is $\bowtie p$. A path formula is built from the typical temporal operators *next* (X), *until* (U), and *bounded until* ($\mathrm{U}^{\leq k}$). Other logic operators can be derived from the basic ones, like *eventually* and *generally* that are denoted respectively by F and G. The semantics of these operators are given by the following equivalence relations.

- $\bot \equiv \neg\top$, $\phi \vee \phi' \equiv \neg(\neg\phi \wedge \neg\phi')$, $\phi \rightarrow \phi' \equiv \neg\phi \vee \phi'$, and $\phi \leftrightarrow \phi' \equiv \phi \rightarrow \phi' \ \wedge \ \phi' \rightarrow \phi$.
- $F\phi \equiv \ \top \ U \ \phi$, $F^{\leq k}\phi \equiv \ \top \ U^{\leq k} \ \phi$, $G\phi \equiv \ \neg(F\neg\phi)$, and $G^{\leq k}\phi \equiv \ \neg(F^{\leq k}\neg\phi)$ where $k \in \mathbb{N}$.
- $\mathrm{P}_{\geq p}[G\phi] \equiv \mathrm{P}_{\leq 1-p}[F\neg\phi]$.

4 FRACTIONATION-BASED VERIFICATION FRAMEWORK

We detail the framework flow depicted in Fig. 1 by presenting first the fractionation algorithm (Section 4.1). Then, the abstraction mechanism including the reduction rules (Section 4.2). Finally, we present the transformation of diagrams to a PRISM source code (Section 4.3).

4.1 Fractionation Algorithm

For an optimal hierarchical representation, first we look for the largest possible sub diagram called *ample*, a sub diagram with single input and single output, that can be extracted from the initial one. Then, we proceed the same process till satisfying the stopping criteria [6] (a sequence of opaque nodes). We fractionate A by extracting a set of sub-diagrams $A_1, \cdots, A_n$ satisfying the stopping criteria such that $A = A'[\mathsf{a}_1 \uparrow A_1, \cdots, \mathsf{a}_n \uparrow A_n]$ where ' = ' means both sides have equivalent behaviors.

To align within the standard syntax of SysML activity diagrams, an *ample* is bounded by a single input node and a single output node. Definition 4.1 stipulates formally an *ample* where $pred(a)$ and $succ(a)$ return respectively the predecessor and successor nodes of a given node a.

Definition 4.1. Let $A = \langle \mathcal{N}, \mathcal{E}, \mathcal{G}, Grd, Prob \rangle$ be an SysML activity diagram, $A' = \langle \mathcal{N}', \mathcal{E}', \mathcal{G}', Grd', Prob' \rangle$ is the largest ample of A such that $A = A''[\mathsf{a} \uparrow A']$ where $A'' = \langle \mathcal{N}'', \mathcal{E}'', \mathcal{G}'', Grd'', Prob'' \rangle$, iff:

- $\mathcal{N}' \subseteq \mathcal{N}$, $\mathcal{N}'' = (\mathcal{N} \setminus \mathcal{N}') \cup \{\mathsf{a} \uparrow\}$, $\mathsf{a}_i = \mathsf{a}''_i \neq \mathsf{a}'_i$, and $\mathsf{a}_f = \mathsf{a}''_f \neq \mathsf{a}'_f$,
- $\mathcal{E}' \subseteq \mathcal{E}$ and $\mathcal{E}'' = (\mathcal{E} \setminus \mathcal{E}') \cup \{pred_A(succ_{A'}(\mathsf{a}'_i)) \rightarrowtail \mathsf{a}, \mathsf{a} \rightarrowtail succ_A(pred_{A'}(\mathsf{a}'_f))\}$,
- $\mathcal{G}' \subseteq \mathcal{G}, \mathcal{G}'' = \mathcal{G} \setminus \mathcal{G}', Grd(\mathcal{E}) = Grd'(\mathcal{E}') \cup Grd''(\mathcal{E}'')$ and $Grd'(\mathcal{E}') \cap Grd''(\mathcal{E}'') = \emptyset$, and
- $Prob''(\mathcal{N}'') = (Prob(\mathcal{N}) \setminus Prob'(\mathcal{N}')) \cup \{pred_A(succ_{A'}(\mathsf{a}'_i)) \rightarrow_1 \mathsf{a}, \mathsf{a} \rightarrow_1 succ_A(pred_{A'}(\mathsf{a}'_f))\}$ and $Prob'(\mathcal{N}') \subseteq Prob(\mathcal{N})$.

We ensure the scalability of the modeling and verification process by proving Corollary 4.2 that shows the associativity property of the call composition operator $\uparrow$. For simplicity, we express $A[\mathsf{a} \uparrow A']$ by $A \uparrow_{\mathsf{a}} A'$.

Corollary 4.2. *The composition relation based on the call operator $\uparrow$ is associative, i.e. : $A_1 \uparrow_{\mathsf{a}_1} (A_2 \uparrow_{\mathsf{a}_2} A_3) = (A_1 \uparrow_{\mathsf{a}_1} A_2) \uparrow_{\mathsf{a}_2} A_3$.*

Proof. Based on Definition 4.1, it is easy to show that $A_1 \uparrow_{\mathsf{a}_1} (A_2 \uparrow_{\mathsf{a}_2} A_3) = (A_1 \uparrow_{\mathsf{a}_1} A_2) \uparrow_{\mathsf{a}_2} A_3$, by comparing the final constructed tuple of each side as defined in Definition 4.1. □

The algorithm "Fractal" illustrated in Algorithm 1 finds the largest possible amples of a given SysML activity diagram A by developing the function Ξ. It is called recursively to find a new ample inside the largest ample found by making it as an SysML activity diagram. The diagram is visited using a depth-first search procedure and the algorithm's output is a fractal diagram. First, the initial node is pushed into the stack of nodes denoted by *nodes* (line 6). The algorithm recursively pops a node from the stack *nodes* into the current node denoted by *cNode* (line 8). For a current node *cNode* with a single input edge (line 11) where its successor nodes have the same output node (line 13) the current node is considered and its marked successors are pushed into the stack of nodes (line 14). After all nodes are visited, the new diagram *newD* to be called in the current node *cNode* is constructed from *nNode* by adding an initial and a final node (lines 21-24). Recursively, the algorithm is called for all obtained diagrams satisfying single input single output (lines 25-27).

By relying on the underlying semantics of activity diagrams proposed through operational rules and resulting a probabilistic automaton [13], we consider M_A as the probabilistic automaton of A; and we denote by $\mathbf{R}_\Xi$ the relation related to the function Ξ that is defined in Definition 4.3.

Definition 4.3 (Fractionation relation). Let $A, A' \in \mathcal{A}$ where $A' = \Xi(A)$; then, $M_A \mathbf{R}_\Xi M_{A'}$ denotes the relation between the probabilistic automata M_A, $M_{A'}$ of A and A'.

The soundness of Ξ speculated in Lemma 4.4 shows that the relation $\mathbf{R}_\Xi$ is a bi-simulation relation [16].

Lemma 4.4 (Fractionation Soundness). *The fractalization algorithm Ξ is sound, i.e. $M_A \mathbf{R}_\Xi M_{A'}$ is a bi-simulation relation.*

[6] Initially, we consider a path of opaque nodes as the smallest sub diagram.

Algorithm 1 The Largest Ample Algorithm (Fractal).

Input: SysML activity diagram A.
Output: Fractal SysML activity diagram $A'[\mathsf{a}_1 \uparrow A_1, \cdots, \mathsf{a}_n \uparrow A_n]$.

1: *newDs* as **list_of_ Diagrams** ▷ List of new diagrams: ▷ $A_1, \cdots, A_n$
2: *curD, newD,nodes* as **Stack** ▷ Current diagram, next diagram, ▷ and all nodes.
3: *cNode* as **Node** ▷ Current node to analyze.
4: *nNode, vNode* as **list_of_ Node** ▷ Nodes and visited ones ▷ for the new diagram.
5: **procedure** FRACTAL(A)
6: *nodes.pop*(*in*); ▷ Consider the initial node.
7: **while** *not nodes.empty*() **do** ▷ Parse all nodes.
8: *cNode* := *nodes.pop*(); ▷ Select a current node.
9: **if** *cNode* not in *vNode* **then**
10: *vNode.add(cNode)*; ▷ Consider the current node ▷ as visited.
11: **if** (*size(pred(cNode))=1*) $\wedge$ Type($cNode$) $\neq \mathsf{a}_i$) **then**
12: *nNode* := *next(cNode)*;
13: **if** (*size(succ(nNode))==1* $\wedge$ Type($cNode$) $\notin \{\mathsf{a}_f, \mathsf{a}_t\}$)) **then**
14: *nNode.add*(*cNode*); ▷ Add the next node ▷ to the new diagram.
15: *nodes.delete*(*cNode*);
16: **end if**
17: **end if**
18: **end if**
19: *nodes.push*(*nNode*);
20: **end while**
21: *newD.push*(*in*);
22: *newD.push*(*nodes*);
23: *newD.push*(*fin*);
24: *newDs.add*(*newD*); ▷ Construct the new diagram.
25: **while** *not newDs.empty()* **do** *curD = newDs.Next*;
26: Call Fractal(curD); ▷ Fractionate another diagram.
27: **end while**
28: **end procedure**

Proof. For a given A where $A' = \Xi(A)$, we prove the soundness of Ξ by showing the correspondence between any state and its related transitions in M_A with its similar associated one in $M_{A'}$.

Algorithm 1 implements Ξ by adding two Dirac transitions (Conditions 2 and 4 in Definition 4.1) having silent actions for each called diagram. Further $\mathbf{R}_\Xi$ is an equivalence relation since Ξ is reflexive, symmetric et transitive. And for any action in M_A we have the same in $M_{A'}$ and we have for every two states s and s' in M_A and M'_A the same probability since Ξ does not affect the probabilistic decisions. Then, M_A and M'_A are probabilistically bi-simulated. □

4.2 The Abstraction Algorithm

The abstraction step of the current framework implements, first Ψ then Υ, by extending the one developed in [13] and collapsing states that have similar behaviors as well by taking advantages from the properties of the operator $\uparrow$.

To implement Ψ, we consider a PCTL expression ϕ to be verified on A where Σ_ϕ is the set of the atomic propositions of ϕ. By assuming $\Sigma_\phi \subseteq \mathcal{N}$, we propose Definition 4.5 that reduces the size of A. The first rule excludes the nodes of the diagram that are unrelated to the activity whereas the second excludes the entire called diagram.

Definition 4.5. For a given SysML activity diagram $A \uparrow_{\mathsf{a}} A'$ and a PCTL expression ϕ such that $\Sigma_\phi \subseteq \mathcal{N}$, we have

- $\forall \mathsf{a}_x \notin \Sigma_\phi \wedge \mathsf{a}_x \in \mathcal{N} \cup \mathcal{N}' : \Psi(\mathsf{a}_x \rightarrowtail N) = N$.
- $\Sigma_\phi \cap \mathcal{N}_{A'} = \emptyset : \Psi(A \uparrow_{\mathsf{a}} A') = A$.

Further, Definition 4.6 develops the set of collapsing rules implemented by function Υ.

Definition 4.6. For a SysML activity diagram A, we define a set of reduction rules that are applicable on the artifacts $\|$, $|$, $\blacklozenge$, and $\Diamond$ as follows.

- $\Upsilon(\|(\mathsf{a}_1, \|(\mathsf{a}_2, \mathsf{a}_3))) = \|(\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3)$,
- $\Upsilon(|(\mathsf{a}_1, |(\mathsf{a}_2, \mathsf{a}_3))) = |(\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3)$,
- $\Upsilon(\blacklozenge(\mathsf{a}_1, \blacklozenge(\mathsf{a}_2, \mathsf{a}_3))) = \blacklozenge(\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3)$,
- $\Upsilon(\Diamond_p(\mathsf{a}_1, \Diamond_{p'}(\mathsf{a}_2, \mathsf{a}_3))) = \Diamond_{p.p', p.(1-p'), (1-p).(1-p')}(\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3)$,
- $\Upsilon(\Diamond_g(\mathsf{a}_1, \Diamond_{g'}(\mathsf{a}_2, \mathsf{a}_3))) = \Diamond_{g \wedge g', \neg g \wedge g', \neg g \wedge \neg g'}(\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3)$.

The algorithm "AbsRed" illustrated in Algorithm 2 abstracts a given SysML activity diagram A by taking into account all atomic propositions of a specification ϕ '$Var(\phi)$'. The diagram is visited using a depth-first search procedure, then, the algorithm's output is a less complex diagram. First, the initial node is pushed into the stack of nodes denoted by *nodes* (line 5). The algorithm recursively pops a node from the stack *nodes* into the current node denoted by *cNode* (line 7) and adds each visited node into the list *vNode* of visited nodes (line 9). Then, it applies the abstraction rules in order to minimize the diagram A (lines 10-12). In each iteration, the destination nodes of the *cNode*'s outgoing edges are explored. When two successive nodes are matched (line 14), then they will be collapsed (line 15) to be pushed into *nodes* (line 21). The condition in line 17 excludes a diagram not overlapped with the atomic propositions of ϕ. The algorithm terminates when all nodes are visited.

The algorithm implements the composed function $\Upsilon \circ \Psi$ by calling first Ψ then Υ. Hence, Definition 4.7 defines the relation $\mathbf{R}_{\Upsilon \circ \Psi}$ between the probabilistic automata of A denoted by M_A and $M_{A'}$ of A' obtained by $A' = \Upsilon(\Psi(A))$.

Definition 4.7 (Abstraction-Reduction relation). Let $A, A' \in \mathcal{A}$ where $A' = \Upsilon(\Psi(A))$; $M_A \mathbf{R}_{\Upsilon \circ \Psi} M_{A'}$ is the relation between the probabilistic automata M_A, M'_A of A and A', respectively.

The soundness of the composed functions speculates the type of relation between M_A and $M_{A'}$ where $A' = \Psi \circ \Upsilon(A)$.

Lemma 4.8 (Abstraction-Reduction Soundness). *$\Psi \circ \Upsilon$ is sound, i.e. $M_A \mathbf{R}_{\Upsilon \circ \Psi} M_{A'}$ is a probabilistic weak simulation relation.*

Proof. For $A' = \Upsilon(\Psi(A))$, we prove the soundness of $\Upsilon \circ \Psi$ by showing the correspondence of a set of transitions in M_A with a weak transition in $M_{A'}$ for a given state in M_A.

We have $\Upsilon \circ \Psi$ implemented in Algorithm 2 hides actions by applying Ψ which replaces a set of transitions in M_A with a weak transition in $M_{A'}$. Further, Υ merges nodes that reduces states in

Algorithm 2 The Abstraction-Reduction Algorithm.

Input: SysML activity diagram A.
Output: New Abstracted and Reduced SysML activity diagram A'.

```
1:  nodes as Stack                                   ▷ All nodes.
2:  cNode as Node                                    ▷ Current node.
3:  nNode, vNode as list_of_ Node                    ▷ New and visited nodes.
4:  procedure AbsRed(M, φ)
5:      nodes.push(in);
6:      while not nodes.empty() do
7:          cNode := nodes.pop();                    ▷ Current node to analyze.
8:          if cNode not in vNode then
9:              vNode.add(cNode);
10:             if cNode ∉ InstOf(fin,in)∪Var(φ)) then
11:                 nNode := next(cNode);            ▷ Abstracting nodes
12:                 nodes.delete(cNode);
13:             end if
14:             if (Match(nNode,succ(nNode))) then
15:                 M.Collapse(nNode,succ(nNode));   ▷ Reducing
                                                     ▷ nodes.
16:             end if
17:             if (Var(φ) ⊄ A') then
18:                 A.delete(A');                    ▷ Excluding a diagram.
19:             end if
20:         end if
21:         nodes.push(nNode);                       ▷ Constructing new diagram.
22:     end while
23: end procedure
```

$M_{A'}$ by preserving the probability distribution for the weak transitions. Then, for the transitions of any state in M_A there exist a corresponding weak transition for a state in $M_{A'}$ to represent the same behavior. Consequently, $M_A \mathbf{R}_{\Upsilon \circ \Psi} M_{A'}$ is a probabilistic weak simulation relation. □

4.3 Transformation Algorithm

The function Γ presented in Fig. 1 translates a diagram A to a PRISM source code P. The latter is a composition of modules, presenting diagrams as processes, and communicate à la CSP algebra [7] over global and local variables. Each module has a finite set of local variables of type boolean or integer. A module's state is defined by the evaluation of its local variables, while the global state of P is defined by the evaluation of all variables, locals and global.

The behavior of each module is described by a set of probabilistic and Dirac commands representing the transitions of the probabilistic automata of A (the underlying semantics of an activity has been provided as probabilistic automata [13]). Probabilistic commands are written as $[\alpha]\ g \rightarrow p_1 : u_1 + ... + p_n : u_n$, and Dirac as $[\alpha]\ g \rightarrow u$ where $p_i \in]0, 1[$, $\sum_{1 \leq i \leq n} p_i = 1$, α is the action label, and the guard g is a proposition logic formula over all variables. The update u stipulates the execution of the action α by specifying the next state of the one satisfying g. An update u_i expresses explicitly the evaluation of the local variables of the next state as $(v'_j = [\![v_j]\!]) \& \cdots \& (v'_k = [\![v_k]\!])$ by assigning a value $[\![v_i]\!]$ to a variable v_i, where & is the conjunction operator.

For the set of activities $\mathcal{A}$ and the set of PRISM programs $\mathcal{P}$, we define the function Γ that assigns for each node and edge a fragment of PRISM source code. The generated code of any diagram, initial or called, is bounded by terms '`module the diagram's name`' and `endmodule`.

$$\Gamma(A) = \begin{cases}
[\alpha_x] x \rightarrow (x' = \bot) \& (y' = \top); & \texttt{iff } x, y \in \mathcal{N}, x \rightarrowtail y \in \mathcal{E}, \\
& \mathsf{Type}(x) \in \{\mathsf{a}, \mathsf{a}_i\}, \text{ and} \\
& \mathsf{Type}(y) \in \{\mathsf{a}, \mathsf{a}_t, \mathsf{a}_f, |, \|, \blacklozenge, \Diamond, \ddagger\} \\
[\alpha_x] x \rightarrow (x' = \bot) \& (y'_1 = \top) \& (y'_2 = \top); & \texttt{iff } x, y_1, y_2 \in \mathcal{N}, x \rightarrowtail y_1 \in \mathcal{E}, \\
& x \rightarrowtail y_2 \in \mathcal{E}, \mathsf{Type}(x) \in \{\|\}, \text{ and} \\
& \mathsf{Type}(y) \in \{\mathsf{a}, \mathsf{a}_t, \mathsf{a}_f, |, \|, \blacklozenge, \Diamond, \ddagger\} \\
[\alpha_x] y_1 \& y_2 \rightarrow (x = \top) \& (y'_1 = \bot) \& (y'_2 = \bot); & \texttt{iff } x, y_1, y_2 \in \mathcal{N}, x \rightarrowtail y_1 \in \mathcal{E}, \\
& x \rightarrowtail y_2 \in \mathcal{E}, \mathsf{Type}(x) \in \{|\}, \text{ and,} \\
& \mathsf{Type}(y) \in \{\mathsf{a}, \mathsf{a}_t, \mathsf{a}_f, |, \|, \blacklozenge, \Diamond, \ddagger\} \\
[\alpha_x] y_1 \vee y_2 \rightarrow (x = \top) \& (y'_1 = \bot) \& (y'_2 = \bot); & \texttt{iff } x, y_1, y_2 \in \mathcal{N}, x \rightarrowtail y_1 \in \mathcal{E}, \\
& x \rightarrowtail y_2 \in \mathcal{E}, \mathsf{Type}(x) \in \{\blacklozenge\}, \text{ and,} \\
& \mathsf{Type}(y) \in \{\mathsf{a}, \mathsf{a}_t, \mathsf{a}_f, |, \|, \blacklozenge, \Diamond, \ddagger\} \\
[\alpha_x] x \rightarrow p : (x = \bot \& y_1 = \top) & \texttt{iff } x \in \mathcal{N}, x \rightarrowtail_p y_1 \in \mathcal{E}, \\
+(1-p) : (x = \bot \& y_2 = \top); & x \rightarrowtail_{1-p} y_2 \in \mathcal{E}, \mathsf{Type}(x) \in \{\Diamond_p\}, \text{ and,} \\
& \mathsf{Type}(y) \in \{\mathsf{a}, \mathsf{a}_t, \mathsf{a}_f, |, \|, \blacklozenge, \Diamond, \ddagger\} \\
[\alpha_x] x \wedge g \rightarrow (x = \bot) \& (y'_1 = \bot); & \texttt{iff } x, y_1, y_2 \in \mathcal{N}, x \rightarrowtail_g y_1 \in \mathcal{E}, \\
[\alpha_x] x \wedge \neg g \rightarrow (x = \bot) \& (y'_2 = \bot); & x \rightarrowtail_{\neg g} y_2 \in \mathcal{E}, \mathsf{Type}(x) \in \{\Diamond_g\}, \text{ and,} \\
& \mathsf{Type}(y) \in \{\mathsf{a}, \mathsf{a}_t, \mathsf{a}_f, |, \|, \blacklozenge, \Diamond, \ddagger\} \\
[\alpha_x] x \rightarrow (x = \bot) \& (y'_1 = \bot); & \texttt{iff } x, y_1, y_2, y_3 \in \mathcal{N}, x \rightarrowtail y_3 \in \mathcal{E}, \\
[\alpha_x] x \rightarrow (x = \bot) \& (y'_2 = \bot); & x \rightarrowtail y_2 \in \mathcal{E}, \mathsf{Type}(x) \in \{\dagger\}, \\
& \mathsf{Type}(y_1) \in \{\mathsf{a}_i\}, \mathsf{Type}(y_2) \in \{\mathsf{a}_f\}, \text{ and,} \\
& \mathsf{Type}(y) \in \{\mathsf{a}, \mathsf{a}_t, \mathsf{a}_f, |, \|, \blacklozenge, \Diamond, \ddagger\} \\
[\alpha_x] x \rightarrow (x' = \bot) & \texttt{iff } x \in \mathcal{N}, \mathsf{Type}(x) \in \{\mathsf{a}_t\} \\
[\alpha_x] x \rightarrow \bigwedge_{i>0} (x'_i = \bot) & \texttt{iff } x \in \mathcal{N}, \mathsf{Type}(x) \in \{\mathsf{a}_f\} \\
[\alpha_x] x_1 \wedge x_2 \rightarrow p : (x'_1 = \bot \& x'_2 = \bot \& y'_1 = \top); & \texttt{iff } x_1, x_2, y_1, y_2 \in \mathcal{N}, x_1 \rightarrowtail_p y_1 \in \mathcal{E}, \\
+1-p : (x'_1 = \bot \& x'_2 = \bot \& y'_2 = \top); & x_2 \rightarrowtail y_2 \in \mathcal{E}, \mathsf{Type}(x) \in \{\ddagger\}, \text{ and,} \\
[\alpha_x] \neg x_1 \wedge x_2 \rightarrow (x'_1 = \bot \& x'_2 = \bot \& y'_2 = \top); & \mathsf{Type}(y) \in \{\mathsf{a}, \mathsf{a}_t, \mathsf{a}_f, |, \|, \blacklozenge, \Diamond\}
\end{cases}$$

Definition 4.9 defines the mapping relation of Γ by relying on the correspondence between M_A, the probabilistic automata of A, and M_P the probabilistic automata of P constructed respectively through the operational semantic rules developed in [13].

Definition 4.9 (Mapping relation). For $A \in \mathcal{A}$ and $P \in \mathcal{P}$ where M_A and M_P are the probabilistic automata of A and P such that $\Gamma(A) = P$; then, $M_A \mathbf{R}_\Gamma M_P$ is the relation between M_A and M_P.

The function Γ is sound means that the relation between M_A and M_P is a strong bi-simulation relation [16].

Lemma 4.10 (Mapping Soundness). *The mapping algorithm Γ is sound, i.e. $M_A \mathbf{R}_\Gamma M_P$ is a strong probabilistic bi-simulation relation.*

Proof. By comparing the transitions of M_A and M_P where $\Gamma(A) = P$, $A \in \mathcal{A}$, and $P \in \mathcal{P}$, we show that for each action of a state in M_A has one and only one equivalent in M_P. Further for two states s_1 and s_2 where $s_1 \mathbf{R}_\Gamma s_2$, we have:

- The initial state of M_A is related to the one of M_P, and
- Each PRISM command produced by Γ is the transitions of an action. Further, for each pair of states s_1 and s_2 where $s_1 \mathbf{R}_\Gamma s_2$ and each transition $s_1 \rightarrow_{p_1} s'_1$ of either M_A or M_P

there exists a transition $s_2 \rightarrow_{p_2} s_2'$ of either M_A or M_P, such that $p_1 = p_2$.

So, $M_A \mathbf{R}_\Gamma M_P$ is a strong probabilistic bi-simulation relation. □

Proposition 4.11 shows that the function Γ preserves the satisfiability of a PCTL expression ϕ.

Proposition 4.11 (Mapping Preservation). *Let $\Gamma(A) = P$ where M_A and M_P are the probabilistic automata of A and P. Then, for a PCTL expression ϕ we have: $M_A \models \phi \Leftrightarrow M_P \models \phi$.*

Proof. Based on the PCTL semantics (Section 3.2) and by following a structural induction on each term of PCTL, we prove the PCTL satisfiability preservation on both M_A and M_P.

We have $M_A \mathbf{R}_\Gamma M_P$ strong bi-simulation relation(Lemma 4.10), which means paths and states behavior are equivalent in both M_A and M_P. Then propositional and temporal operators are satisfied on M_P and M_A for each PCTL operator with respect to its semantics (the equivalence relations defined in Section 3.2). Hence, $M_A \models \phi \Leftrightarrow M_P \models \phi$ □

5 EXPERIMENTAL RESULTS

We show the effectiveness of the proposed framework by applying it on the Open Service Availability Framework[7](OpenSAF), an open source project focused on service availability that goes beyond high availability requirements established to develop a base platform high availability middleware for communication.

OpenSAF's application interface specification code consists of 4,849 files and 1,735,859 lines written in C describing 12 services and two frameworks. For the experiment, we rely on Visustin[8] tool that generates the flow chart diagrams of OpenSAF C++ source code used to derive an equivalent SysML activity diagrams. The obtained diagrams are not probabilistic which are equivalent to UML activity diagrams for this application since the code does not support probabilistic choices.

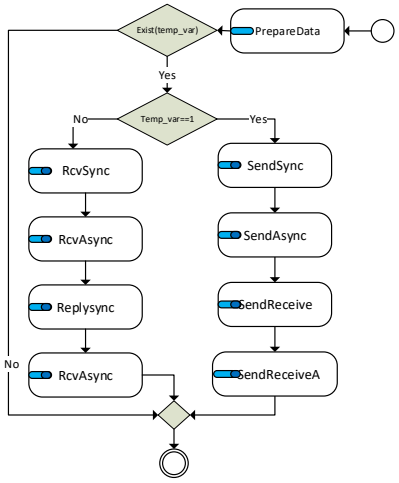

Figure 3: The messaging activity diagram.

[7]http://opensaf.org.

[8]http://www.aivosto.com/visustin.html. It generates flow charts of 36 programming languages

The messaging service is the most important package in OpenSAF. It is a buffered message passing system for processes on the same or different nodes based on message queue. As first step, the function Ξ is applied on the initial one obtained from Visustin. The result is a fractioned diagram calling 9 other diagrams as shown in Figure 3. For the limit of space we focus on the messaging service, the called SendSync diagram is described in Figure 4 and it calls 6 other diagrams including saMsgInitialize behaviour.

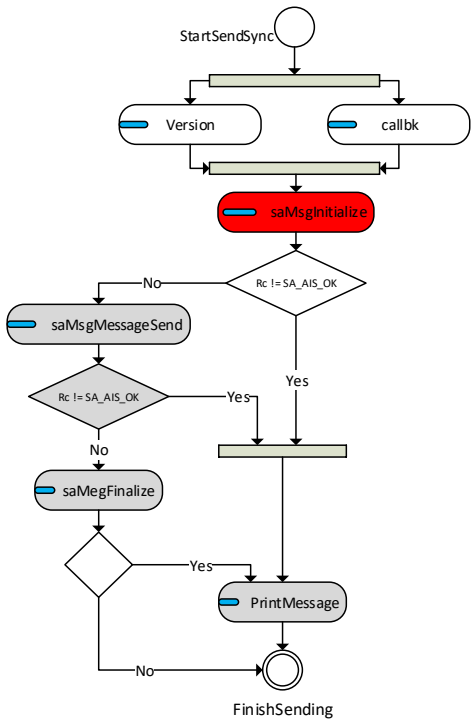

Figure 4: SendSyn Activity diagram.

For verification, we consider two requirements for the behavior related to 'saMsginitialize': 1) The ncsAgents should authenticate before starting up, and also, 2) If it can be shutdown after msInitSync. Both are expressed in PCTL as follows where G stands for the *generally* operator that looks for all paths, X is the *next* operator, and the *eventually* operator F that looks for any state in a path.

- $Pmax = ?G[(InitParameters \rightarrow X(saSecAuthenticate)) \& (saSecAuthenticate \rightarrow X(ncs\_agents\_startup))]$
- $Pmin = ?G[(InitParameters \rightarrow F(ncs\_agents\_shutdown))]$

As a second step, the function Ψ is applied where the nodes and diagrams that do not contain the atomic propositions from the properties are excluded. Then, the function Υ merges nodes of similar kinds. The result is a considerable reduced diagram. Later the function Γ is applied and the PRISM source code is generated. The fragment code listed in Listing 1 shows a part of the code.

```
1 MDP
2 const int step;
3 module Messaging //Module of messaging diagram
4 in_M: bool init true; //The variable of the initial node.
```

```
5  DecxTmpVar: bool init false; // Decision for checking the guard
       tmpvar.
6  xTmpVar: bool init false; // Checking the guard tmpvar.
7  DecTmpVar: bool init false; // Decision to evaluate the guard
       tmpvar.
8  TmpVar: [0..1] init 0; //The initial value of tmpVar.
9  SendSyncCall: bool init false; // SendSyncCall action node.
10 SendSyncBack: bool init false; // SendSyncCall action node.
11 Merge: [0..3] init 0; //The initial value of merge.
12 End: bool init false; // End to stop the behavior.
13 //intial behavior
14 [in_M] (in_M)->(in'_M = false)&(DxTmpVar' = true);
15 //First decision behavior
16 [DxTmpVar] (DxTmpVar)->(DxTmpVar' = false)&(xTmpVar' = true);
17 [DxTmpVar] (DxTmpVar)->(DxTmpVar' = false)&(xTmpVar' = false);
18 [xTmpVar] (xTmpVar)->(xTmpVar' = false)&(TmpVar' = 1);
19 [xTmpVar] (xTmpVar)->(xTmpVar' = false)&(TmpVar' = 0);
20 //Second decision behavior
21 [DxTmpVar] (DxTmpVar)->(DxTmpVar' = false)&(xTmpVar' = true);
22 [DxTmpVar] (DxTmpVar)->(DxTmpVar' = false)&(xTmpVar' = false);
23 [xTmpVar] (xTmpVar)->(xTmpVar' = false)&(TmpVar' = 1);
24 [xTmpVar] (xTmpVar)->(xTmpVar' = false)&(TmpVar' = 0);
25 //Call SendSync behavior
26 [SendSyncCall] (TmpVar = 1)->(TmpVar'0 = 1)&(SendSyncCall' = true);
27 [SendSyncBack] (SendSyncCall)->(SendSyncCall' = false)&(
       SendSyncBack' = true);
28 //Merge behavior
29 [Mergein] (DxTmpVar & !xTmpVar)->(DxTmpVar' = false)&(
       Merge' = 1);
30 [Mergeout] (Merge > 0)->(End' = true);
31 //End behavior
32 [End] (End > 0)->(in'_M = false)...(End' = false);
33 endmodule
```

Listing 1: The PRISM Source Code Fragment of the System.

Finally, we compare the verification results before and after applying the proposed framework. We have obtained false (Pmax=0) for the first property and true (Pmin=1) for the second one which means OpenSAF needs to implement the authentication for agents and nodes. The scalability is the main difference in terms of time verification. Without applying the proposed framework, it costs 39046.98 seconds for the first property and 27154.87 seconds for the second. By applying the presented framework, the verification time costs 234.589 and 122.01 seconds, respectively. It means, the proposed framework reduces the verification time, in this application case, with a considerable rate of more than 90% which is a prominent reduction.

6 CONCLUSION

We presented, in this paper, a promising verification framework that reduces significantly the cost of probabilistic model-checking by fractioning a system initially modeled as SysML activity diagrams. The proposed framework encloses three steps, the first builds a given diagram in its fractal form which facilitates and accelerates later the analysis and verification procedures. Further, the abstraction and reduction steps reduce the diagram size with respect to the requirement under verification. Furthermore, the translating mechanism verifies the system under test by transforming its proper diagrams to PRISM source code. Finally, we proved the soundness of the presented framework, and illustrated its potential effectiveness on a real use case.

In the near future, we would like to extend the presented framework by investigating the directions shown by dashed edges and enumerated steps in Figure 5. First, we would like to include other kinds of diagrams and features (1), and detail the sketched proofs by

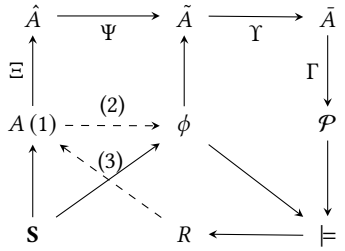

Figure 5: The Extended Fractionation Framework.

presenting the underlying semantics of the fractal SysML activity diagrams, the properties of the implemented functions, and the preservation of the requirements satisfaction. Further, we plan to instantiate automatically the property from the diagrams (2), and to provide a probabilistic counter example guided abstraction refinement approach on the obtained results in case of errors (3). Finally, we target to apply this framework on a large real applications.

REFERENCES

[1] Christel Baier and Joost Pieter Katoen. 2008. *Principles of Model Checking.* The MIT Press.

[2] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. 2001. *Systems and Software Verification.* Springer.

[3] Guillaume Brau, Nicolas Navet, and Jérôme Hugues. 2017. Heterogeneous Models and Analyses in the Design of Real-time Embedded Systems - an Avionic Case-study. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems (RTNS '17).* ACM, 168–177.

[4] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 1999. *Model Checking.* The MIT Press.

[5] Chen Daoxi, Zhang Guangquan, and Fan Jianxi. 2009. Abstraction framework and complexity of model checking based on the Promela models. In *Computer Science Education, 2009. ICCSE '09. 4th International Conference on.* 857 –861.

[6] V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker. 2011. Automated Verification Techniques for Probabilistic Systems. In *Formal Methods for Eternal Networked Software Systems (SFM'11) (LNCS).* Springer.

[7] C.A.R. Hoare. 1985. *Communicating Sequential Processes.* Prentice Hall International, Incorporated.

[8] J. Holt and S. Perry. 2007. *SysML for Systems Engineering.* Institution of Engineering and Technology Press.

[9] Steven Holtzen, Todd D. Millstein, and Guy Van den Broeck. 2017. Probabilistic Program Abstractions. *CoRR* abs/1705.09970 (2017). arXiv:1705.09970

[10] Marius Kloetzer and Cristian Mahulea. 2015. LTL-Based Planning in Environments With Probabilistic Observations. *IEEE Trans. Automation Science and Engineering* 12, 4 (2015), 1407–1420.

[11] OMG. 2007. *OMG Systems Modeling Language (OMG SysML) Specification.* Object Management Group. OMG Available Specification.

[12] Samir Ouchani and Mourad Debbabi. 2015. Specification, Verification, and Quantification of Security in Model-based Systems. *Computing* 97, 7 (01 Jul 2015), 691–711.

[13] Samir Ouchani, Otmane Ait Mohamed, and Mourad Debbabi. 2012. Efficient Probabilistic Abstraction for SysML Activity Diagrams. In *Software Engineering and Formal Methods (SEFM) (LNCS).* Springer, 263–277.

[14] Samir Ouchani, Otmane Aït Mohamed, and Mourad Debbabi. 2014. A property-based Abstraction Framework for SysML Activity Diagrams. *Knowl.-Based Syst.* 56 (2014), 328–343.

[15] Chikmagalur Manjappa Prashanth and K. Chandrashekhar Shet. 2009. Efficient Algorithms for Verification of UML Statechart Models. *Journal of Software* 4 (2009), 175–182.

[16] Lei Song, Lijun Zhang, and Jens Chr. Godskesen. 2011. Bisimulations Meet PCTL Equivalences for Probabilistic Automata. In *CONCUR 2011 – Concurrency Theory*, Joost-Pieter Katoen and Barbara König (Eds.). Springer Berlin Heidelberg, 108–123.

[17] Bernd Westphal. 2006. LSC Verification for UML Models with Unbounded Creation and Destruction. *Electronic Notes in Theoretical Computer Science* 144 (2006), 133–145. https://doi.org/10.1016/j.entcs.2006.01.009

[18] Hongji Yang. 2005. *Software Evolution With UML And XML.* IGI Publishing, Hershey, PA, USA, Chapter Abstracting UML Behavior Diagrams for Verification, 296–320.