Incremental and Formal Verification of SysML Models

Sophie Coudert[1], Ludovic Apvrille[1*], Bastien Sultan[1], Oana Hotescu[2], Pierre de Saqui-Sannes[2]

[1*]LTCI, Telecom Paris, Institut Polytechnique de Paris, 450 route des Chappes, Biot Sophia Antipolis, 06410, France.
[2]ISAE-SUPAERO, Université de Toulouse, France.

*Corresponding author(s). E-mail(s): ludovic.apvrille@telecom-paris.fr;
Contributing authors: sophie.coudert@telecom-paris.fr; bastien.sultan@telecom-paris.fr; oana.hotescu@isae-supaero.fr; pdss@isae-supaero.fr;

Abstract

Agile methods are now commonly used to design critical systems. They consist in progressively doing increments to a model, and subsequently checking that all previously checked properties are still satisfied. Yet, model-checking is not inherently incremental, which means that all proofs must be redone at each stage, where one would expect to redo proofs only for parts of the systems that have been impacted by the modification. This makes model evolution costly and hampers the use of agile development methods. The paper proposes to facilitate model updates (also called mutations): whenever a mutation is performed on a model, the algorithms introduced in this paper can determine which proofs remain valid and which ones must be performed again. The main idea to reduce the proof obligation is to identify new possible execution paths that need to be re-verified. Our algorithm reuses the results of proofs applied to a previous model version. The paper applies this approach on dependency graphs generated from SysML models: our generic propagation algorithm can rework mutated dependency graphs so as to deduce more simple properties to be proved on reduced dependency graphs. Our approach can handle reachability properties and discusses extensions to liveness properties. The embedded system of an autonomous vehicle, characterized by real-time communication constraints, exemplifies the challenges and relevance of our approach.

Keywords: Model Mutation, Model Checking, SysML, Time Sensitive Network

1 Introduction

Model-Based Systems Engineering has opened promising avenues, such as better early simulation and formal verification, while also raising new challenges, as elaborated by [28]. Among these challenges, the incremental development of models stands out. Supported by well-known methods such as Event-B [1], incremental modeling involves a meticulous progression, allowing the capture of complex issues through a process of model mutation and verification. Yet, formal verification is known to be costly, so having to perform a formal check on a system even after a small update is a clear limit to design agility.

The current paper, which extends our works initially published in [9], contributes to the formal verification of SysML [21] models after updates (that we call mutations), enabling for the capture of more profound changes in the models than the Event-B approach allows. This is indeed a common practice to progressively build a system from basic functionalities: mutations can be performed in an agile way by progressively adding new blocks (to a block diagram) or new states and transitions to a state machine

diagram. Assuming that a set of reachability properties (*e.g.*, "state 1" is reachable) have been proven on version n of a model, the paper proposes solutions to simplify the reachability proofs to be performed on version $n+1$. For this purpose, the current paper reminds the notion of *mutations* as defined in [33], and a set of algorithms to handle proof simplification when a mutation has been performed. The main idea behind the algorithm is first to figure out, for a given reachability property of a model element e, if a mutation has modified at least one path between the start of the system (version n) and e. For this, a dependency graph featuring logical dependency of the model is built [27] [8], and paths are investigated in this graph. In case at least one path has been modified, then a more complex algorithm, presented in the current paper, handles this situation. In brief, the main idea is to analyze if one old reachability path may still be executable, or if a new path offers a new way to reach the element of interest. For this, we rely on a multi-labelling graph approach handled by a generic propagation algorithm. A running case study, taken from an autonomous driving system, illustrates the main concepts. In particular, it shows how the different mutations could impact the proof and the labelling techniques used to circumvent these cases. The paper also delves into the strengths and limitations of the theoretical foundations presented, thereby unveiling promising perspectives. These include the complexity of the algorithms, the potential to broaden our approach to include other mutation categories, identifying systems where our contribution could significantly reduce verification time, and potential extensions to other types of properties, notably liveness properties.

The paper is organized as follows. Section 2 introduces the SysML diagrams and model mutations we consider, both informally and formally, with a use case. Section 3 illustrates the problem statement with a case study. The heart of the contribution is presented in Section 4 where main algorithms manipulating paths in reachability graphs and paths in dependency graphs are detailed, and then illustrated on the paper's case study. Section 5 discusses the strength, limits and possible extensions of our work. Section 6 provides a survey of related work. In particular, Subsection 6.4 highlights the distinctions between our current research and our previously published papers, as well as details how it diverges from the state-of-the-art in incremental model verification. Section 7 concludes the paper.

2 Problem formalization

The paper presents an enhanced verification scheme that can be applied to the design stage of systems. In the scope of this paper, a design stage is made upon a SysML block instance diagram and a set of SysML state machine diagrams: each block instance has its behavior defined in a state machine diagram. Mutations concern either addition or extension of a block, or the extension of state machines. Since our proof algorithms use the semantics of models and mutations, this section defines these concepts in both informal and formal ways. For this, a use case is first introduced, then the formalisation is illustrated with this use case, thus leading to the problem statement exposed in the next section.

2.1 Running example: an autonomous automotive system

In this section, the use case helps to illustrate the notion of SysML design and of mutations. In the next section 3, it illustrates the notion of dependency graphs, of reachability properties, of mutations, and of incremental verification. It is also used in section 4. However, for illustrating in detail each concept of our verification algorithms (see Section 4.3.2), we rely on toy systems and graphs because this running case study is not adapted to concisely present precise concepts.

As a case study, we consider the example of an "in-vehicle network" based on Ethernet-based Time-Sensitive Networking (TSN) [17] for autonomous vehicles. Autonomous automotive systems have to perform various functions such as advanced driver assistance, infotainment and autonomous driving. Consequently, they need an important number of Electronic Control Units (ECUs) including sensors such as radars, Lidars and cameras interconnected by a backbone network. To maintain low complexity of wiring, recent studies in [4, 23] propose a zone-based in-vehicle network architecture to group the ECUs in zones according to their physical location as illustrated in Figure 1.

In-vehicle architectures for autonomous vehicles have to satisfy very stringent communication requirements since ECUs for autonomous driving must transmit and receive time-sensitive data. For instance, data from radars and Lidars must be processed in time by the deciding ECU function to avoid accidents. To

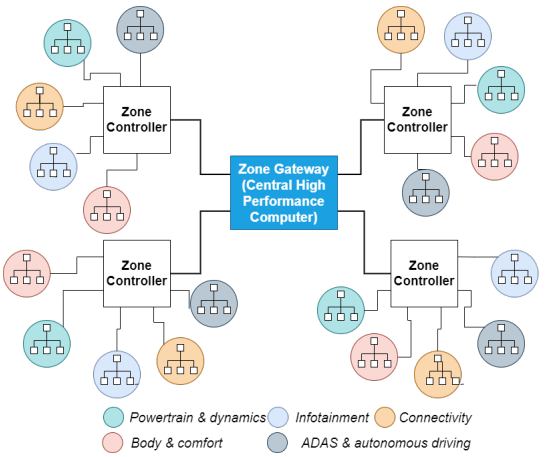

Fig. 1: In-vehicle zonal architecture

guarantee such requirements, specific network protocols have been designed such as the Time-Sensitive Networking (TSN) Ethernet-based standard developed by the IEEE 802.1 Working Group [17]. TSN provides wide bandwidth and low data transmission with accurate time synchronization which is very adapted for the automotive domain.

Since in-vehicle systems are complex critical systems, the formal verification of their properties is needed from the early stages of design. However, these architectures evolve very quickly to provide new and better services and features in vehicles and must adapt accordingly to take into account new functions and ECUs.

For modeling and verification purposes, we have used the latest version of the TTool framework [35], including its ability to generate dependency graphs, its model-checker [11], and its support for model mutations [33].

2.2 A first SysML model

The SysML internal block diagram of our first basic system is given in Figure 2. The main lower block represents a zone controller: it contains two sub blocks, with *ZC1_in1* handling input messages and *ZC1_out2* forwarding messages to an output interface. The top left block represents a sensor sending messages (*ASFD_Tx*) to a data handler (*DID_Rx*) represented at the top right of the Figure. Last, the pink note area corresponds to safety properties that will be discussed in the next section.

More formally, block instances and block instance diagram can be defined as follows.

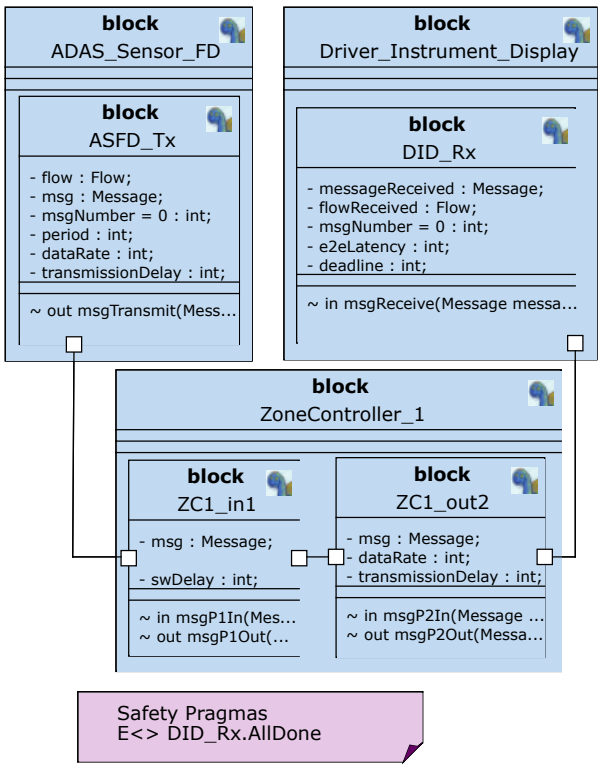

Fig. 2: Block diagram of the basic system (1 stream)

Definition: block instance. A block instance is a 8-tuple $B = \langle id, A, M, P, S_i, S_o, smd, B_p \rangle$ where:

- id is a String that names the block instance.
- A is an attribute list. An attribute is a 3-tuple $\langle identifier, type, initialValue \rangle$ where *identifier* is a String, types include Integer, Boolean, Timer, and user-defined Records, and *initialValue* is the initial value of the attribute.
- M is a set of methods.
- P is a set of ports.
- S_i and S_o are sets of input and output signals.
- smd is a state machine diagram.
- B_p represents the parent block to which B belongs. B_p can be empty.

Definition: Block Instance Diagram. A Block Instance Diagram models the architecture of a system as a graph of interconnected block instances. More formally, a Block Instance Diagram D is a 3-tuple $D = \langle \mathcal{B}, connect, assoc \rangle$.

- $\mathcal{B}$ is a set of block instances. We denote by $\mathcal{S}_i$ the set of all input signals of $\mathcal{B}$, by $\mathcal{S}_o$ the set of all output signals of $\mathcal{B}$ and by $\mathcal{P}$ the set of all ports of $\mathcal{B}$.

- *connect* is a function $\mathcal{P} \times \mathcal{P} \rightarrow \{No, synchronous, asynchronous\}$ that returns the communication semantics between two ports ($\varnothing$, synchronous or asynchronous).
- *assoc* is a function $\mathcal{P}_{B_1} \times \mathcal{S}_o \times \mathcal{P}_{B_2} \times \mathcal{S}_i \rightarrow Bool$ that returns true if an output signal s_o of block B_1 is associated to an input signal s_i of block B_2 via 2 ports $p1$, $p2$ of respectively B_1 and B_2, and if these two ports are connected (*i.e.* $connect(p1, p2) \neq No$).

2.3 State Machine Diagram

Each block instance contains one finite state machine that supports states, transitions, attribute settings and testings, inputs and outputs operations on signals, and temporal operators such as delays and timers.

For instance, the state machine diagram of *ASFD_Tx* is represented in Figure 3. Basically, the state machine features a configuration of parameters, followed with the send of messages, and then looping again on the *Ready* state or going to the *Stop* state once all messages have been sent. The behaviour of *DID_Rx* is quite similar (see Figure 4): the block waits for messages (*Ready* state) until all of them have been received. Then, it goes to the *AllMessagesReceived* state, before going to successive states modeling respectively the computations done from messages and signaling that the processing is done (*AllDone* state).

Definition: State Machine. A finite state machine depicted by a SysML state machine diagram is a bipartite graph $\langle s_0, S, T \rangle$ where

- S is a set of states (s_0 is the initial state).
- T is a set of transitions.

Definition: State Transition. A transition is a 5-tuple $\langle s_{start}, after, condition, Actions, s_{end} \rangle$ where:

- s_{start} is the initial state of the transition.
- $after(t_{min}, t_{max})$ specifies that the transition is enabled only after a duration between t_{min} and t_{max} has elapsed.
- *condition* is a Boolean expression that conditions the execution of the transition.This Boolean expression can use block attributes.
- $action \in \{variable\ affectation, send\ signal, receive\ signal\}$ represents the action attached to the transition. The action can be executed only once the transition has been enabled, *i.e.* when the *after* clause has elapsed and the *condition* equals *true*.

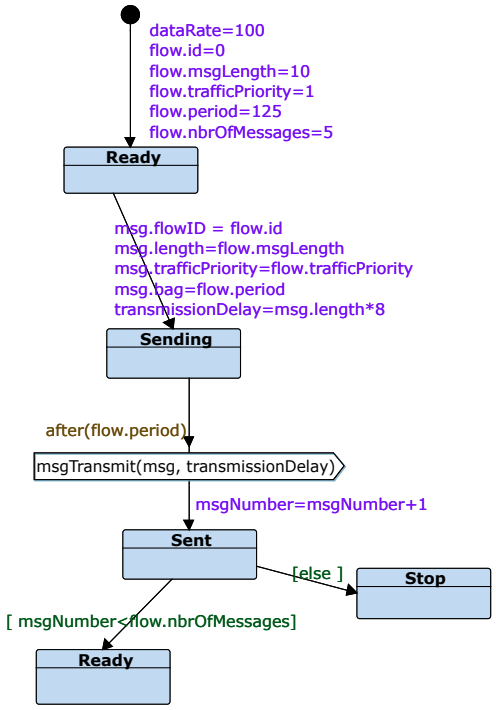

Fig. 3: State machine diagram of ASFD_Tx

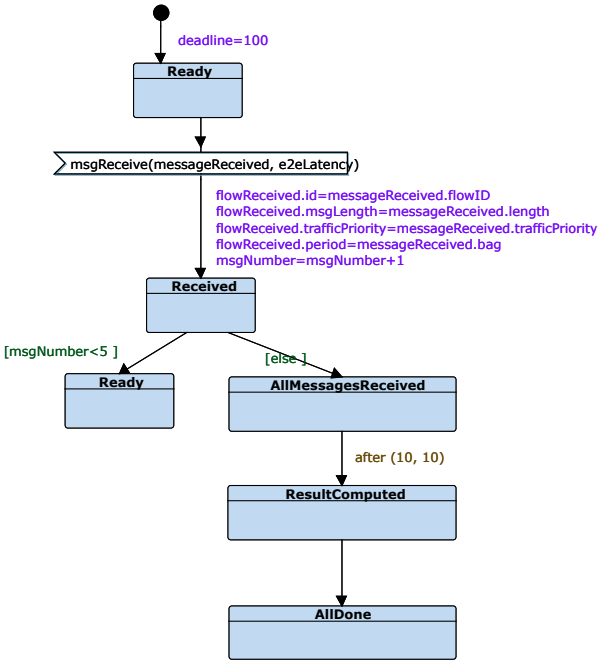

Fig. 4: State machine diagram of DID_Rx

send signal, *receive signal* can use its signals, or the signals of the parent block B_p, and so on.
- s_{end} is the final state of the transition.

2.4 Design mutation

The current paper considers SysML model mutations [33] used to update design diagrams. These mutations rely on elementary functions. We consider five kinds of mutation functions at block instance diagram-level and state machine diagram-level as follows:

- Block instance diagram-level: addition of a new block instance, a new connection between two ports, and a new input or output signal declaration.
- State machine diagram-level: addition of a new state, and a new transition between two states.

For instance, in our running case study, one may want to improve the system architecture with an updated interconnection network by adding new zone controllers (ZoneController_2), or by adding new zone controller ports in ZoneController_1. The behavior of the system could also be updated by adding new behaviors. For example, the display instrument driver may need to display other information coming either from the ADAS_Sensor, or from other sensing blocks.

Let $\mathfrak{D}$ be the set of all block instance diagrams, $\mathfrak{B}$ the set of all block instances, $\mathfrak{A}$ the set of all attributes, $\mathfrak{P}$ the set of all ports, $\mathfrak{S}_o$ (resp. $\mathfrak{S}_i$) be the set of all output (resp. input) signals, $\mathfrak{L}$ be the set of all states and $\mathfrak{T}$ be the set of all transitions.

We now formalize some of the possible model mutations.

2.4.1 Architectural mutations

Function to add a block:

$$add_{Block} : \mathfrak{D} \times \mathfrak{B} \to \mathfrak{D}$$
$$(\langle \mathcal{B}, connect, assoc \rangle, B)$$
$$\mapsto \begin{cases} \langle \mathcal{B} \cup \{B\}, connect, assoc \rangle & \text{if } B \notin \mathcal{B} \\ \langle \mathcal{B}, connect, assoc \rangle & \text{otherwise} \end{cases}$$

Function to add a port connection:

$$add_{Conn} : \mathfrak{D} \times \mathfrak{P} \times \mathfrak{P} \times \{no, sync, async\} \to \mathfrak{D}$$
$$(\langle \mathcal{B}, connect, assoc \rangle, \langle p_1, p_2 \rangle, semantics)$$
$$\mapsto \begin{cases} \langle \mathcal{B}, connect', assoc \rangle & \text{if } (p_1, p_2) \in \mathcal{P} \times \mathcal{P} \\ \langle \mathcal{B}, connect, assoc \rangle & \text{otherwise} \end{cases}$$

where $connect$ and $connect'$ are such that $connect(p_1, p_2) = no$ and $\forall \langle p, q \rangle \in \mathcal{P} \times \mathcal{P} \setminus \{\langle p_1, p_2 \rangle\}, connect'(p, q) = connect(p, q) \wedge connect'(p_1, p_2) = semantics$[1].

Function to add a signal association[2]

$$add_{Assoc} : \mathfrak{D} \times \mathfrak{P} \times \mathfrak{P} \times \mathfrak{S}_o \times \mathfrak{S}_i \mapsto \mathfrak{D}$$
$$(\langle \mathcal{B}, connect, assoc \rangle, \langle p_1, p_2 \rangle, s_o, s_i)$$
$$\mapsto \begin{cases} \langle \mathcal{B}, connect, assoc' \rangle & \text{if } (p_1, p_2) \in \mathcal{P} \times \mathcal{P} \\ \wedge (s_o, s_i) \in \mathcal{S}_o \times \mathcal{S}_i & \\ \langle \mathcal{B}, connect, assoc \rangle & \text{otherwise} \end{cases}$$

where $assoc$ and $assoc'$ are such that $\neg assoc(p_1, s_o, p_2, s_i)$ and $\forall \langle p, q, s, t \rangle \in \mathcal{P} \times \mathcal{P} \times \mathcal{S}_o \times \mathcal{S}_i \setminus \{\langle p_1, p_2, s_o, s_i \rangle\}, assoc'(p, s, q, t) = assoc(p, s, q, t) \wedge assoc'(p_1, s_o, p_2, s_i)$.

Function to add an attribute:

$$add_{Attr} : \mathfrak{B} \times \mathfrak{A} \to \mathfrak{B}$$
$$(\langle id, A, M, P, S_i, S_o, smd, B_p \rangle, a)$$
$$\mapsto \begin{cases} \langle id, A \cup \{a\}, M, P, S_i, S_o, smd, B_p \rangle & \text{if } a \notin A \\ \langle id, A, M, P, S_i, S_o, smd, B_p \rangle & \text{otherwise.} \end{cases}$$

2.4.2 Behavioral mutations

Function to add a state:

$$add_{State} : \mathfrak{B} \times \mathfrak{L} \to \mathfrak{B}$$
$$(\langle id, A, M, P, S_i, S_o, \langle s_0, S, T \rangle, B_p \rangle, s)$$
$$\mapsto \begin{cases} \langle id, A, M, P, S_i, S_o, \langle s_0, S \cup \{s\}, T \rangle, B_p \rangle & \text{if } s \notin S \\ \langle id, A, M, P, S_i, S_o, \langle s_0, S, T \rangle, B_p \rangle & \text{otherwise.} \end{cases}$$

For the needs of the following definition, we define the function

$$parents : B \mapsto \begin{cases} \varnothing & \text{if } B_p \textit{ is empty} \\ \{B_p\} \cup parents(B_p) & \text{otherwise} \end{cases}$$

For a given block instance $B = \langle id, A, M, P, S_i, S_o, \langle s_0, S, T \rangle, B_p \rangle$, we denote with:

- $S_i^+ = S_i \cup \bigcup_{Block \in parents(B)} S_{i_{Block}}$
- $S_o^+ = S_o \cup \bigcup_{Block \in parents(B)} S_{o_{Block}}$

where $S_{i_{Block}}$ (resp. $S_{o_{Block}}$) is the input (resp. output) signals set of *Block*.

[1] $\mathcal{P}$ is the set of all ports of $\mathcal{B}$ such as defined herein above.
[2] $\mathcal{S}_o$ (resp. $\mathcal{S}_i$) is the set of all output (resp. input) signals of $\mathcal{B}$.

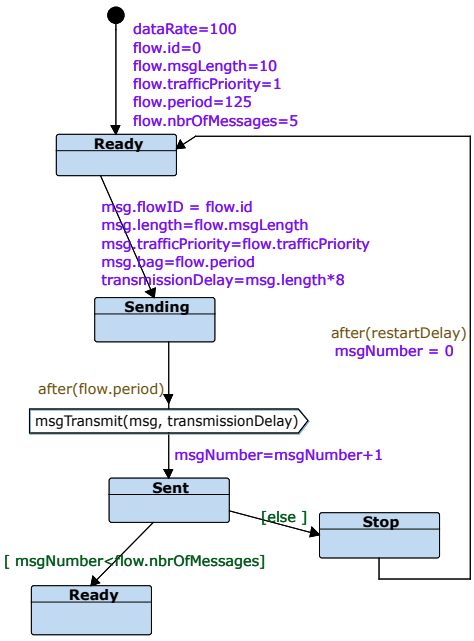

Fig. 5: State machine diagram of ASFD_Tx after mutation

- $\mathfrak{T}_{|B}$ the subset of $\mathfrak{T}$ such that $\forall \langle s_{start}, after, condition, Actions, s_{end} \rangle \in \mathfrak{T}_{|B}$, $(s_{start}, s_{end}) \in S^2 \wedge condition$ is an expression over elements of $A \wedge Actions$ contains only variable affectations over elements of A, receive signal actions over elements of S_i^+ and send signal actions over elements of S_o^+.

Function to add a transition:

$$add_{Trans} : \mathfrak{B} \times \mathfrak{T} \mapsto \mathfrak{B}$$
$$(\langle id, A, M, P, S_i, S_o, \langle s_0, S, T \rangle, B_p \rangle, t)$$
$$\mapsto \begin{cases} \langle id, A, M, P, S_i, S_o, \langle s_0, S, T \cup \{t\} \rangle, B_p \rangle \\ if\ t \in \mathfrak{T}_{|\langle id, A, M, P, S_i, S_o, \langle s_0, S, T \rangle, B_p \rangle} \\ \langle id, A, M, P, S_i, S_o, \langle s_0, S, T \rangle, B_p \rangle \\ otherwise \end{cases}$$

For instance, enhancing the behavior of the ASDF_Tx, as modeled by the SMD depicted in Figure 3 with an automatic reboot feature, can be achieved through the following additive mutations: $ASFD'_{Tx} = add_{Attr}(ASFD_{Tx}, \langle restartDelay, \mathbb{N}, 0 \rangle)$ and then $add_{Trans}(ASFD'_{Tx}, \langle Stop, restartDelay, \top, msgNumber \leftarrow 0, Ready \rangle)$. This results in the SMD illustrated in Figure 5, which now includes the new transition between the states *Stop* and *Ready*.

3 Problem statement and illustration

This section provides a formalized problem statement, and illustrates this problem statement with our running example. As articulated in the preceding section, our aim is to reduce proof complexity when updating a system. Design languages, like SysML, do not inherently provide a straightforward foundation for reasoning. Consequently, we intuitively introduce the concept of dependency graphs in this section, with a formalization presented in Section 4. The use of dependency graphs, constructed from SysML models, has already been explored in a distinct context by [8].

3.1 Problem statement: reachability

The pink box depicted in Figure 2 features the safety property the system shall satisfy. For instance, the property given in the box specifies the reachability of state *AllDone* in block *DID_Rx* : $E <>$ *DID_Rx.AllDone*. This property is expressed using CTL operators. "E" stands for "on at least one path" and "<>" stands for "in at least one state". Reachability properties are at the heart of the paper contribution: our interest is whether reachability properties proven as satisfied remain satisfied, or not, when mutations are applied on the system. Liveness properties are not handled by the present contribution, but they are discussed in section 5.

More formally, we denote a reachability property as $E <> o$ where o is a state or a send/receive action of a state machine of a design D.

We denote with $D_I = \langle \mathcal{B}_{D_I}, connect_{D_I}, assoc_{D_I} \rangle$ the initial design and with $D_M = \langle \mathcal{B}_{D_M}, connect_{D_M}, assoc_{D_M} \rangle$ a mutated design, *i.e.* D_M derives from D_I through a mutation or a composition of mutations among $\{add_{Block}, add_{Conn}, add_{Assoc}, add_{State}, add_{Trans}\}$.

We assume that the reachability property is satisfied in the initial design, i.e., $D_I \models E <> o$[3], *i.e.*, the operator[4] o is reachable in D_I. In our running example, it would mean that state "AllDone" or block "DID_Rx" is reachable in the initial system.

Problem 1. *Instead of reproving if* $D_M \models E <> o$ *using model-checking techniques applied to* D_M *and* $E <> o$*, could we reuse the result* $D_I \models E <> o$ *and*

[3] Symbol $\models$ means "satisfy"

[4] In this context, the term 'operator' refers to a state-machine artifact, encompassing elements such as states and send or receive actions, whose reachability we aim to verify.

$D_M = m(D_I)$ *to lower the complexity of the proof of* $E <> o$ *on* D_M*?*

3.2 Dependency graphs

As explained by Apvrille et al. in [8], a dependency graph is equivalent to a SysML design model. This graph features all the logical dependencies between system elements: *i.e.* state to transition, transition to states, and actions including communication actions. The logical dependencies of communication actions take into consideration the definition of communication between blocks, i.e., in which blocks signals are defined, but also the connection of signals through ports, and finally how blocks are embedded into each other. In the scope of this paper, we consider only asynchronous communications with infinite FIFOs (i.e., a writer is never blocked, while a reader must wait for a message to be available in the FIFO when reading). Basically, all SysML elements have at least one corresponding vertex in the graph such that it is possible to rebuild the original SysML design model from a graph. Such a dependency graph has vertices with no input edges (they correspond to the start states of state machines), vertices with no output edge (states of state machines with no output transitions), and other vertices corresponding to states, to transitions, or to actions of transitions.

We now examine the dependency graph of the running example depicted in Figure 6. This graph depicts each block (*ASFD_Tx*, *ZC1_in1*, *ZC1_out2*, *DID_Rx*) from left to right, respectively. The graph incorporates start states of blocks, represented as green vertices, thus correlating to the initial states of the four state machines. It also features termination vertices (represented in red), aligning with the stop states of *ASFD_Tx* and *DID_Rx*. Given that *ZC1_in1* and *ZC1_out2* perpetually await incoming messages prior to processing, they do not encompass a termination vertex. Vertices correlated to regular states or transitions within the SysML model are indicated in gray. Communication vertices are denoted in purple for sending actions, and in blue for receiving actions. To convey the communication dependency between a sender and a receiver, a green edge is depicted between associated sending and receiving actions. Broadly, all edges illustrate a logical dependency among the operators within the state machines.

3.3 Updating the system

In the design process of such an in-vehicle network, it is common practice to incrementally introduce new network streams. This method allows for a progressive escalation of system complexity. Moreover, it facilitates the consideration of scenarios where network streams are added dynamically during system operation due to the activation of a device by the driver. For instance, upon the driver's initiation of the GPS system, new data streams are established between various Electronic Control Units (ECUs) within the system.

Table 1 showcases four systems, where the initial system is sequentially augmented through three separate mutations. Each mutation increases the complexity of the system, adding numerous blocks, establishing connections between these blocks, and introducing related state machines[5]. Consequently, the execution of each update necessitates the application of multiple mutation functions (addition of each block, addition of their attributes and signals, adding each element of their state machines).

The first update introduces a new network stream using different ports of *ZoneController_1*. This novel stream originates from a new device *IntegratedAntenna* and ends in *ADAS_Sensor_FD*. The second mutation appends a secondary destination to the data stream flowing between *ASFD_Tx* and *DID_Rx*. In this mutation, *ZC1_in1* is in charge of duplicating the stream that now reaches both *DID_Rx* and a new ECU represented by a block named *Engine*. Lastly, the third mutation introduces a GPS transmitter, *GPS_Tx*, that dispatches a data stream to *ASFD_Tx*. Consequently, *ASFD_Tx* now receives dual data streams: one from *IntegratedAntenna* and another from *GPS_Tx*. The final block diagram is illustrated in Figure 7.

3.4 Verification after a system update

Table 1 provides an overview of the dimensions of the dependency graphs and the corresponding reachability graphs. Results were obtained using the latest version of TTool running on a 16-core Macbook pro with 32 GB of RAM. This table exemplifies the benefit of the contribution exposed in Section 4: predominantly operating on dependency graphs facilitates the verification procedure after mutation by reducing the complexity of both the model and properties fed to the model checker. Even in the scope of three mutations, which introduced three new data streams, the

[5] Table 1 specifies only the number of newly added blocks

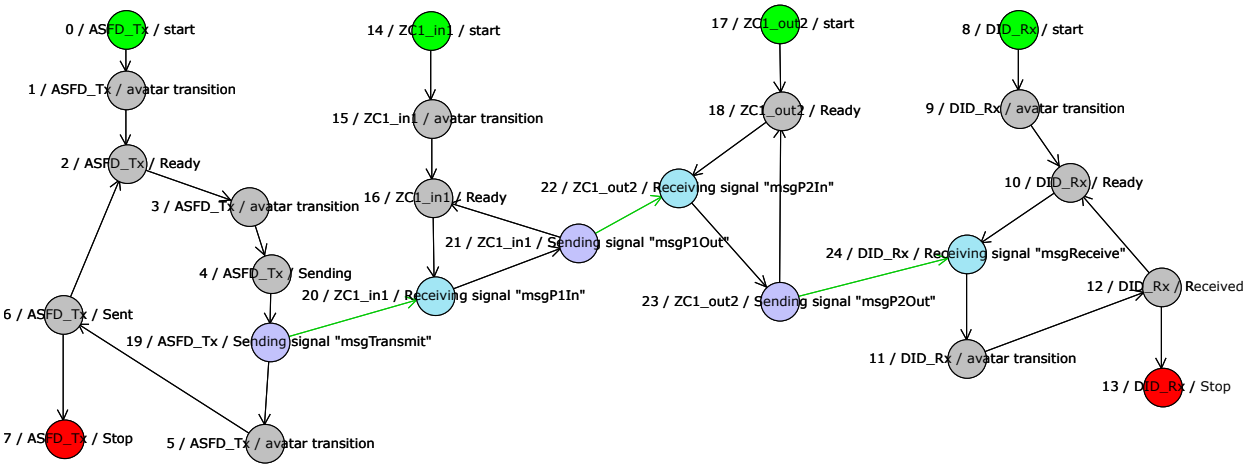

Fig. 6: Dependency graph of the basic system (1 stream). In green: start states ; in red: stop states ; in purple: sending actions ; in blue: receiving actions ; gray: any other state machine elements (states, transitions, actions)

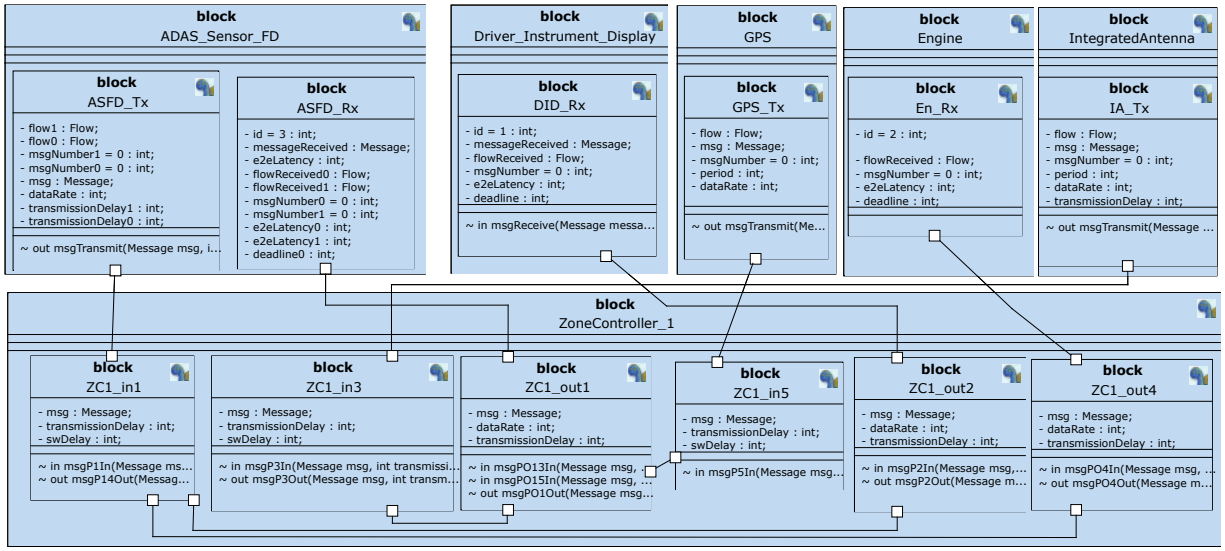

Fig. 7: Block diagram of the system after applying 3 mutations

System	Design	Mutations	DG		RG	
			Vertices	Edges	Vertices	Edges
1	7 blocks: 1 Tx, 1 Rx, 1 in, 1 out, 1 stream	initial system	25	28	581	1072
2	12 blocks: 2 Tx, 2 Rx, 2 in, 2 out, 2 streams	+ 5 blocks	53	59	14k	35k
3	15 blocks: 2 Tx, 3 Rx, 2 in, 3 out, 3 streams	+ 3 blocks	72	84	2.5M	7.6M
4	18 blocks: 3 Tx, 3 Rx, 3 in, 3 out, 4 streams	+ 3 blocks	94	111	$>$ 32M*	$>$ 110M*

**RG generation stopped after 40 minutes*

Table 1: List of systems. System n is created using mutations applied to system $n-1$

dependency graph remains reasonably manageable. Conversely, the reachability graph expands to encompass several dozens of million states and transitions,

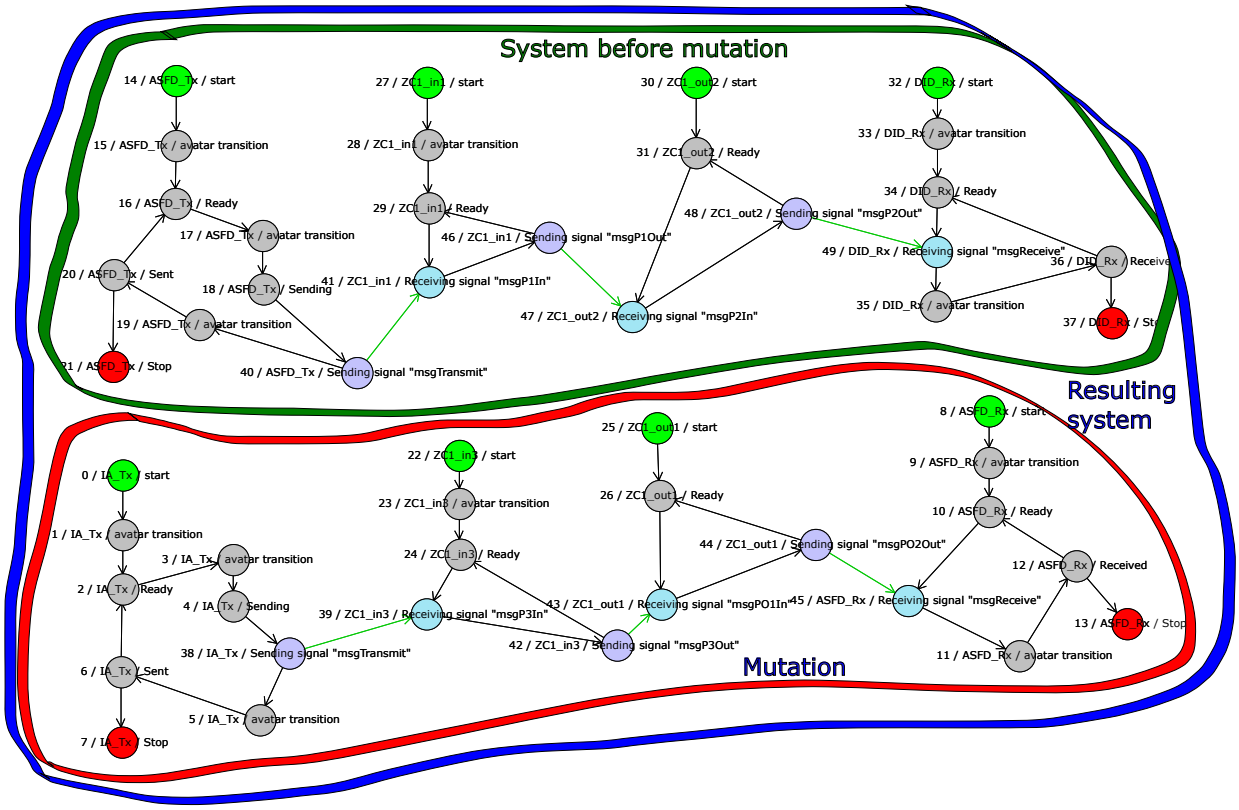

Fig. 8: Dependency graph (circled in blue) after applying one mutation. This graph has two independent parts: the top part of the Figure (circled in green) is exactly the previous system (see Figure 6), the bottom part (circled in red) represents the added network elements. Thus, the logical separation between these two graphs demonstrates that the mutation creates a new network stream logically independent from the network stream of the original model. But again the whole Figure features only a single dependency graph (circled in blue): the model after mutation.

thereby complicating the task of scrutinizing paths within this reachability graph.

As further discussed in section 5.1, the mutations we consider in this paper cannot remove existing logical dependencies: they can only extend the model by adding new blocks, new elements of blocks (for instance, attributes) or new elements of state machines, thus resulting in new logical dependencies. Since deletions are proscribed, logical dependencies of the dependency graph present before the mutation are still present, but obviously the behavior might be altered by additive mutations. Also, the only situation for which an operator o of a state machine would become non reachable is when applying mutations that would prevent all dependency paths to o to be executable. Said differently, previous execution paths would all be "preempted" by the mutations, while no new execution paths would lead to o. The approach exposed in section 4 investigates these two cases (preemption and new path).

Coming back to our example, the dependency graph after applying one mutation is given in Figure 8. One can easily notice that the added data stream creates two concurrent sub dependency graphs: they have no logical dependencies. The top of this graph (circled in blue) corresponds to the dependencies of the former that are still present in the new model, while the lower part (circled in red) corresponds to the new network stream. Both upper and lower graphs (circled in blue, "resulting system") represent the whole dependency graph of the new model. Thus, old execution paths are still valid because they cannot be preempted by the new execution paths. Since *AllDone* state was proved as reachable before mutation, we can easily prove that *AllDone* is still reachable after the first mutation. Our algorithm works similarly: it first investigates how old paths could be impacted by new ones.

The previous mutation added new elements without dependencies with the old elements: this was a trivial example. The dependency graph after three mutations is far more complex, but still totally manageable (94 vertices, 111 edges) with regards to the reachability graph of this updated system. While it also contains a sub-graph totally independent from the old logical paths leading to *AllDone* (this sub-graphs relates to data streams independent from each other), it also add new paths from which *AllDone* logically depends. Consequently, it may not be feasible to infer that the original paths retain their executability, as a newly introduced path could potentially preempt all existing paths leading to *AllDone*. Simultaneously, it is unfeasible to assert with certainty that a new path reaches *AllDone*. Nevertheless, we can undertake the following steps:

- We can remove all vertices and edges that are not in paths leading to *AllDone*.
- If an edge could preempt an old path and that path does not lead to *AllDone*, we can rework that preemption (notion of *Next* explained in Section 4).
- When considering old paths leading to *AllDone*, if a certain point p of an old path is reached and if there is not possible preemption after p, and after p there are no more vertices from which the system blocked at previous verification stage, then reaching p means reaching *AllDone*. The proof of reachability of *AllDone* can thus be replaced by the proof of reachability of p. For instance, the reachability of *AllDone* is ensured as soon as the vertex of *AllMessagesReceived* is reached. So, the reachability of *AllDone* can be replaced by the one of *AllMessagesReceived*. Consequently, some of the vertices and edges can be removed from the dependency graph. Figure 9 gives an excerpt of the dependency graph after 3 mutations. We can cut all vertices and edges between the vertex of *AllDone* (vertex 43) and the vertex of *AllMessagesReceived* (vertex 40). After this cut, a new reachability property must be verified: $E <> DID\_Rx.AllMessagesReceived$.

All these concepts (and others) useful for incremental verification are detailed in next section.

4 Incremental verification: algorithms

In this section, we precisely describe our approach to simplify the verification of a mutated model w.r.t. a

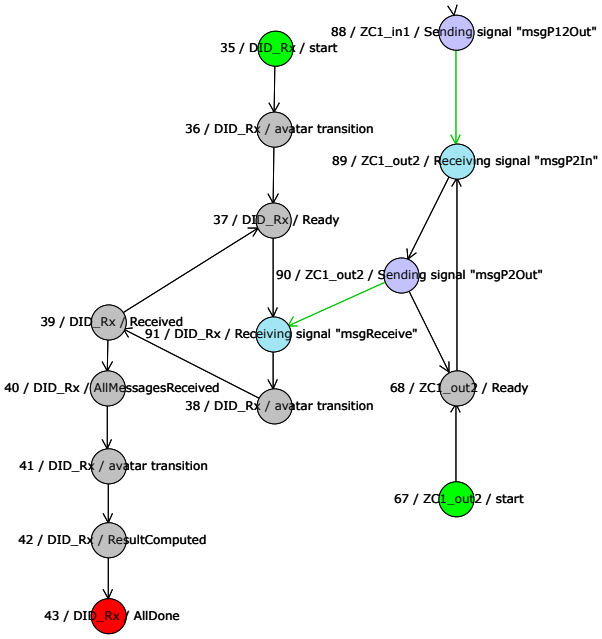

Fig. 9: Excerpt of the dependency graph after applying three mutations

set of reachability properties that have been proven as satisfied on the initial model (before mutation).

Technically, models and dependency graphs are two alternative representations of identical information. We therefore assume: $Model \equiv graphToModel(modelToGraph(Model))$. Then the algorithms presented here rely on these graphs, and for reasons of simplicity, we may occasionally consider graphs as sets of both edges and vertices.

The approach replaces the verification of one reachability property p on a mutated graph with the verification of a set of reachability properties on two smaller graphs: as soon as one of these properties is proved as satisfied, then p is proved as satisfied on the mutated graph. The complexity and interest of proving this set on smaller graphs is discussed in section 5.3. Yet, to compute the two smaller graphs and the related set of reachability properties, we use a central algorithm called *splitGraph*. Subsection 4.1 presents this general approach with the definition of these two graphs and why relying on them for verification instead of the whole mutated graph is sound. Then, section 4.2 is a high level presentation of *splitGraph* algorithms (later, the two last sections provide a complete technical description of them). Section 4.3 presents a **general vertex labelling algorithm** which is widely used in our implementation

of *splitGraph*. This algorithm is later customized to fulfill different types of labelling aiming to achieve our goal. Following this, section 4.4 fully specifies the function *splitGraph* itself. Finally, Subsection 4.5 reuses the main case study to illustrate a few key concepts in a more concrete way.

4.1 Global Approach

The approach considers one single reachability property by iterating on a process that computes a set of reachability properties. The inputs of this process are:

- DG_I: the initial model (before mutations)
- DG_M: the mutated model (after mutation)
- v_p: a vertex in DG_I (and thus in DG_M) that corresponds to a reachability property, denoted by p, that needs to be verified.
- The knowledge that v_p is reachable in DG_I, and the set of blocking states of DG_I (see section 4.4.3), both obtained by model-checking.

Our objective entails verifying the reachability of v_p after mutation, in other words, confirming the accessibility of v_p for system DG_M. Instead of applying model-checking on DG_M, we propose to decompose the verification w.r.t. the following simple (and obviously sound) decomposition principle:

- *assertion:* if v_p is reachable DG_M ($reach(DG_M, v_p)$), there is at least one executable path leading to v_p from start vertices in DG_M and this path is (exhaustively):
  - either an old path,
    i.e all the used vertices and edges are in DG_I.
  - or a new path,
    i.e. at least one used edge is not in DG_I. This edge corresponds to the addition of a new logical dependency.
- *conclusion:* if we have both a method $reach_{old}(DG_M, v_p)$ that decides if there is an old path, and a method $reach_{new}(DG_M, v_p)$ that decides if there is a new path, then $reach(DG_M, v_p) = reach_{old}(DG_M, v_p) \vee reach_{new}(DG_M, v_p)$

Then our main idea is to build two smaller graphs to implement $reach_{old}$ and $reach_{new}$:

- DG_N is dedicated to **new paths** and is such that $reach_{new}(DG_M, v_p) = reach(DG_N, v_p)$
- DG_O **is dedicated to old paths**, together with a set of reachability properties P_O (i.e., a set of vertices of DG_O) such that $reach_{old}(DG_M, v_p) = \exists v \in P_O, reach(DG_O, v)$.

Considering P_O instead of v_p in the latter case is an optimization explained later (cutting some branches in DG_M, c.f. section 4.4.6).

Approximation: building a graph DG_N that accepts new paths without accepting old paths is complex and not required. Thus, in practice, our $reach(DG_N, v_p)$ may accepts old paths and return true also if no new path exists, which is not a false positive w.r.t our main goal: the so detected reachability actually hold in DG_M. In short, our approximation ensures that $reach_{new}(DG_M, v_p) \Rightarrow reach(DG_N, v_p)$ and $reach(DG_N, v_p) \Rightarrow reach(DG_M, v_p)$. One issue is to minimize the residual old paths in DG_N: indeed, smaller dependency graphs mean a faster graph analysis and thus a faster verification process.

Building DG_N, DG_O and P_O is the purpose of the core function of our algorithms *splitGraph* whose high level behaviour can be informally summarized as follows.

> ***function*** $splitGraph(DG_I, DG_M, v_p, blockings)$ ***returns*** (DG_O, P_O, DG_N) ***where***
>
> - DG_O is a reduced graph and P_O is a subset of its vertices, such that v_p is reachable in DG_M through an old path if and only if at least one $p \in P_O$ is reachable in DG_O.
> - DG_N is a reduced graph such that if v_p is reachable through a new path in DG_M, then it is reachable in DG_N, and any reachable path in DG_N is a reachable path in DG_M.

Then, using *splitGraph*, the main process of our approach is described by Algorithm 1 (where blocking information is abstracted). It decides if a set *Prop* of reachability properties that have been proved in the initial model are preserved after mutation. The algorithm iterates on properties: for each reachability property, it first builds the two small graphs mentioned before, then tests if P_O is reachable using old paths, and finally tests if v_p is reachable using a new path. The algorithm stops as soon as one of the tests succeeds. Of course, executing all tests may be computationally intensive: the earlier a test succeeds, the more our approach is efficient.

Algorithm 1: Simplifying reachability proofs after mutation

Data: $DG_I, DG_M, Prop$
Result: $res[Prop]$

```
S foreach p ∈ Prop do
    DG_O, P_O, DG_N = splitGraph(DG_I, DG_M, v_p)
    foreach p_r ∈ P_O do
        res(p) = prover(DG_O, p_r)
        if res(p) then
            break
    end
    if not res(p) then
        res(p) = prover(DG_N, v_p)
end
```

The work presented herein is forward-looking, and the algorithms detailed in this paper are not intended to represent optimized implementations. Rather, they serve to provide a precise specification, fast prototyping, and a breakdown of the computation into manageable steps, thus facilitating understanding, reasoning, and mastery of the concepts. In essence, the focus here is on establishing a proof of concept.

Next section provides a high level description of *splitGraph*'s algorithm (and sub-algorithms) while the following ones contain their precise mathematical description.

4.2 *splitGraph* Overview

A. ***splitGraph*** **computes** DG_O **and** P_O

DG_O and P_O must cover all old paths to v_p and all ways to leave these paths in DG_M, as these ways may compromise the reachability of v_p. The algorithm has four main stages:

1. Keeping only vertices of DG_I that are on paths from start vertices to v_p
2. Identifying branches that are known to eventually lead to v_p: we call these branches "*locally live*". This set of branches defines P_0: vertices in P_0 are the root vertices from which all these branches start.
3. Cutting branches from P_0 because they don't have to be checked since, by definition, they all reach v_p.

At this point, all old paths to v_p have been considered, but events that may lead to diverge from these paths are not yet represented. This is done at the following stage:

4. Adding all "escaping edges" and their targets, i.e., control edges that have their source in the obtained graph but that are not in this graph.

Escaping edges directly lead to deadlock vertices because after them, we are not on an old path any more, but necessarily on a new one. To ensure this, their (added) targets are "copies" of their original targets in DG_M, without any outgoing edge. Then, the updated model is ready for a classical model checking.

B. ***splitGraph*** **computes** DG_N

DG_N must cover all new paths to v_p and all ways to leave these paths in DG_M.

1. Keep only vertices of DG_M that are on new paths from start vertices to v_p
2. Add all "escaping edges".

C. ***splitGraph*** **uses label propagation**

Technically, both algorithms apply a similar approach

1. First, the vertices to keep are identified. This is done by progressively labelling the graphs using a label propagation algorithm. At the end of this process, the vertices to keep are labeled by specific labels $\#DG_O$, $\#P_O$ and $\#DG_N$.
2. The vertices are kept and the relevant edges are added (inherited from DG_I and DG_M), to build the graph that recognized all old or new paths.
3. Finally, escaping edges are added.

Section 4.3 presents a generic label propagation algorithm that may be instantiated by different kinds of labels. Then section 4.4 presents *splitGraph*'s implementation, using relevant labels for this purpose.

4.3 Labelling and Propagation

Here we formalize how to build labellings of graph vertices in a general way by simply composing functions that make evolve these labellings, with associated notations. We call these functions *propagation functions* since they add labels to vertices by looking at the neighbourhood of these vertices and their labelling. Section 4.3.1 precisely defines the labellings and section 4.3.2 presents the algorithm used to build them.

4.3.1 Labelling and Chained Propagations

A labelling associates a set of labels with each vertex of a graph.

Labels is the set of all *labels*.

Vertices is the set of *graph vertices*.

Graphs is the set of all graphs with vertices in **Vertices**

Labellings is the set of *labellings*, *i.e.* total functions lbl : **Labels** $\rightarrow \mathcal{P}$(**Vertices**).

$\varepsilon_{\mathbf{L}}$ is the empty labelling, associating $\varnothing$ to all labels.

Propagations are functions
propag : **Labellings** $\rightarrow$ **Labellings**,

$\mathbf{in_L}$(**propag**) $\subseteq$ **Labels** is the set of labels on which **propag** depends,
i.e., the smallest set L such that for any labelling l, **propag**$(l) =$ **propag**$(l_{|L})$,
where $l_{|L}$ is the restriction of l to the domain $L \subseteq$ **Labels**.

In this paper, we successively apply simple propagation functions, each of them assigning a distinct set of labels L. Once L has been assigned by a step, no subsequent propagation can alter the labels in L. Furthermore, for such a step function to be applied, all the labels it relies on must have been previously handled. Indeed, the step may look at the labels assigned by previous ones to decide which vertices must be labeled by L. This is denoted by $\mathbf{in_L}$(**propag**) above. This leads us to define the subsequent constrained composition along with the associated notation.

Given a set of atomic propagations $\mathbf{propag}_L$,
with $L \subseteq$ **Labels**
and $\mathbf{propag}_L \neq \mathbf{propag}_{L'} \Rightarrow L \cap L' = \varnothing$

- $\xrightarrow{L}$ denotes $\mathbf{propag}_L$, with $assigned(\xrightarrow{L}) = L$
- If pr_1 and pr_2 denote two propagations,with $\mathbf{in_L}(pr_2) \subseteq exported(pr_1)$, then $pr_1 pr_2$ denote $pr_2 \circ pr_1$ and $assigned(pr_1 pr_2) = assigned(pr_1) \cup assigned(pr_2)$.
- If l is a labelling and pr denotes a propagation, $l\,pr$ denotes $pr(l)$.

where $exported(pr) = \mathbf{in_L}(pr) \cup assigned(pr)$ for any propagation pr.

As chained propagations are widely used in the sequel, we adopt a compact notation to denote them, as specified by the last item above. For example, if ***#a***, ***#b*** and ***#c*** are labels assigned by propagations $\mathbf{propag}_{\#a}$ and $\mathbf{propag}_{\#b,\#c}$ such that $\mathbf{in_L}(\mathbf{propag}_{\#a}) = \varnothing$ and $\mathbf{in_L}(\mathbf{propag}_{\#b,\#c}) = \{\#a\}$, then $\varepsilon_{\mathbf{L}} \xrightarrow{\#a} \xrightarrow{\#b,\#c}$ denotes the labelling $\mathbf{propag}_{\#b,\#c}(\mathbf{propag}_{\#a}(\varepsilon_{\mathbf{L}}))$, which assigns exactly the labels ***#a***, ***#b*** and ***#c***.

4.3.2 Propagation Algorithm

Algorithm 2 implements a generic propagation that assigns a set of labels L. It first initializes the labels it handles (line 2) by means of a function $init_L$ (meant to be provided for each particular instantiation). Then, it propagates labels by iterating over a loop in which each vertex v decides which labels it gets by looking at its neighbours and their relative labels. This is the purpose of functions $test_{\#l}(v, Labelling)$, line 9, also specific to each instantiation. Test functions rely on the close neighbourhood of v in some graphs (in our application, DG_M or DG_I), and the labelling of this neighbourhood. Moreover, we use per-label initialisations that can be applied in any order.

Instantiating algorithm 2 specifies the set V of vertices to handle and provides for each $\#l$ in L:
- $Init_{\#l}$: $Labelling \rightarrow \mathcal{P}$(**Vertices**)
- $test_{\#l}$: **Vertices** $\times$ **Labellings** $\rightarrow$ **Bool**.
Considering
- $init_{\#l}(l) = l$, except that $init_{\#l}(l)(\#l) = Init_{\#l}(l)$
- $init_L$: composition of the $init_{\#l}$ functions
This defines $\mathbf{propag}_L^V$: **Labellings** $\rightarrow$ **Labellings**

For so-defined propagations, we have $\mathbf{propag}_L = \mathbf{propag}_L^V$. We can write $\xrightarrow{L^V}$ instead of $\xrightarrow{L}$ for a more informative notation, and V can be a graph G, as a shortcut for $vertices(G)$.

In general, the order of initialisations may matter, as well as the order of the vertices and labels in the loop. Actually, in our instantiations, this order has no impact.

As an obvious example, we can assign a label ***#rch*** to (statically) reachable vertices of a usual control graph. $Init_{\#rch}(l)$ contains the start vertices, and $test_{\#rch}(v,l)$ is true iff l assigns ***#rch*** to some direct predecessor of v. Figure 10 unrolls this instantiation of algorithm 2 on a toy example graph. Iteration 3 is the last one as it does not assign any new vertex.

In a similar way, we could label paths leading to a vertex v by initially labelling v and then apply backward propagation for labelling direct predecessors of

Algorithm 2: Propagation algorithm

Inputs:
  $V \subseteq$ **Vertices**
  $InLabelling \in$ **Labellings**
  $L \subseteq$ **Labels**
Result: $OutLabelling$
$Labelling = init_L(InLabelling)$
$Continue = true$
while $Continue$ **do**
  $Continue = false$
  foreach $v \in V$ **do**
    foreach $\#l \in L$ **do**
      if $\#l \notin Labelling(v)$ **then**
        if $test_{\#l}(v, Labelling)$ **then**
          $Labelling(v) = Labelling(v) \cup \{\#l\}$
          $Continue = true$
    end
  end
end
$OutLabelling = Labelling$

already labeled vertices. Chaining both propagations allows to identify the reachable paths to v. This is one of the intuitions underlying the approach in the next section.

4.4 Computing P_O, DG_O and DG_N

Here, we precisely present the *splitGraph*'s algorithms, i.e., we detail the three steps summarized in section 4.2.C. Propagation algorithms used in step 1 are applied on dependency graphs, thus up from here, **Graphs** contains dependency graphs and we rely on their specific notion of neighbourhood presented in section 4.4.1. Section 4.4.2 presents the "adding-edge" tools used to implement steps 2 and 3. Finally, the following sections details the three steps, i.e. the applied propagations and the way functions of section 4.4.2 are used to obtain P_O, DG_O and DG_N.

Respecting our approach, we make a preliminary simplification: paths downstream of v_p are not useful because as soon as v_p has been reached while model-checking DG_O or DG_N, the reachability query is solved and it is not useful to continue on the path. Thus, we remove edges starting from v_p (paths after P_O are also removed, technically in section 4.4.8).

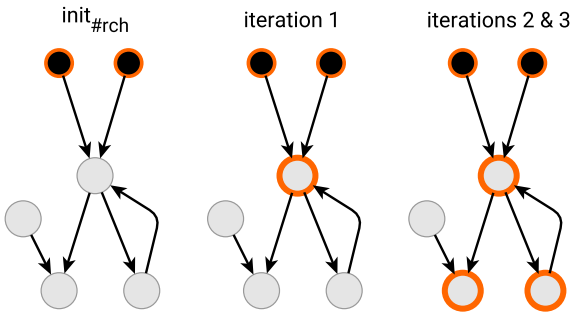

Fig. 10: ***#rch*** label (orange circles) propagation

We remove all edges starting from v_p:
$DG_M := DG_M \backslash \{(v,v') \in Edges(DG_M) \mid v = v_p\}$
$DG_I := DG_I \backslash \{(v,v') \in Edges(DG_I) \mid v = v_p\}$

4.4.1 Neighbourhood in Dependency Graphs:

In dependency graphs, neighbourhood is defined by two binary relations on vertices: control-related order and communication dependency. In the sequel, $Edges(DG)$ is $Ctrl(DG) \cup Com(DG)$. $\overrightarrow{ctrl_{DG}}(v)$ and $\overleftarrow{ctrl_{DG}}(v)$ respectively denote the sets of successors and predecessors of the vertex v with respect to the control relation, *i.e.*, $\{v' \in DG \mid (v',v) \in Ctrl(DG)\}$ and $\{v' \in DG \mid (v,v') \in Ctrl(DG)\}$. Similarly, $\overrightarrow{com_{DG}}(v)$ and $\overleftarrow{com_{DG}}(v)$ respectively denote the sets of successors and predecessors of the vertex v with respect to the communication relation.

4.4.2 Some Tools: *addAllEdges*, *Nexts*

The two following functions define concepts used to build graphs. The first one completes a target graph TG with all the relevant edges of another one, the source SG (com/ctrl edge typing is preserved):

$addAllEdges(SG,TG) = TG \cup \{(nd,nd') \in Edges(SG) \mid \{nd,nd'\} \subseteq vertices(TG)\}$

The second one is a way to test if an execution can escape a subgraph DG of DG_M. Indeed, in the sequel, we build the graph of old paths and the graph of new paths leading to v_p, and we expect to test the reachability of v_p with respect to these subcategories of paths. Thus, in these specialised graphs (where all paths lead to v_p), we must add all edges of DG_M that may "preempt" these paths, *i.e.*, disrupt the path leading to v_p which may compromise the reachability of

v_p. For this, we suggest to enrich DG with all control edges e that exit from DG and their target vertex. For each e:

1. Either e has its source in DG but not its target. In that case, e and its target are added to DG.
2. Or e is in $Ctrl(DG_M)$ but not in $Ctrl(DG)$ and its target is in DG. Then the copy of e's target is added to DG, together with a control edge from the source of e to the added copy (so, a new edge is created).

For added edges that were initially directed towards vertices of DG, we use copies of targets because it ensures that we can't come back to the graph after exiting: all new paths in enriched DG, *i.e.*, paths having at least a new edge, terminate just after this new edge, in an added vertex that is not in DG and has no control successors (in particular in DG). More formally,

- Let $NextOut(DG) = \{(v,v') \in Ctrl(DG_M) \mid v \in DG \wedge v' \notin DG\}$.
- Let $NextNew(DG) = \{(v,v') \in Ctrl(DG_M) \mid v \in DG \wedge (v,v') \notin DG\} \backslash NextOut(DG)$.

$$Nexts(DG) = DG \cup targets(NextOut(DG)) \cup NextOut(DG) \cup copy(targets(NextNew(DG))) \cup \{(v,v') \mid \exists (v,v'') \in NextNew(DG), v' = copy(v'')\}$$

Nexts is exemplified in Figure 11. Vertices i and j illustrate the above-defined point 1, whereas vertex f illustrates point 2, *e.g.*, the addition of a copy f'.

In fact, this version of the *Nexts* function relies on a (temporary) strong simplification we make in the scope of this paper: "next" vertices must not have incoming communications, otherwise our approach fails. Indeed, if j has an incoming communication from a vertex x, we would have to add the part P of the graph useful to potentially reach x, as reaching x would be a condition to reach j. But adding P could unfortunately add paths that don't lead to v_p, which contradicts the elaboration of this graph. It could also add new paths to v_p that are not in DG (using edges in $P \backslash DG$), which again contradicts the purpose of this graph. As later explained (section 5.2), work is in progress to tackle this limitation, at least partially, thanks to a subtle definition of P.

4.4.3 Initial Labelling: *#blocking*

A *path* is a sequence of vertices that begins with a start vertex and continues following our dependency relation. Thus paths are static and provide an over-approximation of dynamic paths (which are actually executable). In general, static analysis cannot decide if a vertex is live, that is if it necessarily leads to some another vertex of interest. The existence of a static path between the two vertices is not sufficient as this path may never be dynamically used.

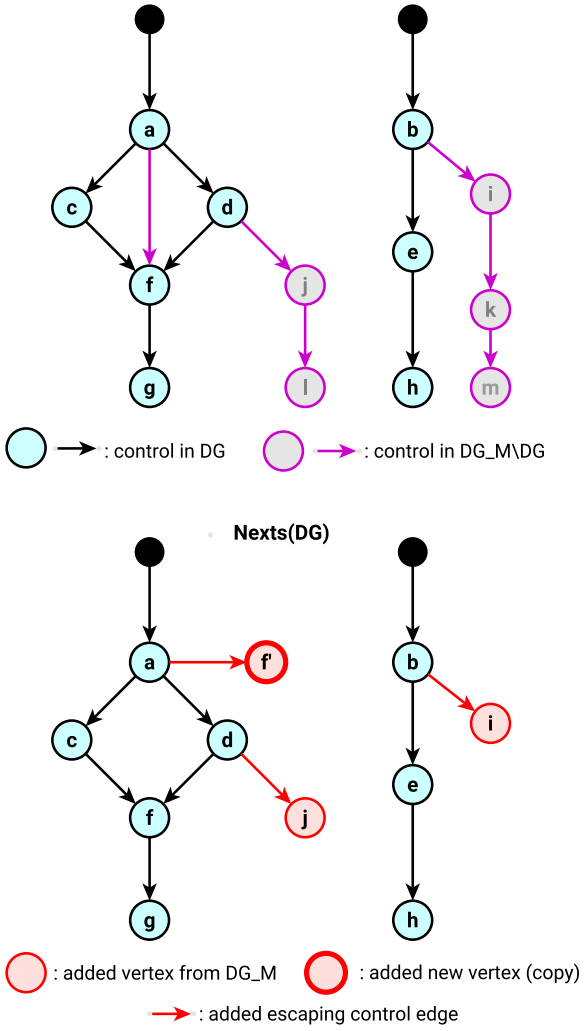

Fig. 11: Use of the *Nexts* function

But since we assume that dynamic execution was performed on the initial system DG_I, we can reuse some of the dynamic information from this execution, such as local liveness. This liveness information we need is identified with ***#blocking*** labels. ***#blocking*** labels the vertices that may be blocking in DG_I, *i.e.*, for which there is a way to reach them with all output edges disabled. Note that we could also use a statically computed over-approximation of blocking vertices but

the loss of precision would be important. We compute ***#blocking*** labels at the beginning of the algorithmic chain. For consistency's sake, it can be treated as a constant propagation that doesn't have dependencies.

> DG_I enhanced with the label ***#blocking*** defines the following propagation:
> $\mathbf{propag}_{\#blocking}$: **Labellings** $\rightarrow$ **Labellings**
> with $\mathbf{in_L}(\mathbf{propag}_{\#blocking}) = \varnothing$.

4.4.4 Identifying paths: *#path*,*#path*$_I$,*#path*$_M$

We use the following parameterized definition to characterize labels vertices of ***#path***(DG) that are statically reachable by using only vertices of DG. Also, if a vertex relies on incoming communications, it can only execute if at least one of these incoming communications is reachable—that is, the source of the corresponding edge is reachable.

Let DG be in **Graphs**. $\#p$ is isomorphic to ***#path***(DG), denoted $\#p \equiv$ ***#path***(DG), if

- $Init_p(l) = \{DG\text{'s start vertices}\}$
- $test_p(v,l) =$
  $(\exists v' \in DG, v' \in l(\mathbf{\#p}) \wedge (v',v) \in Ctrl(DG))$
  $\wedge\ (\overleftarrow{com_{DG_M}}(v) = \varnothing \vee \exists v' \in \overleftarrow{com_{DG}}(v), v' \in l(\mathbf{\#p}))$

We use two times this parameterised label: ***#path***$_I \equiv$ ***#path***(DG_I) and ***#path***$_M \equiv$ ***#path***(DG_M).

> ***#path***$_I$ labels the paths in DG_I, *i.e.*, the statically reachable vertices in DG_I. To be used with $vertices(DG_I) \subseteq V$.
> ***#path***$_M$ labels the paths in DG_M, *i.e.*, the statically reachable vertices in DG_M. To be used with $vertices(DG_M) \subseteq V$.
> Both labels have no dependencies $(\mathbf{in_L}(\mathbf{propag}^{DG}_{\mathbf{\#path}(DG)}) = \varnothing)$.

#path is illustrated in Figure 12. In this Figure, vertices c, d, g are not (statically) reachable in DG because they are not on a control path from a start vertex. Vertex k cannot be reached as the incoming communication of h cannot occur. Finally, vertex o is unreachable because j depends on a communication that can only occur in DG_M and not in DG.

4.4.5 Identifying old local liveness: *#live*

We are now interested in vertices from which v_p is live in DG_M relying only on old paths. Intuitively, v_p is live

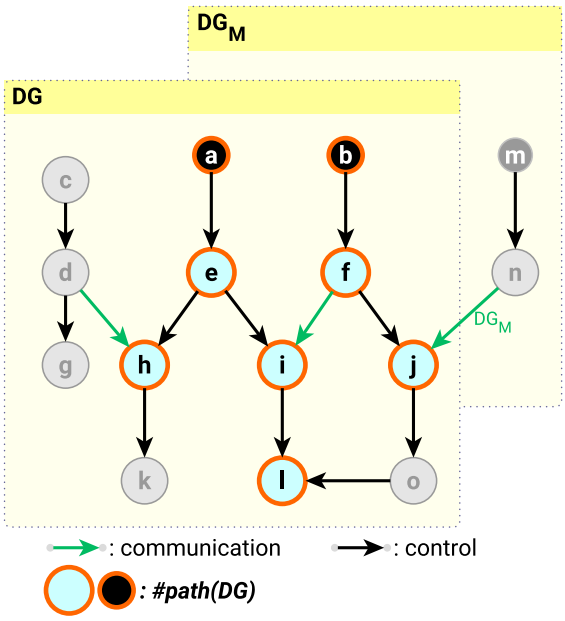

Fig. 12: ***#path*** Illustration

from a vertex v if reaching v ensures to reach v_p, i.e., all executable paths from v lead to v_p. The live vertices we identify here are the ones from which v_p was live in DG_I and this liveness could not have been compromised by diverging edges in DG_M since only additive mutations are considered. Identifying these live vertices in DG_M avoids re-exploring the successors of these vertices while model-checking the old paths of DG_O in DG_M.

Actually, label ***#live*** is an underestimation of this local liveness, which impacts the efficiency but not the correctness.

- $Init_{live}(l) = \{v_p\}$
- $test_{live}(v,l) = v \in l(\mathbf{\#path}_I) \wedge$
  $v \notin l(\mathbf{\#blocking}) \wedge$
  $(\forall v' \in \overrightarrow{ctrl_{DG_M}}(v), v' \in l(\mathbf{\#live}))$

> ***#live*** labels vertices that have local liveness to v_p in DG_M when reached from old paths in DG_I.
> To be called with $vertices(DG_I) \subseteq V$.
> $\mathbf{in_L}(\mathbf{propag}^{DG_M}_{\mathbf{\#live}}) = \{\mathbf{\#path}_I, \mathbf{\#blocking}\}$

#live is illustrated in Figure 13 which only represents the part of DG_I that is relevant, *i.e.* reachable control paths in the component of v_p. ***#live*** vertices have an orange circle: h, j, k, l, n, and v_p. They certainly

lead to v_p using an old path. Vertices like i that do not have all their outgoing paths leading to v_p are not live. Vertex g leads to v_p but is not live since it could block. Vertex e is not live as the loop that originates from it may be infinite. And finally vertices like f may seem to be live but one of its outgoing paths (to p) goes through $DG_M \backslash DG_I$. Thus it has not been tested while model-checking DG_I and cannot be ensured to be dynamically live.

The dynamic information issued from the model checking of DG_I is transmitted by ***#blocking*** in order to ensure that v_p is live in DG_I for all vertices tagged ***#live***. Thus, if a vertex v labelled ***#live*** is reachable in DG_M through old paths, then v_p is reachable in DG_M through old paths (because in this case, v_p remains live through old paths at v in DG_M). Conversely, if v_p is reachable in DG_M through old paths, then there are some v, labelled ***#live***, that are reachable in DG_M through old paths (trivial: v_p is labelled ***#live***).

Of course, using model-checking to decide if there exists a ***#live***-labelled v that is reachable in DG_O may be much more complex than directly checking the reachability of v_p in DG_M. This is particularly true if v_p is the only ***#live*** vertex. But in some cases, finding a relevant v early makes the approach we propose much less complex. Thus, defining efficient strategies to find such a v is of utmost importance: our approach is a first step in this direction. Basically, our approach identifies P_O which is the minimal subset of ***#live*** labelled vertices that it is sufficient to explore while searching v. Developing new strategies is part of our future work.

Handling of loops is another limitation: as soon as a vertex precedes a loop, it cannot be live because termination of loops is not known. Using known static information to compute loop termination would enhance knowledge on live vertices. For instance, vertices b and e defined in Figure 13 could be tagged ***#live*** if the loop can be proved as always terminating. Handling loops is also part of our future work.

We now characterize P_O, and the graph DG_O which are such that P_O is reachable in DG_O if v_p is reachable in DG_M using an old path.

4.4.6 Computing Output P_O: $\#P_O$

- If all reachable control predecessors of v are live, we don't need to test the reachability of v because the existence of any path to v is detected when testing the existence of a path to its predecessors.

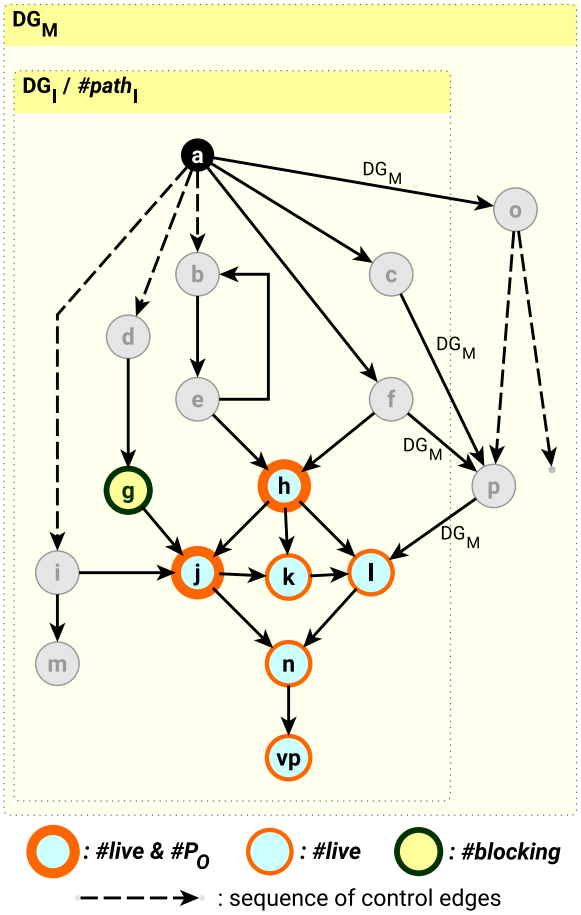

Fig. 13: ***#live*** and P_O Illustration

- If any control predecessor v' of v is not live, the reachability of v may have to be tested, as v was perhaps only reachable in DG_I through a path to v' that may be compromised in DG_M.

As ***#live*** is an underestimation of true local liveness, applying this principle with ***#live***-labelling instead of true liveness leads to a loss of efficiency, as non-identified live vertices will be re-explored while model-checking. As a matter of fact, this does not prevent the tool from identifying whether v_p is reachable or not.

Then P_O is the set of ***#live***-labelled vertices that have at least one predecessor which is not ***#live***-labelled. Since label $\#P_O$ is not recursive, it can be defined with a custom propagation function.

$\mathbf{propag}_{\#P_O}(l)$ is equal to l, except that
$\mathbf{propag}_{\#P_O}(l)(\#P_O) = \{v \in DG_I \mid v \in l(\mathbf{\#live}) \wedge$
$\exists v' \in DG_I, v' \in (l(\mathbf{\#path}_I) \cap \overleftarrow{ctrl_{DG_I}}(v)) \backslash l(\mathbf{\#live})\}$

$\#P_O$ labels the vertices of the output P_O
$\mathbf{in_L}(\mathbf{propag}_{\#P_O}) = \{\mathbf{\#path}_I, \mathbf{\#live}\}$

Thus, $\varepsilon_L \xrightarrow{\mathbf{\#blocking}} \xrightarrow{\mathbf{\#path}_I} \xrightarrow{\mathbf{\#live}} \xrightarrow{\#P_O}$ is a complete computation of P_O: P_O is the set of vertices labelled by $\#P_O$ at the end of the computation.

P_O is illustrated in Figure 13: bold-circled vertices are the ***#live***-labelled vertices that have at least one predecessor in DG_I that is not ***#live***-labelled.

The challenge is now to recognize old paths that lead to P_O.

4.4.7 Identifying ways: *#way*

A "way" to v is a sequence of vertices that leads to the vertex v following our dependency relation in some graph, and conforming to a set of labels. As this notion of "leading to" is used in different contexts, its definition is parameterized.

Let DG be in **Graphs**. Let $\#l$ and $\#w$ be labels, and X a set of vertices. $\#w$ is isomorphic to $\mathbf{\#way}(DG, \#l, X)$, denoted $\#w \equiv \mathbf{\#way}(DG, \#l, X)$, if

- $Init_w(l) = X \cap l(\mathbf{\#l})$
- $test_w(v, l) = v \in l(\mathbf{\#l}) \wedge$
  $(\exists v' \in DG, v' \in l(\mathbf{\#w}) \wedge (v, v') \in Edges(DG))$

$\mathbf{\#way}(DG, \#l, X)$ labels vertices on ways to X by following the dependency relation in DG through ***#l***-labeled vertices.
To be used with $vertices(DG) \subseteq V$.
$\mathbf{in_L}(\mathbf{propag}^{DG}_{\mathbf{\#way}(DG,\#l,X)}) = \{\mathbf{\#l}\}$

#way is illustrated in Figure 14 by orange-circled vertices. Only ***#l***-labelled vertices (blue) can be orange circled. They are circled if some vertex of X is their direct or indirect successor, using both types of edges and going only through blue vertices.

4.4.8 Computing Output DG_O

DG_O must be such that P_O is reachable in DG_O iff P_O is reachable in DG_M using an old path, *i.e.*, a path in DG_I. As a consequence:

1. DG_O must contain all (useful) old paths to P_O.
2. DG_O must not contain any new path to P_O.

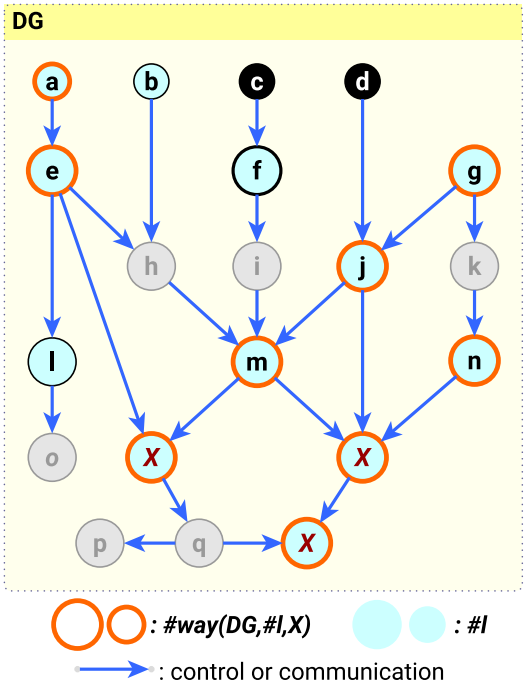

Fig. 14: ***#way*** Illustration

3. DG_O must contain any edge of DG_M that allows to escape from old paths to P_O and thus may compromise its reachability.

$\mathbf{\#way}(DG_I, \mathbf{\#path}_I, P_O)$ computes all old paths to P_O in DG_I. Unfortunately, it also captures paths starting from P_O. These parts of paths are useless since reaching P_O ensures the reachability of v_p by construction. Our approach thus starts by removing the edges starting from P_O, so that these sub-paths are eliminated from the result. Then, we re-use ***#path*** to update reachability in DG_{Iopt}.

- $DG_{Iopt} = DG_I \backslash (P_O \times P_O)$
- $\mathbf{\#path}_{P_O} \equiv \mathbf{\#path}(DG_{Iopt})$.
- $\#DG_O \equiv \mathbf{\#way}(DG_I, \mathbf{\#path}_{P_O}, P_O)$

Label $\#DG_O$ is depicted in Figure 15, as a continuation of Figure 13. It describes all paths to P_O, *i.e.* bold circled vertices in Figure 13, except that vertex i has been excluded because checking the reachability of j or h is sufficient. Here communication paths (that were abstracted in Figure 13) are relevant, thus they appear relying v, w, y and f with green and blue edges.

The following expression ensures that we build an optimized graph DG_{P_O} that respects points 1 and 2

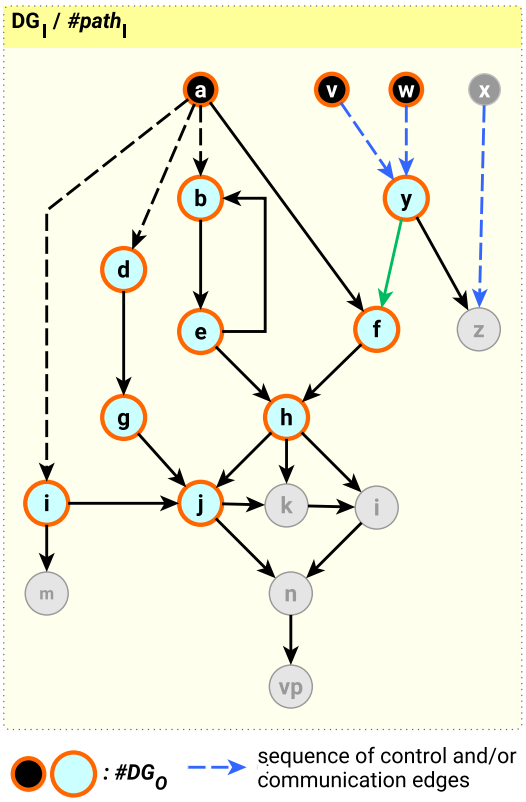

Fig. 15: Illustration of $\#DG_O$ w.r.t. Fig. 13

above, that is DG_{P_O} contains all useful old paths to P_O and no new path.

- $lbl_O = \varepsilon_L \xrightarrow{\textbf{\#blocking}} \xrightarrow{\textbf{\#path}_I} \xrightarrow{\textbf{\#live}} \xrightarrow{\#P_O} \xrightarrow{\textbf{\#path}_{P_O}} \xrightarrow{\#DG_O}$
- $DG_{P_O} = addAllEdges(DG_{Iopt}, lbl_O(\#DG_O))$

Finally, adding the escaping edges mentioned in point 3 above is the role of function *Nexts* defined in Section 4.4.2.

$$DG_O = Nexts(DG_{P_O})$$

A side-effect of the "*Nexts*" function is to reintroduce edges from P_O. Fortunately, these edges do not modify the reachability of P_O. Yet, since these useless edges could make the model-checking process less efficient, we can remove them as follows:

$$DG_O := DG_O \backslash \{(v,v') \in DG_O \mid v \in P_O\}$$

4.4.9 Computing Output DG_N

The purpose of DG_N is to deal with new paths: DG_N must be guaranteed to contain at least all new paths to v_p. It may also contain false positives, *i.e.*, old paths leading to v_p. Our approach intends to minimize them for efficiency purposes.

We first identify vertices on path to v_p in DG_M, labeled $\#\textbf{\textit{tovp}}_M$:

- $\#\textbf{\textit{tovp}}_M \equiv \#\textbf{\textit{way}}(DG_M, \#\textbf{\textit{path}}_M, \{v_p\})$.

Then, among all these vertices, we intend to keep only the ones that are on at least one modified path. These are the vertices v that have at least either a modified edge upstream (between some start vertex and v) or a modified continuation downstream (between v and v_p), or both. We respectively label these two categories of vertices by $\#\delta\textbf{\textit{up}}$ and $\#\delta\textbf{\textit{down}}$, as defined by the propagations below.

Computing the $\#\delta\textbf{\textit{up}}$ label is simple: modified edges contaminate all their descendants through control. However, we also need to identify vertices that have been contaminated by their incoming communication edges, which is the purpose of $\#\delta\textbf{\textit{com}}$. Vertices in $\#\delta\textbf{\textit{com}}$ are not necessarily in $\#\delta\textbf{\textit{up}}$ or $\#\delta\textbf{\textit{down}}$. They also make their control descendants labelled by $\#\delta\textbf{\textit{up}}$. For these labels, the propagation algorithm is called with $V = vertices(DG_M)$.

- $Init_{\delta com}(l) = \varnothing$
- $Init_{\delta up}(l) = \{v \in l(\#\textbf{\textit{tovp}}_M) \mid \exists (v', v'') \in Ctrl(DG_M) \backslash Ctrl(DG_I),\ v'' = v\}$
- $test_{\delta com}(v,l) = v \in l(\#\textbf{\textit{tovp}}_M) \wedge \exists (v', v'') \in Com(DG_M) \wedge v'' = v \wedge v' \in l(\#\delta\textbf{\textit{up}}) \cup l(\#\delta\textbf{\textit{com}})$
- $test_{\delta up}(v,l) = v \in l(\#\textbf{\textit{tovp}}_M) \wedge \exists (v', v'') \in Ctrl(DG_M),\ v'' = v \wedge v' \in l(\#\delta\textbf{\textit{up}}) \cup l(\#\delta\textbf{\textit{com}})$

$\#\delta\textbf{\textit{up}}$ and $\#\delta\textbf{\textit{com}}$ are illustrated in Figure 16. Edges $e \to d$ and $h \to j$ make their descendants pink (*i.e.* $\#\delta\textbf{\textit{up}}$-labelled). Green circled k is the single $\#\delta\textbf{\textit{com}}$-labelled vertex, due to the edge $i \to k$ with i labelled by $\#\delta\textbf{\textit{up}}$. Its descendants are pink.

Vertices labelled by $\#\delta\textbf{\textit{down}}$ are the ancestors of modified vertices, *i.e.*, vertices that have a new outgoing edge or that are in $\#\delta\textbf{\textit{com}}$.

- $Init_{\delta down}(l) = \{\ v \in l(\#\textbf{\textit{tovp}}_M) \mid \exists (v', v'') \in Edges(DG_M), (v', v'') \notin Edges(DG_I) \wedge v = v' \wedge v'' \in l(\#\textbf{\textit{tovp}}_M)\}$
- $test_{\delta down}(v,l) = v \in l(\#\textbf{\textit{tovp}}_M) \wedge (\exists (v', v'') \in Ctrl(DG_M),$

$$v' = v \wedge v'' \in l(\#\delta\textbf{down}) \cup l(\#\delta\textbf{com}) \quad \vee$$
$$\exists (v', v'') \in Com(DG_M),\ v' = v \wedge v'' \in l(\#\delta\textbf{down}) \quad)$$

#δdown-labelled vertices are red circled vertices in Figure 16. All vertices having either a new outgoing edge or a ***#δcom***-labelled vertex in some control path starting from them are concerned. Although it is not illustrated, ***#δdown*** can also be propagated by communication edges: for example, if $k \to m$ were purple, k would be red circled and then i would be red circled too, as it would have a modified outgoing path.

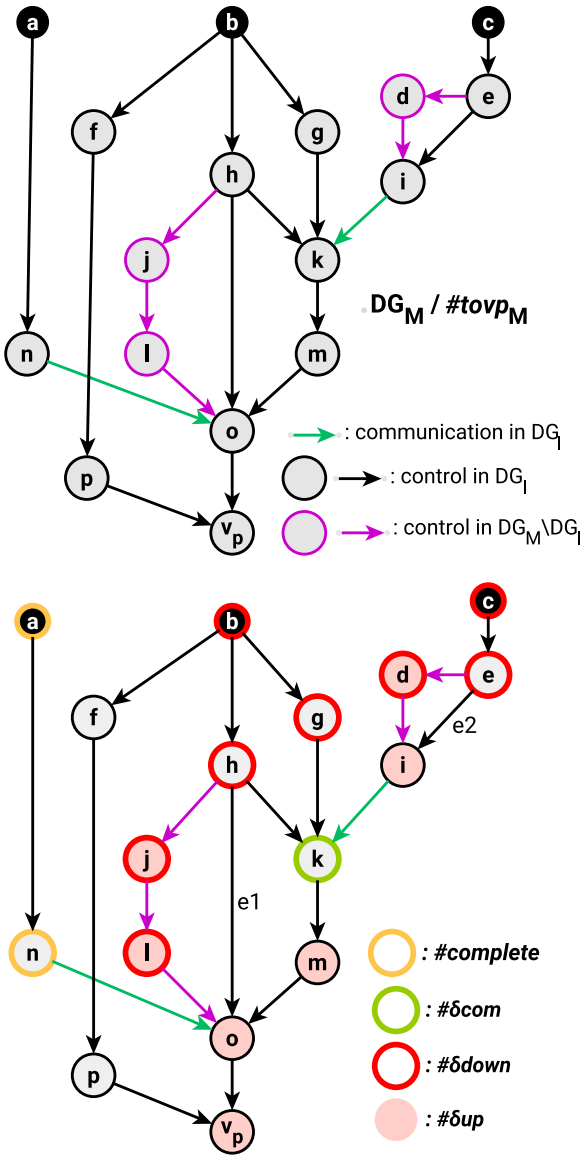

Fig. 16: ***#δup*** and ***#δdown*** illustration

Vertices in ***#δcom***, ***#δup*** or ***#δdown*** represent all the vertices on modified paths. Together they define a graph containing all new paths, but adding all edges of DG_M between these vertices also adds edges that are only in old paths, and so not in the new ones. This is the case for edges $e1$ and $e2$ in Figure 16. Removing these edges leads to an optimized graph (defined as DG_{new}, see below). These edges are such that all paths using them are neither modified in upstream nor in downstream.

- $lbl_{new} = \varepsilon_L \xrightarrow{\#path_M} \xrightarrow{\#tovp_M} \xrightarrow{\#\delta up, \#\delta com} \xrightarrow{\#\delta down}$.
- $DG_{new} = DG \backslash \{(v,v') \in Edges(DG) \mid v \notin lbl_{new}(\#\delta up) \wedge v' \notin lbl_{new}(\#\delta down)\}$, where $DG = addAllEdges(DG_M, l(\#\delta com) \cup l(\#\delta up) \cup l(\#\delta down))$.

Next, to obtain a complete graph where all these paths are statically (or logically) executable, we have to add the incoming communications these paths depend on. This includes the paths that lead to the source of these incoming communications. Indeed, these paths could be part of old paths that have not been labelled by the previous steps. Building this complete graph is the purpose of label ***#complete***

- ***#complete*** = ***#way***$(DG_M; \#path_M, lbl(\#\delta up) \cup lbl_{new}(\#\delta down))$
- $DG_{complete} = DG_{new} \cup addAllEdges(DG_M, lbl(\#complete))$, where $lbl = lbl_{new} \xrightarrow{\#complete}$.

In Figure 16, the edge $n \to o$ leads to label n and a with ***#complete***.

Thanks to these labellings, we have characterized a graph that allows the execution of all new paths to v_p but no path that is not a path to v_p in DG_M. Finally, to obtain a model determining if one of these old paths could be executed, we add all escaping edges.

$$DG_N = Nexts(DG_{p2})$$

Unfortunately, "*Nexts*" reintroduces edges from v_p that can later be easily removed to decrease the complexity of model-checking. For this, we do: $DG_N := DG_N \backslash \{(v,v') \in DG_N \mid v = v_p\}$.

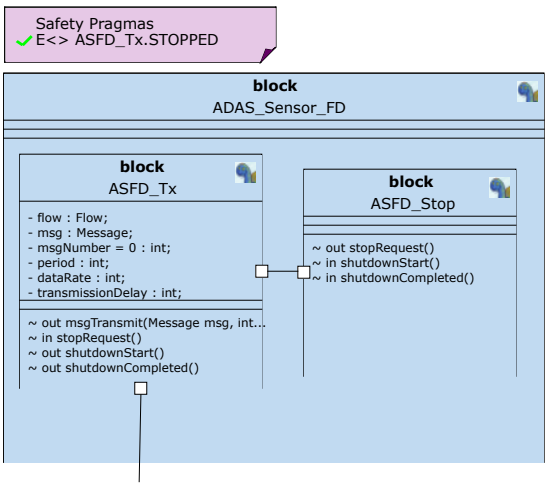

Fig. 17: Architectural modifications within ASFD (mutation $m1$)

4.5 Application to our use case

We now come back to our use case, and show more concretely some of the notions and notations that were presented in this formal section.

4.5.1 A first mutation $m1$

Let us consider the initial system presented in section 2 on which we introduce a mutation. This first mutation, called $m1$, adds a new block (i.e., *ASFD_Stop*) to handle the shutdown of the sensor by shutting down the transmitter of this sensor (i.e., block *ASFD_Tx*) as shown in Figure 17. Thus, we add a new block, called *ASFD_Stop*, and three signals to *ASFD_Tx* that connect to *ASFD_Stop* with a new connection between *ASFD_Tx* and *ASFD_Stop*. The updated state machine diagram of *ASFD_Tx* is presented in Figure 18. Basically, a new transition makes it possible to exit the main loop of *ASFD_Tx* from the *Ready* state. We also consider a new reachability property: $E <>$ *ASFD_Tx.STOPPED* which corresponds to the accessibility of the last state of the shutdown procedure.

The dependency graph of this updated application is shown in Figure 19. Yet, since the reachability property concerns a model element not present in the previous model version, the reachability property cannot be optimized with our approach (the verification is not incremental). The verification takes 46 ms, and the reachability graph has around 11k states and 25k transitions.

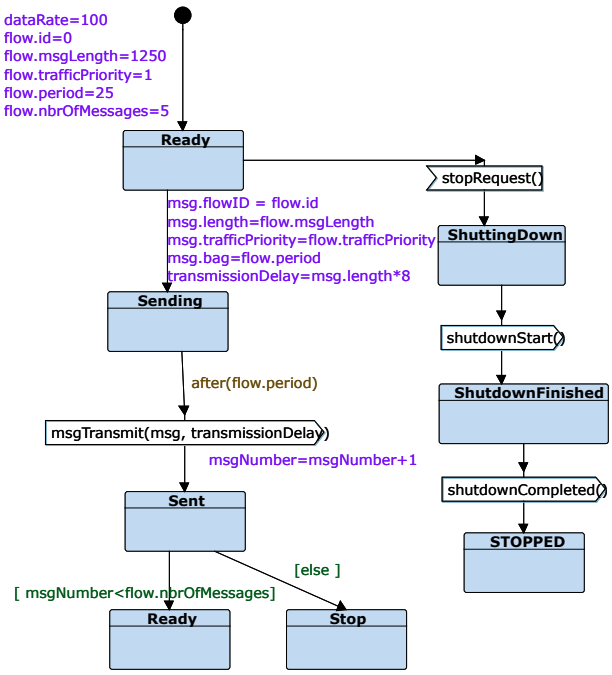

Fig. 18: State machine diagram of ASFD_Tx after mutation $m1$

4.5.2 Applying a second mutation $m2$

Now, we add a shutdown facility to the Display (block *DID_Rx*) as well as a new block to handle this shutdown, *DID_Stop*. The state machine of *DID_Rx* is similar to the one of *ASFD_Tx* in mutation $m1$ in which a new *stopRequest* signal can be received from the main loop of *DID_Rx*.

Let's verify this updated system. Proving $E <>$ *ASFD_Tx.STOPPED* **without** using our approach now takes 968 ms.

Now, let us consider our approach. The dependency graph of the updated system is given in Figure 20. Generating this dependency graph takes less than 1 ms. Let us apply our reduction algorithm on this graph. DG_I is the dependency graph given in Figure 19 while Figure 20 depicts DG_M.

By applying our incremental verification algorithm, we can cut most of DG_M since only *ASFD_Tx* and *ASFD_Stop* tasks are determined as relevant. Moreover, a part of *ASFD_Stop*'s state machine is cut as it does not impact the checked reachability, and a "live-to-vp" branch is cut as it does not have to be checked again. Figure 20 illustrates P_O and DG_O (*ASFD_Tx.STOPPED* is v_p). DG_N is empty as there is no new path to v_p. The ***#live*** label labels the live

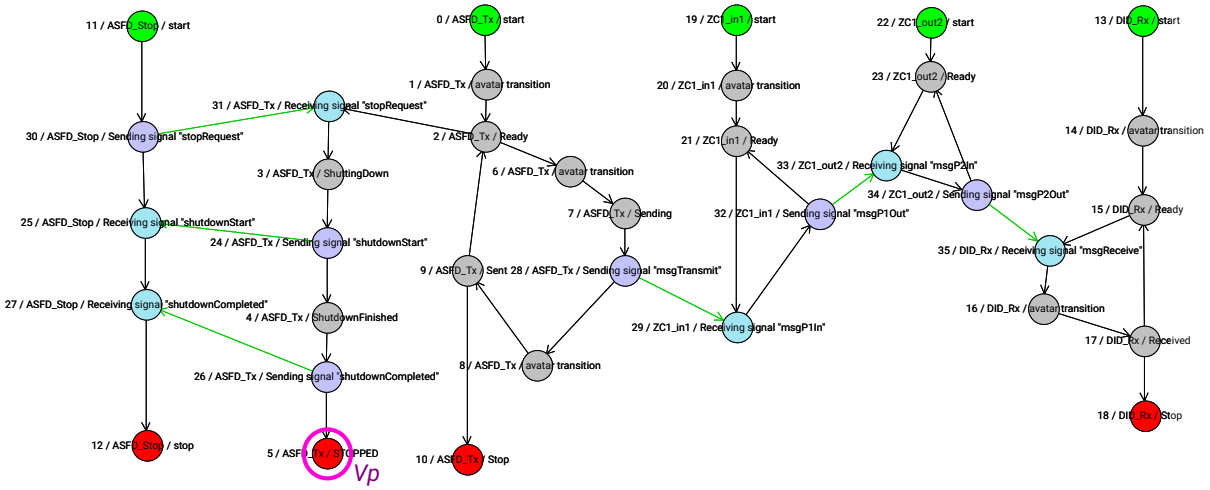

Fig. 19: Dependency graph of the system after applying $m1$

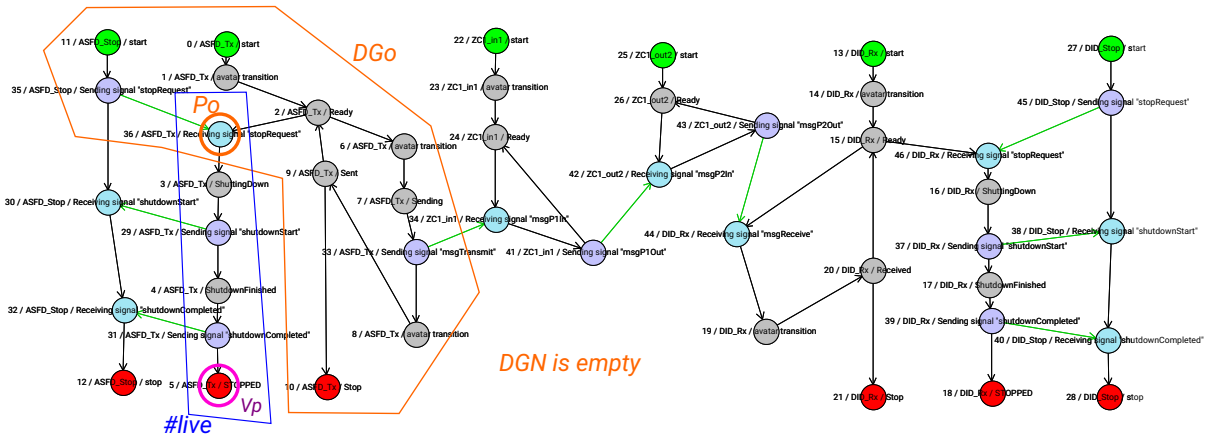

Fig. 20: Dependency graph of the system after applying $m1$ and $m2$

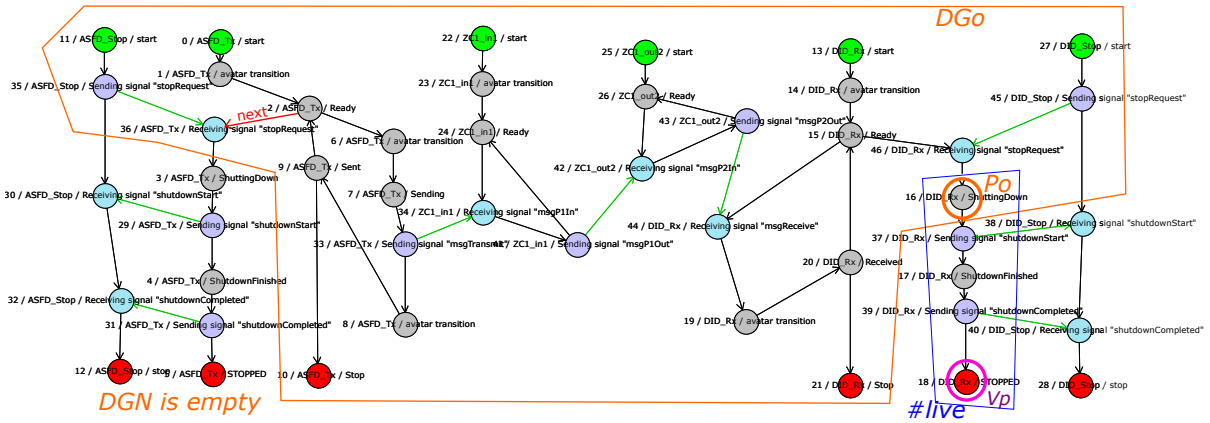

Fig. 21: Dependency graph of the system after applying $m2$ then $m1$

branch that is cut (except P_O). Finally, proving the reachability of P_0 in DG_0 with TTool takes 5 ms.

4.5.3 $m2$ **then** $m1$

Let us now consider another modeling scheme: we apply $m2$ on the initial model, and then $m1$. After applying $m2$, the model has a new stopping procedure for *DID_Rx* and the property to be proved is $E <>$ *DID_Rx.STOPPED* (indeed, *DID_Rx.STOPPED* is v_p).

After applying $m1$, the system has both stopping procedures, and our objective is to prove $E <>$ *DID_Rx.STOPPED* in an incremental way. Figure 21 presents DG_M reworked with our approach. As for previous mutation, Figure 21 shows P_O, DG_O and ***#live***. This example also features escaping edges: the red edge called "next" is an escaping edge. The cut graph of mutations "$m1$ then $m2$" is much smaller: yet, proving $E <>$ *DID_Rx.STOPPED* is reduced to the proof of reachability of state "ShuttingDown" in DID/Rx. For instance, many communications can be removed for this proof, as illustrated by all the elements outside of DG_O. The proof with increment takes around 900 ms, and the one using our approach takes 300 ms. Indeed, even if the system on which the proof is applied is not smaller, many interleavings mostly due to communication are removed. Even if the gain is not as interesting as in the case $m1$ then $m2$, the dependency graph reduction still makes the proof lighter.

5 Discussion

This section discusses how limitations on input models, mutations and algorithms could be addressed. The complexity of our approach is also discussed, as well as enhancements, completions and extensions of this forward-looking work. Part of this discussion relates to work in progress or future work.

5.1 Subtractive mutations

As a first step, the approach presented in previous section has been designed for additive mutations. An additive mutation adds edges or vertices to the original dependency graph. By contrast, subtractive mutations can remove a whole block or simply some of the transitions, states or actions of a state machine: in all cases, this results in the removal of vertices or edges in the dependency graph. To handle such modification, we noted two major impacts on the current version of our algorithms.

First, the ***#live*** label estimates the local liveness with the notion of addition in mind. We could surely define this local liveness differently so as to make it compatible with the removing of edges in the dependency graph. Indeed, removing edges could remove paths to v_p, thus breaking the local liveness of some of the vertices. Thus, with respect to the current version, we must remove from the ***#live*** label of all the vertices from which at least one path to v_p has been removed. Unfortunately, the computation of this updated local liveness is more complex, and may require extra algorithms that our generic propagation algorithm cannot handle. However, this is a decidable problem that we are currently working on.

Secondly, because of the removal of vertices/edges, it is possible that what we name "old paths" do not refer anymore to paths in DG_I. More precisely, since vertices/edges have been removed, the correct reference model to consider is $DG_{del} = DG_I \cap DG_M$. Thus the computation of DG_O must be implemented using DG_{del} instead of DG_I, in particular when labelling old paths (section 4.4.4) and when implementing the steps defined in section 4.4.8. By doing so, we should be able to handle subtractive mutations, and perhaps some other minor modifications. Indeed, this part is still not fully proved.

However, it's important to note that our approach, even focusing on additive mutations, remains relevant and practically applicable in various contexts. Indeed, while incremental modeling within this model-based software engineering methods may occasionally necessitate the removal of certain features, it predominantly involves the addition of new components. This typically includes incorporating new blocks and state-machine diagrams, as well as introducing new states and transitions within these diagrams, as illustrated by the example given in Section 3 where the addition of a new behavior to a subsystem consisted in applying two additive mutations (see Figures 3 and 5). Note also that adding new elements in state-machine diagrams can, paradoxically, limit the model's behavior. Consider a scenario where we have two states, s_1 and s_2, connected by a transition that is triggered by an $after(3)$. If we were to add a third state, s_3, and a transition from s_1 to s_3 that features an $after(1)$, this modification would render state s_2 inaccessible. For all these situations that cover a wide-spectrum of practical modeling circumstances, our approach is fully valid.

5.2 Handling more additive mutations by enhancing the *Nexts* function

As pointed out at the end of section 4.4.2, some of the additive mutations cannot yet be handled. Indeed, some graphs cannot be handled with the current definition of the *Nexts* function. Unfortunately, there is no simple solution.

Let us illustrate this issue with Figure 22 with a simple example. Basically, once we have built the graph DG_{P_O} (yellow area in Figure 22) that contains all old paths to P_O (c.f. section 4.4.8), we add all the edges that escape the graph (red edges): they correspond to edges that have their source in the (yellow) graph and their target outside the (yellow) graph. The target vertices of these edges are added too (red *next* vertex). Then, all the possible paths not going to P_O are taken into account: the model is finally ready to be checked against the reachability of P_O old paths despite these added escaping possibilities.

The problem arises when one of the "next" vertices has an incoming communication edge. In that case, we need more information to decide if the red arrow could be taken by the model-checker. Indeed, as reading from a channel is a blocking action, a "next" vertex can only be reached if data to be read is available, *i.e.*, if the x vertex (the blue vertex in Figure 22) has been reached before. Thus, in this example, not only we should add the red edge and the pink vertex to the graph, but also all elements in the purple triangle, *i.e.*, all the paths that may lead to x being reachable. Unfortunately, the purple triangle may also contain new edges (due to mutations): these edges may themselves produce new paths to v_p in the graph, which compromises the approach because reaching P_O through a new path compromises local liveness. Intuitively, these so-called "dangerous" new edges reside at the intersection of the purple triangle and the yellow region. In left part of figure 23, the purple triangle is built upon nodes *new*, *e* and their attached edges. In the context of this article, we do not address such graphs. However, two methodologies are currently being explored to solve this issue. A first approach consists in being able to precisely identify when the graph contains "dangerous edges". For this, any more accurate overestimation than the current one is relevant. With such an overestimation, we could then compute the corresponding relevant purple triangles. For instance, a basic rough approximation consists in excluding graphs where some new edge can lead both to an "x" vertex (see Figure 22) and

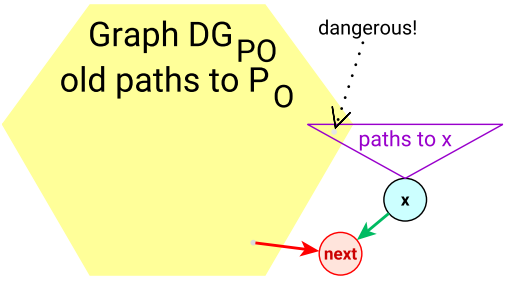

Fig. 22: Issue with the *Nexts* function

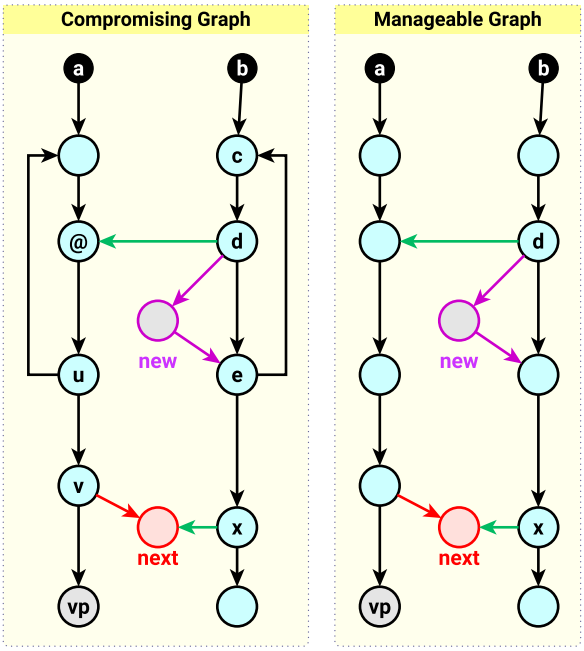

Fig. 23: "Compromising" and "Manageable" graph situations that *Nexts* function can handle with our approximation

to an old communication reaching DG_{P_O}. Figure 23 illustrates typical graphs that we can and cannot handle with this approximation. For instance, the graph on the left cannot be handled because of the purple edges (indirectly) leading both to x and d. These edges are not in DG_{P_O} but they are still reintroduced by the purple triangle, making this new (forbidden) path $b.c.d.new.e.c.d.@.u.v.v_p$ potentially feasible, when it should not be feasible. The graph on the right can be handled because the purple edges introduced by the purple triangle do not lead to d.

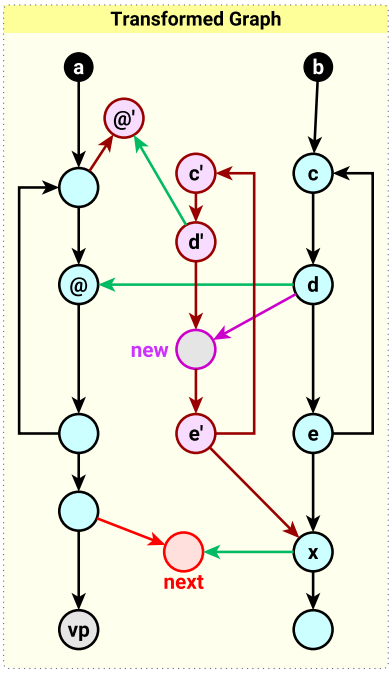

Fig. 24: A solution to the *Nexts* problem

A second approach consists in computing a new graph where going through a new edge makes it definitively impossible to communicate data to DG_{P_O}. This approach seems to be more powerful, but the so-built graph may be more complex than the original one (at most, about twice), so making our incremental verification less interesting. Figure 24 represents such a transformed graph for the graph on the left in Figure 23. The purple edge now leads to a duplicate of the loop in which the (forbidden) edge $d' \rightarrow @$ is replaced by $d' \rightarrow @'$ where $@'$ is a new vertex without any successor. The graph is built in such a way that, after adding the purple edge, v_p is not reachable any more: no new path to v_p is now introduced by our approach, while still ensuring the reachability of x through (new or old) original paths that led to it.

5.3 Complexity and Methodology

As previously highlighted, our propagation algorithm serves as a proof of concept and is admittedly sub-optimal when it comes to the computation of certain labels. It operates akin to a backward propagation mechanism, where a vertex references its neighbours for self-labelling, potentially necessitating multiple scans of all vertices, albeit with room for minor optimization. In certain situations, forward propagation could be used, whereby a labeled vertex propagates its labels to its neighbours, circumventing the need for extensive vertex browsing and consequently enhancing efficiency. Custom algorithms tailored for specific labels could also be a feasible approach. Nevertheless, it is important to note that such algorithms operating on graphs are usually complex and are generally categorized as NP-complete.

In our approach, this complexity remains reasonable since almost all our algorithms work on the model graph, which is expected to be much smaller than the reachability graph. One exception is the ***#blocking*** label which is computed from the reachability graph of DG_I. However, this reachability graph must be computed for the model-checking of DG_I, so the only added complexity is the computation of paths for ***#blocking*** labels.

Once the different graphs of our approach have been built (DG_O and DG_N, cf section 4), they are used for model-checking, instead of the original one (DG_M). We are well aware that using these graphs for model-checking will not always be more efficient, and sometimes this will be less efficient. For instance, if the initial system is rather tiny, and many mutations are applied on the initial system, it is very likely that execution paths will be strongly impacted.

More generally, our algorithms' efficiency is impacted when mutations concern several old paths and introduce several new paths, including paths modifying vertices and edges on paths just before v_p. Thus the approach more likely to be efficient for mutations having little impact on properties of v_p. We are presently developing a more formal characterization of promising instances, by crafting metrics and heuristics. We have already evaluated our approach on various case studies, which have given confidence in the relevance of our approach. This confidence stems especially from the fact that mutations are typically localized within a system, and do not trigger significant alterations. Our forthcoming research includes performance evaluation of our algorithms on systems and mutations generated randomly. Additionally, we plan to categorize the results in relation to the original system and the class of mutations.

Work is also planned to decrease the complexity of the approach by allowing a more important graph reduction. The approximation of local liveness is of main interest for cutting branches in DG_O (section 4.4.8). This approximation could be strongly

enhanced by handling terminating loops. For this some information must be available (for example user-provided) about which loops terminate. Research is underway to identify the relevant form of such information together with algorithms to exploit it.

Lastly, regarding our algorithms, they have been defined to accommodate all additive mutations, regardless of their position within the initial dependency graph. However, in practice, certain cases yield rather straightforward or easily determinable results. For instance, when no mutations occur on all paths from the starting states to a model element e, the mutation does not influence the reachability of e. An optimization strategy in such cases would involve early identification of these scenarios, thereby circumventing the need for a more complex approach. Regardless, our existing algorithms are already equipped to handle these situations.

5.4 Other Properties

While the paper predominantly addresses reachability properties, our methodology could also be advantageous for a variety of other properties. For instance, we might be willing to know if reachability is *not* conserved, meaning that an (un)desirable state has become unreachable due to a mutation, or continues to be unreachable post-mutation. Our approach is adapted for such inquiries. For instance, if v_p has been validated as unreachable in DG_I, then it certainly remains unreachable via old paths in DG_O, hence not necessitating its construction. It only requires verification that v_p isn't reachable via new paths in DG_N. On a broader scale, distinguishing old paths from new paths could yield critical insights for certain purposes. For example, identifying a shift in the path to reach a vertex could signify a potential threat from a security or safety perspective, as this vertex could then be accessed with potentially new attribute values.

The liveness of a model element e is another common property of interest. The liveness of a model element e is satisfied when all executable paths eventually reach v_p. Without entering into many details (*viz.*, no definition nor algorithm), we hereafter provide an idea of how liveness could be taken into account in an incremental way, in the case of additive mutations.

Figure 25 illustrates the general idea. Basically, we assume that the prover can output all executable paths leading to v_p on the initial model: DG_I can thus be reduced to all elements of these paths (black paths in

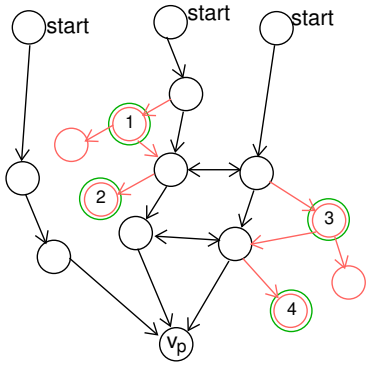

Fig. 25: Proving liveness after model mutation

Figure 25), thus leading to *RDG*. Let us now assume that mutations are performed on the initial model, thus leading to a new model DG_M. New paths in DG_M that depend upon at least one element in *RDG* are added to *RDG* (red paths), leading to build a RDG_M graph. Black paths, such that there are no mutations reaching them or starting from them, can be ignored since their ability to reach e was examined for DG_I. In contrast, other paths are to be re-evaluated. For this, we compute the next elements n (rounded in green in the Figure) of divergent paths (*i.e.*, the first elements of a red path starting from a black path). Then, for each element n, the prover is used, roughly speaking, on the model reduced to all paths leading to n in DG_M to figure out whether n is reachable or not. If n is reachable, then there are two cases. First, if from n there exists no path to v_p, then the liveness is not satisfied anymore. Otherwise, the liveness of v_p from n must be evaluated with the prover. For instance, if the next "2" (Figure 25) is reachable, then the liveness is not satisfied. If the next "1" is reachable, then the liveness from "1" to v_p must be evaluated. From "3", since there is at least one path leading to v_p, the liveness to v_p must also be evaluated.

Finally, if all reachable next elements n eventually reach v_p, then the liveness is satisfied.

6 Related Work

6.1 Formal Verification of SysML Models

The sooner one detects design errors in the life cycle of systems, the lower is the price of fixing these errors. Assuming a system S is modeled in SysML, checking the SysML model of S against design errors may rely on two complementary techniques: simulation or formal verification. The former randomly traverses the

state space of the system. The latter more systematically explores the state space of the system. This paper focuses on formal verification as does the current section on related work.

Several verification techniques have been applied to SysML models. In [26] it is proposed to explore the SysML model entirely and to perform abstractions of that complete behavior to focus verification results on a limited subset of events occurring during the life of the system. Another approach consists in looking for invariants [6] in particular to address mutual exclusion problems. Besides abstractions and search for invariants, model checking is definitely the most commonly investigated technique among papers that address formal verification of SysML models.

Model checking [27] generalizes application of exhaustive exploration of models state space. Fisman and Pnuelli define model checking as the method by which a desired behavioral property of a reactive system is verified over a given system (the model) through exhaustive enumeration (explicit or implicit) of all the states reachable by the system and the behaviors that traverse through them [14]. The model checker is provided with a model of the system and a formal expression of the properties to be verified. The model checker processes the model and the properties, and outputs a "yes/no" answer stating whether the property is verified or not. The model checker also traces execution paths that lead to property violations. The tool must indeed help the system designer with the interpretation of the verification results with respect to the system model.

A survey of the literature indicates that formal verification has been applied to SysML activity diagrams [16, 22, 31] and state machine diagrams [7, 13, 29], respectively. TTool, which is the SysML tool considered in the current paper, applies formal verification to state machine diagrams in the context of SysML models where each block defining the architecture of the system embodies a state machine.

SysML models formal verification tools usually transform a SysML model into a formal language that may provide an external and preexisting formal verification tool. Examples include Petri nets [13, 16, 25, 34], automata for NuSMV model checker [37], timed automata [29] [15] for UPPAAL model checker, hybrid automata [3], model checker NuSMV [19], probabilistic model checker PRISM [3, 22], and a theorem prover [18]. Translation from UML to process algebra has been investigated for RT-LOTOS [7] and CSP [5]. The family of correct by construction specifications has been addressed with Event B [10].

The aforementioned papers essentially apply model checking techniques where a SysML model is checked against a set of properties. User friendliness of formal verification tools therefore depends on the way properties can be easily expressed or not. Users of TTool may insert properties inside the SysML model itself in the form of specific comments [27, 30].

In terms of user friendliness, users of SysML verification tools are further concerned by verification results interpretation [24]. How to come back from verification results to the initial state machines is an issue. It is worth pointing out that the native model checker of TTool can backtrack verification results to the initial SysML model with no obligation for developers of the SysML diagrams to understand the inner workings of TTool's model checker.

6.2 Incremental Modeling and verification

Event-B [1] is a well-know formalism to support incrementality through the modeling and verification processes, enabling the gradual development of systems with a correct-by-construction approach. In the realm of modeling, Event-B allows system architects to start with an abstract model and iteratively refine it, adding details and complexities in a stepwise manner. This incremental approach ensures that each refinement maintains consistency with the abstract model, facilitating a systematic development process. A refinement consists in adding a precision to a given action. On the verification side, Event-B employs mathematical proofs to validate the correctness of both the initial model and its subsequent refinements. As the system evolves, new proofs are generated for each refinement, ensuring that the added details do not violate the established properties of the system.

In [10] Bougacha, Laleau, Collart-Dutilleul and Ben Ayed suggest to translate SysML models into Event-B specifications, and to reuse the refinement mechanisms of Event-B to formally verify the SysML models. The work in [10] follows a correct by construction approach. Conversely, the current paper develops an incremental modeling and verification approach where each increment in models construction results in an incremental model-checking, that is, a model-checking based on reachability properties proved on more simple application graphs.

In [12] Carrillo, Chouali and Mountassir focuses discussion on relationships between requirements and component-based systems architectures. They use requirement, sequence and block diagrams to represent systems requirements, components behaviors, and systems architectures, respectively. Atomic requirements are one by one extracted from the requirement diagrams to incrementally build an architecture of the system relying on components libraries. Model checking enables verification of atomic components modeled in Promela [20] against properties expressed in the form of LTL formulas.

In [38] Xie, Tan, Yang, Li, Xing and Huang present an integrated SysML modelling and verification approach where compositional verification is used to verify the nominal behaviour of the SysML model and FTA (Fault Tree Analysis) is used for safety analysis. SysML is extended with contract information. SysML models are transformed into OCRA specifications.

6.3 Model Mutation

Alterations of formal models are commonly called mutations [36]. Model mutations are particularly used for model-based testing purposes: for instance, Aichernig et al. [2] present a method where a large set of mutations is applied to a model of a system in order to detect the implementation mistakes that can invalidate the specification of the system. Model mutations can also be used in a security impact assessment context, since a vulnerability disclosure, an attack or a countermeasure deployment on a given system can be modeled with a mutation of the system's model: a vulnerability discovery leads to a change in the knowledge we have of the system, and an attack or a countermeasure leads to a change in the system itself. In particular, the W-Sec method [32] relies on SysML models for assessing the (positive or negative) impacts of security countermeasures. The approach introduced in this paper can therefore help in reducing the complexity of the model-checking stages of testing and impact assessment methods for SysML models.

6.4 Differential Positioning of the Contribution

With respect to Event-B, our contribution enables the utilization of a high-level language, SysML, to achieve greater depth in model mutations. Unlike refinements in Event-B, which are limited to refining existing actions into sub-actions or adding independent actions (i.e., actions not impacting previous ones), our approach facilitates the introduction of new behaviors into the model, whether these new behaviours are independent from the previous ones or not. Moreover, it supports incremental verification, ensuring that each addition or modification maintains the system's overall reachability properties.

Contrasting with [12], our approach achieves incrementality not only through the successive composition of SysML blocks, but also through internal modifications within these blocks (like adding attributes and signals), as well as in the associated state-machines (including the addition of states and transitions). Additionally, our model-checking algorithms operate directly on the SysML models, bypassing the need for an intermediate formal language like Promela or OCRA as in [38].

Lastly, in comparison to the methodology presented in our previous conference paper [9], this article introduces a fundamentally new algorithmic strategy. Instead of depending on a mere dependency graphs reduction, we now employ a novel graph splitting approach. This approach is further enhanced by a new labeling and propagation algorithm, which we apply at multiple stages, resulting in a more cohesive and integrated algorithmic framework.

7 Conclusions

Apvrille et al. [8] demonstrated an enhancement in the performance of a SysML model checker by computing a dependency graph of the SysML models prior to applying model checking. Building on this, the present paper advances this first idea by establishing a theoretical foundation for the incremental verification for SysML models, thereby fostering the effectiveness of agile design methods.

The new algorithms of the paper can decide how a reachability property proved on a model before an additive mutation can be proven on the mutated model without the need for exhaustive examination of the entire mutated model. The fundamental concept revolves around discerning the influence of new execution paths on their predecessors, and evaluating how these novel paths could present additional execution paths potentially capable of affirming or negating a property. An autonomous automotive system's real-time communication architecture serves as a case

study to illustrate this complex issue. A discussion draws promising perspectives.

Our vision for future work, which has been partially outlined in the discussion subsection, is multifaceted. A paramount aspect of this vision is optimization, with the aim of ensuring that our algorithms compute more swiftly than conventional model-checking approaches. While we plan to better classify and pinpoint systems where this advancement is most beneficial, preliminary explorations on several complex systems have already revealed significant reductions in verification time for random additive mutations.

Expanding our approach to accommodate liveness and, more broadly, CTL properties is also part of our current work. For liveness, preliminary algorithms are already in development.

Despite incremental modeling primarily involving the introduction of new details, it occasionally necessitates the removal of features that have become redundant or deprecated. At present, our algorithms cannot process the elimination of modeling elements, a challenge which our theoretical framework addresses by extending the concept of local liveness.

Finally, while the current contribution is exclusively focused on safety properties, performance properties and security properties such as confidentiality, integrity, and authenticity can also be influenced by mutations. Our future endeavors will aim to address these properties, thereby facilitating an agile integration of potentially opposing safety/security countermeasures. Incorporating in a safe way such mechanisms is a crucial facet for system architects, both during design and maintenance phases.

Declarations

On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

[1] Abrial JR (2010) Modeling in Event-B: System and Software Engineering, 1st edn. Cambridge University Press, USA

[2] Aichernig BK, Lorber F, Ničković D (2013) Time for mutants—model-based mutation testing with timed automata. In: International Conference on Tests and Proofs, Springer, pp 20–38

[3] Ali S (2018) Formal verification of SysML diagram using case studies of real-time system. Innovations in Systems and Software Engineering 14(6):245–262. https://doi.org/10.1007/s11334-018-0318-5

[4] Alparslan O, Arakawa S, Murata M (2023) A zone-based optical intra-vehicle backbone network architecture with dynamic slot scheduling. Optical Switching and Networking p 100753

[5] Ando T, Yatsu H, Kong W, et al (2013) Formalization and model checking of SysML state machine diagrams by csp#. In: Computational Science and Its Applications (ICCSA), p 114–127, https://doi.org/10.1007/978-3-642-39646-5_9

[6] Apvrille L, de Saqui-Sannes P (2013) Analysis Techniques to Verify Mutual Exclusion Situations within SysML Models. In: SDL 2013: Model-Driven Dependability Engineering. SDL 2013. Lecture Notes in Computer Science, vol 7916. Springer, Berlin, Heidelberg, https://doi.org/10.1007/978-3-642-38911-5_6

[7] Apvrille L, Courtiat JP, Lohr C, et al (2004) TURTLE: A real-time UML profile supported by a formal validation toolkit. IEEE Transactions on Software Engineering 30(7):473–487

[8] Apvrille L, de Saqui-Sannes P, Hotescu O, et al (2022) SysML Models Verification Relying on Dependency Graphs. In: 10th International Conference on Model-Driven Engineering and Software Development, Vienna, Austria, https://doi.org/10.5220/0010792900003119, URL https://hal.telecom-paris.fr/hal-03575960

[9] Apvrille L, Sultan B, Hotescu O, et al (2023) Mutation of Formally Verified SysML Models. In: 11th internationl conference on Model-Based Software and Systems Engineering (Modelsward'2023)

[10] Bougacha R, Laleau R, Collart-Dutilleul S, et al (2022) Extending SysML with Refinement and Decomposition Mechanisms to Generate Event-B Specifications. In: TASE 2022: Theoretical Aspects of Software Engineering, Lecture Notes in Computer Science, vol 13299. Springer, pp 256–273, https://doi.org/10.1007/

978-3-540-88387-6

[11] Calvino AT, Apvrille L (2021) Direct model-checking of SysML models. In: Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (Modelsward'2021), Vienna, Autrichia (online)

[12] Carrillo O, Chouali S, Mountassir H (2014) Incremental Modeling of System Architecture Satisfying SysML Functional Requirements. In: Fiadeiro JL, Liu Z, Xue J (eds) Formal Aspects of Component Software (FACS 2013. Springer, Lecture Notes in Computer Science, pp 79–99

[13] Delatour J, Paludetto M (1998) UML/PNO: A way to merge UML and Petri net objects for the analysis of real-time systems. In: Oriented Technology: ECOOP'98 Workshop Reader, p 511–514, https://doi.org/10.1007/3-540-49255-0_169

[14] Fisman D, Pnueli A (2001) Beyond regular model checking. 21st conference on Foundations of Software Technology and Theoretical Computer Science LNCS 2245

[15] Horváth B, Molnár V, Graics B, et al (2023) Pragmatic verification and validation of industrial executable sysml models. Systems Engineering 1(22):1–22. https://doi.org/10.1002/sys21679

[16] Huang E, McGinnis L, Mitchell S (2019) Verifying sysml activity diagrams using formal transformation to Petri nets. Systems Engineering 23(1):118–135

[17] IEEE (2018) 802.1Q - IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks. " https://standardsieeeorg/standard/802 1Q-2018html

[18] Kausch1 M, Pfeiffer1, Raco1 D, et al (2021) Model-based design of correct safety-critical systems using dataflow languages on the example of SysML architecture and behavior diagrams. In: AVIOSE'2021, Software Engineering 2021 Satellite Events, Bonn, Germany (virtual), Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, pp 1–22

[19] Mahani M, Rizzo D, Paredis C, et al (2021) Automatic formal verification of SysML state machine diagrams for vehicular control system. SAE Technical Paper https://doi.org/doi.org/10.4271/2021-01-0260

[20] Neumann R (2014) Using Promela in a Fully Verified Executable LTL Model Checker. In: Working Conference on Verified Software: Theories, Tools, and Experiments, Springer, pp 105–114

[21] OMG (2017) OMG Systems Modeling Language. Object Management Group, https://www.omg.org/spec/SysML/1.5

[22] Ouchani S, Ait Mohamed O, Debbabi M (2014) A formal verification framework for SysML activity diagrams. Expert Systems with Applications 41(6). https://doi.org/10.1016/j.eswa.2013.10.064

[23] Park C, Park S (2023) Performance evaluation of zone-based in-vehicle network architecture for autonomous vehicles. Sensors 23(2):669

[24] Rahim M, Hammad A, Ioualalen M (2017) A methodology for verifying SysML requirements using activity diagrams. Innovations in Systems and Software Engineering 13:19–33

[25] Rahim M, Boukala-Loualalen M, Hammad A (2020) Hierarchical colored Petri nets for the verification of SysML designs - activity-based slicing approach. In: 4th Conf. on Computing Systems and Appli. (CSA 2020), Algiers, Algeria, pp 131–142, URL https://publiweb.femto-st.fr/tntnet/entries/17274/documents/author/data

[26] de Saqui-Sannes P, Vingerhoeds R, Apvrille L (2018) Early checking of SysML models applied to protocols. In: 12th International Conference on Modeling, Optimisation and Simulation (Mosim 2018), Toulouse, France, pp 1–8

[27] de Saqui-Sannes P, Apvrille L, Vingerhoeds RA (2021) Checking SysML Models against Safety and Security Properties. Journal of Aerospace Information Systems pp 1–13

[28] de Saqui-Sannes P, Vingerhoeds RA, Garion C, et al (2022) A taxonomy of MBSE

approaches by languages, tools and methods. IEEE Access 10:120936–120950. https://doi.org/10.1109/ACCESS.2022.3222387

[29] Schafer T, Knapp A, Merz S (2001) Model checking UML state machines and collaborations. Electronic Notes in Theoretical Computer Science 55:357–369. https://doi.org/10.1016/S1571-0661(04)00262-2

[30] Rey de Souza FG, Hirata CM, Nadjm-Tehrani S (2022) Synthesis of a controller algorithm for safety-critical systems. IEEE Access 10:76351–76375. https://doi.org/10.1109/ACCESS.2022.3192436

[31] Staskal O, Simac J, Swayne L, et al (2022) Translating sysml activity diagrams for nuxmv verification of an autonomous pancreas. In: SESS22), pp 1–6

[32] Sultan B, Apvrille L, Jaillon P, et al (2021) W-Sec: a Model-Based Formal Method for Assessing the Impacts of Security Countermeasures. In: International Conference on Model-Driven Engineering and Software Development, Springer, pp 203–229

[33] Sultan B, Frénot L, Apvrille L, et al (2023) AMULET: a Mutation Language Enabling Automatic Enrichment of SysML Models. ACM Transactions on Embedded Computing Systems

[34] Szmuc W, Szmuc T (2018) Towards embedded systems formal verification translation from SysML into Petri nets. In: 25th International Conference Mixed Design of Integrated Circuits and System (MIXDES), pp 420–423, https://doi.org/10.23919/MIXDES.2018.843687

[35] TTool (2022) https://ttool.telecom-paris.fr/. Retrieved May 11, 2022

[36] Von Neumann J, Burks AW, et al (1966) Theory of self-reproducing automata. IEEE Transactions on Neural Networks 5(1):3–14

[37] Wang H, Zhong D, Zhao T, et al (2019) Integrating model checking with sysml in complex system safety analysis. IEEE Access 7:16561–16571. https://doi.org/10.1109/ACCESS.2019.2892745

[38] Xie J, Tan W, Yang Z, et al (2022) Sysml-based compositional verification and safety analysis for safety-critical cyber-physical systems. Connection Science 34(1):911–941. https://doi.org/10.1080/09540091.2021.2017853