Introduction
metrics. Section 5 surveys related work. Section 6 concludes the paper.

2 PROBLEM FORMALIZATION

The paper applies an enhanced verification scheme to the design stage of systems. A design stage is built upon a SysML block instance diagram and a set of SysML state machine diagrams: each block instance has a 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 (Sultan. et al., 2022) used to perform updates (Sultan. et al., 2022) used to perform updates to design diagrams. We consider five kinds of model mutations:

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

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

Function to add a block:

\[ \text{add} : \mathcal{D} \times \mathcal{B} \rightarrow \mathcal{D} \]

\[ (\mathcal{B}, \text{connect}, \text{assoc}, \mathcal{B}) \mapsto (\mathcal{B} \cup \{\mathcal{B}\}, \text{connect}, \text{assoc}) \]

2.1 Block Instance Diagram

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

- \( id \) is a String that names the block instance.
- \( A \) is an attribute list. The attribute types include Integer, Boolean, Timer, and user-defined Records. An attribute may be defined with an initial value.
- \( M \) is a set of methods.
- \( P \) is a set of ports.
- \( S_i \) and \( S_o \) are sets of input and output signals.
- \( \text{smd} \) is a state machine diagram.
- \( B_p \) represents the parent block to which \( B \) belongs. \( B_p \) is empty. We denote by \( S_p \) the set of all input signals of \( B \), by \( S_o \) the set of all output signals of \( B \) and by \( \mathcal{P} \) the set of all ports of \( B \).

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 \( D = \langle \mathcal{B}, \text{connect}, \text{assoc} \rangle \),

- \( \mathcal{B} \) is a set of block instances.
- \( \text{connect} \) is a function \( \mathcal{P} \times \mathcal{P} \rightarrow \{\text{No}, \text{synchronous}, \text{asynchronous}\} \) that returns the communication semantics between two ports (\( \varnothing \), synchronous or asynchronous).
- \( \text{assoc} \) is a function \( \mathcal{P}_{B_i} \times \mathcal{S}_i \times \mathcal{P}_{B_j} \times \mathcal{S}_j \rightarrow \text{Bool} \) 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 \( p_1, p_2 \) of respectively of \( B_1 \) and \( B_2 \), and if these two ports are connected (i.e. \( \text{connect}(p_1, p_2) \neq \text{No} \)).

2.2 State Machine Diagram

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

Definition: State Machine. A finite state machine depicted by a SysML state machine diagram is a bipartite graph \( \langle \mathcal{S}_0, \mathcal{T} \rangle \) where:

- \( \mathcal{S} \) is a set of states (\( \mathcal{S}_0 \) is the initial state).
- \( T \) is a set of transitions.

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

- \( s_{\text{start}} \) is the initial state of the transition.
- \( \text{after} \) specifies that the transition is enabled only after a duration between \( t_{\text{min}} \) and \( t_{\text{max}} \) has elapsed.
- \( \text{condition} \) is a Boolean expression that conditions the execution of the transition. This Boolean expression can use block attributes.
- \( \text{action} \in \{\text{variable affectation, send signal, receive signal}\} \) 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. \( \text{send signal, receive signal} \) can use its signals, or the signals of the parent block \( B_p \), and so on.
- \( s_{\text{end}} \) is the final state of the transition.

2.3 Design mutation

The current paper considers SysML model mutations (Sultan. et al., 2022) used to perform updates to design diagrams.
Function to add a port connection:

\[\text{add}_{\text{conn}} : \mathcal{D} \times \mathcal{P}^2 \to \mathcal{D} \]
\[
\begin{cases}
(B, \text{connect}, \text{assoc}, (p_1, p_2), \text{semantics}) \\
(B, \text{connect'}, \text{assoc}) & \text{if } (p_1, p_2) \in \mathcal{P} \\
(B, \text{connect}, \text{assoc}) & \text{otherwise}
\end{cases}
\]

where connect and connect' are such that \(\text{connect}(p_1, p_2) = \text{no}\) and \(\forall (p, p') \in \mathcal{P} \setminus \{(p_1, p_2)\}, \text{connect}(p, p') = \text{connect}(p, p') \land \text{connect}(p_1, p_2) = \text{semantics}'.

Function to add a signal association:

\[\text{add}_{\text{assoc}} : \mathcal{D} \times \mathcal{P}^2 \times \mathcal{S}_o \times \mathcal{S}_i \to \mathcal{D} \]
\[
\begin{cases}
(B, \text{connect}, \text{assoc}, (p_1, p_2), s_0, s_1) \\
(B, \text{connect}, \text{assoc'}) & \text{if } (p_1, p_2) \in \mathcal{P} \\
(B, \text{connect}, \text{assoc}) & \text{otherwise}
\end{cases}
\]

where assoc and assoc' are such that \(\text{assoc}(p_1, p_2, s_0, s_1) = \text{false}\) and \(\forall (p, p', s, s') \in \mathcal{P}^2 \times \mathcal{S}_o \times \mathcal{S}_i \setminus \{(p_1, p_2, s_0, s_1)\}, \text{assoc}(p, p', s, s') = \text{assoc}(p, s, p', s') \land \text{assoc}(p_1, s_0, p_2, s_1) = \text{true}.

Function to add a state:

\[\text{add}_{\text{state}} : \mathcal{D} \times \mathcal{S} \to \mathcal{D} \]
\[
\begin{cases}
(id, A, M, P, S_i, S_o, \langle s, 0, S, T \rangle, B_p), s \\
(id, A, M, P, S_i, S_o, \langle s, 0, S, 0 \rangle, B_p) & \text{otherwise}
\end{cases}
\]

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

\[\text{parents} : B \to \{\emptyset\} \cup \text{parents}(B) \]

where \(B\) is a given block instance \(B = (id, A, M, P, S_i, S_o, \langle s, 0, S, T \rangle, B_p)\), we denote with:

- \(S_i^+ = S_i \cup \bigcup_{\text{Block}\in \text{parents}(B)} S_{\text{block}}\)
- \(S_o^+ = S_o \cup \bigcup_{\text{Block}\in \text{parents}(B)} S_{\text{block}}\)

where \(S_{\text{block}}\) (resp. \(S_{\text{block}}\)) is the input (resp. output) signals set of \(\text{Block}\).

\[\mathcal{B}_{\text{after}} \]

the subset of \(\mathcal{B}\) such that \(\forall (s_{\text{start}}, \text{after}, \text{condition}, \text{Actions}, s_{\text{end}}) \in \mathcal{B}_{\text{after}}\), \(s_{\text{start}}, s_{\text{end}} \in \mathcal{S}^2 \wedge \text{condition is an expression over elements of } A \land \text{Actions}\) contains only variable affectation over elements of \(A\), receive signal actions over elements of \(S_i^+\) and send signal actions over elements of \(S_o^+\).

\[1\] \(\mathcal{P}\) is the set of all ports of \(B\) such as defined herein above.

\[2\] \(S_o\) (resp. \(S_i\)) is the set of all output (resp. input) signals of \(B\).

Function to add a transition:

\[\text{add}_{\text{trans}} : \mathcal{B} \times \mathcal{S} \to \mathcal{B} \]
\[
\begin{cases}
(id, A, M, P, S_i, S_o, \langle s, 0, S, T \rangle, B_p), t \\
(id, A, M, P, S_i, S_o, \langle s, 0, S, T \cup \{\{\}, \rangle, B_p) & \text{otherwise}
\end{cases}
\]

In the next sections, we will denote with \(D_1 = (\mathcal{B}_1, \text{connect}_1, \text{assoc}_1)\) the initial design and with \(D_M = (\mathcal{B}_M, \text{connect}_M, \text{assoc}_M)\) a mutated design, i.e., \(D_M\) derives from \(D_1\) through a mutation or a composition of mutations among \(\{\text{add}_{\text{Block}}, \text{add}_{\text{Conn}}, \text{add}_{\text{Assoc}}, \text{add}_{\text{State}}, \text{add}_{\text{Trans}}\}\).

2.4 Reachability

Let \(o\) be a state or an send/receive action of a state machine of \(D\). \(E <> o\) denotes the reachability of \(o\) that is \(o\) is executed in at least one execution path. We assume that \(D_1 |\!\!\|= E <> o\), i.e., the operator \(o\) is reachable in \(D_1\). \(\models\) symbol means “satisfies”.

Problem 1. Instead of reproving if \(D_M |\!\!\|= E <> o\) using model-checking techniques applied to \(D_M\) and \(E <> o\), could we reuse the result \(D_1 |\!\!\|= E <> o\) and \(D_M = m(D_1)\) to lower the complexity of the proof of \(E <> o\) on \(D_M\)?

3 CONTRIBUTIONS

3.1 Overview

This section defines how a model update can impact reachability properties, and what exactly has to be proved when performing an update to ensure that reachability properties proved at stage \(n\) are still valid at stage \(n + 1\). For this, this section introduces the notion of Dependency Graphs that are useful to simplify the computation of logical paths in models. The interest of using dependency graphs built from SysML models has already been discussed by (Apvrille et al., 2022): we reuse dependency graphs in a slightly different scope. Thus, after presenting dependency graphs, the section focuses on the main algorithms to simplify proofs of reachability properties after model mutation. A discussion on optimization, complexity and liveness ends this section.

3.2 Dependency graphs

As explained by Apvrille et al. in (Apvrille et al., 2022), a dependency graph can be built from a
SysML model. This graph features the communication between blocks (synchronous and asynchronous communication) as well as all logical dependencies of the state machines (i.e. state to transition, transition to states, actions including communication actions). All these SysML elements have a corresponding vertex in the graph so that it is possible to rebuild the original SysML 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 or transitions.

3.3 Reachability properties of mutated designs

Since the mutations we study in this paper cannot remove existing dependencies (they can also extend the model, not suppress model elements), the only case for which a reachability property for an operator \( o \) would not be satisfied anymore is the case where a new dependency would prevent all dependency paths to \( o \) from being executable. Thus, the main idea is to compute the new dependencies induced by the mutations, and then to perform a model-checking on a (hopefully very) reduced system: the one containing only the new dependencies plus all the first elements of the dependency paths that necessarily lead to \( o \) and then has no forward mutation. So, if at least one of them is still reachable, then the reachability property is still valid on the new system.

Let us illustrate our approach on a toy system. Since this is a toy system, using this reduction technique brings only trivial benefits: the idea is not to illustrate the gain in performance but rather to illustrate the principle.

Basically, the toy system has two senders (\( S1, S2 \)) that attempt to synchronize via the \( send \) signal with the \( recv \) signal of the \( Receiver \) block (see Figure 1). This synchronisation is delayed in the state machines of senders and receivers with an after clause, as illustrated in Figure 2. The \( Other \) block does not synchronize with the other blocks: it only waits for a given delay before stopping. As illustrated by the \( RL \) in green, Figure 2, both the reachability and liveness of \( End1 \) are satisfied. We use CTL to denote the reachability of an element: \( D_f \models E < > Receiver.End1 \) means that the \( End1 \) state of the SysML block \( Receiver \) is reachable in \( D_f \).

Let us now apply a mutation to this toy system. We add a new behaviour to \( Receiver \): we add a new state called \( End3 \) and we add a transition from state \( Waiting \) to state \( End3 \) with an after clause \( after(2) \) (Figure 3).

Now, to study if \( End1 \) is still reachable after the mutation, we first need to generate the dependency graph of the mutated design (Figure 4). This graph illustrates all logical dependencies of the mutated design. The green states represent start states of the state machines. The red states represent end states, and other states represent other elements (states, transitions, and sending/receiving actions). The top right part of the graph represents the \( Other \) block. The middle top part represents the dependencies of \( S1 \), while \( S2 \)'s dependencies are depicted in the left top part. \( Receiver \) is displayed in the middle and bottom of the graph. One can notice the two read/write dependencies between \( S1/Receiver \) and \( S2/Receiver \) leading to two possible execution paths to reach the \( End1 \) state. We have also rounded in brown the new dependencies due to the mutation.

Now, let us apply our approach to figure out how
to prove whether the reachability of End\(1\) still holds after the mutation or not. Since we know that End\(1\) is reachable in \(D_I\), we know that one of the logical paths between Receiver/start (i.e., the start state of the state machine of Receiver 3, displayed as vertex "3" in the dependency graph) and Receiver/End1 (vertex "9") is executable. Let us consider these two paths, and their intersection with the path from Receiver/End3 to Receiver/state: this intersection is at state Receiver/Waiting (vertex "4"). Thus, to know if End\(1\) is still reachable, we need to know if from Receiver/Waiting, it is still possible to use one of the paths leading to End\(1\): since these two paths have to go through vertex 7 first, if vertex 7 is still reachable, then End\(1\) is still reachable. Otherwise, End\(1\) is not reachable anymore. So, we can reduce the graph to all new paths and all paths leading to vertex 7: this reduced graph is given in Figure 5.

From this reduced graph, we can easily reconstruct a reduced design \(D_R\) (illustrated in Figure 6) from which the reachability of state 'Transition' must now be studied using a model-checker. Obviously, since the \(D_R\) model is less complex than \(D_M\), we expect that studying the reachability of state 'Transition' is much faster than studying the reachability of state End\(1\) in model \(D_M\). Since the reachability of 'Transition' is not satisfied, we can deduce that \(D_M \models E \leftrightarrow \) Receiver.End\(1\) is false. Finally, the approach consists in cutting as much as possible the paths in \(D_M\) that have already been explored when proving the property in \(D_I\) so as to obtain a reduced model on which a simplified proof can be performed. Note that the algorithm does not need to re-display the reduced SysML model: this is only to better illustrate how the model has been cut to reduce the proof complexity.

### 3.4 Formalization

Models and dependency graphs are two different representations of the same information. Thus in the sequel we only consider dependency graphs (denoted by "\(DG\)"), which abstracts explicit conversions used in algorithms (i.e., we assume we have \(Model \equiv graphToModel(modelToGraph(Model))\)).

Algorithm 1 decides if a set \(Prop\) of reachability properties that have been proved in the initial model \(DG_I\) are preserved after mutation, i.e. in the mutated model \(DG_M\).

For each reachability property \(p\) (with associated vertex \(v_p \in DG_I\)), the algorithm first reduces the graph with \(reduceGraph\), which returns the reduced graph \(DG_R\), a new set \(P_R\) of reachability properties to be proven, and a boolean \(P\) which is true if \(p\) must be re-proven on the other returned graph, \(DG_P\), using paths to \(v_p\) that are new in \(DG_M\) w.r.t. \(DG_I\). We first assume that the property is not satisfied anymore. We...
Runnable paths are not preserved.

- **Algorithm 1:** Simplifying reachability proofs after mutation

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

  1. foreach $p \in$ Prop do
     2. $DG_R, P_R, DG_p = reduceGraph(DG_I, DG_M, v_p)$
     3. res($p$) = $P \land \text{prover}(DG_p, p)$
  4. if not res($p$) then
     5. foreach $p_r \in P_R$ do
     6. result$_{p_r} = \text{prover}(DG_R, p_r)$
     7. if result$_{p_r} == \text{true}$ then
     8. res($p$) = true
     9. break
  10. end

Algorithm 2 implements the graph reduction. In this simplified presentation, graphs are handled as sets containing both vertices and edges. Some explicit functions and data are not detailed in the scope of this paper. Figure 7 illustrates some of the corresponding concepts.

- **Path**($DG, V$) computes all paths from start vertices of to vertices of $V$ in graph $DG$.
- $v(X)$ is the set of vertices in Path($X, v$).
- Modified denotes the set of vertices – corresponding to states in state machines – from which a new transition has been added with mutations.

**addEdges**($DG, DG'$) completes $DG$ with all edges of $DG'$ between vertices common to $DG$ and $DG'$.

- **next**($V, DG$) denotes the set of all direct successors of any vertex $v \in V$ in $DG$.

- **last**($w, DG_M, DG_I, Modified$) is the first vertex in path $w$ that has no (direct or indirect) successor $v$ w.r.t. $DG$ verifying $v \in Modified$ or $v$ is connected with a vertex of another path of $DG_I$ that contains modified nodes: this connection corresponds to a synchronous or asynchronous communication between two blocks that may be compromised by mutations.

- **prefix**($w, DG_M, DG_I, Modified$) is the prefix of $w$ that ends at **last**($w, DG, Modified$) (without containing it).

- **runGraph**($DG, v_p$) (illustrated in Figure 8) is the Reachable Dependency Graph $RDG$ that only contains paths of $DG$ that lead from start vertices to $v_p$ and are "executable" (i.e. $v_p$ is actually reachable w.r.t. these paths).

In Algorithm 2, graphs $DG_R, DG_p$ and properties $P_R$ are build from empty sets. Vertices are added first, and edges are added at the end.

1. **Lines 1-7.** New paths to $v_p$ in graph $DG_M$ are computed (and added to $W_{new}$): indeed, we need to evaluate if $v_p$ could be reachable through these paths. Thus, if these paths are non empty, they are added to $DG_G$ with all the immediate successors (next) of their vertices. So, only transitions that may prevent the path to be executed are kept.

2. **Lines 8** We compute executable paths able to reach $v_p$ in $DG_I$.

3. **Line 9-15** We then identify the shortest prefixes ($W_{prev}$) of these paths that cannot lead to any modified vertices or sensible connection any more. The idea is that if the remaining suffix (with first vertex $v_{last}$) is reachable then $p$ is reachable, as this suffix has already been tested on $DG_I$. Thus, we add the reachability of $v_{last}$ in $P_R$, and the prefixes (and their "next") in $DG_G$ (lines 12-13).
However, these vertices may also be reachable from new paths with different values for variables, which could compromise the execution of the suffix. When such a risk exists (line 14), we require to re-test the reachability of $v_p$ and thus add the prefix to $DG_p$ (line 15).

4. **Line 18-21** We complete both graphs with the vertices of all paths leading to their current vertices: if there is a synchronization needed for a path to execute, the vertices of the synchronized block need to be added so that the synchronization can occur. Finally, we add corresponding edges.

---

### Algorithm 2: Reducing graph to a given reachability property $p$

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$DG_R, P_R, P, DG_p = \emptyset, \emptyset, \text{false}, \emptyset$</td>
<td></td>
</tr>
<tr>
<td>$W_{dg} = \text{Paths}(DG_I, v_p)$</td>
<td></td>
</tr>
<tr>
<td>$W_{dgm} = \text{Paths}(DG_M, v_p)$</td>
<td></td>
</tr>
<tr>
<td>$W_{newp} = W_{dgm} \setminus W_{dg}$</td>
<td></td>
</tr>
<tr>
<td>if $W_{newp} \neq \emptyset$ then</td>
<td></td>
</tr>
<tr>
<td>$DG_p = v(W_{newp}) \cup \text{next}(v(W_{newp}), DG_M)$</td>
<td></td>
</tr>
<tr>
<td>$P = \text{true}$</td>
<td></td>
</tr>
<tr>
<td>$W_{RDG_I} = \text{Paths}(\text{runGraph}(DG_I, v_p), v_p)$</td>
<td></td>
</tr>
<tr>
<td>foreach $w \in W_{RDG_I}$, $v_{last} =$</td>
<td></td>
</tr>
<tr>
<td>$v_{pre} = \text{prefix}(w, DG_M, DG_I, \text{Modified})$ do</td>
<td></td>
</tr>
<tr>
<td>if $v_{last} \not\in v(W_{newp})$ then</td>
<td></td>
</tr>
<tr>
<td>$DG_R = DG_R \cup v(W_{pre}) \cup \text{next}(v(W_{pre}), DG_M)$</td>
<td></td>
</tr>
<tr>
<td>$P_R = P_R \cup {E &lt; v_{last}}$</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>$DG_p = DG_p \cup v(w) \cup \text{next}(v(w), DG_M)$</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>$DG_R = DG_R \cup \text{Paths}(DG_M, v(DG_R))$</td>
<td></td>
</tr>
<tr>
<td>addEdges($DG_R, DG_M$)</td>
<td></td>
</tr>
<tr>
<td>$DG_p = DG_p \cup \text{Paths}(DG_M, v(DG_p))$</td>
<td></td>
</tr>
<tr>
<td>addEdges($DG_p, DG_M$)</td>
<td></td>
</tr>
</tbody>
</table>

---

### 3.5 Discussion

#### 3.5.1 Optimization

The algorithm has been written to cover all possible mutations integrated at whatever position in the initial dependency graph. When there are no mutations on all paths from the start states to a model element $e$, the mutation has no impact on the reachability of $e$: this is a property that would be easy to check at first, i.e., before our algorithm is run. But again, our algorithm covers this case, but not in an explicit way and it needs to compute reachable paths first.

#### 3.5.2 Complexity

The complexity of our algorithm depends on many factors: the size of the dependency graph, the size of the reachability graph, the number of paths, the number of mutations, how mutations impact the dependency graph, and so on. Our approach assumes that the Dependency Graph (i.e., the model) is much smaller than the reachability graph of the model.

As a future work, we suggest to evaluate the algorithm, from a performance point of view, as follows: generate a huge random number of models and of mutations, then randomly select accessibility properties in these models, and apply our algorithm to these systems, and finally compute the min, average and maximal execution time with regards to the size of the model and the number of mutations.

Obviously, after too many mutations, or in some corner cases yet to be defined, directly re-proving the reachability will be faster than applying our algorithm. Indeed, more mutations mean more paths to compute and evaluate.

#### 3.5.3 Liveness

While this paper focuses on the reachability property, 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 explain how liveness could be taken into account in an incremental way.

---

Figure 9: Proving liveness after model mutation

Figure 9 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 Figure 9), thus leading to $RDG$. Let us now assume...
that mutations are performed on the initial 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 was examined for \( DG_I \). On the contrary, other paths are to be re-evaluated. For this, we compute the next elements \( n \) (rounded in green on 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 if \( n \) 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 9) is reachable, the 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.

4 CASE STUDY

This section illustrates the paper’s contribution in the scope of a complex case study: an industrial Ethernet-based Time-Sensitive Networking (TSN).

4.1 Time-Sensitive Networking

Time-Sensitive Networking (TSN) (IEEE, 2018) is a set of standards defined by IEEE 802.1 Working Group to provide deterministic services through IEEE 802 Ethernet networks, i.e. guaranteed packet transport with bounded low latency, low packet delay variation, and low packet loss. Modern embedded systems and cyber-physical systems, such as safety-critical industrial, automotive and avionics networks require deterministic real-time communication.

A TSN network is built upon transmitting End Systems (Tx ES) sending flows with a given priority and a period in a network composed of switches (SW), with a predefined network path for each flow, and multiple routes handled by frame replication and elimination. Each end system has a network interface interconnected with communication switches via full-duplex physical links (Figure 10). A network path finishes with one receiving End System (Rx ES). The network supports unicast and multicast communications between a set of applications distributed on different end systems.

In the past few years, several research work on Time-Sensitive networking has relied on model-based approaches to formally verify properties of the network. In (Guo et al., 2021; Lv et al., 2020), the UP-PAAI model checker is used for timing analysis of TSN, while in (Samson et al., 2022; Farzaneh et al., 2017), network models described in MARTE, respectively EMF, are proposed to serve for automatic generation of TSN network configurations. By reducing the verification time of TSN models, the current contribution facilitates the impact of the addition of new flows or SW in an existing TSN network.

4.2 TSN model in SysML

In our case study, we propose several TSN network designs models differing in few mutations (i.e., new components or behaviours are added from a design to another) for which we need to verify reachability properties with the model checker, e.g., the receiving of a packet is still reachable. These models are intended to be used as a decision helper for dimensioning TSN networks for safety-critical applications. We illustrate different kinds of mutations and show how our algorithm consider these situations to ease property proving.

4.3 TSN Systems, mutations and performance evaluation

We consider the SysML models of the systems listed in Table 1. Models contain more than 20 blocks and complex state machines (the dependency graph contains around 500 vertices corresponding to around 20 blocks and hundred of states and transitions). Performance are given for the system before mutation and after mutation. After mutation, we show timings without our approach (no reduction) and with our approach (with reduction), to indicate the benefits of our method. The time with reduction includes the time to generate \( DG_I, DG_M \) and the proof time for elements.
of \( P \), but not the time to compute \( P \) nor \( DG_{\text{R}} \). The performance is computed with the latest version of TTool (TTool, 2022) (de Saqui-Sannes et al., 2021), using the internal model-checker of TTool (Calvino and Apvrille, 2021), and using Oracle Java 11, and a MacOS laptop with an i7 processor running at 2.3 GHz. For each performance evaluation, we run the proof 10 times and take the lowest value.

In the first evaluation, we start from System 1, then we combine System 1 and System 2 into two independent networks. This case is obviously favorable to our contribution since \( P \) contains only the reachability of the beginning of the Tx ES blocks since there is no forward divergence in the dependency graph path. The proof is therefore straightforward. On the contrary, the proof considering the whole system is much longer. The generation of the dependency graph after mutation takes around 5 ms and must be done only once for the three reachability evaluations.

In the second evaluation, we add to System 2 a new flow. Then, in the third evaluation, we add a second mutation to System 2 + previous mutation (also another flow). In both cases, our approach decreases the proof time. Also, dependency graphs are of much smaller size than the reachability graph, which was part of our assumption for definition of our reduction algorithm.

In the three investigated mutations, important additions have been performed on the model: adding many blocks in the case of the first mutation, adding several states and transitions in the case of the second and third mutations.

As a result, we can conclude that Algorithm 1 decreases the proof time for the considered mutations. Since proving reachability mutations is quite fast, the gain is not so important but, if we assume that the reachability shall be proven for all system states, then the total gain would be much higher. Yet, the approach is promising and we do hope to apply the same principle for properties which are much more com-

\[\begin{array}{|c|c|c|c|c|c|}
\hline
\text{Reachability} & \text{States/} & \text{Proof} & \text{Mutation} & \text{States} & \text{Proof} \\
\text{} & \text{Transi-} & \text{time} & \text{(ms)} & \text{Transitions} & \text{time} \\
\text{} & \text{tions} & \text{(ms)} & \text{and \/} & \text{(ms)} & \text{and \/} \\
\text{} & \text{} & \text{} & \text{edges} & \text{} & \text{} \\
\hline
\text{DG:} & \text{time to generate} & \text{vertices/} & \text{DG:} & \text{time to generate} & \text{vertices/} \\
\text{vertices/} & \text{edges/} & \text{DG:} & \text{vertices/} & \text{edges/} & \text{DG:} \\
\text{transitions/} & \text{edges/} & \text{time} & \text{transitions/} & \text{edges/} & \text{time} \\
\text{transitions} & \text{transitions} & \text{(ms)} & \text{transitions} & \text{transitions} & \text{(ms)} \\
\text{} & \text{} & \text{(ms)} & \text{} & \text{} & \text{(ms)} \\
\text{} & \text{} & \text{no red.} & \text{} & \text{no red.} & \text{} \\
\hline
\text{System 1:} & \text{1 Tx ES, 2SW, 1 Rx ES, 2 flows} & \text{1763/3238} & \text{16} & \text{Adding System 2} & \text{13k/50k} \\
\text{Full reachability} & \text{graph generation} & \text{16} & \text{(30 blocks)} & \text{1212} & \text{240126} \\
\text{Receiving \ a} & \text{227} & \text{2} \\
\text{packet in flow 0} & \text{227} & \text{2} \\
\text{Receiving \ a} & \text{231} & \text{2} \\
\text{packet in flow 2} & \text{231} & \text{2} \\
\text{Packet received \ in SW#2} & \text{232} & \text{2} \\
\text{System 2:} & \text{2 Tx ES, 3SW, 1 Rx ES, 4 flows} & \text{80k/200k} & \text{292} & \text{Adding a flow} & \text{300k/677k} \\
\text{Full reachability} & \text{graph generation} & \text{292} & \text{sent by Tx ES#1} & \text{11170} & \text{389/594/2ms} \\
\text{Receiving \ a} & \text{11} & \text{7} \\
\text{packet in flow 3} & \text{11} & \text{7} \\
\text{Receiving \ a} & \text{10} & \text{5} \\
\text{packet in flow 0} & \text{10} & \text{5} \\
\text{Packet received \ in SW#3} & \text{10} & \text{4} \\
\text{Full reachability} & \text{Applies to previous \ mutation} & \text{1.4k/3.4k} & \text{4983} & \text{395/605/2ms} \\
\text{graph generation} & \text{} & \text{} & \text{} & \text{} \\
\text{Receiving a} & \text{12} & \text{4} \\
\text{packet in flow 3} & \text{12} & \text{4} \\
\text{Receiving a} & \text{14} & \text{4} \\
\text{packet in flow 0} & \text{14} & \text{4} \\
\text{Packet received \ in SW#3} & \text{14} & \text{4} \\
\hline
\end{array}\]

Table 1: Systems, mutations and performance results
plex to evaluate, e.g., liveness properties.

5 RELATED WORK

5.1 Formal Verification of SysML Models

A survey of the literature indicates that formal verification has been applied to SysML activity diagrams (Ouchani et al., 2014; Huang et al., 2019; Staskal et al., 2022) and state machine diagrams (Delatour and Paludetto, 1998; Schafer et al., 2001; Aprville et al., 2004), 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 eat an external and preexisting formal verification tool. Examples include Petri nets (Delatour and Paludetto, 1998; Szmuc and Szmuc, 2018; Huang et al., 2019; Rahim et al., 2020), automata for NuSMV model checker (Wang et al., 2019), timed automata (Schafer et al., 2001) for UPPAAL model checker, hybrid automata (Ali, 2018), model checker NuSMV (Mahani et al., 2021), probabilistic model checker PRISM (Ouchani et al., 2014; Ali, 2018), and a theorem prover (Kausch1 et al., 2021). Translation from UML to process algebra has been investigated for RT-LOTOS (Aprville et al., 2004) and CSP (Ando et al., 2013). The family of correct by construction specifications has been addressed with Event B (Bougacha et al., 2022).

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 (de Saqui-Sannes et al., 2021; Rey de Souza et al., 2022).

In terms of user friendliness, users of SysML verification tools are further concerned by verification results interpretation (Zoor et al., 2021). How to come back from verification results to the initial state machines is an issue. It is worth being noticed 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.

5.2 Incremental Modeling with SysML

In (Carrillo et al., 2014) 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 against properties expressed in the form of LTL formulas.

In (Xie et al., 2022) 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 safety analysis. SysML is extended with contract information. SysML models are transformed into OCRA specifications.

In (Bougacha et al., 2022) Bougacha, Laleau, Collart-Dutilleul and Ben Ayed translate SysML models into Event-B specifications, and reuse the refinement mechanisms of Event-B to formally verify the SysML models. The work in (Bougacha et al., 2022) follows a correct by construction approach. Conversely, the current paper develops an approach where each increment in models construction requires new application of model checking even though the amount of properties to be verified from an increment to next one is reduced thanks the mutation principles presented in the current paper.

5.3 Model Mutation

Alterations of formal models are commonly called mutations (Von Neumann et al., 1966). Model mutations are particularly used for model-based testing purposes: for instance, Aichernig et al. (Aichernig et al., 2013) 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 always 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 (Sultan et al., 2017). In particular, the W-Sec method (Sultan et al., 2022) 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 CONCLUSIONS

(Apvrille et al., 2022) has shown how the performance of a SysML model checker can be improved by computing a dependency graph of the SysML models before applying model checking to a reduced model. The current paper goes one step forward: it addresses agility in the context of SysML modeling. Indeed, the algorithm introduced in the current paper decides how a reachability property proved on a model before an addition mutation can be proven on the mutated model without having to consider the entire mutated model. The main principle is to identify how the new execution paths impact the former ones. A real-time communication architecture based on TSN (Time Sensitive Networking) serves as a case study to illustrate different mutations and shows how our algorithm performs.

Our vision of future work has already been partly covered in the discussion subsection. Optimization is obviously part of our future work to decrease the complexity: our contribution will increase in interest when multiple mutations will be taken into account. Handling liveness and more complex properties is also part of our future work. Also, addressing only addition mutation can be seen as a limit. Indeed, if incremental modeling mostly consists in adding new details, it does not exclude to remove features that are no longer necessary. Today, our algorithms cannot handle the removal of modeling elements: this is part of our future work. Last, the current contribution concerns only safety properties. Yet, performance (Zoor et al., 2021) and security properties (e.g., confidentiality, integrity, authenticity), as defined in SysML-Sec (Apvrille and Roudier, 2013), can also be impacted by mutations. We do intend to address these properties in the future.

REFERENCES


Apvrille, L. and Roudier, Y. (2013). Sysml-sec: A sysml environment for the design and development of secure embedded systems. In IEEE, editor, APCOSEC 2013, Asia-Pacific Council on Systems Engineering, September 8-11, 2013, Yokohama, Japan, Yokohama. © 2013 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.


