Enabling Incremental SysML Model Verification: Managing Variability and Complexity through Tagging and Model Reduction

Bastien Sultan [1][a], Ludovic Apvrille [1][b], Oana Hotescu [2][c] and Pierre de Saqui-Sannes [2][d]

[1] *LTCI, Télécom Paris, Institut Polytechnique de Paris, France*

[2] *Fédération ENAC ISAE-SUPAERO ONERA, Université de Toulouse, France*

Keywords: MBSE, SysML, Model Checking, Model Reduction, Software Product Lines

Abstract: Designing complex software systems with model-based approaches encounters the recognized state space explosion problem. Typically, only a subset of models can be formally verified, forcing reliance on simulation or testing to verify the entire system. Furthermore, most formal verification tools require a complete reevaluation of properties after even minor modifications to a model. Although incremental formal verification, particularly the incremental model-checking approach of TTool, has been proposed, it still requires modelers to manually select sub-models not facing state space explosion. Unfortunately, this manual model selection is susceptible to errors. This paper presents a twofold contribution to SysML models of software product lines. First, we introduce a SysML model tagging feature that enables designers to explicitly differentiate between various subsystems, such as core and optional features. Second, we develop and implement a model reduction algorithm using dependency graphs (DGs). This algorithm automatically deactivate model elements linked to specific tags, removing both the specified elements and all their logical dependencies provided the DG is acyclic. These two contributions are evaluated for their effectiveness in generating model variants. Together, they facilitate the creation of a core model and an associated set of models, each extended by additional model elements, and make it possible to rely on incremental model-checking. We have implemented the contributions in TTool and applied it to an integrated modular avionics system. This application enables to compare—both manual and automated—model reduction strategies and assess their benefits for TTool users.

1 Introduction

Over the past decade, systems engineering has switched from a document-based paradigm to a model-based paradigm. The acronym MBSE (Model-Based Software and Systems Engineering) was coined accordingly. The expected benefits of using MBSE are many (de Saqui-Sannes et al., 2022), and include early detection of design errors in the life cycle of systems. This early detection may be achieved using model-based formal verification techniques (Cederbladh et al., 2024). One of the well-established formal verification techniques is model-checking: the model of the system is checked against a set of properties and a "satisfied/unsatisfied" answer is returned for each property.

Since SysML (OMG, 2022) is a common modeling language employed in Model-Driven Engineering (MDE) and MBSE methodologies, various methods, algorithms, and tools have been developed to support formal verification of SysML models (Wang et al., 2019; Horváth et al., 2020; de Saqui-Sannes et al., 2021). Among these, TTool[1] integrates a model checker that enables for direct verification of CTL properties on SysML semantics (Calvino and Apvrille, 2021), unlike SysML tools that rely on external verification tools (Cederbladh et al., 2024). Model checking is however hindered by combinatorial explosion issues, due to the complexity of the models being verified. Verifying relevant sub-models (or sub-systems) is one way to circumvent combinatorial explosion. Incremental verification, as the one introduced in (Coudert et al., 2024), also made a step towards addressing this challenge. SysML model design can be seen as an incremental process and incremental model-checking proposes solutions to not entirely verify the complete set of model properties when an additive increment partly modifies the SysML model under design. Building on that work,

[a] https://orcid.org/0000-0002-5031-5794

[b] https://orcid.org/0000-0002-1167-4639

[c] https://orcid.org/0000-0001-6612-8574

[d] https://orcid.org/0000-0002-1404-0148

[1] https://ttool.telecom-paris.fr

the current paper presents a complementary solution to further address these issues. It considers systems designed with a core set of functions that can be augmented with optional features, a common practice in e.g., automotive systems (Hohl et al., 2018), and more broadly within software product lines. In this context, these different system versions are typically referred to as *variants* (Garmendia et al., 2024). Verifying models of such systems can be approached incrementally: first, by verifying the model's subpart that represent the core functions, and subsequently, by adding and verifying the optional functions one at a time (or several among all), and verifying if the properties of the core model are still satisfied by utilizing incremental model checking approaches that reuse the previous proof results (Cordy et al., 2012; Coudert et al., 2024). The paper introduces two contributions to support this incremental verification process:

1. A tagging feature for SysML models that enables model designers to easily classify the different model elements for various purpose: history, feature characteristics, core features *vs.* optional features, abstraction level—enabling the generation of multiple model variants with different degrees of abstraction from a single model, etc. In this paper, we demonstrate the interest of this tagging approach for differentiating core functions from optional features, facilitating the distinction between different system variants.
2. A model reduction algorithm based on dependency graphs (DG)—graphs we assume to be equivalent to SysML models as defined in the paper—that enables the deactivation of elements associated with specified input tags, including their logical dependencies in the model[2]. In complex models, performing this model reduction manually can be both tedious and error-prone. Indeed, some logical dependencies of a tagged element may span across multiple diagrams—typically, removing a tagged *send signal* operator in a state-machine diagram involves removing the related *receive signal* operators from any other state-machine diagrams where they are present.

These contributions have been integrated into the SysML modeling and verification toolkit TTool.

This paper is organized as follows. Section 2 defines the SysML diagrams used throughout the current paper. Section 3 presents the contributions of the current paper, namely SysML models tagging and the reduction algorithm. Section 4 evaluates our contributions on a case study: an avionics system. Section 5 surveys related work, and Section 6 concludes the paper.

2 SysML Background

This section introduces key definitions for a subset of SysML, borrowed from (Apvrille et al., 2021; Coudert et al., 2024), for the sake of self-containment.

Definition 1 (Block instance).

A block instance is a 7-tuple $B = \langle A, M, P, S_i, S_o, smd, B_p \rangle$ *where A is a set of integer or boolean* attributes*, 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, and* B_p *represents the parent block B belongs to.*

Each block instance contains one finite state machine.

Definition 2 (State machine).

A finite state machine is a directed graph $\langle s_0, S, T \rangle$ *where:*

- *S is a set of states (*$s_0 \in S$ *is the initial state).*
- *T is a set of transitions* $\langle s_{start}, after, condition, Actions, s_{end} \rangle$ *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* $true$*.*
  - s_{end} *is the final state of the transition.*

Definition 3 (Block instance diagram).

A block instance diagram *(or* SysML model*) is a 3-tuple* $\langle \mathcal{B}, connect, assoc \rangle$ *where:*

- $\mathcal{B}$ *is a set of block instances. We denote with* $\mathcal{P} = \bigcup\limits_{B \in \mathcal{B}} P_B$ *the set of all ports of* $\mathcal{B}$ *and with* $\mathcal{S}_o = \bigcup\limits_{B \in \mathcal{B}} S_{oB}$ *(resp.* $\mathcal{S}_i = \bigcup\limits_{B \in \mathcal{B}} S_{iB}$*) the set of all output (resp. input) signals of* $\mathcal{B}$*.*
- *connect is a function* $\mathcal{P} \times \mathcal{P} \rightarrow \{No, synchronous, asynchronous\}$ *that returns the communication semantics between two ports.*

[2] The algorithm ensures the deletion of *all* the logical dependencies if the DG is acyclic, and of all the *path dependencies*, as defined below, if the graph contains cycles.

- *assoc is a function $\mathcal{P} \times \mathcal{S}_o \times \mathcal{P} \times \mathcal{S}_i \to Bool$ 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. In addition, a signal cannot be associated with more that one port and one signal.*

3 Tagging and reducing SysML models

The current paper primarily introduces two new features into TTool: (1) the ability to manually assign tags to elements of a SysML model and (2) a one-click model reduction feature. The model reduction feature, given a set of input tags and a model, generates a new model that excludes elements associated with the specified tags and (at least part of) their logical dependencies. This reduction process relies on model-to-DG and graph-to-model generation algorithms already implemented in TTool (Apvrille et al., 2021), as well as a new graph reduction algorithm proposed in the paper.

3.1 Background on Dependency Graphs

The main contributions of the current paper are based on DGs (Apvrille et al., 2021). A DG is a directed graph (V_{DG}, E_{DG}) constructed from a SysML model as defined in Definition 3. In this graph, the vertices V_{DG} represent the behavioral elements of the SysML model (*i.e.*, states and transitions), while the edges $E_{DG} \subseteq V_{DG}^2$ represent the sequential dependencies between these elements (*e.g.*, which state is connected to which transition) and the communication semantics between blocks. Some of the vertices have no incoming edges: they represent the initial states of the SysML model's state machine diagrams, and all the other vertices are reachable from at least one path originating from one of these source vertices. The vertices are labeled with information that enables reconstructing the SysML model from the graph: *e.g.*, the block to which the state/transition represented by the vertex belongs, or the type of element the vertex represents. For the needs of the following definitions, we introduce a function $isCommVertex : V_{DG} \to \mathbb{B}ool$ which indicates whether a vertex represents a transition containing a send/receive signal within its action set. Last, we assume that this graph is semantically equivalent to the SysML model: for a given SysML model M, $graphToModel(modelToGraph(M)) \equiv M$. Several reduction algorithms for these DGs were previously proposed (Apvrille et al., 2021; Apvrille et al., 2023) in the context of model checking. They are used to reduce a DG with respect to a given property, enabling model-checking algorithms to work on smaller models. The algorithm introduced in this section operates differently, reducing the DG based on a set of input tags, in a manner that is entirely agnostic to the properties to be verified.

3.2 Tagging

We define below the tagging operation. Tags can be assigned to the following elements of a SysML model: blocks, associations between blocks (*i.e.*, considering a SysML model such as defined in Definition 3, elements of $\{(p_1, p_2) \in \mathcal{P}^2 | connect(p_1, p_2) \neq No\}$), states, and state-machine transitions.

Definition 4 (Tagging)**.**

Let $\mathcal{A}* = \{a, A, \ldots, z, Z\}*$ be the set of finite words *and $M = \langle \mathcal{B}, connect, assoc \rangle$ be a SysML model. For each $B \in \mathcal{B}$, we denote its state-machine diagram with $\langle s_{0B}, S_B, T_B \rangle$. Let $\mathcal{T} = \bigcup_{B \in \mathcal{B}} T_B$ and $\mathcal{S} = \bigcup_{B \in \mathcal{B}} S_B$.*

A tagging *is a total function $tag : \mathcal{B} \sqcup \mathcal{P}^2 \sqcup \mathcal{S} \sqcup \mathcal{T} \to \wp(\mathcal{A}*)$[3] that associates a set of* tags *with blocks, port associations, states and transitions of M.*

Since DGs shall be semantically equivalent to SysML models, we have also introduced the ability to tag vertices.

Definition 5 (Graph tagging)**.**

Let $\mathcal{A}* = \{a, A, \ldots, z, Z\}*$ be the set of finite words *and (V, E) be a graph.*

A graph tagging *is a total function $tag_G : V \to \wp(\mathcal{A}*)$.*

The tags are preserved when a model is converted to a DG. Given a SysML model M and its corresponding graph $(V, E) = modelToGraph(M)$, for every vertex $v \in V$, if e denotes the block, state, or transition represented by v, then $tag_G(v) = tag(e)$.

These definitions form the basis for the graph reduction algorithm.

3.3 Tag-Driven Reduction

3.3.1 Preliminary definitions

The reduction algorithm introduced in the current paper relies on the concept of *logical dependency* in DGs. The logical dependencies of a given vertex are defined as the union of its *path dependencies* and *communication dependencies*, as described below. In

[3] $\wp(\mathcal{A}*)$ is the powerset of $\mathcal{A}*$ and $A \sqcup B$ denotes the disjoint union of A and B ($A \cup B$ with $A \cap B = \emptyset$).

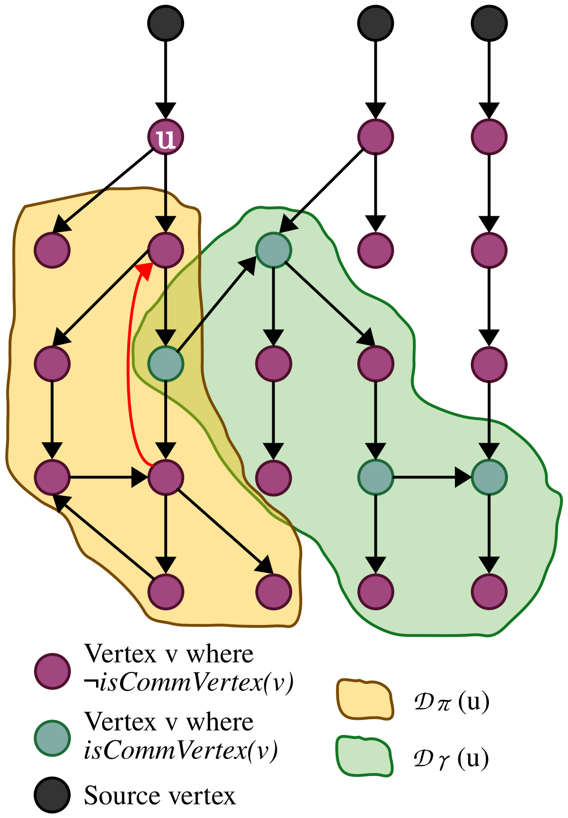

Figure 1: Example DG for illustrating Definitions 6 to 8

those three definitions, (V,E) is a (finite) directed graph.

Definition 6 (Path dependency).

Let $V_s = \{v \in V \mid \forall u \in V, (u,v) \notin E\}$.

A vertex t is a path dependency *of a vertex u if $\forall s \in V_s, \forall \{(s,v_1),(v_1,v_2)\dots,(v_n,t)\} \subseteq E, (s = u \vee \exists i \in [\![1,n]\!]$ s.t. $v_i = u)$.*

The function $\mathcal{D}_\pi : V \to \wp(V)$ associates each vertex with the set of its path dependencies.

Definition 7 (Communication dependency).

We define the function $\gamma : V \times \mathbb{N} \to \wp(V)$ by the following recurrence relation:

- $\gamma(u,0) = \{v \in V \mid isCommVertex(v) \wedge \exists w \in \{u\} \cup \mathcal{D}_\pi(u), (w,v) \in E\}$.
- $\gamma(u,n+1) = \gamma(u,n) \cup \{v \in V \mid isCommVertex(v) \wedge \exists w \in \gamma(u,n), (w,v) \in E\} \cup \bigcup_{v \in \gamma(u,n)} \mathcal{D}_\pi(v)$.

The set of communication dependencies of a vertex $u \in V$ is then defined as $\gamma(u,i)$ where i is such that $\gamma(u,i+1) = \gamma(u,i)$.

The function $\mathcal{D}_\gamma : V \to \wp(V)$ associates each vertex with the set of its communication dependencies.

Definition 8 (Logical dependency).

The set of logical dependencies *of a vertex $u \in V$ is defined as $\mathcal{D}_\pi(u) \cup \mathcal{D}_\gamma(u)$. For the rest of the paper, we define the total function*

$$logdep: \quad V^2 \to \mathbb{B}ool$$
$$(t,u) \mapsto \begin{cases} \top & if\, t \in \mathcal{D}_\pi(u) \cup \mathcal{D}_\gamma(u) \\ \bot & otherwise. \end{cases}$$

3.3.2 Tag-driven reduction algorithm

Algorithm 1: Tag-driven graph reduction

```
Data: Dependency graph $DG_I = (V_I, E_I)$,
      $T \in \wp(\mathcal{A}*)$, graph tagging function
      $tag_G : V_I \to \wp(\mathcal{A}*)$
Result: Reduced graph $G_R = (V_R, E_R)$
1  $(V_R, E_R) \leftarrow (\emptyset, \emptyset)$
2  $V_{rem} \leftarrow \emptyset$
3  Boolean $continue \leftarrow \top$
4  foreach $v \in V_I$ do
5      if $T \cap tag_G(v) \neq \emptyset$ then
6          $V_{rem} \leftarrow V_{rem} \cup \{v\}$
7  end
8  while $continue$ do
9      $continue \leftarrow \bot$
10     foreach $v \in V_I$ do
11         if $v \notin V_{rem} \wedge (E_I \cap V_I \times \{v\} \subseteq$
             $V_{rem} \times \{v\}) \vee (isCommVertex(v) \wedge$
             $(E_I \cap V_I \times \{v\}) \cap V_{rem} \times \{v\} \neq \emptyset)$
             then
12             $V_{rem} \leftarrow V_{rem} \cup \{v\}$
13             $continue \leftarrow \top$
14     end
15 end
16 $V_R \leftarrow V_I \setminus V_{rem}$
17 $E_R \leftarrow E_I \cap V_R^2$
```

Algorithm 1 generates a graph by removing a set of vertices, V_{rem} (line 2), from the input DG (V_I, E_I). The set V_{rem} is initially populated with vertices from V_I that are associated with at least one tag from the input tag set T (lines 4-7). Subsequently (lines 8-14), the algorithm adds to V_{rem} any vertex from V_I whose incoming edges all originate from vertices in V_{rem}. This enrichment of V_{rem} is performed recursively until no additional vertex in $V_I \setminus V_{rem}$ has all its incoming edges originating from vertices in V_{rem}. Finally, a new graph (V_R, E_R) is constructed by subtracting V_{rem} from V_I and by deleting from E_I every edge involving a vertex in V_{rem} (lines 16-17). Afterwards, a graph-to-model reconstruction algorithm already implemented in TTool builds a model based on the reduced graph.

Overall, our two contributions interrelate as follows. The process begins with the user providing a SysML model (*i.e.*, a block diagram and a set of state-machine diagrams). Using the tagging feature in the TTool's graphical user interface or command-line interface, the user assigns tags to elements of the model. When necessary (*e.g.*, when only the core model, or a variant including a subset of optional features, is needed), the user then specifies a list of tags referring to the elements to deactivate in the model. Based on this input, TTool computes the DG, applies Algorithm 1 to derive a new graph, and outputs the reduced SysML model built from this graph thanks to the pre-existing graph-to-model reconstruction algorithm. Currently, this reduction process guarantees the removal of all logical dependencies only for certain classes of SysML models, as discussed below.

Proposition 1. *In the absence of additional assumptions about the input DG, Algorithm 1 guarantees the deletion of* all *the logical dependencies of a vertex tagged with an input tag iff the input DG is acyclic.*

Proof. Let (E_I,V_I), $T \in \wp(\mathcal{A}*)$ and $tag_G : V_I \to \wp(\mathcal{A}*)$ be the input DG, set of tags and graph tagging function. Assume that $\exists v \in V_I$ s.t. $\exists v_t \in \{v \in V_I | T \cap tag_G(v) \neq \emptyset\}, logdep(v,v_t) \wedge v \notin V_{rem}$. Therefore $\neg((E_I \cap V_I \times \{v\} \subseteq V_{rem} \times \{v\}) \vee (isCommVertex(v) \wedge (E_I \cap V_I \times \{v\}) \cap V_{rem} \times \{v\} \neq \emptyset))$ (lines 8-14 of Algorithm 1). Equivalently, this can be expressed as:

$$\neg isCommVertex(v) \wedge \exists (v',v) \in E_I, v' \notin V_{rem} \\ \vee \forall (v',v) \in E_I, v' \notin V_{rem} \tag{1}$$

We first prove the following lemma:

Lemma 1. *In an acyclic DG, Algorithm 1 removes every path dependency of a vertex tagged with an input tag.*

We start by noting that (1) $\implies \exists (v',v) \in E_I, v' \notin V_{rem}$ and will prove that a contradiction arises from this weaker condition. At this point, we have two possible cases:

1. $tag_G(v') \in T$: in that case, $v' \in V_{rem}$ (lines 5-6 of Algorithm 1) and we have a contradiction.
2. $tag_G(v') \notin T$. Since $v' \notin V_{rem}$, we can apply the same reasoning and deduce that $\exists (v'',v') \in E_I$ s.t. $v'' \notin V_{rem}$. By induction, since (V_I,E_I) is a directed acyclic graph we can continue this process to build a path leading to v and originating from a vertex with no incoming edges, where no vertex on this path is associated with any tag from T. We can here invoke Definition 6 to note a contradiction and conclude that $\nexists v \in V_I$ s.t. $\exists v_t \in V_T, v \in \mathcal{D}_\pi(v_t) \wedge v \notin V_{rem}$, which proves Lemma 1.

From the above, we now infer that the vertex v satisfying (1) is necessarily a communication dependency of v_t. Using an inductive argument analogous to the one used in the proof of Lemma 1, it follows from (1) that no edge (u,w) where $isCommVertex(w)$ and $u \in V_{rem} \supseteq \{v_t\} \cup \mathcal{D}_\pi(v_t)$[4], can exist on any path from a source vertex to v, which contradicts Definition 7. Finally, recalling Definition 8 ends the proof of the sufficient condition.

Now, if the input graph is not acyclic, a logical dependency v may have an incoming edge originating from a cycle, where all vertices on the cycle are path dependencies of v. Such a vertex invalidates the condition $E_I \cap V_I \times \{v\} \subseteq V_{rem} \times \{v\}$ (Algorithm 1, line 11) since the vertices of the cycle, only reachable thanks to paths originating from v, do not belong to V_{rem}. If $\neg isCommVertex(v)$, v will not be added to V_{rem} by the algorithm, which proves the necessary condition. □

However, even if Algorithm 1 overlooks any logical dependencies of a vertex v_t associated with an input tag, lines 4–6 ensure that v_t is removed from the graph. Therefore, since by Definition 6, any path originating from a vertex representing the initial states of a state machine diagram and leading to a path dependency of v_t must pass through v_t, no path remains in the reduced graph that leads to one of these overlooked dependencies from a source vertex. TTool's reconstruction algorithms will ignore these vertices during the model reconstruction phase, producing a model from the partially reduced graph that is free from all elements corresponding to the path dependencies associated with the input tags. From the perspective of model reduction, Algorithm 1 thus addresses a broader range of cases than models associated with an acyclic DG. However, some communication dependencies remain in the output graph, and their corresponding elements in the reduced model, if the input model is such that its DG contains a cycle of logical dependencies of a tagged vertex, involving a vertex associated with a communication label (e.g., the cycle closed by the red arrow in Figure 1).

3.4 Methodological Insight

The tag-based variability management approach presented in the current paper is designed to integrate into a broader model-driven engineering framework. The anticipated benefits, given below, vary depending on whether the model captures high-level system aspects or low-level algorithmic details.

Formal Verification (for both systems and low-level algorithmic models): by generating portions of the model that can be independently verified, the complexity of the proof can be reduced. Additionally, in a software product line engineering context, the tagging

[4] $\{v_t\} \cup \mathcal{D}_\pi(v_t) \subseteq V_{rem}$ is guaranteed by Lemma 1.

and reduction approach enables for the generation of a family of models from a complete model. Each model element corresponding to a feature is first tagged (one tag per feature). The reduction is then performed by excluding all tags (to generate a core model), followed by excluding specific tags selected by the user, depending on the feature content of each variant. This results in a set of models comprising a core model and several variants that extend the core. It is now possible to rely on TTool's incremental model-checking algorithm: instead of verifying all properties on the entire model at once, this algorithm reuses previous proof results (e.g., from the core), checking if the properties still hold, rather than performing a full exploration of the state space each time an new optional feature is evaluated on each variant. With this approach, verifying a model variant does not require a comprehensive exploration of the state space, similar to the method proposed by (Cordy et al., 2012).

Code Generation (for low-level algorithmic models): the tagging and reduction approach enables users to generate multiple applications with just a few clicks from a same model. Moreover, the code generated from a reduced model focuses only on the required functionality, optimizing both the size and complexity of the software. This is particularly important in application domains such as embedded systems.

4 Evaluation and Discussion

An Integrated Modular Architecture, or IMA for short, serves a case study. An IMA offers computing, communication, and I/O services for modern embedded systems in domains such as automotive (Fürst et al., 2009) and avionics (Bieber et al., 2012). IMA design requires rigorous development processes that include formal verification techniques. Because of frequent changes during the development of IMA architectures, the approach proposed in the current paper may be applied for it simplifies formal verification when including new features or existing ones.

4.1 IMA Architecture

Let us consider an IMA architecture model composed of 4 computing modules (represented by blocks CM1, CM2, CM3 and CM4, see Figure 2) and a network (block Network) interconnecting the modules and allowing communication between them via messages. 9 tasks are distributed among the 4 computing modules contributing to the execution of 4 applications:

- Application 1: *Task1* → *Task3* → *Task6*
- Application 2: *Task2* → *Task7* → *Task4* → *Task5*
- Application 3: *Task2* → *Task6* → *Task8*
- Application 4: *Task9* → *Task5* → *Task1*

Each task requires for its execution an amount of computing resources defined as the workload. The total workload on a module cannot exceed the total amount of resources available on that module.

4.2 Tagging and Reduction of the IMA Model

The IMA model may be reduced manually by the user of TTool or using the tagging and the algorithmic reduction approach presented in the current paper. The core features of the system are considered to be the applications provided by the modules $CM1$, $CM2$, and $CM4$, with an optional variant of the system that additionally includes the applications from module $CM3$. The reduction process therefore involves disabling $Application$ 1 and $Application$ 2, and by consequence the software tasks $task3$ and $task4$ executed by the module $CM3$. We evaluate two reduction strategies:

(1) Over-approximation strategy: this approach removes the entire $CM3$ block from the model, along with all transitions in the state-machine diagrams of other blocks that are synchronized with the transitions of $CM3$'s state-machine diagram. This approach requires:

- **By hand**: removing the $CM3$ block, and then removing every transition in the state-machine diagrams involving an incoming or outgoing signal linked to $CM3$'s signals as well as states only reachable from these transitions.
- **With the algorithmic reduction approach**: tagging CM3 with a tag t, and launching the reduction algorithm for the tag t.

(2) Fine reduction strategy: in addition to the above one, this strategy deletes elements in the other state-machine diagrams that depend to some extent on the $CM3$ block, but are not synchronized with transitions in its state-machine diagram (*e.g.*, transitions with guards involving values that can only be reached when a signal is received from $CM3$'s state-machine). This approach requires:

- **By hand**: removing every transition, and subsequently removing the states that are only reachable via those transitions, where the transitions depend on values carried by signals from the $CM3$ block.

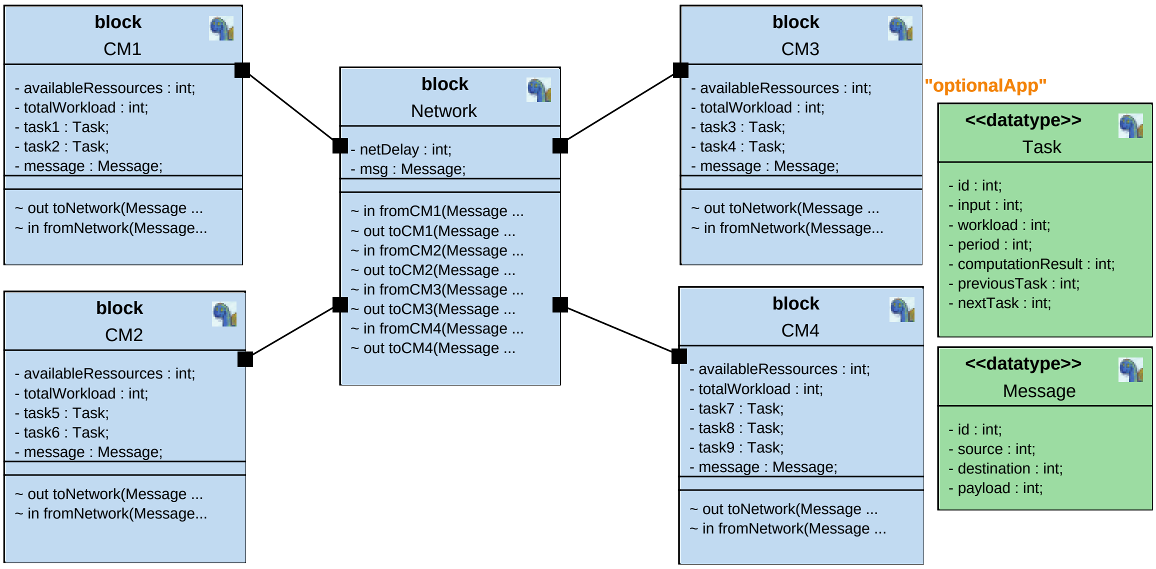

Figure 2: Block diagram of an IMA architecture

Table 1: Comparison of manual and tag-driven automated model reductions

	Nb. of manual operations for model reduction		DG	Reachability graph
	Manual approach	Automated approach		
Initial model	—		109 vertices 162 edges	6,256,658 vertices 9,015,420 edges
Reduced model (fine reduction)	17	**9**	**67 vertices** **94 edges**	**8,458 vertices** **10,362 edges**
Reduced model (over-approximation)	4	**2**	90 vertices 131 edges	50,120 vertices 68,634 edges

- **With the tagging and algorithmic approach**: tagging these transitions with a tag t, and launching the reduction algorithm for the tag t.

4.3 Results and Discussion

Table 1 summarizes key figures of our evaluation. We can draw the following conclusions:

- Regardless of the reduction strategy (fine/over-approximation reduction), the tagging and algorithmic approach reduces the number of operations required in TTool's GUI from the user, thereby reducing as well the risk of errors.
- The sizes of the reachability graphs—those analyzed by TTool's model-checking algorithms for verifying CTL properties—indicate that model reduction is effective in lowering proof complexity. The over-approximation reduction reduces the number of edges in the reachability graph by a factor of 131, and the fine reduction decreases this number by a factor of 870. For example, proving the reachability of the same state in the state machine of the block *Network* requires exploring 985 vertices/1,529 edges in the original model, 530/1,063 in the over-approximated reduced model, and 128/177 in the fine reduced model.
- Additionally, an advantage of this approach is that once different tags are assigned to model elements corresponding to different versions of the modeled system, models of these versions can be generated with a single click, thanks to our reduction algorithm. Users of TTool no longer need to store large model files containing a full SysML model for each system version, but only a single tagged model.

However, our approach has several limitations that shall be addressed in future work. First, as demonstrated by Proposition 1 and the subsequent discussion, Algorithm 1 does not eliminate all communication dependencies when the input DG contains cycles of logical dependencies of a tagged vertex, involving a vertex associated with a communication label: for those graphs, the reduction is only partial. A naive approach to removing *path* dependencies based on the logical condition expressed in Definition 6 and thus involving an exhaustive examination of all paths leading to each vertex, would guarantee the complete removal of the remaining *communication* dependencies. However, its computational complexity makes it highly inefficient for application to large SysML models. Two avenues may be explored: (1) Adding complementary reduction algorithms and (2) improving the graph-to-model reconstruction algorithm currently implemented in TTool.

As discussed earlier, while the fine reduction strategy produces more optimized model variants, it currently requires an in-depth user analysis. This is because our algorithm does not evaluate guards, focusing only on logical dependencies as defined per Definition 8. To enhance our approach, we should expand the definition of a logical dependency for a vertex v to include vertices that can be reached only through the guards which are evaluated at *true* due to an action triggered by the model element represented by v. By doing so, the fine reduction strategy would demand the same tagging operations as those currently required by the over-approximation strategy.

In addition, the task of tagging model elements is currently left to the user. To automate this process, we could implement automatic pre-assignment of tags to model elements based on a use case diagram, e.g., by adapting TTool's capability to check consistency between diagrams using formal rules combined with LLMs (Sultan and Apvrille, 2024).

A potential limitation to the validity of our conclusions should also be considered. In our evaluation, model reduction resulted in a significant decrease in the size of the reachability graphs. However, there might be cases where model reduction, particularly due to the removal of synchronizations between state-machine diagram transitions, could instead cause an increase in the state space. Nevertheless, since TTool computes reachability graph sizes, the users of TTool can easily verify whether model reduction has led to a reduction in state space and accordingly decide whether to perform formal verification on the reduced or the entire model.

5 Related work

5.1 Verifying SysML Models

The authors of (Cederbladh et al., 2024) review applications of formal verification in a MBSE context, emphasizing on SysML. SysML verification tools are categorized in two directions: (1) SysML tools connected to an external model checker (e.g., UPPAAL (Wang et al., 2023)), and (2) SysML tools that integrates a native model checker. TTool belongs to the second group and its native model checker favorably compares to UPPAAL (Tempia Calvino and Apvrille, 2021).

With its native model checker, TTool offers a platform for optimizing formal verification of SysML models (see, *e.g.*, (Coudert et al., 2024). Optimizing formal verification may be viewed into two directions: first, optimizing the model checker itself and, second, annotating the SysML model for driving the model checker. The contributions of the current paper fall in the second category. Whatever the direction, formal verification may be discussed in terms of transfer to industry (Horvath et al., 2023) and lowering its complexity is crucial in the context of software product lines engineering which involve the reuse of system subparts (Cordy et al., 2012).

5.2 Verifying and Handling Variants in Software Product Lines

Given the importance of reducing verification complexity in software product line engineering, numerous research contributions aim at addressing this issue. Among these, a significant body of research has concentrated on verifying Featured Transition Systems (FTS), an extension of labeled transition systems designed to model software product lines (Classen et al., 2011). Classen et al. proposed a symbolic model checking method to reduce the complexity of FTS verification. Building upon this work, an incremental approach (Cordy et al., 2012) was introduced, reusing previous proof results on FTS variants by assessing whether newly added elements to the model invalidate properties satisfied in the previously verified version. Lochau et al. proposed an incremental model-checking process based on CCS process calculus (Milner, 1980), where variability between variants is represented as deltas between the CCS processes. The approach involves analyzing these deltas to verify whether the properties verified prior to their application are preserved (Lochau et al., 2016). These approaches share with TTool's incremental model checking algorithms the principle of performing a preliminary analysis of model variations to identify preserved properties and determine which elements require re-verification. However, the supported modeling languages differ and TTool's model verification operates directly on SysML semantics, reducing potential translation bias between the user-designed model and the formal model being verified (Coudert et al., 2024).

In order to use these incremental model-checking algorithms, incremental variants of the SysML model must be provided. Our paper addresses this challenge with the model tagging and reduction approach. Variability in SysML models can be managed through different strategies, including rule-based (Barbie et al., 2023) and *annotation* strategies (Schäfer et al., 2021), as employed in (Webber and Gomaa, 2004; Bilic et al., 2019). Actually, our approach can be considered as a form of annotation strategy, where the initial model serves as a "150% model". A significant advantage of annotation strategies is their ability to maintain consistency across different variants when modifications are made to the core model (Schäfer et al., 2021), as seen in source code tagging in (Pfofe et al., 2016). However, variability tagging in SysML models is often applied at the block diagram level, as emphasized by Bilic et al., who underscore the importance of tagging behavioral models (*e.g.*, state machine diagrams). Our approach handles the tagging of behavioral models. Another key aspect of our approach is the cross-diagram propagation of tagged element removals using DGs, which reduces the number of tags the user needs to add to the SysML model.

In addition, while our tagging approach is applied to variant identification and generation in this paper, it is polymorphic. No constraints are imposed on the tagging operation: the tags are not explicitly used to model variability points but are simply strings attached to model elements. Our approach therefore can be adapted for other purposes, such as model versioning to maintain a history of model evolution.

6 Conclusions

The paper introduces a SysML model tagging feature along with model reduction algorithms based on DGs. These contributions enable deleting tagged elements from a SysML model, along with (at least part of) their associated logical dependencies. The combined use of these two features helps with the generation of a core system model from a full system model, as well as the creation of functional variant models that represent optional features. Since these variants are additive modifications to the core model, this tagging and reduction approach enables the use of TTool's incremental model checking algorithm, simplifying the proof process for such families of models used, for instance, in software product line engineering. Our evaluation demonstrates improvements over manual reduction approaches.

However, there is potential for further enhancement. Future work could expand the definition of logical dependencies to also include the evaluation of guards, and adapt our algorithms accordingly, and design new algorithm to effectively delete all communication dependencies in cyclic DGs. Furthermore, the development of AI-assisted mechanisms to aid users in the tagging process appears to be a promising avenue. Future evaluations should include testing our approach on a wider range of models and, beyond comparing our method to manual reduction approaches, evaluating it against model reduction techniques based on mutation scripts and compilers such as those implemented in TTool (Sultan et al., 2023).

REFERENCES

Apvrille, L., de Saqui-Sannes, P., Hotescu, O., and Calvino, A. T. (2021). Dependency graphs to boost the verification of sysml models. In *International Conference on Model-Driven Engineering and Software Development*, pages 109–134. Springer.

Apvrille, L., Sultan, B., Hotescu, O., de Saqui-Sannes, P., and Coudert, S. (2023). Mutation of Formally Verified SysML Models. In *11th internationl conference on Model-Based Software and Systems Engineering (Modelsward'2023)*.

Barbie, P., Tenev, V., and Becker, M. (2023). Intra: Automatic reduction of model complexity and generation of system variants - a tool demonstration. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume B*, SPLC '23, page 25–29, New York, NY, USA. Association for Computing Machinery.

Bieber, P., Boniol, F., Boyer, M., Noulard, E., and Pagetti, C. (2012). New Challenges for Future Avionic Architectures. *Aerospace Lab*, (4):p. 1–10.

Bilic, D., Brosse, E., Sadovykh, A., Truscan, D., Bruneliere, H., and Ryssel, U. (2019). An integrated model-based tool chain for managing variability in complex system design. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 288–293.

Calvino, A. and Apvrille, L. (2021). Direct Model-Checking of SysML Models. In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development - MODELSWARD*, pages 216–223. INSTICC, SciTePress.

Cederbladh, J., Cicchetti, A., and Suryadevara, J. (2024). Early validation and verification of system behaviour in model-based systems engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, 33(3):1 – 67.

Classen, A., Heymans, P., Schobbens, P.-Y., and Legay, A. (2011). Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 321–330, New York, NY, USA. Association for Computing Machinery.

Cordy, M., Schobbens, P.-Y., Heymans, P., and Legay, A. (2012). Towards an incremental automata-based approach for software product-line model checking. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC '12, page 74–81, New York, NY, USA. Association for Computing Machinery.

Coudert, S., Apvrille, L., Sultan, B., Hotescu, O., and de Saqui-Sannes, P. (2024). Incremental and Formal Verification of SysML Models. *SN Computer Science*, 5(6):714.

de Saqui-Sannes, P., Apvrille, L., and Vingerhoeds, R. A. (2021). Checking SysML Models against Safety and Security Properties. *Journal of Aerospace Information Systems*, pages 1–13.

de Saqui-Sannes, P., Vingerhoeds, R. A., Garion, C., and Thirioux, X. (2022). A taxonomy of MBSE approaches by languages, tools and methods. *IEEE Access*, 10:120936–120950.

Fürst, S., Mössinger, J., Bunzel, S., Weber, T., Kirschke-Biller, F., Heitkämper, P., Kinkelin, G., Nishikawa, K., and Lange, K. (2009). AUTOSAR–A Worldwide Standard is on the Road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, volume 62. Citeseer.

Garmendia, A., Guerra, E., and de Lara, J. (2024). Product lines of graphical modelling languages. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, MODELS '24, page 69–79, New York, NY, USA. Association for Computing Machinery.

Hohl, P., Stupperich, M., Münch, J., and Schneider, K. (2018). Combining agile development and software product lines in automotive: challenges and recommendations. In *2018 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*, pages 1–10. IEEE.

Horváth, B., Graics, B., Hajdu, A., Micskei, Z., Molnár, V., Ráth, I., Andolfato, L., Gomes, I., and Karban, R. (2020). Model checking as a service: towards pragmatic hidden formal methods. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, New York, NY, USA. Association for Computing Machinery.

Horvath, B., Molnar, V., Graics, B., Hajdu, A., Rath, I., Horvah, A., Karban, R., Trancho, G., and Micskei, Z. (2023). Pragmatic verification and validation of industrial executable SysML models. *Systems Engineering*, 26(6):693–714.

Lochau, M., Mennicke, S., Baller, H., and Ribbeck, L. (2016). Incremental model checking of delta-oriented software product lines. *Journal of Logical and Algebraic Methods in Programming*, 85(1, Part 2):245–267. Formal Methods for Software Product Line Engineering.

Milner, R. (1980). *A calculus of communicating systems*. Springer.

OMG (2022). *Systems Modeling Language (SysML), Version 1.7 beta 1*. Object Management Group.

Pfofe, T., Thüm, T., Schulze, S., Fenske, W., and Schaefer, I. (2016). Synchronizing Software Variants with VariantSync. In *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, page 329–332, New York, NY, USA. Association for Computing Machinery.

Schäfer, A., Becker, M., Andres, M., Kistenfeger, T., and Rohlf, F. (2021). Variability realization in model-based system engineering using software product line techniques: an industrial perspective. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '21, page 25–34, New York, NY, USA. Association for Computing Machinery.

Sultan, B. and Apvrille, L. (2024). AI-Driven Consistency of SysML Diagrams. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, MODELS '24, page 149–159, New York, NY, USA. Association for Computing Machinery.

Sultan, B., Frénot, L., Apvrille, L., Jaillon, P., and Coudert, S. (2023). AMULET: a Mutation Language Enabling Automatic Enrichment of SysML Models. *ACM Trans. Embed. Comput. Syst.* Just Accepted.

Tempia Calvino, A. and Apvrille, L. (2021). Direct Model-Checking of SysML Models. In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development - MODELSWARD*, pages 216–223. INSTICC, SciTePress.

Wang, H., Zhong, D., Zhao, T., and Ren, F. (2019). Integrating model checking with SysML in complex system safety analysis. *IEEE Access*, 7:16561–16571.

Wang, S., Shi, J., Huang, Y., and Yang, Y. (2023). A tool for transforming SysML state machine into uppaal automatically. In *2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2471–2476.

Webber, D. L. and Gomaa, H. (2004). Modeling variability in software product lines with the variation point model. *Science of Computer Programming*, 53(3):305–331. Software Variability Management.