

Dependency Graphs to Boost the Verification of SysML Models

Ludovic Apvrille¹[0000-0002-1167-4639], Pierre de Saqui-Sannes²[0000-0002-1404-0148],
Oana Hotescu²[0000-0001-6612-8574], and Alessandro Tempia
Calvino³[0000-0003-1312-2907]

¹ LTCI, Telecom Paris, Institut Polytechnique de Paris, France

² ISAE-SUPAERO, Université de Toulouse, France

³ LSI, École Polytechnique Fédérale de Lausanne, Switzerland

Abstract. Model-Based Systems Engineering has often been associated with the Systems Modeling Language. Several SysML tools offer formal verification capabilities, and therefore enable early detection of design errors in the life cycle of systems. Model-checking is a common formal verification approach used to assess the satisfiability of properties. Thus, a SysML model and a property can be injected into a model-checker returning a true/false result. A drawback of this approach is that the entire SysML model is used for the verification, even if the property targets a sub-system of the model. In this paper, it is suggested to rely on dependency graphs to avoid applying model checking to the entire system when only a subset of the latter needs to be taken into account. We formalize SysML models and properties, then we present new algorithms to generate and reduce dependency graphs, so as to perform verification on reduced models. A case study on Time-Sensitive Networking is used to demonstrate the efficiency and limits of this approach. The algorithms described in the paper are fully implemented by the free software TTool. Our method enables an improvement in run time between 3% and 90% depending on the state space to be traversed to verify the property.

Keywords: MBSE · SysML · Formal Verification · Model Checking · TSN

1 Introduction

Over the past two decades, Systems Engineering has transitioned from document centric approaches to model based ones. The ‘MBSE’ acronym was coined to denote a form of systems engineering where models serve as references for a set of activities as various as requirement capture, use-case driven analysis, and system design. With a system life cycle made up of a requirement capture, analysis and design steps, one major concern is to detect design errors as early as possible in the life cycle of the system.

Checking a model against design errors can be achieved using formal verification techniques. The latter have first been developed for formal methods, such as timed automata, Finite State Machines, and Petri Nets, just to mention a few. Formal verification techniques such as model checking have more recently been adapted to semi-formal languages, in particular SysML [8], [38], [10], [36].

As far as SysML is concerned, a model-checker takes a SysML model and a property as input, and outputs a true/false result. To make model checking practicable, the designer of the SysML model must be able to identify the properties to be verified and to express them in a form that is processable by the model checker. Model checking SysML models further requires to interpret the results output by the model checker, and to eventually relate the true/false answers to the original SysML Model. With its in-built model checker, TTool [43] handles the two issues. First, the properties are expressed in a CTL-like language and located in specific comments inside the SysML model. Second, the results of model checking the SysML models are reported in the comments containing the properties, by indicating which property holds or not.

Despite of its user-friendliness in terms of properties expression and verification, TTool shares one limitation with other SysML model checkers: the verification process uses the entire model as input, even if the property of interest concerns only a sub-part of it, thus leading to extra verification time, and possibly to combinatorial explosion. The purpose of this paper is to address this issue in the context of SysML and to assess the efficiency of the proposed approach by using TTool as a prototyping platform. The proposed approach relies on the following statement: many parts in the block and state machine diagrams of the SysML model are two by two dependent. For instance, a dependency does exist between two blocks $B1$ and $B2$ that synchronize by respectively sending and receiving a message m .

In the current paper, it is argued that dependencies may be expressed using a graph that we call *dependency graph*. It is further proposed to compute a reduced model of the SysML model that is sufficient to prove the property of interest. Because the resulting model is smaller, the proof is expected to be faster, as illustrated in this paper with a case study.

The current paper sketches the dependency graph generation algorithm and relies on a case study to show the efficiency of its implementation in the free software TTool [43, 36], both for the proof of reachability and liveness properties.

The current paper extends a paper co-authored by the same authors and published at Modelswards 2022 conference [5]. The current paper differs from [5]. The related work section has been substantially extended with recently released papers. Algorithms 2 and 3, at the root of this new contribution, have been substantially improved and their implementation in TTool has been updated. Last but not least, the TNS (Time Sensitive Networking) protocol [21] has been selected as a case study for it is the successor of the AFDX protocol addressed in [5]. More complete performance measures, based on the next algorithms, are provided: they better demonstrate the interest of our approach.

The current paper is organized as follows. Section 2 formally defines a subset of SysML. Section 3 introduces dependency graphs. It also presents the algorithms implemented by TTool to generate dependency graphs from SysML models. Section 4 discusses a case study based on the IEEE 802.1 TSN protocol. Section 5 surveys related work. Section 6 concludes the paper and outlines future work.

2 SysML

The Systems Modeling Language (SysML [30]) is an international standard [30] at OMG (*Object Management Group*) and originates from joint efforts of OMG and INCOSE (*International Council on Systems Engineering*) to define a modeling language for systems engineers. Version 1.6 of SysML enables covering the requirement capture, analysis and design steps in the life trajectory of systems.

The main objective of the design phase is to define the architecture of the system using the Block Definition Diagrams (BDD) and the Internal Block Diagrams (IBD) defined by the SysML standard [30]. In this paper, the BDD and IBD are merged into a Block Instance Diagram (BID). Each block instance in the BID has a behavior expressed in the form of a SysML state machine diagram.

2.1 Block Instance Diagram

A Block Instance Diagram contains a set of block instances that can be composed together, and associated through port relations.

Definition: block instance. A block instance is a 7-tuple $B = \langle id, A, M, P, S_i, S_o, smd \rangle$ 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.
- 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 $D = \langle \mathcal{B}, connect, assoc \rangle$. 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} .

- \mathcal{B} is a set of block instances.
- $connect$ is a function $\mathcal{P} \times \mathcal{P} \rightarrow \{No, synchronous, asynchronous\}$ that returns the communication semantics between two ports (\emptyset , synchronous or asynchronous).
- $assoc$ is a function $(\mathcal{P}_{\mathcal{B}_1} \times \mathcal{S}_o \times \mathcal{P}_{\mathcal{B}_2} \times \mathcal{S}_i) \rightarrow Bool$ that returns true if an output signal \mathcal{S}_o of block \mathcal{B}_1 is associated to an input signal \mathcal{S}_i of block \mathcal{B}_2 via 2 ports $p1, p2$ of respectively of \mathcal{B}_1 and \mathcal{B}_2 , and if these two ports are connected (*i.e.*, $connect(p1, p2) = true$);

2.2 State Machine Diagram

Each block instance contains one finite state machine that supports states, transitions, attribute settings, 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 $\langle s_0, S, T \rangle$ 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*. *send signal*, *receive signal* can use its signals, or the signals of the parent block B_p , or the signals of the parent block of B_p , and so on.
- s_{end} is the final state of the transition.

2.3 Formal Verification with TTool

A SysML model is made up of one or several diagrams expressed in a graphic fashion for SysML V1 [30] or by a combination of graphics and text for SysML V2 [30]. Whatever the version of SysML, a SysML tool must offer a diagram editor. The open source Papyrus tool [32] offers a complete editor that strictly follows the SysML standard [30]. Other SysML tools commonly support variants of the OMG-based SysML syntax, and offer extensions to supports various classes of systems. Examples of tools applied to real-time systems include Cameo Systems Modeler [29], Rhapsody [34], If-Omega [13], and TTool [43].

SysML diagrams editors usually save SysML diagrams in a form that becomes processable by external tools or in-built modules in charges of checking the SysML diagrams, especially the block and state machine diagrams, against design errors. Simulation enables early debugging of SysML diagrams by randomly firing transitions. Model checking goes one step further with a more systematic and mathematically grounded analysis of the SysML models.

TTool [43, 6] is a free and open source framework for the design and verification of embedded systems. The TTool model checker [11] inputs SysML models enriched with safety properties to be verified and outputs a yes-no answer for each property. In practice, the TTool model checker takes as input (1) a block instance diagram and the state machine diagrams modeling the inner workings of the blocks, and (2) properties formally expressed using a CTL-based language. TTool’s model checker computes properties expressed inside the SysML model and returns the feedback in the same SysML model. Users of TTool are therefore not obliged to use external tools or to inspect the inner workings of the model checker.

The benefits and potential of using TTool for model-checking SysML models have been discussed in [6, 36]. The remainder of the current paper explains how the model checker of TTool has recently been extended with the purpose to reduce the amount of time allocated to model checking of SysML models.

3 Dependency Graphs

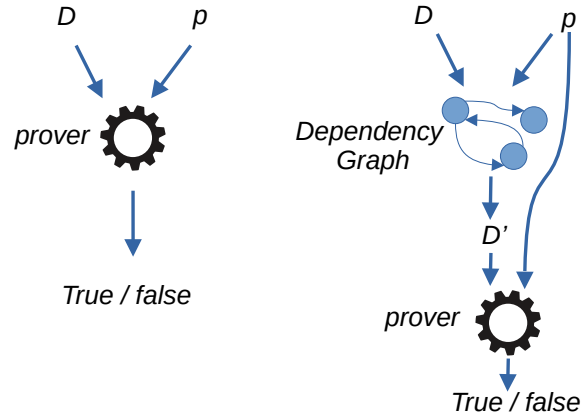


Fig. 1: Proofs without/with dependency graphs [5]

This section shows how dependency graphs can be used in the scope of the verification problem addressed by the paper. A classical verification process takes as input a design, a property, and outputs false or true, as illustrated by the left part of Figure 1. Basically, a design is made up of a (1) a SysML block instance diagram and its associated state machines, and (2) a set of properties. Using a dependency graphs is expected to decrease the complexity of the proof with regards to the proof without using dependency graphs. This section first gives a definition related to systems' verification. Then, dependency graphs are introduced along with algorithms, and illustrated on a toy system.

3.1 Definition of a system

Definition 1. A *System* has a *Design* and a set of *Properties* to be verified, as defined in [11].

$$S = \langle D, P \rangle \quad (1)$$

3.2 Proving a property over a system

Definition 2. Let us define *prover* as a function that takes a *Design* D and a *Property* p as input. The *prover* function returns true if p is satisfied by D (also denoted as $D \models p$),

false otherwise.

$$prover(D, p \in P) = \begin{cases} true & \text{if } p \text{ is satisfied by } D \\ false & \text{Otherwise} \end{cases} \quad (2)$$

The objective of this work is to decrease the complexity of the *prover()* function (2).

3.3 Decreasing the proof complexity

To prove a property, a prover considers all design elements, even if some of these elements are not involved in this proof, as depicted by the left part of Figure 1.

The right part of Figure 1 illustrates the main idea behind the paper's contribution: to eliminate parts of the models that may slow down the proof without impacting its result. The proposed solution, detailed in Algorithm 1, is (i) to compute a dependency graph DG from the input model D , (ii) to reduce DG to DG_p according to property p to be proven, (iii) to rebuild a model D_p from DG_p and finally (iv) to use DG_p and p as input for the *prover* to figure out if $D_p \models p$, and by deduction if $d \models p$.

Algorithm 1: Use of dependency graphs to simplify proofs

Data: D, P
Result: $\forall p \in P, result_p = prover(D, p)$

- 1 $DG = computeGraph(D)$
- 2 **foreach** $p \in P$ **do**
- 3 $DG_p = reduceGraph(DG, p)$
- 4 $D_p = graphToModel(DG_p)$
- 5 $result_p = prover(D_p, p)$
- 6 **end**

The section now formalizes the different stages of Algorithm 1.

3.4 From a Design to a dependency graph

We now assume that a design $d \in D$ is a block instance diagram B . A dependency graph DG can be computed from $D = B$ (Algorithm 2):

$$DG = computeGraph(D)$$

For each *smd* of $B \in Bl$, for each element of the state machine (states, transitions, send/receive actions), we generate one vertex v_e in DG (line 4).

Then, the algorithm looks for all couples of *read* and *write* operators connected through the same channel (line 7, *cond*₁). If the channel is synchronous, the two operators must belong to different blocks. For all such couples, a new vertex is added for each element (line 9, v_1 and v_2 , and an edge is created in line 10 between the vertex of the writer (v_1) to the vertex of the reader (v_2). Then, the new vertices v_1 and v_2 are

Algorithm 2: Building a dependency graph from a model

```

Data:  $D$ 
Result:  $DG$ 
1  $DG = \text{emptyGraph}$ 
2 foreach  $smd$  of  $Bl$  of  $D$  do
3   foreach  $elt \in smd$  do
4      $DG \uplus \text{vertex}(elt)$ 
5   end
6   foreach  $elt_1, elt_2 \in smd^2$  do
7      $c = \text{connect}(\text{block}(elt_1), \text{signal}(elt_1), \text{block}(elt_2), \text{signal}(elt_2))$ 
8      $cond_1 = \text{isSending}(elt_1) \wedge \text{isReceiving}(elt_2) \wedge c! = \text{"No"} \wedge c == \text{synchronous}$ 
9      $\implies \text{block}(elt_1)! = \text{block}(elt_2)$ 
10    if  $cond_1$  then
11       $DG \uplus v_1 = \text{vertex}(elt_1\_to\_elt_2)$ 
12       $\uplus v_2 = \text{vertex}(elt_2\_to\_elt_1)$ 
13       $\uplus \text{edge}(v_1, v_2)$ 
14       $\uplus \text{edge}(\text{vertex}(elt_1), v_1)$ 
15       $\uplus \text{edge}(v_1, \text{vertex}(\text{next}(elt_1)))$ 
16       $\uplus \text{edge}(\text{vertex}(elt_2), v_2)$ 
17       $\uplus \text{edge}(v_2, \text{vertex}(\text{next}(elt_2)))$ 
18       $cond_2 =$ 
19       $\text{isSending}(elt_1) \wedge \text{isReceiving}(elt_2) \wedge \text{connect}(\text{block}(elt_1), \text{signal}(elt_1),$ 
20       $\text{block}(elt_2), \text{signal}(elt_2)) == \text{"synchronous"}$ 
21       $cond_2 \implies DG \uplus \text{edge}(v_2, v_1)$ 
22    else
23       $\text{link}(elt_1, elt_2) \implies DG \uplus \text{edge}(\text{vertex}(elt_1), \text{vertex}(elt_2))$ 
24    end
25  end
26  // Optimization: removing empty transitions
27  foreach  $elt \in smd$  do
28    if  $elt == \text{"empty transition"}$  then
29       $DG = DG \text{ vertex}(elt)$ 
30      foreach  $l = \text{link}(elt_1, elt)$  do
31         $DG = DG \text{ edge}(l)$ 
32         $DG \uplus \text{edge}(\text{vertex}(elt_1), \text{vertex}(\text{next}(elt)))$ 
33      end
34    end
35  end
36 end

```

connected to the rest of the graph as follows. (i) Edges from the element vertex (e.g., $\text{vertex}(elt_1)$ and $\text{vertex}(elt_2)$ created at line 4) are respectively connected to v_1 and v_2 . (ii) Edges are created from v_1 / v_2 vertices are respectively connected to the vertex of the next element of elt_1 and elt_2 (lines 12 to 15). Last, if the communication between the reader to the writer: indeed, the latter must wait for the former to be ready to perform the (synchronous) write operation. If the two selected elements do not correspond to

a pair (writer, reader), then an edge is simply added between their respective vertices according to the links specified in their state machines (line 19).

Further, if a transition is empty, then its corresponding vertex can be seen as a simple logical dependency: so it can be captured with an edge. This optimization is taken into account at the end of the algorithm by the optimisation stage: the algorithm removes useless vertices, and updates edges accordingly.

Finally, the dependency graph is built upon control flow dependencies (transitions of the state machines) and communication dependencies (asynchronous, synchronous).

Let us use a toy example to illustrate the construction of a dependency graph with a simple sensor monitoring system. Two sensors provide data to a remote filtering system. The role of the filtering is to decide to store data in a data center, or to drop them.

Blocks *Sensors* and *DataCenter* are synchronously connected by their respective ports to convey signals *value* and *stored* (Figure 2). The two sub-blocks *Sensor1* and *Sensor2* can use both signals. *Filter* and *Center1* blocks are connected with a *query* signal via an asynchronous port connection. The state machines are given in Figure 3. *Sensor1* and *Sensor2* have the same state machine diagram. Note that in these state machines, actions for sending or receiving messages are depicted with a dedicated graphical operator. Nonetheless, sending or receiving a message is considered as an action of a transition between two states.

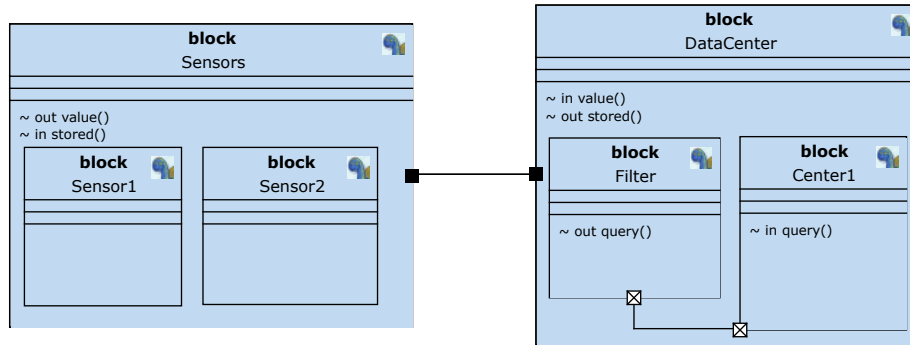


Fig. 2: Internal block diagram of the sensor system

The resulting dependency graph of this toy system (Figure 4) was built using TTool. The graph shows start states of blocks in green, stop states in red, other states in grey, and communication actions in blue. All double arrows between communication states depict possible synchronous communications between a sender and a receiver using the same signal, *e.g.*, between states 37 and 38. On the contrary, an asynchronous communication has a unique dependency arrow from the writer to the reader, *e.g.*, between states 29 and 30. The *Filter* block can synchronize on signals "value" and "stored" either with *Sensor1*, or with *Sensor2*: the graph depicts these two logical dependencies. Last but not least, *Filter* can stop in two different situations: either after getting a value

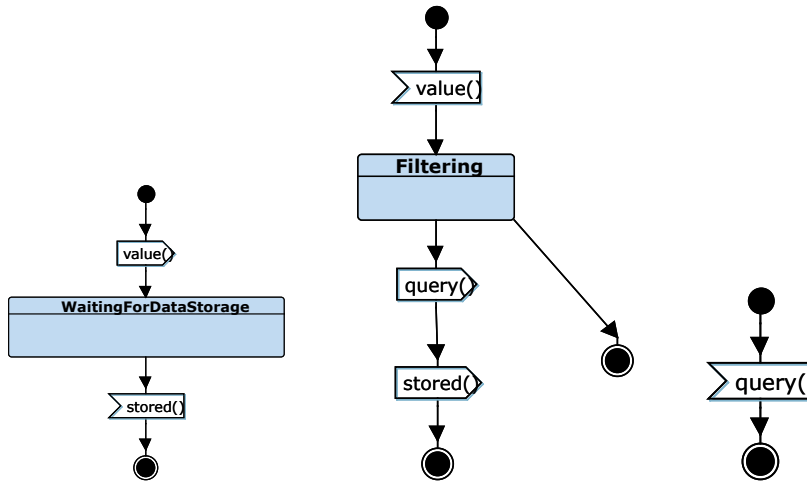


Fig. 3: State machine diagrams: sensors (left), Filter (middle) and Center1 (right)

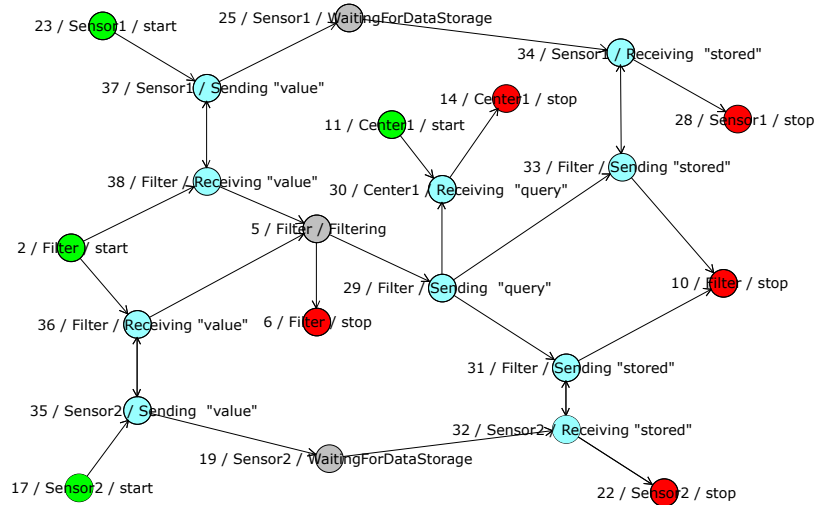


Fig. 4: Dependency graph of the toy system

and dropping it, or after storing the data in the center and informing sensors about the fact that the data has been stored. This explains why *Filter* has two different stop states.

3.5 Reducing a graph with regards to a property

As explained before, reducing the graph w.r.t. to a property decreases the proof complexity. Yet, if a property refers to the whole graph then there is no reduction. Parts of

the graph that are not related to the property can instead be pruned: this is the objective of graph reduction.

Graph reduction consists in marking the vertices related to the selected CTL property p , and then removing all vertices that are not marked, *i.e.*, that are not on a path between start vertices and property vertices. CTL properties explicitly refer to a list of elements in D . In TTool, CTL properties can either relate to a state of a block, (*e.g.*, $E \langle \rangle Block_1.state_1$ means the reachability of $state_1$ in $Block_1$) or can refer to attributes of blocks, (*e.g.*, $A \square Controller.pressure > 0 \& \& Controller.pressure < Controller.threshold$ expresses that in all the system states, the *pressure* attribute of block *Controller* must be between 0 and *threshold*). Another CTL property, called *leads-to* and denoted as " $expr_1 \rightarrow expr_2$ " expresses that the if $expr_1$ is reached then $expr_2$ will eventually be reached. Currently, the reduction works for $A \langle \rangle expr$ and $E \langle \rangle expr$ properties with $expr$ referring to an sending/receiving action or the the state of a state machine.

Algorithm 3 first computes V_p , the list of vertices corresponding to the elements referenced by a CTL property p (*e.g.*, states, sending/receiving actions). Then, each vertex v of DG is added to the reduced graph DG_p if there exists a path from v to at least one vertex in V_p . For liveness properties, we also have to add all the vertices that are connected by one edge to all the vertices v on the path V_p , in order to take into account the beginning of paths not leading to v_p . Finally, if two vertices of DG are in DG_p , then all edges between these two vertices are also added to DG_p .

Algorithm 3: Reduction of dependency Graphs: reduceGraph()

Data: D, DG, p
Result: DG_p

- 1 $next(v)$ denotes $\{v_1 \in DG / edge(v, v_1) \in DG\}$
- 2 $V_p = listOfVertices(D, p)$
- 3 **foreach** vertex $v \in DG$ **do**
- 4 $path(v, V_p) \rightarrow DG_p \uplus v \uplus next(v)$
- 5 **end**
- 6 **foreach** vertex $v_1, v_2 \in DG^2$ **do**
- 7 $e = edge(v_1, v_2) \neq \emptyset \rightarrow DG_p \uplus e$
- 8 **end**

Let us apply this algorithm to the graph given in Figure 4 and to the following property: the liveness of the "Filtering" state of *Filter*, *i.e.*, in CTL: $A \langle \rangle Filter.Filtering$. The resulting graph is given in Figure 5. The graph shows that only the path leading to the Filtering state have been kept, thus cleaning the initial model from useless elements, *e.g.*, the *Center1* block, the behaviour of sensors after sending "value", and the behavior of *Filter* once the Filtering state has been reached.

3.6 Back to a (SysML) model from a dependency graph

Since a dependency graph references all the model elements, it is possible to reconstruct the initial model from a dependency graph. As our prover takes as input a model (and

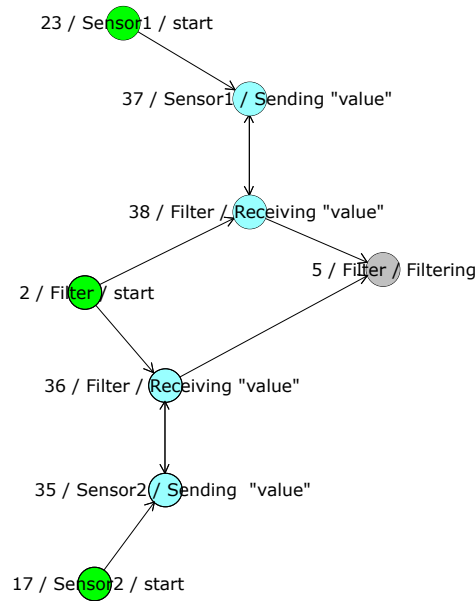


Fig. 5: Dependency graph of the toy system after reduction

a property), once the dependency graph has been reduced according to a given property, we can rebuild a new model from the reduced graph. The new model is reduced with regards to the original one, which means it contains fewer, or the same number of elements as the original model does.

Let us come back to our toy system. As said in the previous subsection, the reduced dependency graph shows that both the structure (blocks and their connections) and the behaviour (state machines) have been impacted. As shown in Figure 6, the *Center1* block has been removed, and the declaration and connection of signals "stored" and "query" has also been removed.

Similarly, the states machines have less states and sending / receiving actions, as shown in Figure 7. Moreover, the state machine of *Center1* has been removed since its behaviour is now empty.

3.7 Dependency graphs for model updates

Figure 8 depicts another usage of dependency graphs. The goal is to avoid re-proving properties after a SysML model was updated. Those properties impacted by the model update are the only ones that need to be proven again. For this, as shown on the left part of Figure 8, a property p is first proved on a design D using a dependency graph DG . Then, D is updated as D' . To know whether p must be proved again on D' , the dependency graph DG' is generated and then compared with DG . If DG is equivalent to DG' according to a bisimulation relation, then the proof of p made on D is still valid for D' . Otherwise, p must be proved for D' . This approach is summarized by algorithm 4 (which is implemented by TTool).

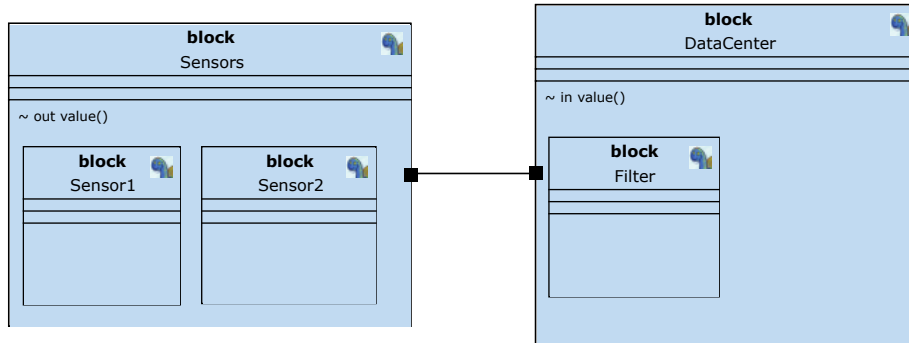


Fig. 6: Internal block diagram of the sensor system after reduction

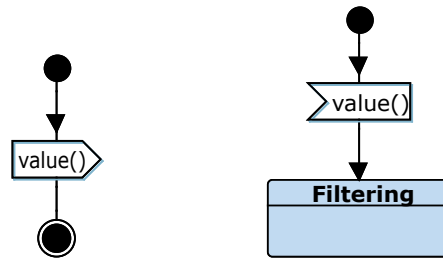


Fig. 7: State machine diagrams built from the reduced dependency graph. From left to right: Sensor1 (and Sensor2) and Filter. Center1 has been removed because its behaviour is now empty.

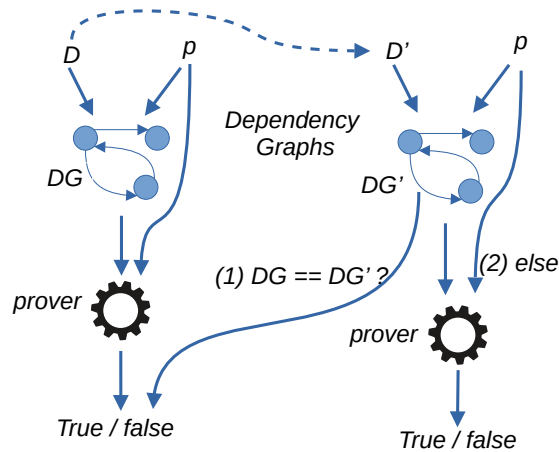


Fig. 8: Decreasing proof complexity using dependency graphs [5]

4 Case Study

The purpose of this section is to evaluate the gain when using model-checking with dependency graphs with regards to model-checking without dependency graphs. Here, model-checking relates to the internal model-checker of TTool.

Algorithm 4: Use of dependency graphs to simplify proofs

Data: D, D', P
Result: DG

```

1  $DG' = computeGraph$ 
2 foreach  $p \in P$  do
3    $DG = computeGraph(D, p)$ 
4    $DG' = computeGraph(D', p)$ 
5   if  $DG \equiv DG'$  then
6      $result_{p'} = result_p$ 
7   else
8      $result_{p'} = prover(DG', p)$ 
9   end
10 end

```

The selected case study is an industrial Ethernet-based Time-Sensitive Networking (TSN) [21] that serves as communication mean for distributed safety-critical applications.

4.1 Time-Sensitive Networking

Time-Sensitive Networking (TSN) [21] 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. Deterministic real-time communication is a crucial requirement in modern embedded systems and cyber-physical systems, *e.g.*, safety-critical industrial, automotive and avionics networks.

The topology of such networks consists of a set of end systems and communication switches. Each end system has a network interface interconnected with communication switches via full-duplex physical links. A TSN network architecture is depicted in Figure 9. The network supports unicast and multicast communications between a set of applications distributed over a number of end systems.

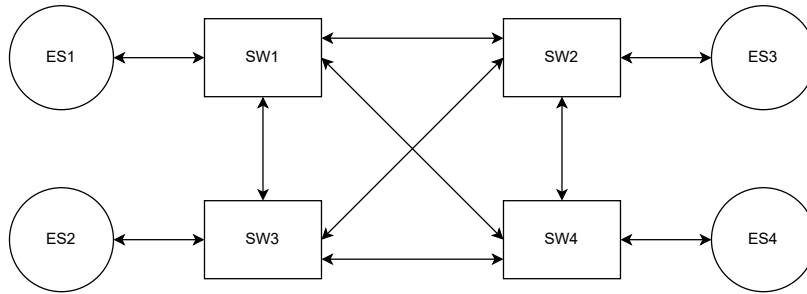


Fig. 9: A TSN network architecture

Reliability and fault tolerance. To achieve determinism, TSN enables transmission of Time-Triggered (TT) flows with bounded end-to-end delay guarantees. TT flows share the network with less critical non-TT flows. Since TT flows carry safety-critical traffic, if a TT flow cannot be delivered correctly (*e.g.*, because of a fault) and in a timely manner, disastrous consequences may occur in safety-critical systems.

The reliability of TT flows may be compromised by two types of faults: *permanent faults* and *transient faults*. Permanent faults may cause link or switch failure and disturb the transmission service, while *transient faults* include packet losses or bit-flips caused by electromagnetic interference, and may compromise the transmission of a message without affecting successive messages. For fault tolerance, TSN enhances redundancy with Frame Replication and Elimination for Reliability (FRER) (IEEE 802.1CB) [20]. According to FRER, multiple routes that do not share any common switches, are allocated for each TT flow. Frames are replicated at the source and transmitted through separate paths to the destination as depicted in Figure 10. Duplicates are eliminated at destinations.

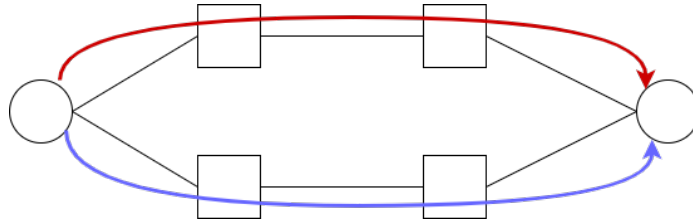


Fig. 10: Frame Replication and Elimination for Reliability (FRER)

In the past few years, several research work on Time-Sensitive networking has been using model-based approaches to formally verify properties of the network. In [16, 27], the UPPAAL model checker is used for timing analysis of TSN, while in [35, 14], network models described in MARTE, respectively EMF are proposed to serve for automatic generation of TSN network configurations. As for the FRER mechanisms on which we focus in this paper, most of the existing research work focus on time scheduling and routing in case of faults and propose optimal or heuristic-based algorithms to compute flows schedules in case recovery is needed, such as presented in [24, 45]. In this paper, we do not focus on aspects related to the timing requirements of flows. Instead, we focus on the behavior of the network when dealing with a failure. We will address the timing analysis of flows in a future work.

4.2 TSN FRER model in SysML

Our aim is to model the FRER mechanism for TSN with SysML and TTool, and then to verify properties on the SysML model. This model is intended to be used as a decision helper for dimensioning TSN networks for safety-critical applications.

Our model considers a communication scenario with two emitting end systems and a receiving end system. End systems are interconnected by three switches and six com-

munication links. Emitting end systems inject data flows into the network on different priority levels as described in Table 1. In our model, we consider flows with only 3 levels of priority (0 - high, 2 - low) of the 8 priority levels available in TSN. High priority level is intended for the transmission of safe-critical flows that also require fault tolerance. So, in our example, the FRER replication/elimination mechanism is applied for flows F10 and F20 of priority 0 for which two different paths are established in the network.

Table 1: TSN flows profile considered in the model

Emitting ES	Flow	Priority	Period	Path	FRER
ES1	F10	0	20	EmittingES1 -> Switch1 -> ReceivingES	Yes
				EmittingES1 -> Switch2 -> ReceivingES	
	F11	1	5	EmittingES1 -> Switch1 -> ReceivingES	No
	F12	2	5	EmittingES1 -> Switch2 -> ReceivingES	No
ES2	F20	0	20	EmittingES2 -> Switch3 -> ReceivingES	Yes
				EmittingES2 -> Switch2 -> ReceivingES	
	F21	1	5	EmittingES2 -> Switch3 -> ReceivingES	No
	F22	2	5	EmittingES2 -> Switch2 -> ReceivingES	No

Figure 11 depicts the SysML internal block diagram of the case study presented in this section. The model is made up of (1) blocks that describe the end systems, (2) switches, and (3) communication links of the network. Each emitting end system is modelled by a set of blocks representing the emission of flows, the classification of flows by priority, the replication of flows in case of safety-critical traffic, and the scheduling mechanisms for the selection of messages on the output ports. The switch model focuses on mechanisms for switching, for priority filtering and for selecting messages based on the Time Aware Scheduling scheduling policy of TSN. The receiving end system is defined by two blocks: one block corresponds to the elimination of duplicate messages in case of redundant transmission and the second block models the reconstruction of flows from the received sequence of messages. Examples of state machines of blocks *FrameReplication* and *FrameElimination* related to the FRER mechanism are given in Figure 12, respectively Figure 13⁴.

4.3 Property verification with (and without) dependency graph

Let us now apply the approach of Figure 1 and Algorithm 1 to a set of properties (section 4.3) we ought to prove on our model. We then compare the proof time with and without dependency graphs and discuss the results.

Evaluated properties We have studied the reachability and liveness of states corresponding to frame generation and sending, frame routing and frame receiving:

⁴ The complete model can be retrieved under TTool -> Examples -> TSN.

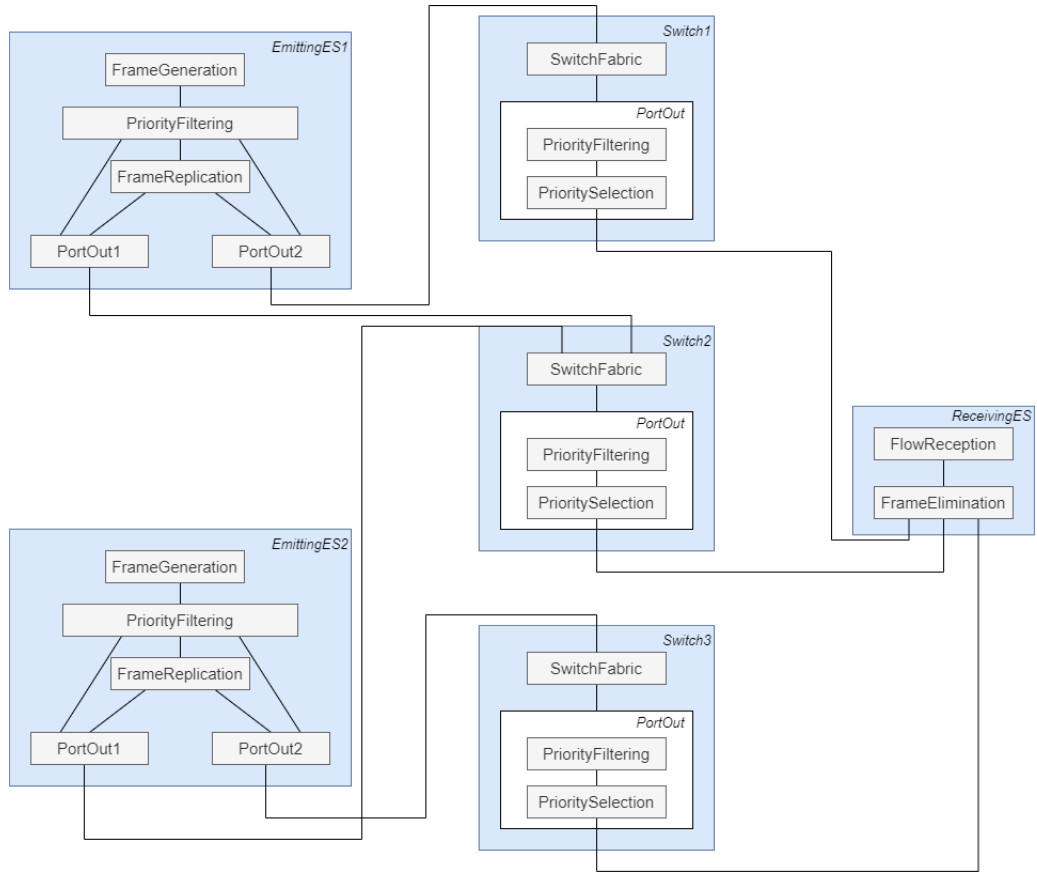


Fig. 11: TSN model with FRER

1. State *Sending1* in the *FrameGeneration*
2. State *Queue0* in *PrioritySelection2*
3. State *Filter2* in block *PriorityFiltering1*
4. State *SendingMessages* in block *FrameReplication_0*
5. State *HandlingMessage1* in block *SwitchFabric1*
6. State *TestingSequence* in block *FrameElimination*
7. State *MessageToFIFO11* in block *PriorityFilteringEndSystem*
8. State *ForwardMessage* in block *CommunicationLink22_0*

These states were selected in order to cover the different networking mechanisms (frame generation, frame replication and elimination, priority filtering and selection, message switching).

Results Table 2 compares results for reachability and liveness analysis. The results do not take into account the time to generate the dependency graph, which is around 10ms:

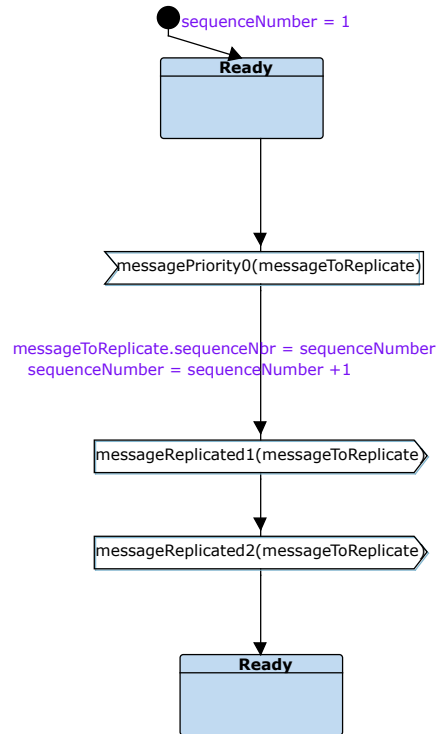


Fig. 12: Example of state machine diagram: block FrameReplication

this generation is made once for all the verification process, thus this time is negligible. But obviously, the time to reduce the dependency graph is taken into account.

The left part of the table concerns the verification time without using a dependency graph, while the right part relies on the dependency graph, and its reduction to the property of interest, for performing the verification. In this table, the verification addresses two kinds of CTL properties, related to states listed at the beginning of this subsection:

- $E \langle \rangle Block.state$: the reachability of a state, *i.e.* there exists at least one execution path that goes through this state, *i.e.* all execution paths have go through this state at some point.

For the case with dependency graph, we provide the time to reduce the dependency graph, which is not negligible for some properties, and we compute the gain, that is:

$$gain = (totalNoDG - totalWithDG) / totalNoDG.$$

Verification times were obtained on a macbook pro running "Big Sur" with 2.3 GHz 8-Core Intel Core i9 and 32 GB of RAM. The oracle JRE 11 was used to execute TTool build 14145 date: 2022/06/30 03:22:06 CET. To avoid Just-In-time compilation delays and load of the machine, each verification was run 10 times and the lowest value was used.

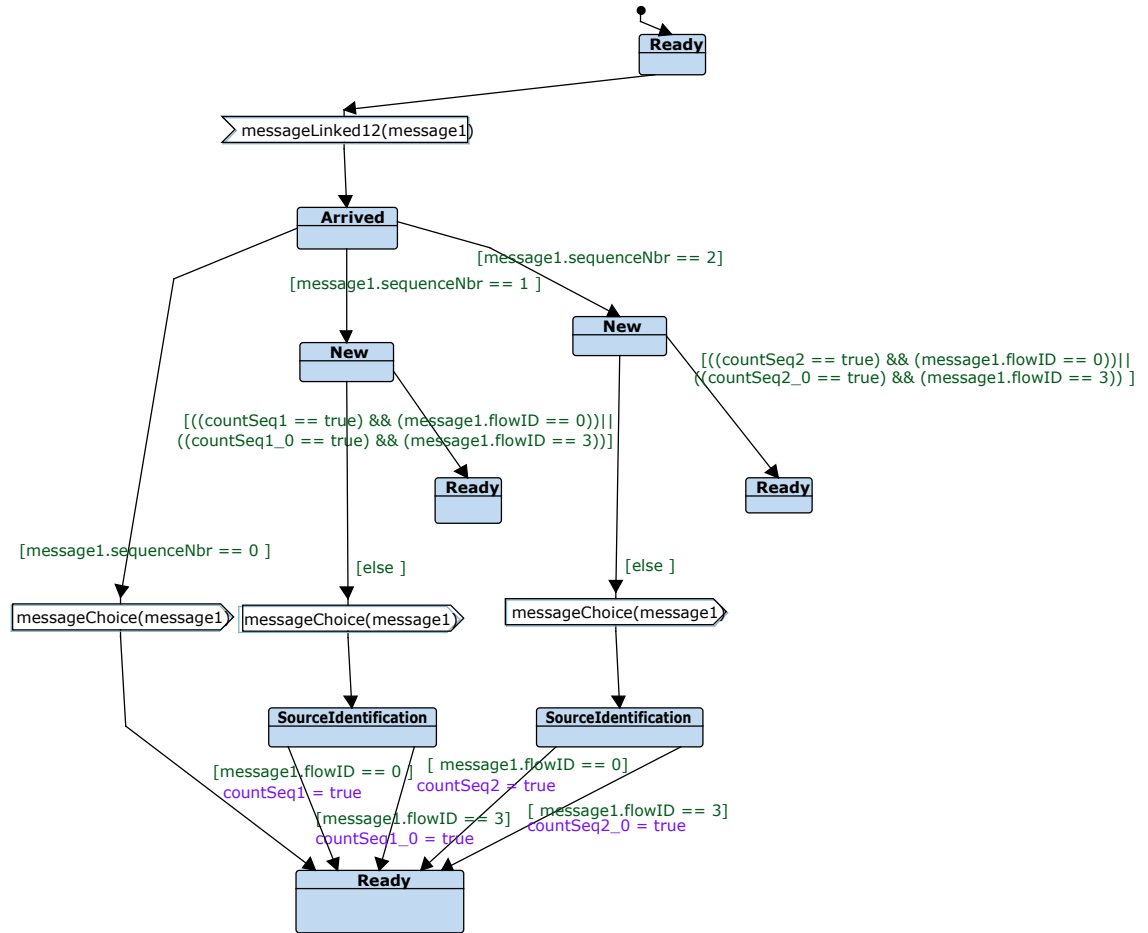


Fig. 13: Example of state machine diagram: block FrameElimination

Results demonstrate a systematic gain, which is more obvious for liveness than for reachability. The gain is far better when the verification process is long: this is exactly what we expect since very short verification (*i.e.*, a few ms) do not need to be accelerated while a gain of 80% is obtained for long verification (a few seconds), as depicted in Figure 14. Thus, for proving one simple property like reachability, it is not worth using dependency graphs. On the contrary, when verification takes more time, like for liveness properties, using dependency graphs always brings a gain.

As a whole, the full verification process takes more than 21 seconds without the dependency graph, and around 4 seconds with the dependency graph, including the time to generate the dependency graph and to reduce it to different graphs.

Table 2: Execution duration (in ms) of the reachability and liveness proof with and without dependency graph

Block/State	Proof duration (ms)							Gain
	No DG			With DG				
	Reachability	Liveness	Total	Reachability	Liveness	Graph reduction	Total	
FrameGeneration/Sending1	7	284	291	6	187	40	233	18%
PrioritySelection2/Queue0	13	16035	16048	11	1939	30	1980	88%
PriorityFiltering1/Filter2	9	343	352	7	249	40	286	18%
FrameReplication_0/SendingMessages	7	566	573	7	359	33	135	30%
SwitchFabric1/HandlingMessage1	11	81	92	7	57	25	89	3%
FrameElimination/TestingSequence	11	984	995	9	718	20	747	24%
PriorityFilteringEndSystem/MessageToFIFO11	8	2768	2776	5	206	34	245	91%
CommunicationLink22_0/ForwardMessage	8	191	199	7	112	23	142	28%
Total	74	21252	21325	59	3827	255	4141	81%

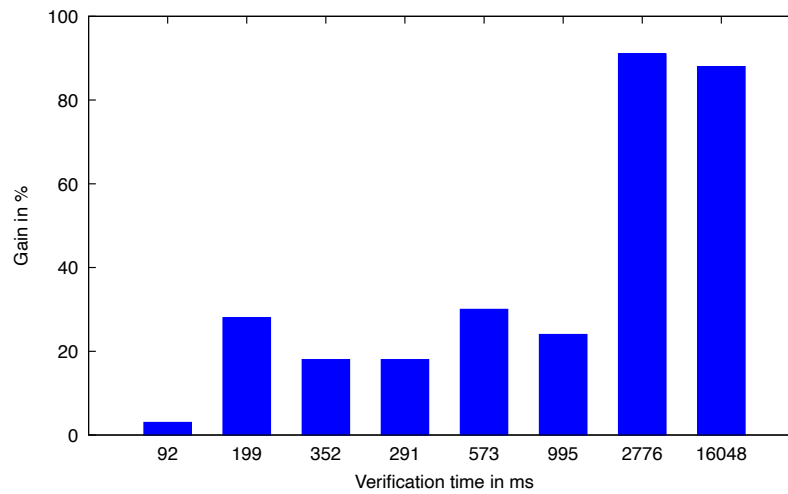
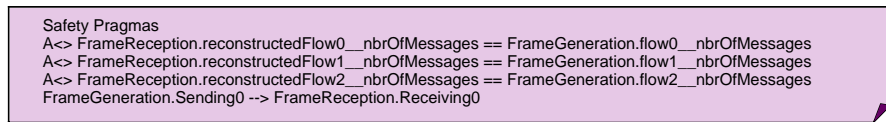


Fig. 14: Gain in function of the verification time. The higher the gain is, the more the use of dependency graphs saves verification time.

Extension to more complex properties In this case study, some properties cannot be expressed as simple reachability or liveness properties. Two properties we ought to verify are:

- Property 1: **arrival of messages**. At least one of the duplicate messages must arrive at destination.
- Property 2: **order of messages**. Messages that take separate paths may arrive earlier on a path than another message that was sent after, thus leading to out of order situations. Depending on the service policy, the out of order can have an impact on the worst case delay analysis as shown in [41].

In TTool, CTL properties can be captured with so-called "safety pragmas". Figure 15 illustrates the CTL properties related to the two properties listed above: the three first pragmas correspond to the first property, while the last one captures the last property with a "leads-to".



```

Safety Pragmas
A<> FrameReception.reconstructedFlow0__nbrOfMessages == FrameGeneration.flow0__nbrOfMessages
A<> FrameReception.reconstructedFlow1__nbrOfMessages == FrameGeneration.flow1__nbrOfMessages
A<> FrameReception.reconstructedFlow2__nbrOfMessages == FrameGeneration.flow2__nbrOfMessages
FrameGeneration.Sending0 --> FrameReception.Receiving0

```

Fig. 15: More advanced properties captured with safety pragmas

Currently, the graph reduction algorithm (Algorithm 3) cannot handle these properties:

- Property 1 uses block attributes. To handle this property, our algorithm would need to identify all the vertices of the dependability graph susceptible to modify the value of all the attributes related in the property.
- Property 2 uses a lead-to, currently not supported. Handling a leads-to property, *e.g.*, $expr1 \text{ --> } expr2$, would mean to identify not only the paths starting from the initial state, but also all paths from all elements of $expr1$ to all elements of $expr2$.

5 Related Work

Formal verification of models and programs has been the subject of many papers and books. Since the current paper relies on dependency inside SysML models to optimize verification of these models, this state of the art section specifically surveys papers that include discussions and proposals for optimizing model verification.

We may first distinguish between two formal verification approaches: static analysis [7, 3] and state space enumeration [4, 36]. The former relies on the structure of the model and avoids explicit enumeration of the states the systems may reach from its initial state. The latter explicitly characterizes the states the system may reach from its

initial state. A direct consequence is that state space enumeration faces the well-known explosion problem, which in practice means the graph of reachable states may turn impossible to compute.

Many solutions have been investigated to lower the state explosion risk by partly exploring the state space of the system without loss of property verification capacity. For instance, in [9], Bourdil *et al.* explore symmetries in systems modeled by Time Petri Nets and implement their proposal in the TINA tool [42]. The current paper explores another avenue in the context of SysML, by looking for dependencies inside the block and state machine diagrams defined by SysML models.

Besides the way the model's state space is traversed to compute a reachability graph, efficiently storing the states of the graph is also an issue. Work in this area has been pioneered by Holzman and implemented into SPIN [17]. Such type of optimized state storage is not yet implemented by the TTool tool considered in the current paper.

SPIN falls in the family of verification tools that we term as 'model checkers'. A Model Checker inputs a model and a property, processes them, and outputs a 'yes/no' answer stating whether the property is satisfied or not. Basically, a model checker explores the state space of the model and potentially identifies states where the property is not satisfied.

Using a model checker first requires to express the properties to be verified. It is a common practice to express the properties in the form of logic formulas expressed, *e.g.*, using Temporal Logic. Property expression is not an easy task and automatic generation of the properties to be verified is an issue [15].

As far as the model checker has been catered with a set of properties and a model, the model checking process may start, raising the following question: understanding the reasons why one or several properties are not satisfied. Identifying counter examples is an issue [22].

More generally, tracing verification results back to the initial model is a complex issue, and regularly the subject of questions asked for to researchers who present their model checkers in papers or talks. As far as the models are expressed in SysML, difficulties in tracing verification results back to the SysML model stems from the fact that SysML tools use external model checkers [26, 34] that had been developed for formal methods such as Petri nets. Translation from UML/SysML to state/transition models has been formalized in the context of Petri nets [12, 40, 19, 33], automata for NuSMV model checker [44], timed automata [37] for UPPAAL model checker, hybrid automata [1], model checker NuSMV [28], model checker nuXmv [39], probabilistic model checker PRISM [31, 1], and a theorem prover [23]. Translation from UML to process algebra has been investigated for RT-LOTOS [4] and CSP [2]. The family of correct by construction specification has been addressed with B [25]. Other contributions such as [46], target a better understanding of verification results output, especially when the property of interest is not satisfied.

Conversely the TTool tool considered in the current paper includes a native model checker and returns yes/no answers inside the SysML model, for properties that are themselves expressed inside the block diagram of the SysML model. In terms of performances, the native model checker of TTool favorably compares to the performance of the first version of TTool where the latter was interfaced with UPPAAL [11].

To be more precise with respect to TTool, let us add that TTool applies model checking to the block and state diagrams of SysML. This is a common point with other research work published, *e.g.*, practice [12, 37, 4]. Nevertheless, some authors apply formal verification to SysML activity diagrams [31, 19, 39].

6 Conclusions

The expected benefits of using an MBSE approach includes early detection of design errors in the life cycle of systems. One or several models are checked against their expected properties using a model checker that takes the models and the properties as inputs, and answers stating whether each property holds or not.

Such a model checking approach has been implemented for SysML. The TTool software implements a user-friendly approach where the properties to be verified are expressed inside the SysML model and their evaluation is reported at the same place in the SysML model. Thus, users of TTool work at the SysML level with no need to learn about the inner workings of the model checker.

The model-checker of TTool, its performance, and its increased user-friendliness had already been discussed in [11, 36]. The current paper proposes a new optimization for the model checking of SysML models. The goal is to avoid a prover to handle a large model when only a subset of it is necessary to evaluate a property. The proposed idea relies on dependencies internal to blocks (control flows) or between blocks (communications).

An early paper [5] by the authors of the current paper used an AFDX (Avionics Full Duplex [18]) network to demonstrate the benefits of relying formal verification of SysML models on dependency graph generation. The current paper optimizes the dependency graph generation algorithm and illustrates the proposed approach using a new generation of real-time network: TSN (Time-Sensitive Networking).

Future work includes the definition of a bisimulation relation to compare dependency graphs. Handling more CTL properties, like leads-to, is also part of our future work. We also intend to use new case studies to demonstrate the efficiency of our approach at larger scale. Last, we intend to support the new syntax (including the textual syntax) and semantics of SysML V2.

Acknowledgement

François Genauzeau has contributed to the SysML diagrams presented in this paper.

References

1. Ali, S.: Formal verification of SysML diagram using case studies of real-time system. *Innovations in Systems and Software Engineering* **14**(6), 245–262 (December 2018). <https://doi.org/10.1007/s11334-018-0318-5>
2. Ando, T., Yatsu, H., Kong, W., Hisazumi, K., Fukuda, A.: Formalization and model checking of SysML state machine diagrams by csp#. In: *Computational Science and Its Applications (ICCSA)*. p. 114–127 (2013). https://doi.org/10.1007/978-3-642-39646-5_9

3. Apvrille, L., de Saqui-Sannes, P.: 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 (2013). https://doi.org/10.1007/978-3-642-38911-5_6
4. Apvrille, L., Courtiat, J.P., Lohr, C., de Saqui-Sannes, P.: TURTLE: A real-time UML profile supported by a formal validation toolkit. *IEEE Transactions on Software Engineering* **30**(7), 473–487 (2004)
5. Apvrille, L., de Saqui-Sannes, P., Hotescu, O., Calvino, A.T.: SysML Models Verification Relying on Dependency Graphs. In: *10th International Conference on Model-Driven Engineering and Software Development*. Vienna, Austria (2022). <https://doi.org/10.5220/0010792900003119>, <https://hal.telecom-paris.fr/hal-03575960>
6. Apvrille, L., de Saqui-Sannes, P., Vingerhoeds, R.A.: An educational case study of using SysML and ttool for unmanned aerial vehicles design. *IEEE Journal on Miniaturization for Air and Space Systems* **1**(2), 117–129 (2020)
7. Ayache, J.M., Courtiat, J.P., Diaz, M.: Rebus, a fault-tolerant distributed system for industrial real-time control. *IEEE Transactions on Computers* **C-31**(7), 637–647 (July 1982). <https://doi.org/10.1109/TC.1982.1676061>
8. Baduel, R., Chami, M., Bruel, J.-M., Ober, I.: Validation in an industrial context: Challenges and experimentation. In: *European Conference on Modelling Foundations and Applications*, Toulouse, France (June 2021)
9. Bourdil, P., Berthomieu, B., Dal Zilio, S., Vernadat, F.: Symmetry reduced state classes for time Petri nets. In: *30th Annual ACM Symposium on Applied Computing*. p. 1751–1758. ACM (2015)
10. Brisacier-Porchon, L., Hammami, O., Boutemy, R.: Modeling a uav in practice: A comparison between rhapsody and capella. In: *IEEE International Symposium on Systems Engineering (ISSE)*. pp. 1–8 (2021). <https://doi.org/10.1109/ISSE51541.2021.9582553>
11. Calvino, A.T., Apvrille, L.: 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) (2021)
12. Delatour, J., Paludetto, M.: 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 (1998). https://doi.org/10.1007/3-540-49255-0_169
13. Dragomir, I., Ober, I., Percebois, C.: Contract-based modeling and verification of timed safety requirements within sysml. *Softw. Syst. Model.* **16**(2), 587–624 (2017). <https://doi.org/10.1007/s10270-015-0481-1>, <https://doi.org/10.1007/s10270-015-0481-1>
14. Farzaneh, M.H., Kugele, S., Knoll, A.: A graphical modeling tool supporting automated schedule synthesis for time-sensitive networking. In: *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. pp. 1–8. IEEE (2017)
15. Gao, H., Dai, B., Miao, H., Yang, X., Duran Barroso, R.J., Walayat, H.: A novel gap approach to automatic property generation for formal verification: The gan perspective. *ACM Transactions on Multimedia Computing, Communications, and Applications* (February 2022). <https://doi.org/10.1145/3517154>
16. Guo, W., Huang, Y., Shi, J., Hou, Z., Yang, Y.: A formal method for evaluating the performance of tsn traffic shapers using uppaal. In: *2021 IEEE 46th Conference on Local Computer Networks (LCN)*. pp. 241–248. IEEE (2021)
17. Holzmann, G.J.: *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley (2004)
18. Hotescu, O., Jaffrès-Runser, K., Scharbarg, J.L., Fraboul, C.: Multiplexing avionics and additional flows on a qos-aware AFDX network. In: *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. pp. 282–289. IEEE (2019)

19. Huang, E., McGinnis, L., Mitchell, S.: Verifying sysml activity diagrams using formal transformation to Petri nets. *Systems Engineering* **23**(1), 118–135 (2019)
20. IEEE: IEEE Standard for Local and metropolitan area networks—Frame Replication and Elimination for Reliability (2017)
21. IEEE: 802.1Q - IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks. ” https://standards.ieee.org/standard/802_1Q-2018.html (2018)
22. Kaleeswaran, A., Nordmann, A., Vogel, T., Grunske, L.: A systematic literature review on counterexample explanation. *Information and Software Technology* **145** (2022). <https://doi.org/10.1016/j.infsof.2021.106800>
23. Kauschl, M., Pfeiffer, R., Raco, D., Rumpe, B.: 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). pp. 1–22. Lecture Notes in Informatics (LNI), Gesellschaft für Informatik (2021)
24. Kong, W., Nabi, M., Goossens, K.: Run-time recovery and failure analysis of time-triggered traffic in time sensitive networks. *IEEE Access* **9**, 91710–91722 (2021)
25. Laleau, R., Mammar, A.: An overview of a method and its support tool for generating B specifications from UML notations. In: ASE2000. Fifteenth IEEE International Conference on Automated Software Engineering. p. 269–272 (2000). <https://doi.org/10.1109/ASE.2000.873675>
26. Leroux-Beaudout, R., Pantel, M., Ober, I., Bruel, J.M.: Model-based systems engineering for systems simulation. In: Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018), Limassol, Cyprus (2018)
27. Lv, J., Zhao, Y., Wu, X., Li, Y., Wang, Q.: Formal analysis of tsn scheduler for real-time communications. *IEEE Transactions on Reliability* **70**(3), 1286–1294 (2020)
28. Mahani, M., Rizzo, D., Paredis, C., Wang, Y.: Automatic formal verification of SysML state machine diagrams for vehicular control system. SAE Technical Paper (2021). <https://doi.org/10.4271/2021-01-0260>
29. Modeler, C.S.: <https://www.3ds.com/products-services/catia/products/no-magic/comeo-systems-modeler/>. Retrieved May 16, 2022 (2022)
30. OMG: OMG Systems Modeling Language. Object Management Group, <https://www.omg.org/spec/SysML/1.5> (2017)
31. Ouchani, S., Ait Mohamed, O., Debbabi, M.: A formal verification framework for SysML activity diagrams. *Expert Systems with Applications* **41**(6) (2014). <https://doi.org/10.1016/j.eswa.2013.10.064>
32. Papyrus: <https://www.eclipse.org/papyrus/>. Retrieved May 16, 2022 (2022)
33. Rahim, M., Boukala-Loulalen, M., Hammad, A.: Hierarchical colored Petri nets for the verification of SysML designs - activity-based slicing approach. In: 4th Conf. on Computing Systems and Appli. (CSA 2020). Lecture Notes in Networks and Systems, vol. 199, pp. 131–142. Algiers, Algeria (dec 2020), <https://publiweb.femto-st.fr/tntnet/entries/17274/documents/author/data>
34. Rhapsody: <https://www.ibm.com/fr-fr/products/architect-for-systems-engineers>. Retrieved May 16, 2022 (2022)
35. Samson, M., Vergnaud, T., Dujardin, É., Ciarletta, L., Song, Y.Q.: A model-based approach to automatic generation of tsn network simulations. In: 2022 IEEE 18th International Conference on Factory Communication Systems (WFCS). pp. 1–8. IEEE (2022)
36. de Saqui-Sannes, P., Apvrille, L., Vingerhoeds, R.A.: Checking SysML Models against Safety and Security Properties. *Journal of Aerospace Information Systems* pp. 1–13 (Nov 2021)
37. Schafer, T., Knapp, A., Merz, S.: Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science* **55**, 357–369 (2001). [https://doi.org/10.1016/S1571-0661\(04\)00262-2](https://doi.org/10.1016/S1571-0661(04)00262-2)

38. de Souza, F.G.R., de Melo Bezerra, J., Hirata, C.M., de Saqui-Sannes, P., Apvrille, L.: Combining stpa with sysml modeling. In: IEEE International Systems Conference (SysCon). pp. 1–8 (2020). <https://doi.org/10.1109/SysCon47679.2020.9275867>
39. Staskal, O., Simac, J., Swayne, L., Rozier, K.Y.: Translating sysml activity diagrams for nuxmv verification of an autonomous pancreas. In: SESS22). pp. 1–6 (2022)
40. Szmuc, W., Szmuc, T.: 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 (2018). <https://doi.org/10.23919/MIXDES.2018.843687>
41. Thomas, L., Mifdaoui, A., Boudec, J.Y.L.: Worst-case delay bounds in time-sensitive networks with packet replication and elimination. arXiv preprint arXiv:2110.05808 (2021)
42. TINA: Time Petri net analyzer, <http://projects.laas.fr/tina/>. Retrieved October 31, 2020 (2020)
43. TTool: <https://ttool.telecom-paris.fr/>. Retrieved May 11, 2022 (2022)
44. Wang, H., Zhong, D., Zhao, T., Ren, F.: Integrating model checking with sysml in complex system safety analysis. IEEE Access 7, 16561–16571 (2019). <https://doi.org/10.1109/ACCESS.2019.2892745>
45. Zhou, Y., Samii, S., Eles, P., Peng, Z.: Reliability-aware scheduling and routing for messages in time-sensitive networking. ACM Transactions on Embedded Computing Systems (TECS) 20(5), 1–24 (2021)
46. Zoor, M., Apvrille, L., Pacalet, R.: Execution Trace Analysis for a Precise Understanding of Latency Violations. In: International Conference on Model Driven Engineering Languages and Systems (MODELS). Fukuoka (virtual), Japan (Oct 2021), <https://hal.telecom-paris.fr/hal-03349254>