SysML Models Verification Relying on Dependency Graphs

Ludovic Apvrille[1][a], Pierre De Saqui-Sannes[2][b], Oana Hotescu[2][c]
and Alessandro Tempia Calvino[3][d]
[1]*LTCI, Telecom Paris, Institut Polytechnique de Paris, France*
[2]*ISAE-SUPAERO, Université de Toulouse, France*
[3]*LSI, École Polytechnique Fédérale de Lausanne, Switzerland*

Keywords: MBSE, SysML, Formal Verification, Model Checking.

Abstract: Formal verification of SysML models contributes to early detection of design errors early in the life cycle of systems. Incremental modeling of systems leads to the repeated verification of parts of systems models that were already verified in previous versions of the SysML model. This paper proposes to optimize the verification process by generating dependency graphs from SysML models. This revisited verification technique is now supported by TTool. It is illustrated with an Avionics Full DupleX network.

1 INTRODUCTION

The Systems Modeling Language, (OMG, 2017) or SysML for short, is an international standard at OMG and currently under revision to give birth to a new standard named SysML v2 (OMG, 2020). Modeling with SysML is more than just drawing the different diagrams. Formal verification tools, such as model checkers, offer possibilities to check SysML models against their expected properties.

A model-checker takes as input a model and a property, and outputs a result (true/false). The paper shows how model-checking from SysML models can be more efficient by using dependency graphs. The main idea behind this contribution is 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.

Moreover, assuming a model m of system s has been checked and needs to be updated to integrate new requirements, what will be the price of formally verifying an updated model m' of s? In other words, could we avoid entirely re-verifying m'? This paper positively answers to these questions. Again, the proposed solution relies on dependency graphs.

This 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 (TTool, 2021; de Saqui-Sannes et al., 2021), both for the proof of reachability and liveness properties.

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. Section 5 surveys related work. Section 6 concludes the paper and outlines future work.

[a] https://orcid.org/0000-0002-1167-4639
[b] https://orcid.org/0000-0002-1404-0148
[c] https://orcid.org/0000-0001-6612-8574
[d] https://orcid.org/0000-0003-1312-2907

2 SysML

The SysML V1 standard (OMG, 2017) defines nine types of diagrams that may be used inside one model to cover the requirement capture, analysis and design phases in the life cycle of systems. During the design phase, one defines the architecture of the system using Block Definition Diagrams (BDD) and Internal Block Diagrams (IBD). This paper merges BDD and IBD into one diagram: the Block Instance Diagram (BID). Each block instance 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

Apvrille, L., De Saqui-Sannes, P., Hotescu, O. and Calvino, A.
SysML Models Verification Relying on Dependency Graphs.
DOI: 10.5220/0010792900003119
In *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2022)*, pages 174-181
ISBN: 978-989-758-550-0; ISSN: 2184-4348

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.

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 B, by $\mathcal{S}_o$ the set of all output signals of 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 $(\mathcal{B} \times \mathcal{S}_o \times \mathcal{B} \times \mathcal{S}_i) \rightarrow Bool$ that returns true if one output signal of a block is associated to an input signal of a block.

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 $\langle s0, S, T \rangle$ 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.
- $after(t_{min}, t_{max})$ enables firing the transition only after between t_{min} and t_{max} time units have elapsed.
- *condition* is a Boolean expression that conditions the execution of the transition.This Boolean expression can use block attributes.
- *Actions* is a ordered set of *action*. These actions 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.

2.3 Formal Verification with TTool

TTool (TTool, 2021; Apvrille et al., 2020) is a free and open source framework for the design and verification of embedded systems. The TTool model checker (Calvino and Apvrille, 2021) 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.

3 DEPENDENCY GRAPHS

This section formalizes the verification problem addressed by the paper. A design is made up of a (1) SysML block instance diagram and its associated state machines, and (2) a set of properties. A verification process is defined and dependency graphs are used to simplify the proof associated with that verification process.

3.1 Definition of a System

Definition 1. A **System** has a Design and a set of Properties to be verified, as defined in (Calvino and Apvrille, 2021) .

$$S = \langle D, P \rangle \tag{1}$$

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. *prover* returns $true$ if p is satisfied by D (also denoted as $D \models p$), $false$ otherwise.

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

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

3.3 Optimizing Proofs

To prove a property, a prover considers the whole design, even if some elements of the design 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 this paper's contribution. The purpose of this contribution is to eliminate parts of the models that may slow down the proof without impacting the proof result. The proposed solution is to compute a dependency graph from the input model D, then to reduce D to D' according to p, and then to use D' and p as input for the *prover* (see Algorithm 1).

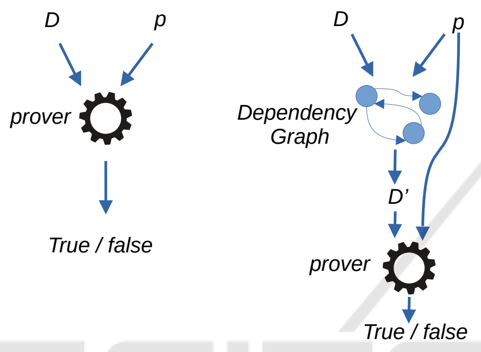

Figure 1: Proofs without/with dependency graphs.

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
```

3.4 Design to Dependency Graph

A dependency graph can be computed from a block instance diagram D (Algorithm 2).

$$DG = computeGraph(D)$$

For each *smd* of $B \in Bl$, for each element of the state machine (states, afters, conditions, actions), we generate a vertex v_e in DG (line 4). Then, the algorithm looks for two elements and checks whether these two elements are *read* and *write* operators connected through a channel (line 7, $cond_1$). If this is the case, then a new vertex is added for each operator (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 connected to the rest of the graph as follows. (i) Edges from the operator vertex to v_1 and v_2 and (ii) edges between the v_1 / v_2 to the vertex of the next operators of the writer and the reader in the state machine (lines 11 to 14). Last, if the communication between the two operators is synchronous ($cond_2$, line 16) then an edge is also added from 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 18). Finally, the dependency graph is built upon control flow dependencies (links of the state machines) and communication dependencies (asynchronous, synchronous). Finally, the dependency graph captures all the elements of the system (blocks, state machines, communications), with a focus on the dependencies between these elements.

Let us use a toy example to illustrate the construction of a dependency graph with a simple client-webserver system. To handle two requests in parallel, the webserver needs two instances. Blocks *Client* and *Webserver* are connected by their respective ports to convey signals *query* and *answer*. The state machines of Client and Instance1 are depicted on the left and right parts of Figure 3, respectively. Instance2 and Instance1 have the same state machine.

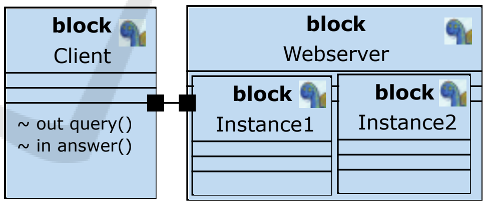

Figure 2: Internal block diagram of the toy system.

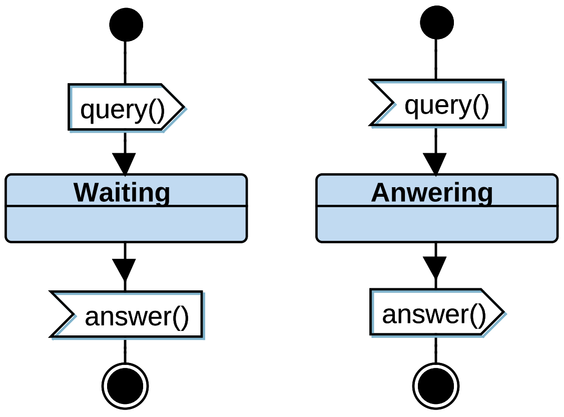

Figure 3: State machine diagrams: Client (left) and Instances (right).

Algorithm 2: Building a dependency graph from a model.

```
Data: $D$
Result: $DG$
$DG = emptyGraph$
foreach $smd$ of $Bl$ of $D$ do
    foreach $elt \in smd$ do
        $DG \uplus vertex(elt)$
    end
    foreach $elt_1, elt_2 \in smd^2$ do
        $cond_1 = isSending(elt_1) \wedge$
          $isReceiving(elt_2) \wedge connect($
          $block(elt_1), signal(elt_1), block(elt_2),$
          $signal(elt_2))! = "No"$
        if $cond_1$ then
            $DG \uplus v_1 = vertex(elt_1\_to\_elt_2) \uplus$
              $v_2 = vertex(elt_2\_to\_elt_1)$
            $\uplus edge(v_1, v_2)$
            $\uplus edge(vertex(elt_1), v_1)$
            $\uplus edge(v_1, vertex(next(elt_1))$
            $\uplus edge(vertex(elt_2), v_2)$
            $\uplus edge(v_2, vertex(next(elt_2))$
            $cond_2 = isSending(elt_1) \wedge$
              $isReceiving(elt_2) \wedge$
              $connect(block(elt_1), signal(elt_1),$
              $block(elt_2), signal(elt_2)) ==$
              $"synchronous"$
            $cond_2 \implies DG \uplus edge(v_2, v_1)$
        else
            $link(elt_1, elt_2) \implies DG \uplus$
              $edge(vertex(elt_1), vertex(elt_2))$
        end
    end
end
```

The resulting dependency graph (Figure 4) shows in green the start of Client. One can notice that the Client depends on a synchronization with either Instance1 (top left) or Instance2 (down right).

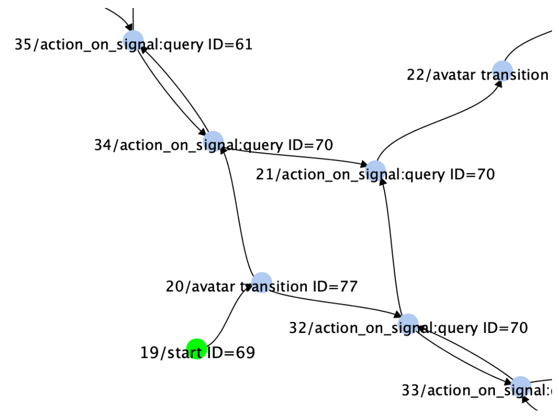
35/action_on_signal:query ID=61
22/avatar transition
34/action_on_signal:query ID=70
21/action_on_signal:query ID=70
20/avatar transition ID=77
32/action_on_signal:query ID=70
19/start ID=69
33/action_on_signal:

Figure 4: Sub dependency graph of the toy system.

3.5 Reducing a Graph with Regards to a Property

Graph reduction consists in marking the vertices related to the selected property p, and then removing all vertices that are on no path between start vertices and property vertices. The CTL property explicitly refers to a list of operators in D. Indeed, in TTool, CTL properties can either relate to a state of a block (*e.g.,* $E <> Block_1.state_1$ means the reachability of $state_1$ in $Block_1$) or can refer to attributes of blocks (*e.g.,* $A[]Controller.pressure > 0\&\&Controller.pressure < Controller.threshold$ expresses that in all the system states, the *pressure* attribute of block *Controller* must be between 0 and *threshold*). Algorithm 3 first computes V_p, the list of vertices corresponding to the operators (*e.g.,* states) referred to by p. Then, each vertex of D is added to the reduced graph DG_p if there exists a path from v to at least one vertex in 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. Finally, we expect that proving a property on the reduced system is equivalent to proving it on the original system. Obviously, if a property influences the whole graph then there is no reduction. Parts of the graph that are not influenced by the property can instead be pruned.

Algorithm 3: Reduction of dependency Graphs: reduceGraph().

```
Data: $D$, $DG$, $p$
Result: $DG_p$
$V_p = listOfVertices(D, p)$
foreach vertex $v \in DG$ do
    $path(v, V_p) \rightarrow DG_p \uplus v$
end
foreach vertex $v1, v2 \in DG^2$ do
    $e = edge(v1, v2) \neq \emptyset \rightarrow DG_p \uplus e$
end
```

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. Since our prover takes as input a model (and 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 operators as the original model.

3.7 Dependency Graphs for Model Updates

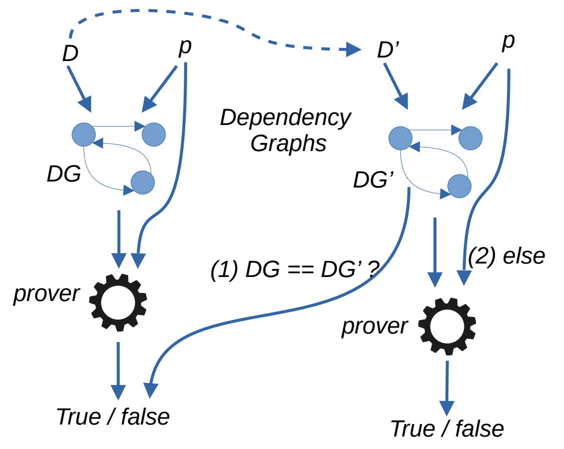

Figure 5: Decreasing proof complexity using dependency graphs.

Figure 5 depicts another usage of dependency graphs. The goal is to avoid reproving 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 5, 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 according to a bisimulation relation to DG', 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).

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
```

4 CASE STUDY

A distributed system illustrates how dependency graphs can reduce the proof complexity.

4.1 AFDX Network

AFDX (Avionics Full DupleX switched Ethernet) is the *de facto* embedded avionics network, deployed in Airbus A380 and A350 civilian aircrafts, to transport command and control data flows between distributed systems running avionics applications in the flight domain. These applications are time-critical. Therefore, the AFDX communication network has to provide a set of guarantees, such as the arrival of messages at the destination application before a deadline, the reception of messages in the transmission order, the absence of loss of messages due to buffer overflow or to some failure. The first requirement can be ensured with worst-case delay analysis methods such as the well-known Network Calculus (Frances et al., 2006), while the others can be verified with formal methods such as the model-based approach discussed in this paper. An AFDX network typically has a set of network interfaces called 'End Systems' (ES), each one corresponding to an avionics module, interconnected by switches and communication links. Dedicated unidirectional communication channels called Virtual Links (VLs) connect ES. A VL is specified with a minimum and maximum frame length (between 64 bytes and 1,518 bytes) and the Bandwidth Allocation Gap (BAG), *i.e.*, the minimum interval between consecutive frames. This interval is enforced at the emitting ES by a traffic regulator and verified at switch level by a filtering and policing function. Possible BAG values are in the set of powers of $2^k, k \in \{0, 1, \ldots, 7\}$. Multicast routes are defined for each VL by static switching tables implemented by each switch in the network. ES using multiple VLs and switches output ports crossed by several VLs need a scheduling mechanism to select frames requiring access to the same physical link. Typical Scheduling policies are FIFO, Static Priority Queuing (SPQ), and local table scheduling (Hotescu et al., 2019).

4.2 AFDX Model in SysML

To formally verify the above listed properties of the AFDX network, we model in SysML the AFDX mechanisms enforcing communication flows between end systems[1]. Our model considers two emitting end systems, two switches and a receiving end system.

[1]The model is available from TTool: File, then "Open project from TTool repository". Then, select "AFDX"

Flows of data are transmitted via 3 VLs. Main characteristics are given in Table 1 and Figure 6.

Table 1: Characteristic of VLs in the network architecture.

VL	BAG	Lmax	Source ES	Destination ES
VL1	8	1379	*ES*_1	*ES*_3
VL2	16	424	*ES*_1	*ES*_3
VL3	8	1385	*ES*_2	*ES*_3

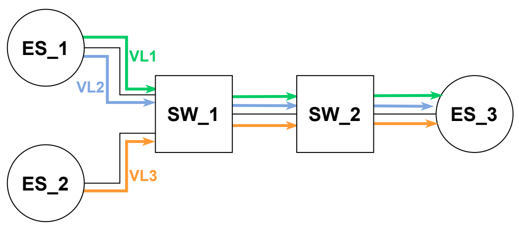

Figure 6: AFDX network architecture.

Our SysML model capture each equipment of the network described before. The model of an end system has blocks modelling the emission of flows as well as blocks modeling traffic regulation and scheduling mechanisms (Figure 7). The switch model focuses on mechanisms of filtering, switching, classification and scheduling of messages. The state machine of an emitting end system is given in Figure 8.

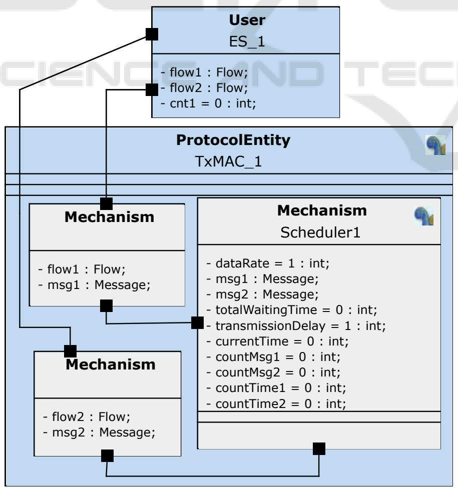

Figure 7: Emitting end system model in SysML.

4.3 Properties Verification with (and without) Dependency Graph

We now apply the approach of Figure 1, Algorithm 1, to a set of properties (section 4.3.1) we ought to prove on our model. We then compare the proof time with and without dependency graphs. The tests were run on an Intel core i9 computer, with 8 cores at 2.3GHz,

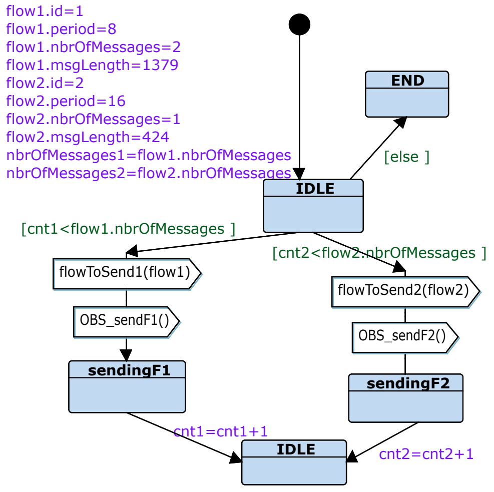

Figure 8: State machine for an emitting end system. A flow correspond to a VL of period equal to BAG.

32 Go RAM, running MacOS and Java 8, with TTool build version 13854 (August 2021).

4.3.1 Evaluated Properties

We have studied the reachability and liveness of four different meaningful states of the model:

1. State *RegulatedF3* in the *TrafficRegulator* block
2. State *ScheduledMsg1* in *SWScheduler*
3. State *FilteredF3* in block *Filtering*
4. State *ReceivedF2* in block *Demultiplexer*

These states were selected so as to cover the different networking mechanisms (scheduling, regulation, filtering, demultiplexing) and to cover all virtual channels: the number at the end of the state refers to the virtual channel number. For instance, *RegulatedF3* refers to the regulation state for virtual channel 3.

4.3.2 Results

Table 2 compares results for reachability and liveness analysis. For the case with dependency graph, we have also added a ‘Graph processing’ column that corresponds to the time taken to generate the dependency graph from the model, and to select a sub-part of this graph for the considered property. While this graph is the same for all properties, we have assumed that we regenerate it each time for each property. Note that the same reduced graph can be used both for reachability and liveness without being recomputed, because the property concerns the same model element, so the dependencies are the same. Whatever the considered state, the time to generate the dependency graph (Algorithm 2) and to compute its relevant sub-part (Algorithm 3) was equal to around 15ms.

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 processing	Total	
TrafficRegulator/ RegulatedF3	6	50	56	6	8	15	29	48%
SWScheduler/ ScheduledMsg1	104	495	599	18	36	15	69	88%
Filtering/ FilteredF3	98	737	835	19	51	15	85	89%
Demultiplexer/ ReceivedF2	114	1734	1848	34	86	15	135	92%

The results show, for this case study, that using the dependency graph is always better, except when proving only the reachability of *TrafficRegular/RegulatedF3*. But even for the this state, proving both its reachability and liveness leads to an important time gain (58%) when using the dependency graph with regard to the usual way. Overall, using dependency graphs leads to a time gain of 80%. Obviously, we have used only a few properties for one given (quite complex) model, but these results are already very encouraging, and correspond to what we expected: cutting a model according to a property by capturing dependencies help reducing proof complexity. Moreover, since our approach leads to consider reduced models, we do expect to be able to perform reachability and liveness proofs in models with combinatory explosion.

5 RELATED WORK

A survey of the literature indicates that papers on the formal verification of SysML models mostly address either activity diagrams (Ouchani et al., 2014; Huang et al., 2019) or state machine diagrams (Delatour and Paludetto, 1998; Schafer et al., 2001; Apvrille et al., 2004). In this paper, the model checker of TTool falls in the category of tools that take state machine diagrams as input. The current version of TTool does not enable formal verification of activity diagrams.

A survey of the literature also indicates that many authors propose to translate SysML activity or state machine diagrams into a formal model, and to reuse a model checker supporting that formal model. Translation from UML/SysML to state/transition models has been formalized in the context of Petri nets (Delatour and Paludetto, 1998; Szmuc and Szmuc, 2018; Huang et al., 2019; Rahim et al., 2020), automata for NuSMV model checker (Wang et al., 2019), timed automata (Schafer et al., 2001) for UPPAAL model checker, hybrid automata (Ali, 2018), model checker NuSMV (Mahani et al., 2021), probabilistic model checker PRISM (Ouchani et al., 2014; Ali, 2018), and a theorem prover (Kausch1 et al., 2021). Translation from UML to process algebra has been investigated for RT-LOTOS (Apvrille et al., 2004) and CSP (Ando et al., 2013). The family of correct by construction specification has been addressed with B (Laleau and Mammar, 2000). Other contributions such as (Zoor et al., 2021), target a better understanding of verification results output, especially when the property of interest is not satisfied.

Translating a SysML model into a formal one raises the issue of coming back to the original model (*e.g.,* blocks, state machines). This issue is not systematically addressed in the literature (De Antoni and Mallet, 2012). By contrast, the native model checker of TTool can backtrack verification results to the initial SysML model and does not oblige the developers of the SysML diagrams to go into the inner workings of the model checker. Further, 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 (Calvino and Apvrille, 2021).

The use of dependency graphs could apply to other model-checking approaches since the idea is rather to feed into the prover a reduced model —so without modifying the prover— that is sufficient to prove the selected properties.

6 CONCLUSIONS

Model checkers state whether a system satisfies properties or not. The paper presents a new technique, based on dependency graphs, to reduce the complexity of model-checking performed from SysML models. Moreover, the same approach can be used to check whether a property must be checked again when a modification has been done on a model. The interest of dependency graphs is demonstrated for an

AFDX network subset.

Future works include the definition of a bisimulation relation to compare dependency graphs. 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.

REFERENCES

Ali, S. (2018). Formal verification of SysML diagram using case studies of real-time system. *Innovations in Systems and Software Engineering*, 14(6):245–262.

Ando, T., Yatsu, H., Kong, W., Hisazumi, K., and Fukuda, A. (2013). Formalization and model checking of SysML state machine diagrams by csp#. In *Computational Science and Its Applications (ICCSA)*, page 114–127.

Apvrille, L., Courtiat, J.-P., Lohr, C., and de Saqui-Sannes, P. (2004). TURTLE: A real-time UML profile supported by a formal validation toolkit. *IEEE Transactions on Software Engineering*, 30(7):473–487.

Apvrille, L., de Saqui-Sannes, P., and Vingerhoeds, R. A. (2020). 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.

Calvino, A. T. 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'2021), Vienna, Autrichia (online)*.

De Antoni, J. and Mallet, F. (2012). Timesquare: Treat your models with logical time. In *International Conferenceon Modelling Techniques and Tools for Computer Per-formance Evaluation*, page 34–41. Springer.

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.

Delatour, J. and Paludetto, M. (1998). UML/PNO: A way to merge UML and Petri net objects for the analysis of real-time systems. In *Oriented Technology: ECOOP'98 Workshop Reader*, page 511–514.

Frances, F., Fraboul, C., and Grieu, J. (2006). Using network calculus to optimize the AFDX network. In *ERTS*, Toulouse, France.

Hotescu, O., Jaffrès-Runser, K., Scharbarg, J.-L., and Fraboul, C. (2019). Multiplexing avionics and additional flows on a qos-aware AFDX network. In *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 282–289. IEEE.

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

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

Laleau, R. and Mammar, A. (2000). 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*, page 269–272.

Mahani, M., Rizzo, D., Paredis, C., and Wang, Y. (2021). Automatic formal verification of SysML state machine diagrams for vehicular control system. *SAE Technical Paper*.

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

OMG (2020). *SysML v2, https://mbse4u.com/2020/10/17/new-incremental-sysml-v2-release-2020-09/*.

Ouchani, S., Ait Mohamed, O., and Debbabi, M. (2014). A formal verification framework for SysML activity diagrams. *Expert Systems with Applications*, 41(6).

Rahim, M., Boukala-Loualalen, M., and Hammad, A. (2020). Hierarchical colored Petri nets for the verification of SysML designs - activity-based slicing approach. In *4th Conf. on Computing Systems and Appli. (CSA 2020)*, volume 199 of *Lecture Notes in Networks and Systems*, pages 131–142, Algiers, Algeria.

Schafer, T., Knapp, A., and Merz, S. (2001). Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55:357–369.

Szmuc, W. and Szmuc, T. (2018). Towards embedded systems formal verification translation from SysML into Petri nets. In *25th International Conference Mixed Design of Integrated Circuits and System (MIXDES)*, pages 420–423.

TTool (2021). https://ttool.telecom-paris.fr/. *Retrieved September 10, 2021*.

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.

Zoor, M., Apvrille, L., and Pacalet, R. (2021). Execution Trace Analysis for a Precise Understanding of Latency Violations. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Fukuoka (virtual), Japan.