Direct Model-checking of SysML Models

Alessandro Tempia Calvino[1,2] and Ludovic Apvrille[1]
[1]*LTCI, Telecom Paris, Institut Polytechnique de Paris, Sophia-Antipolis, France*
[2]*Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland*

Keywords: SysML, Model-checking, Formal Verification, Embedded Systems.

Abstract: Model-checking intends to verify whether a property is satisfied by a model, or not. Model-checking of high-level models, e.g. SysML models, usually first requires a model transformation to a low level formal specification. The present papers proposes a new model-checker that can be applied (almost) directly to the SysML model. The paper first explains how this model-checker works. Then, we explain how it can efficiently check CTL-like properties. Finally, the paper discusses the performance of this model-checker integrated in the TTool framework.

1 INTRODUCTION

Tools for designing critical systems with UML/SysML usually offer a way to either simulate the model or to perform verification from the models. Simulation helps understanding and debugging the model, while formal verification offers strong guarantees on the system. Typical properties that are verified are the absence of deadlocks, the non reachability of error states, and the reachability or liveness of all expected model elements.

UML/SysML tools usually rely on external verification tools (e.g. model-checkers) to perform formal proofs. Commonly, the model to verify is first translated to an intermediate formal specification (e.g. in UPPAAL[1]) before being checked against properties, also expressed in a formal language (e.g. CTL). Finally, results are traced back to the model. The transformation to a formal specification impacts the performance, raises semantic issues, and makes the backward transformation less obvious. For instance, UPPAAL supports only 16-bit integer, so if the original model relies on 32-bit integer, there is a semantic gap between the original model and the formal specification. Similarly, communication semantics of the model (infinite/finite FIFO, synchronization, etc.) sometimes force to ignore some operators or to translate them in an approximate way.

To tackle this issue, we introduce in this paper a new model-checker that can directly work from a SysML model. We assume the structure of the system is described with block definition diagrams and internal block diagrams. The behavior of each block is described with SysML state machines. Our model-checker is SysML-aware in the sense that it directly works with blocks, ports of blocks, connectors between ports, states, guards, actions, etc. The paper shows how the model-checker works, and how it can prove CTL-like properties (reachability, liveness, leads to, deadlock freeness) from a SysML model, and outputs counterexamples when a property is not satisfied. We also show that our model-checker compares to UPPAAL in terms of performance, while avoiding model transformations.

Section 2 presents similar contributions. Section 3 introduces the SysML profile (AVATAR) at the root of the present work. Section 4 introduces the model-checker and explains its main features. Then, section 5 shows how properties can be described in SysML and evaluated with the model-checker. Section 6 presents the main implementation tricks to enhance performance, and compares our model-checker to UPPAAL. Section 7 concludes the paper.

2 RELATED WORK

2.1 Systems Analysis

Systems analysis can be performed using different techniques. The latter can be classified into different categories, including emulation (Thiele et al.,

[1]http://www.uppaal.org/

Calvino, A. and Apvrille, L.
Direct Model-checking of SysML Models.
DOI: 10.5220/0010256302160223
In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2021)*, pages 216-223
ISBN: 978-989-758-487-9

2007), implementation (Thiele et al., 2007), simulation (Stemmer et al., 2019) (Thiele et al., 2007) or hybrid analysis which combines formal and simulation approaches (Stemmer et al., 2019).

Simulation and formal approaches are the most used ones in the domain of performance estimation of embedded systems (Thiele et al., 2007) (Viehl et al., 2006). Simulation tools and industrial frameworks e.g. Koski (Kangas et al., 2006) can only consider a limited set of execution traces and corner cases are usually unknown (Thiele et al., 2007). Formal approaches like timed automata are usually limited in scope to the model under analysis where sharing of resources, leading to complex interactions among components, is difficult to take into consideration.

To overcome the limitations encountered when using either methods, (Thiele et al., 2007) and (Viehl et al., 2006) combined simulation and formal approaches to analyze system performance.

2.2 Formal Verification of UML/SysML Models

Transforming UML/SysML models to a formal specification is a very popular (and old) way to ensure the (formal) verification of these models, as first discussed in (Bruel, 1998), and more recently in (Gabmeyer, 2019) where they have proposed a taxonomy of formal verification techniques for software models with a focus on recent papers addressing simulation and formal verification in the context of SysML.

Delatour et al. were among the first to propose a formal way to verify UML models with Petri Nets (Delatour and Paludetto, 1998). Laleau et al. discussed in (Laleau and Mammar, 2000) the use of annotations in UML so as to generate a B specification. Also, in (Schäfer et al., 2001), Schäfer et al. proposed a translator from UML state machines to UPPAAL. Time was also addressed in (Apvrille et al., 2004) with a transformation from UML to RT-LOTOS. The model-based environment TimeSquare (DeAntoni and Mallet, 2012), supporting the Clock Constraint Specification Language (CCSL), provides facilities for (formal) performance analysis, in particular for the analysis of execution traces. In (Ouchani et al., 2013), Ouchani et al. presented a formal verification framework for checking SysML activity diagrams. The latter are mapped into the input language of the probabilistic model checker PRISM. A calculus dedicated to activity diagrams is proposed and the mapping to PRISM is formalized. The approach is applied to an online shopping system and to real time streaming protocols. (Ando et al., 2013) proposed to formalize SysML state machines with CSP#. (Wang et al., 2019) et al. explored the transformation of SysML models to NuSMV for safety analysis. They applied their approach to a flap control system. Yet, they did not propose any backtracing way. In the present paper, model checking is applied to block and state machine diagrams: block diagrams capture the architecture of the system while state machine diagrams model the inner behavior of blocks.

3 CONTEXT: TTOOL, AVATAR

3.1 TTool

TTool is a free and open source framework targeting the design of embedded systems with UML/SysML. It supports different methodological stages, including system analysis (requirements, fault and attack trees, use cases, sequence diagrams, ...), system hardware / software partitioning with the DIPLODOCUS profile (Apvrille et al., 2006), and embedded software design with the AVATAR SysML environment (Apvrille et al., 2020). SysML-Sec (Apvrille and Li, 2019) is also supported by TTool. SysML-Sec gathers together all methodological stages and adds security features (operators, formal proof) to diagrams. The model-checker presented in this paper concerns AVATAR models. Yet, we intend to extend it to the DIPLODOCUS environment.

3.2 AVATAR

In this paper, by AVATAR, we mean the design part of AVATAR. A design is built upon 3 diagrams:

- A block definition diagram.
- A internal block diagram. To make the structure of the design easier to visualize, block definition and internal block diagrams can be showed in the same view.
- State machine diagrams are used to give a behavior to each block.

3.2.1 Extensions to SyML

AVATAR extends standard SysML as follows:

- Block diagrams support synchronous and asynchronous communications with different flavors (lossy, non lossy, etc.).
- State machines are extended with an *after(min, max)* clause, with timer operators (set, expire, reset), and with a "*x = random(min, max)*" operator.

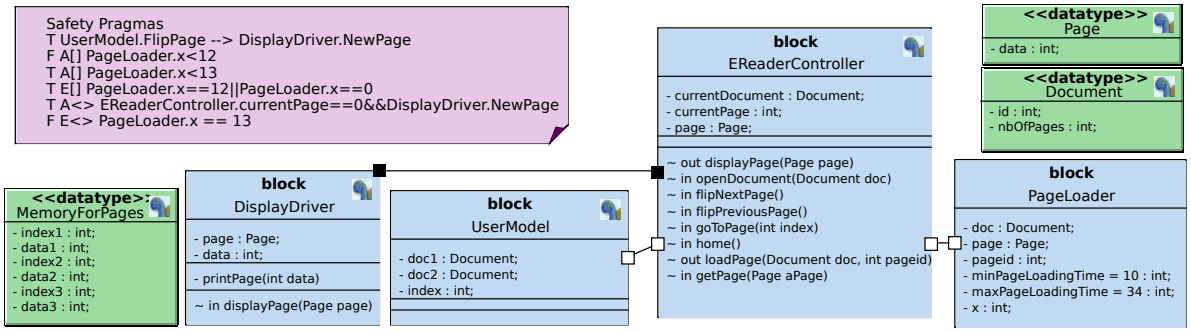

Figure 1: Block diagram with safety CTL properties.

Figure 1 shows a block diagram and the safety CTL properties to be verified on the model. The state machine of block *eReaderController* is shown in Figure 3.

3.2.2 fUML, PSSM and SysML v2

fUML[2], lastest version 1.4, is an OMG standard defining a formal semantics for a subset of UML. Unfortunately, this subset does not consider state machines (but rather activities), so we could not rely on the fUML standard. Also, types such as String or Real are not handled by our model-checker. Once they have been released, the Precise Semantics of UML State Machines standard[3] or SysML v2 may offer new perspectives to the work described in this paper.

4 MODEL-CHECKER

The main contribution of this paper is the development of an open-source model-checker that works almost directly on SysML models without a low-level transformation. Our principal goal is to obtain a flexible and easily upgradable verification engine that can be used efficiently to analyze and verify the correct behavior of a model. Properties are proven with an on-the-fly method while exploring the state space.

The heart of the model-checker is the construction of the reachability graph of a model. A reachability graph is a directed cyclic graph where each node represents a possible state of the model. A node depends on the current pointed states, one for each state machine, the blocks attribute values, and the time progress. Each edge (a,b) represents a direct reachability connection from node a to node b. Any subtree starting from a node a shows the states which are directly reachable from a. Moreover, analyzing one or multiple paths that may connect two nodes, it is possible to find the sequence of actions that, if executed, connect the two. The graph is created by combining the state machines belonging to different blocks in the SysML model. In this paper we refer to a reachability graph node also using the term "r-state". In this context, it has to be interpreted as a state of the reachability graph and not a local state of a state machine.

The reachability graph is constructed using a forward traversal BFS (Breadth-First Search) or DFS (Depth-First Search) strategy. Algorithm 1 shows a simplified pseudo-code of the main loop. The algorithm starts from the given initial states (one for each state machine) creating a r-state S_0. A map *STATES* keeps track of the visited r-states. *PENDING* works as a queue (breadth priority) or a stack (depth priority). Then, every cycle, it extracts a r-state S from the pending queue and it looks for all the available transitions which are currently executable from S. Each of them is executed and leads to a new r-state. If a new r-state is equal to another already visited, it is redundant and only a new link connection is created. If, instead, it is not equal to any other, it is also added to the pending queue in order to be later explored. This procedure cannot be directly used for model-checking but it can be adapted as it will be explained in the next section.

Algorithm 1: Reachability graph algorithm.

```
1:  S_0 = {s_0^0, s_0^1, ..., s_0^N}                ▷ Initial r-state
2:  STATES={S_0}, PENDING={S_0}
3:  while PENDING≠ {} do
4:      Extract a state S from the PENDING queue
5:      T = set of executable transitions from r-state S
6:      for each transition t_i in T do
7:          Execute t_i obtaining a r-state P
8:          if P ∉ STATES then                      ▷ New r-state
9:              Add P to STATES
10:             Append P to PENDING
11:         end if
12:         Add a new edge S → P
13:     end for
14: end while
```

[2]https://www.omg/org/spec/fUML
[3]https://www.omg.org/spec/PSSM

Each r-state in the reachability graph is uniquely represented by a tuple called **r-state context** $(B_1, B_2, \cdots, B_n)$ where each B_i refers to a particular block. The behavior of a block is represented by a state machine. B_i is the state machine context which is also a tuple $(p, clock_{min}, clock_{max}, a_1, a_2, \cdots, a_m)$ where:

- p is the current state pointer
- $clock_{min}$ is the minimum clock value
- $clock_{max}$ is the maximum clock value
- a_j is the value of j-th attribute of the block

From the context definition, a r-state in the reachability graph contains a pointer for each state machine to keep track of the current state in each of them. It saves also the internal minimum and maximum clocks which are used to select the executable transitions in time dependency order. Two r-states are considered equivalent if their contexts are equal.

Several important information must to be considered when fetching and executing new transitions. Checking for acceptable transitions involves checking conditions on attributes (guards), on time (e.g. *after(min, max)* clause) and on signals (signal sending/receiving).

State machines in different blocks have communication channels so that different blocks can exchange information or synchronize their execution. For signal communications, our model-checker transforms asynchronous channels into synchronous ones adding channel queues as a bridge between connected blocks. Transitions with signal communications are executed only when at least a sender and a receiver are active on the same channel. A channel queue acts as either a sender or a receiver depending on the type of request.

Transitions between states can have conditions and actions. Our model-checker solves guards (i.e. boolean conditions on attributes) to activate a transition and actions to modify attributes values. This is achieved by temporarily saving the attributes value of each block in the context of a r-state.

SysML models are usually also timed state machines. In fact, transitions may have a time duration and a defined uncertainty (i.e. *after(min, max)*). Thus, each transition is associated with a time interval (t_{min}, t_{max}) where t_{min} is the shortest execution time and t_{max} is the longest one. The execution may happen any time inside that interval[4]. The model-checker must keep track of the time to select executable transitions. Block clocks are used for this purpose. Time is not saved in absolute value, as it would be quite inefficient, but in relative value with respect to the block time progress. Moreover, saving time in absolute value would not allow to detect easily when some states, at different time periods are actually equivalent. By saving the clock in a relative value, equivalent states encountered at a different time period have the same clock values. Transitions in different blocks must be converted into the same timeline. This is achieved by subtracting the block clock interval from the transitions in the respective block. The executable transitions are extracted doing the intersection with the first smallest time domain $(\min_{\forall t \in \mathcal{T}} t_{c_{min}}, \min_{\forall t \in \mathcal{T}} t_{c_{max}})$. For instance, if we are at clock $(0,0)$ and we have three available transitions $a : (0,3)$, $b : (1,2)$, $c : (3,5)$, the first smallest time domain would be $(0,2)$. Thus, the two possible transitions that can be executed are a with the new interval $(0,2)$ and $b : (1,2)$. Transition c could never be executed since it lies outside the maximum execution time limit of other transitions. The same thing happens for a at time 3.

Finally, the following operations are applied to transitions in order to extract the executable ones:

- for a transition acting on a channel, check for an active sender or receiver
- solve transitions guards
- update transitions clock w.r.t. the local clock
- select the executable transitions in the first smallest clock interval (i.e. the time domain).

The execution of a transition creates a new r-state with an updated context. In the block of the just executed transition, the state pointer is updated with the new one and the internal clock is reset. For the other blocks, the internal clock is incremented by the transition interval $(clock_{min} + t_{c_{min}}, clock_{max} + t'_{c_{max}})$ [5].

The reachability problem is covered on-the-fly with the construction of the reachability graph. In fact, all the reachable states and properties are encountered during its creation. If at the graph creation, or at a given depth bound, a state is not traversed, or a property is not satisfied, their reachability is false.

5 PROPERTIES

The reachability check, presented at the end of the previous section, allows only a basic verification of a model. Several other checks like liveness, safety properties, deadlocks, livelocks, etc. are necessary to

[4] A transition may occur also after max clock value has elapsed in case the action of the transition is not possible, e.g. waiting for a signal that is not yet available.

[5] $t'_{c_{max}}$ considers the maximum time limited at $\min_{\forall t \in \mathcal{T}} t_{c_{max}}$

have a clear idea if the behavior of the SysML model is as expected. The reachability algorithm shown in the previous section can be used as a basis for generic formal verification on SysML models. Our model-checker framework can also prove CTL formulae and the general checks as: liveness and safety properties, reachability, and deadlock freedom.

Most of these properties are way more challenging to prove than reachability. Usually, the standard approach consists in detecting execution cycles that present specific conditions. Loops are usually found using the strong connected components (SCC) algorithm. In our case, loops can be found while constructing the reachability graph. Every time a new r-state P in the reachability graph is equivalent to another S that has been already found, a depth first search from S to $father(P)$[6] is used to find if a path that connects them exists. If it exists, a cycle is detected. This is true by construction of the reachability graph since state P is equivalent to S and reachable from it. Some properties may require the model-checker to execute to completion or until a deadlock is detected.

Deadlocks can be easily detected while building the reachability graph. Every time a reachable r-state has not enabled transitions, that r-state is a deadlock.

The supported CTL formulae are defined accordingly to the UPPAAL standard syntax:

- **A[] p:** property p is always true for all r-states of each path (other notation **AG p**)
- **A<> p:** for all the paths, property p will eventually be true for some r-states (other common notation **AF p**)
- **E[] p:** there exists at least one path for which property p is always true for each r-state in that path (other common notation **EG p**)
- **E<> p:** there exists a path in which property p will eventually be true for some r-states of that path (other common notation **EF p**)
- **p → q:** for all the paths and r-states on that paths, if p becomes true then for all the paths and some r-states on these paths, q will eventually become true (other common notation **AG(p ⇒ AFq)** called *leadsTo*)

with p and q boolean properties. A property can test states of state machines, variables or a mix of them using standard arithmetic and boolean operators.

Figure 2 shows how CTL formulae are expressed in TTool and their result after the verification has run. Before a property, an expected value (T for True, or F for false) may be expressed in order to show the verification result accordingly. The model corresponds to the one shown in Figure 1. A green tick is displayed when the property is satisfied, while a red cross is displayed in the opposite case. If the answer is unknown, in case of a timeout in the execution or if the verification is terminated by the user, a question mark is displayed.

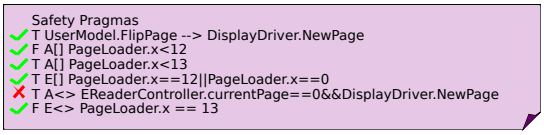

Figure 2: CTL properties after verification.

Figure 3 shows how reachability and liveness properties are backtracked and displayed in the state machines for each state element, using green or red "R" and "L" attached to states.

Here we show some examples of possible CTL properties on SysML models and their meaning:

- *A<> Controller.Received:* tests if the *Controller* block on all the paths will eventually reach a "Received" state, e.g. once an information from a sensor has been received.
- *A[] SpeedSensor.sampledSpeed <= SpeedController.speedLimit + SpeedController.tolerance:* checks that the measured speed of a vehicle never exceeds the defined limit
- *WatchDog.Timer == 0 → System.Restart:* checks that every time a watchdog timer reaches zero, the system goes to the *Restart* state

Given a test on property p, we define a *true (false) loop* as a loop with condition on property p satisfied (unsatisfied). In the same manner, we define also a *true (false) deadlock* as a deadlock where the condition on property p is satisfied (unsatisfied). These properties are now described in the model-checking context that has been presented using a combination of reachability and cycles detection:

- **A[] p:** if during the reachability graph creation a new r-state has p false, then the property is false (reachability of $\neg p$)
- **A<> p:** if p is true for a r-state, stop the search on that path as it is already valid. If a false loop or a false deadlock are found, the property is false. If no false loops or false deadlocks are found, the property is true.
- **E[] p:** if p is false for a r-state, stop the search on that path as it is invalid. If a true loop or a true deadlock are encountered, the property is true. If

[6] $father(P)$ is the father from which P has just been discovered

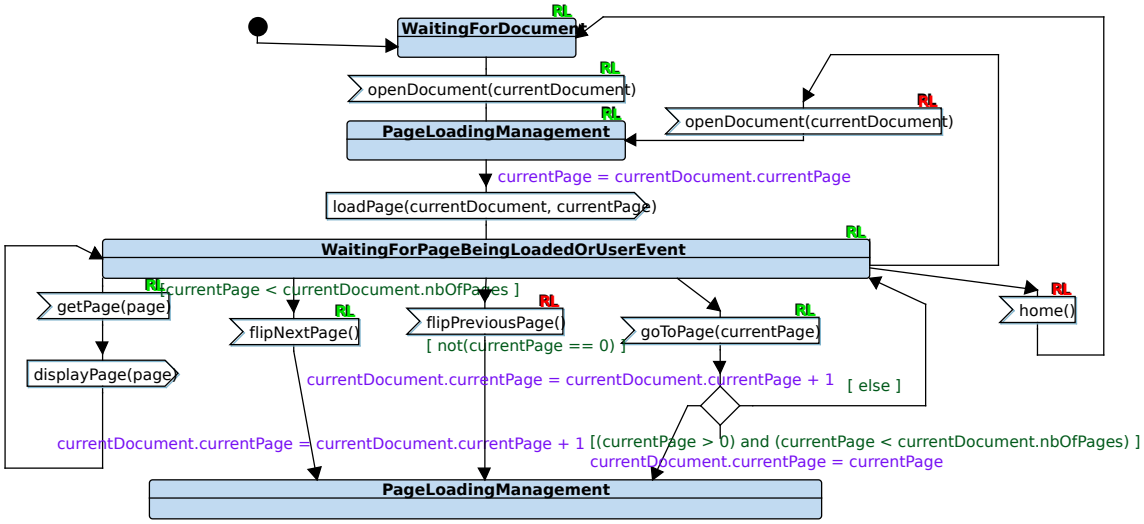

Figure 3: State machine with reachability and liveness information for each state element.

no true loops or false deadlocks are found, the property is false.

- **E<> p:** if during the reachability graph creation, p is true in a new r-state, then the property is true.
- **p → q:** every time p is reached, save the r-state context. Then, for each saved r-state, start a liveness check $A<>\ q$ with that r-state as a starting state[7]. If the liveness of q is true for all the saved r-states or p is always false, the property is true, else it is false.

Thanks to the integration of a cycle detection algorithm and an expression solver to test the properties while building the reachability graph, our model-checker can effectively prove the correctness of a model allowing a breadth set of tests that are generally used in this field.

When proving properties, it is a good practice to provide a report trace which is reported as a list of r-states and actions, extracted as a sub-path from the reachability graph. Our model-checker can generate such a trace, but techniques to do so are not explained in this paper.

6 IMPLEMENTATION AND EVALUATION

The main issue model-checking faces is probably the combinatorial explosion. Optimizing the computation power and memory size necessary for a given SysML model is therefore of prime importance. In this section we present some of the techniques that we have used to limit the exploration space and to enhance performance.

6.1 State and Transition Merging

Since even reasonably small models may have a huge number of r-states, it is important to adopt state merging techniques both as a pre-computation process and on-the-fly. The first step is to merge states which are linked by empty transitions[8]. A second step, on-the-fly, merges non-empty transitions with no multiple choice and time dependencies. Each transition is associated with an unique ID. The merge is executed in ascending order by ID. This operation supports a notable reduction in the number of r-states without changing the verification result. There is an exception. This optimization approximates the concurrency since it merges by ID some possible concurrent r-states. On nested properties, like the *leadsTo*, the result may change since a second pass iteration could check for a concurrency on a merged r-state. The merging order cannot be reconstructed backwards since old transitions are not saved (to save memory). However, the verification does not normally depend on these cases since the nested properties are usually not concurrency related. In fact, the idea behind nested properties is to check something in the future. Anyway, with an option this optimization can be disabled. On the *leadsTo* property, a further optimization is also active. When a r-state at the first

[7]This process of saving the context when a first property is satisfied and iterate to the next ones, is valid and can be adopted for generic nested properties.

[8]With empty transitions we refer to state machine transitions with no actions, no time dependency, no multiple valid choices transitions.

iteration has a true property, available non-concurrent transitions are continuously merged if they keep the property satisfied. In this way, the number of nodes for the second pass iteration is notably reduced.

6.2 Impact of the Exploration Approach

The reachability graph algorithm could be used in a multi-thread BFS or DFS mode. For some properties, such as liveness, we noticed that a DFS exploration is generally more effective than a BFS one as the main proof of unsatisfiability is given by loops or deadlocks. A search in depth increases the odds to find quickly those features.

6.3 State Encoding

The context of a r-state is necessary to save the basic local information. It is also needed to fetch the transitions starting from the current state in order to search for new reachable nodes. The context defines uniquely a r-state and it is used to generate an hash value. All the r-states are inserted in an hash table using the context hash as a key. To find if a new r-state has already been discovered, its context hash is generated. Then, the hash table is accessed checking if it contains the generated hash. To save memory, once a node generates all its subsequent next reachable r-states, its context can be freed.

The r-state context saves also the values for each attribute of each block in the model. It is necessary so that, when a guard is evaluated, the attribute current value is read directly from the context. Those values are usually updated by the model during its execution using signals or assignment actions. If this update never occurs, the attribute is a constant. In precomputation, constants are removed from the context and they are directly substituted as immediate values into the expressions they are involved. Moreover, boolean attributes are grouped and represented on 1-bit while integers can be represented on 8, 16 or 32 bits.

6.4 Handling of Properties

Expressions and properties are constructed as a syntax tree. Each node is an operator while each leaf is an attribute or an immediate value. By vising the tree from the root, the result of the expression can be easily and efficiently retrieved. Leaves have a direct pointer to their corresponding location in the r-state context to have fast access to their values. Leaves containing attributes are saved in a hash table so that leaves can be shared among different expressions or properties.

6.5 Overall Performance

Before developing this framework, TTool relied on UPPAAL to verify its models. Thus, the performance of the new internal model-checker is evaluated by comparing the two. Table 1 shows some results comparing our model-checker to UPPAAL. Time values for UPPAAL don't include the translation time of a model which takes around *225 ms*. The properties are tested on various SysML models, most of them available in the public TTool repository[9]. The *ebook*, *telecom_final*, and *CoffeeMachine_async* (CM_async) models contain asynchronous channels while *AirbusDoor_V2* (AD_V2), *UAV*, and *PressureController* (PC) contain only synchronous channels. *R*, *L*, and *D* refer in order to reachability, liveness, and deadlock freedom.

Our model-checker allows to build a reachability graph (1386541 r-states in *ebook* model) that can be opened for a visual representation that captures the global model behavior. Moreover, the reachability graph can be minimized to represent only the states chosen by the user. This feature is not available in UPPAAL.

Generally, our model-checker is faster than UPPAAL over synchronous channels and for short time verification over asynchronous channels. UPPAAL performs generally better in expensive and hard verification, like the *leadsTo* property, and over asynchronous channels. As a future work, we plan to improve the performance using additional optimization techniques, especially on asynchronous communications, and moving our implementation from Java to C++ since memory management is one of the main bottlenecks. There is still margin for improvement. Nevertheless, our implementation obtains good results while being able to link the verification to the model representations, traces and simulation.

7 CONCLUSION

The paper introduces a model-checker directly working from SysML model, thus without a prior transformation to a formal specification. Our model-checker can handle CTL-like properties. Our evaluation shows that it compares to a state-of-the-art model-checker.

This model-checker is already available in TTool. All properties described in this paper can be directly captured in the AVATAR block diagrams. Once the

[9]Some models have been modified to increase the combinatorial space of the verification.

Table 1: Model-checker timing results (low values are better).

Model	*Property*	Time MC (ms)	Time UPPAAL (ms)
ebook	*RG*	3832	/
ebook	*D*	14	755
ebook	*LeadsTo*	5659	2235
ebook	*A[]x<12 (BFS)*	10	287
ebook	*A[]x<13 (BFS)*	3478	1303
ebook	*E[] (BFS)*	14	282
ebook	*A<>(DFS)*	29	279
ebook	*E<>(BFS)*	3604	1292
ebook	*R, L, D, CTL*	17529	12068
telecom_final	*R, L*	1242	2927
AD_V2	*R, L, D*	82	433
CM_async	*R, L, D*	126	393
CM_async	*CTL*	2776	1328
UAV	*R, L, D, CTL*	688	2505
PC	*R, L, D, CTL*	198	748

model-checker has verified them, a green check or a red cross indicates for each property whether they are satisfied or not. Additionally, a counter example can be generated in some situations. We now intend to improve this trace generation facility. We also intend to extend this model-checker to other profiles supported by TTool. Furthermore, we plan to tackle new optimization techniques to make our model-checker even more competitive among the available ones. Linking this model-checker to other UML/SysML framework is also part of our future work.

ACKNOWLEDGEMENTS

The AQUAS project is funded by ECSEL JU under grant agreement No 737475.

REFERENCES

Ando, T., Yatsu, H., Kong, W., Hisazumi, K., and Fukuda, A. (2013). Formalization and model checking of sysml state machine diagrams by csp#. In Murgante, B., Misra, S., Carlini, M., Torre, C. M., Nguyen, H.-Q., Taniar, D., Apduhan, B. O., and Gervasi, O., editors, *Computational Science and Its Applications – ICCSA 2013*, pages 114–127, Berlin, Heidelberg. Springer Berlin Heidelberg.

Apvrille, L., Courtiat, J. ., 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. (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.

Apvrille, L. and Li, L. W. (2019). Harmonizing safety, security and performance requirements in embedded systems. In *Design, Automation and Test in Europe (DATE'2019)*, Florence, Italy.

Apvrille, L., Muhammad, W., Ameur-Boulifa, R., Coudert, S., and Pacalet, R. (2006). A uml-based environment for system design space exploration. In *2006 13th IEEE International Conference on Electronics, Circuits and Systems*, pages 1272–1275.

Bruel, C. (1998). Integrating formal and informal specification techniques. why? how? In *Industrial-Strength Formal Specification Techniques, Workshop on*, page 50, Los Alamitos, CA, USA. IEEE Computer Society.

DeAntoni, J. and Mallet, F. (2012). Timesquare: Treat your models with logical time. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 34–41. Springer.

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 Demeyer, S. and Bosch, J., editors, *Object-Oriented Technology: ECOOP'98 Workshop Reader*, pages 511–514, Berlin, Heidelberg. Springer Berlin Heidelberg.

Gabmeyer, Sebastian, K. P. S. M. G. M. K. G. (2019). A feature-based classification of formal verification techniques for software models. *Software & Systems Modeling*.

Kangas, T., Kukkala, P., Orsila, H., Salminen, E., Hännikäinen, M., Hämäläinen, T. D., Riihimäki, J., and Kuusilinna, K. (2006). Uml-based multiprocessor soc design framework. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(2):281–320.

Laleau, R. and Mammar, A. (2000). An overview of a method and its support tool for generating b specifications from uml notations. In *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, pages 269–272.

Ouchani, S., Ait Mohamed, O., and Debbabi, M. (2013). A probabilistic verification framework for sysml activity diagrams. volume 246, pages 165–170.

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

Stemmer, R., Schlender, H., Fakih, M., Grüttner, K., and Nebel, W. (2019). Probabilistic state-based rt-analysis of sdfgs on mpsocs with shared memory communication. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1715–1720.

Thiele, L., Wandeler, E., and Haid, W. (2007). Performance analysis of distributed embedded systems. In *International Conference On Embedded Software: Proceedings of the 7 th ACM & IEEE international conference on Embedded software*, volume 30, pages 10–10. Citeseer.

Viehl, A., Schönwald, T., Bringmann, O., and Rosenstiel, W. (2006). Formal performance analysis and simulation of uml/sysml models for esl design. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 242–247. European Design and Automation Association.

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.