Static analysis techniques to verify mutual exclusion situations within SysML models

Ludovic Apvrille[1] and Pierre de Saqui-Sannes[2]

[1] Institut Mines-Telecom, Telecom ParisTech, LTCI CNRS,
Campus SophiaTech, 450 route des Chappes, 06410 Biot, France
ludovic.apvrille@telecom-paristech.fr
[2] Université de Toulouse, ISAE,
10 av. Edouard Belin, B.P. 54032, 31055 Toulouse Cedex 4, France
pdss@isae.fr

Abstract. AVATAR is a real-time extension of SysML supported by the TTool open-source toolkit. So far, formal verification of AVATAR models has relied on reachability techniques that face a state explosion problem. The paper explores a new avenue: applying structural analysis to AVATAR model, so as to identify mutual exclusion situations. In practice, TTool translates a subset of an AVATAR model into a Petri net and solves an equation system built upon the incidence matrix of the net. TTool implements a push-button approach and displays verification results at the AVATAR model level. The approach is not restricted to AVATAR and my be adapted to other UML profiles.

Keywords: Modeling, Model verification, Structural analysis, SysML, Petri Nets, Invariants, Mutual exclusion.

1 Introduction

UML and SysML tools that implement formal verification of real-time systems models commonly reuse reachability analysis techniques and therefore face the state explosion problem [1]. Examples include Artisan Studio [2], SysML Companion [3], OMEGA SysML [4], TOPCASED [5], and TTool [6]. The latest release of TTool, which is addressed in the paper, contrasts with the aforementioned tools by the static analysis it implements for AVATAR, a real-time systems modeling language based on SysML. The paper indeed investigates static analysis of AVATAR models and even focuses discussion on proving mutual exclusion, for instance of shared resources.

The paper reuses the "invariant search" technique originally developed for Petri nets. Unlike papers that propose to write invariants and to check the model against them, the paper generates invariants from the model. In practice, TTool translates an AVATAR model into a Petri net and solves an equation system built upon the incidence matrix of the net. Then, TTool displays verification results at the AVATAR model level. Also, usual questions regarding states in

mutual exclusion are asked at the AVATAR model level, not at the Petri net level, and mutual exclusions results are directly given at model level.

The paper is organized as follows. Section 2 introduces the AVATAR modeling language and the TTool tool. Section 3 reminds the principles of invariant search from a Petri net. Section 4 details how mutual exclusion situations can be detected on AVATAR models. Section 5 presents a case study. Section 6 surveys related work. Section 7 concludes the paper.

2 Avatar: A SysML Environment

2.1 AVATAR diagrams and method

The AVATAR language reuses all SysML diagrams, excepted the package diagram. In the early stages of the method associated with AVATAR, a requirement diagram organizes captured requirements in a tree-like structure that shows their attributes, their interrelations and their connections with other elements of the model.

A text diagram lists the modeling assumptions that apply to the system's environment and to the system itself. Incremental modeling starts with strong assumptions that are progressively lowered.

As far as the AVATAR model is built up following an incremental approach, part of the limitations associated with the original modeling assumptions will progressively be removed from the list.

Analysis is use-case driven. A use-case diagram identifies the main functions or services the system offers in relation with external actors. Scenarios (sequence diagrams) and flowcharts (activity diagrams) document the use-cases. Sequence diagrams handle synchronous/asynchronous communications, absolute dates and time intervals. Activity diagrams depict basic actions, tests and loops.

An AVATAR Design captures both architectural and behavioural matters [7–9]. First, a block instance diagram depicts the architecture of the system as a set of communicating block instances defined by their attributes, methods, and input/output ports. Each block instance has a behaviour defined in terms of a finite state machine that supports most SysML state machine elements: input and output signals, variables, timers, time intervals on transitions, enabling conditions and composite states. Examples of non supported elements are history in composite states, and fork/join pseudo states.

The block instance diagram and its associated state machine diagrams have a formal semantics expressed by translation to timed automata for safety proofs, and to pi-calculus processes for security proofs. Design diagrams may be simulated and formally verified from TTool.

2.2 TTool

The AVATAR language is wholly supported by the open software tool *TTool* [6] developed for Linux, Windows and MacOS. The default installation of TTool

comes with a diagram editor and a simulator. TTool implements gateways towards three tools that are developed by other laboratories : UPPAAL for the formal verification of the logical and temporal properties [10], ProVerif for the formal verification of security properties [8, 11], and SocLib for the virtual prototyping of the software and hardware of real-time systems [9]. The simulator enables step-by-step and random transition firing. All results are given at the AVATAR level: simulation traces in the form of sequence diagrams and on-the-model identification of the explored transitions. Similarly, the strong advantage of TTool as far as formal verification is concerned is the user-friendliness of the interface to UPPAAL. The user of TTool may indeed check for deadlock freedom, as well as for the reachability and liveness of actions and states, by mere identification of the actions and states on the AVATAR model itself, with no need for an inspection of the UPPAAL "code". Further, there is not need for writing logic formulae.

User friendliness - that is, not needing to know about underlying formal models and proof techniques - has also been a key concern in implementing the new verification approach presented in the paper.

3 Petri Nets and invariants

This section is a short reminder about Petri nets and P-invariants, that is sufficient to understand our contribution. More information on Petri nets and invariants may be found for example in [12–14].

3.1 Verification techniques for Petri nets

A Petri net is a bi-partite graph made up of places and transitions. The transition firing policy and the way tokens move from place to place enable the representation of the operation semantics of systems modeled by Petri nets. More formally, Petri nets have been defined as follows [12]:

Definition 1 ***Petri net***
A Petri net is a 5-uple, $PN = (P, T, F, W, M_0)$ *where:*

- $P = \{p_1, p_2, \ldots, p_m\}$ *is a finite set of places,*
- $T = \{t_1, t_2, \ldots, t_n\}$ *is a finite set of transitions,*
- $F \subseteq (P \times T) \bigcup (T \times P)$ *is a set of arcs (called "flow relation" in [12])*
- $W : F \mapsto \{1, 2, 3, \ldots\}$ *is a weight function,*
- $M_o : P \mapsto \{0, 1, 2, \ldots\}$ *is the initial marking.*

By definition, $P \bigcap T = \emptyset$ *and* $P \bigcup T \neq \emptyset$.

In this paper, Petri nets are used to verify AVATAR models that may contain data and time. Nevertheless, the paper restricts discussion to basic Petri nets, as defined above: the limitations of our approach are discussed in section 4. Similarly, the purpose of the paper is not to survey all the verification techniques available for Petri Nets (see for example [12]), and the paper therefore restricts discussion to structural analysis based on place invariants (also called "*P-invariants*").

3.2 P-invariants

P-invariants are defined from the Petri net incidence matrix. An incidence matrix represents the various transitions of each place. Rows are used for places p_i, and columns for transitions t_j. For example, the value v_{ij} at (p_i, t_j) means that v_{ij} tokens are added (or removed if the value is negative) from p_i whenever transition t_i is fired. More formally, an incidence matrix can be defined as follows [12]:

Definition 2 ***Incidence matrix***

The incidence matrix of a Petri net PN with n transitions and m places is a $n \times m$ matrix $A = [a_{ij}]$ of integers with $a_{ij} = a_{ij}^{+} - a_{ij}^{-}$ where:

- *$a_{ij}^{+} = w(i,j)$ represents the weight of the arc from transition i to the output place j,*
- *$a_{ij}^{-} = w(j,i)$ represents the weight of the arc from the input place j to transition i.*

Definition 3 ***P-invariants***

P-invariants of a Petri net PN are usually defined as $\mathcal{W}.A = 0$ with $\mathcal{W}$ being a set of m weighted places of PN and A being the incidence matrix of PN.

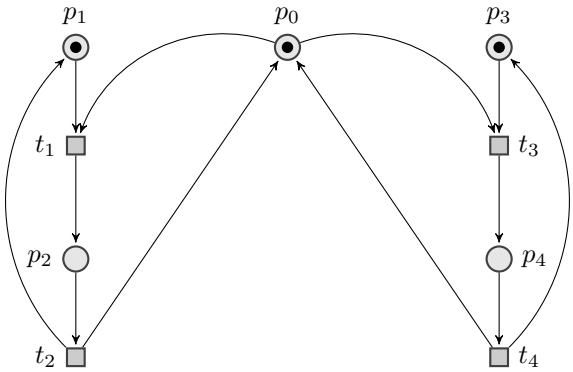
Finally, a P-invariant models a set of places in which the total number of tokens is constant in all reachable markings.

3.3 Algorithms for P-invariants

Again, P-invariants are defined as $\mathcal{W}.A = 0$. This set of equations can be solved with the Farkas algorithm [15]. The latter allows to compute a set of minimal P-invariants. The complexity of this algorithm is exponential, but heuristics have been proposed in order to reduce this complexity [16].

3.4 Example

P-invariants can be used to prove mutual exclusion situations. Indeed, the mutual exclusion between two subnets s_1 and s_2 of a Petri net PN can be proved by showing that at most 1 token is present in the marking of places of s_1 and s_2. Let's illustrate mutual exclusion with the following Petri net:



The transpose incidence matrix A^t of this Petri net is as follows:

$$A^t = \begin{array}{c} \\ p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \end{array} \begin{array}{c} \begin{array}{cccc} t_1 & t_2 & t_3 & t_4 \end{array} \\ \begin{pmatrix} -1 & 1 & -1 & 1 \\ -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix} \end{array}$$

To resolve $\mathcal{W}.A = 0$ (see Definition 3), the matrix is made triangular, as for solving a linear system: lines can be exchanged, multiplied by a given integer value, or one line can be added to another one. Applying this to the A matrix gives:

$$A^t_{triangular} = \begin{array}{l} \\ p_0 \\ p_4 \\ p_0 + p_2 + p_4 \\ p_3 + p_4 \\ p_1 + p_2 \end{array} \begin{array}{c} \begin{array}{cccc} t_1 & t_2 & t_3 & t_4 \end{array} \\ \begin{pmatrix} -1 & 1 & -1 & 1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Finally, the P-invariants are p_1+p_2, p_3+p_4, $p_0+p_2+p_4$. Each of them represents a mutual exclusion situation. For example $p_1 + p_2$ models the fact that either there is a token in p_1 or in p_2. Similarly, $p_0 + p_2 + p_4$ proves a mutual exclusion between, in particular, places $p2$ and p_4.

4 Our approach

4.1 Overview

The main contribution of this paper is the computations of "SysML model invariants". The latter are a list of SysML state machine elements of a model that are all executed in mutual exclusion. We propose to compute these invariants

using P-invariants as defined in previous section. Fig. 1 depicts the approach implemented in TTool. As stated before, there are already two possibilities for proving properties from AVATAR design models. The first possibility is to generate a pi-calculus specification, and then using ProVerif for verifying security properties. The second possibility is to generate timed automata that are taken as input by UPPAAL to verify safety properties. The contribution based on P-invariants is displayed at the right of the figure:

1. AVATAR design models to be studied are first translated to a Petri net. The translation to a Petri Net could be applied to all Domain Specific Languages defined to model sets of entities communicating with synchronous or asynchronous channels, and whose behaviour is described with state machines.
2. The Petri net is in turn translated to an incidence matrix.
3. The Farkas algorithm [15] is used to compute the set of minimal invariants.
4. Invariants are filtered so as to keep only the relevant ones. Relevant invariants are the ones that are not concerning only one block.
5. Invariants are finally back-traced to the SysML model under the form of a SysML model invariant.

4.2 SysML model invariant

We have defined SysML model invariants for design diagrams only. A SysML model invariant is a list of model elements of the state machines: it may refer to a message sending, a message reception or a state. Each element is translated into a set of places and transitions, and so, for back-tracing P-invariants to the SysML model, it is necessary to keep track of the correspondence between operators and places/transitions. The translation of an AVATAR design to a Petri net is now further detailed.

4.3 AVATAR design model translation to Petri nets

Again, an AVATAR model is made up, on the one hand, of the definition the architecture in terms of blocks and communication between blocks (synchronous, asynchronous), and on the other hand, of one state machine per block. AVATAR security-specific operators [8] in both architecture and behavioral diagrams are totally ignored in this translation process.

Translation of state machines. AVATAR states machines are built upon the following operators: states, transitions (guards, actions on variables, time constraints), timers (set, wait for expiration, reset), non-deterministic choices, and communication operators (message sending, message receiving). Let's investigate the basics of the translation for all of them.

– **States**. Each state is translated as one place.

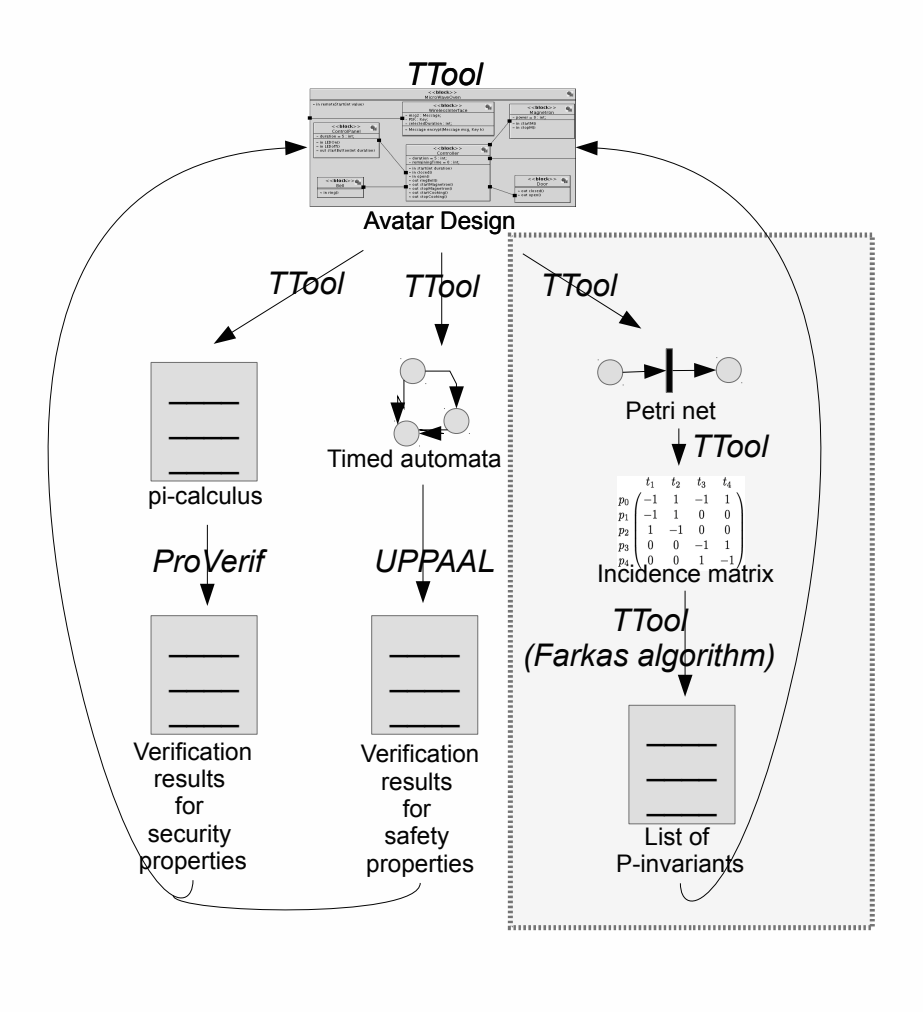


Fig. 1. P-invariants from SysML Models

- **Transitions**. Transitions are constrained with time delays and Boolean guards. Both constraints are ignored by our translation process since the type of Petri nets we rely on does not support time nor variables. Similarly, actions on variables inside transitions are ignored. Finally, transitions are translated into a Petri net transition.
- **Timers**. Since time constraints cannot be represented in the Petri nets we rely on, timers are ignored.
- **Non-deterministic choices**. Each of them is translated to exactly one place.
- **Communication operators**. Their translation is of utmost importance since they model the synchronization between tasks. Their translation is the most complex one and is further explained in next subsection.

Finally, the translation process ignores time constraints and variables, which means the translation to Petri nets only takes into account block to block communications and state to state transitions as described by the state machines of the blocks. We now explain how AVATAR synchronous and asynchronous communications are translated.

Translation of synchronous communications between blocks. Synchronous channels are declared at SysML block diagram levels. A synchronous channel can apply to several blocks. The translation of a given synchronous communication channel ch is a two-step process:

1. For each communication operator c (sending in ch, receiving from ch) of all state machines, two places are generated: p_{cb} models the waiting for the synchronization, ("b" means before) and p_{ca} ("a" means after) models the situation after synchronization occurrence. For example, Fig. 2 represents three communications involving the same channel msg. Thus, for each operator of a design involving msg (one sending operator and two receiving operators) two corresponding places are created.
2. For possible synchronization of ch, i.e., for each possible couple c_{sr} (sending, receiving) of ch, we do the following: We create a new transition t_{csr} with two incoming edges from the p_{cb} places of sending and receiving, and two outgoing edges from t_{csr} to the after-synchronization places p_{ca} of the sender and the receiver. Figure 3 depicts how the places generated in the first step (see Fig. 2) are linked together through two transitions modeling the two possible synchronizations.

Translation of asynchronous communications between blocks. AVATAR asynchronous communication is based on a finite FIFO, with two different writing policies:

1. Writers are blocked when the FIFO is full.

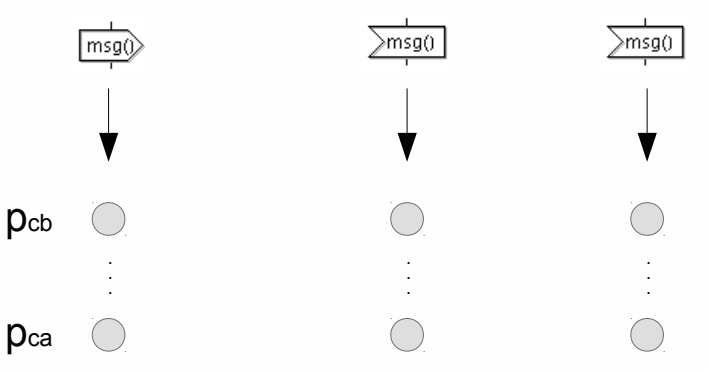


Fig. 2. Translating synchronous communications: step 1

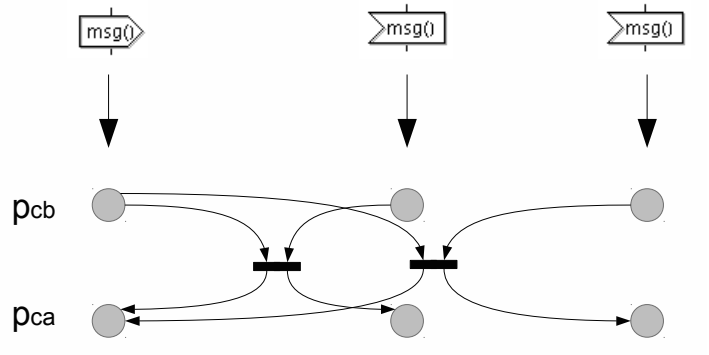


Fig. 3. Translating synchronous communications: step 2

2. Writers are not blocked when the FIFO is full, that is, an element of the FIFO is considered to be removed whenever a write operation in a full FIFO is performed.

The two policies are translated as follows. For the first one (writers are blocked), the main idea is to have a place containing n tokens, with n being the maximum capacity of the FIFO (see Fig. 4). Then, one token is moved to another place when a write operation is performed, and one token is moved back to its initial location whenever a read operation occurs. Thus, write and read operations are blocked when the FIFO is full or empty, respectively.
The second FIFO policy is translated quite similarly to the previous one (see Fig. 5): the main difference relies in an additional transition that is used whenever the FIFO is full so as to "unblock" the writer.

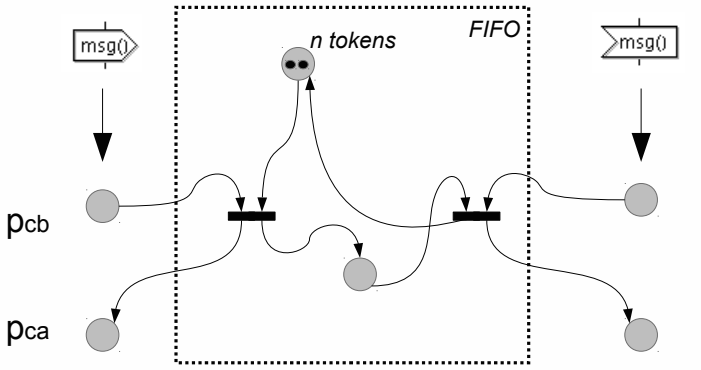


Fig. 4. Translating asynchronous blocking communications

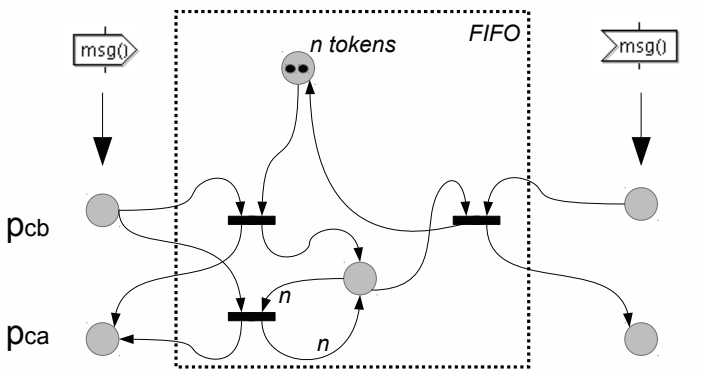


Fig. 5. Translating asynchronous non-blocking communications

5 Case study

The objective of this case study is twofold. First, it intends to demonstrate the effectiveness of P-invariants to identify mutual exclusion situations in SysML diagrams. Second, it illustrates how P-invariants have been integrated in TTool to facilitate their usage.

5.1 Description of the system: a microwave oven

The microwave can be started using a button, or a remote control. Whenever the door is opened, the magnetron must be turned off (safety constraint). Also, the remote control must be secured, that is, a remote control must be attached to only one specific oven (authenticity constraint), and messages sent from the remote control must not be disclosed (confidentiality constraint). Finally, the oven model shall satisfy both safety and security constraints. Yet, security matters are out-of-scope of this paper, but have been explained in [8].

5.2 Design

The design is made upon several types of blocks and elements (see Fig. 6):

- A main block named "RemotelyControlledMicrowave" contains all other blocks modeling the system: the remote control, and the microwave oven itself composed of a wireless interface, a magnetron, a door, a bell and a control panel. Each block declares attributes, methods and signals.
- The declaration of two data types (Key, Message) at the lower left part of the diagram.
- The declaration of security-related constraints and properties in the note located at the top of the diagram.
- The declaration of communication channels between blocks. Ports filled in black represent synchronous communication whereas ports filled in white represent asynchronous communications. Signals and ports can be used by the block declaring them, and by the blocks it contains. For example, all blocks may use the asynchronous channels connecting "RemotelyControlled-Microwave" to itself.
- The declaration of an observer whose purpose is explained hereafter.

5.3 Formal verification of safety properties

One safety property is at stake in this system: "the magnetron must be off whenever the door is opened".
A first way to prove this property in TTool is the usual way to do: adding an observer in the model (see Fig. 6 "ObserverProp1" block). The latter contains an "error" state whose reachability can be studied directly from TTool using UPPAAL. Of course, this technique requires to "execute" the model, and explore

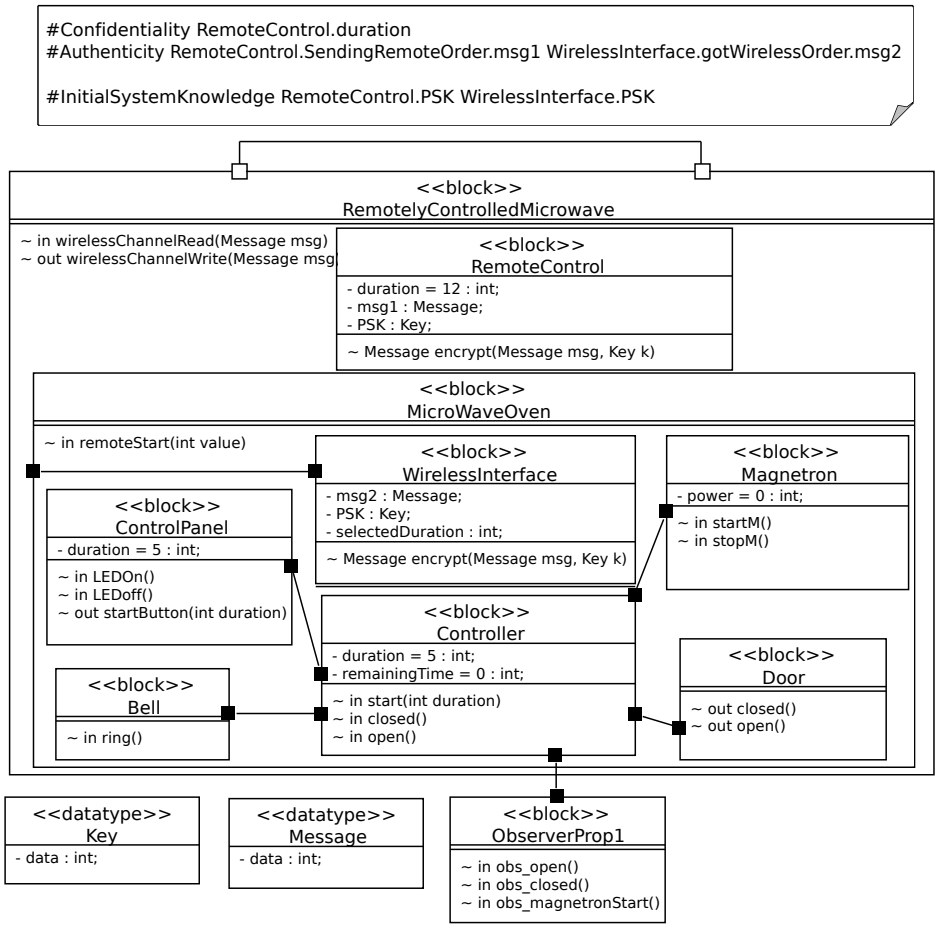


Fig. 6. AVATAR block diagram of the microwave system

all its branches to be certain that none of them contains that "error" action.
A second way to do relies on P-invariants. TTool computes model invariants. Once computed, invariants are listed on the left part of the main window of TTool (see Fig. 7). The user of TTool may select one invariant. Then, all graphical elements of that invariant are underlined with an "inv" annotation (see the circle in Fig. 7), making it very easy to parse all elements of each invariant. All elements of the same invariant are mutually exclusive.

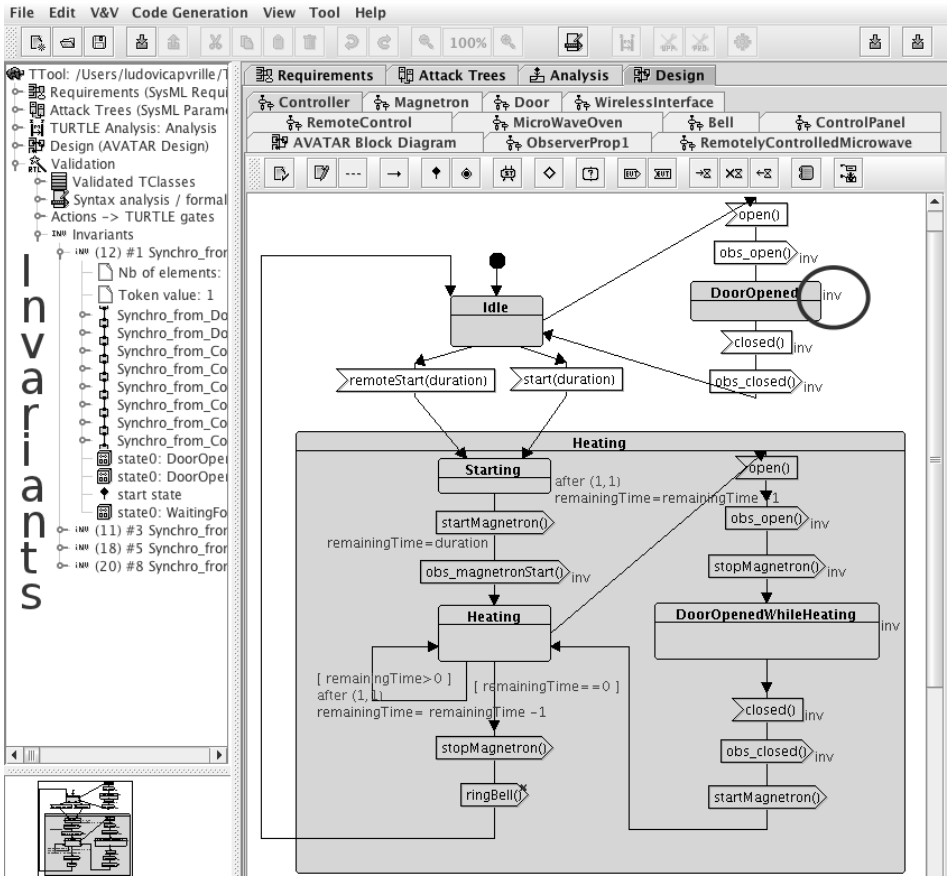


Fig. 7. Invariants as displayed in TTool

From invariants, TTool offers another nice graphical way to visualize mutual exclusive situations: putting the mouse on a given state displays the list of all states that are mutually exclusive with the selected one. For example, the "DoorIsOpened" state in the Door block (see Fig. 8), no state in mutual exclusion could be identified. Indeed, in a first model, it is possible to have the

door opened while the magnetron is on. We have therefore modified the model as follows: we added a lock on the door, i.e., when the user wishes to open the door, the microwave first turns off the magnetron before releasing a lock on the door. With this model, the "DoorIsOpened" state is mutually exclusive with the state "Running" of Magnetron, see Fig. 9, which proves that the magnetron is off whenever the door is opened.

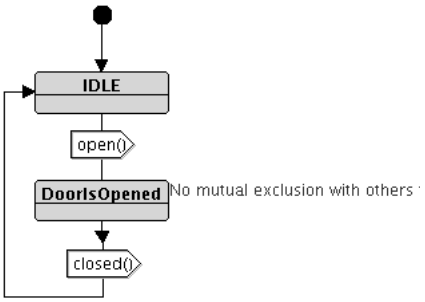


Fig. 8. Mutual exclusions of the state "DoorIsOpened" of the Door block (first version)

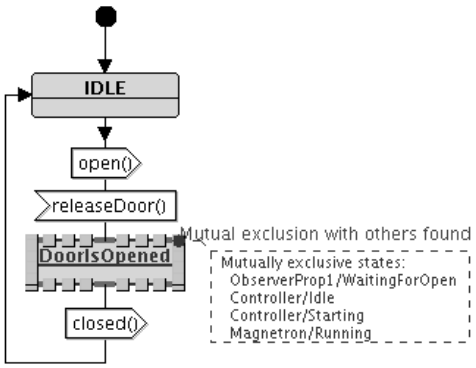


Fig. 9. Mutual exclusions of the state "DoorIsOpened" of the Door block (second version)

5.4 Discussion

Identifying mutual exclusion situations has now become a piece of cake in TTool thanks to the invariants. Heuristics we have defined and implemented - that are not detailed in this paper - make it possible to compute invariants in a few seconds on the most complex models we have made with AVATAR. This is in particular the case for an automotive application published in [9], and developed in the scope of the EVITA European project [17].

Yet, one must be aware of the main current limitation: not all model elements are taken into account to compute invariants, as explained in section 4. In particular, time constraints and variables are not considered in the translation process. In other words, invariants are computed for a reduced model that contains more traces than the original model. And so, two states identified as mutually exclusive by invariants are really mutually exclusive ... but two states not identified as mutually exclusive might still be mutually exclusive. To address that issue, TTool puts a warning textnote ("Mutual exclusion could not be studied") next to the states for which the mutual exclusion could not be proved with invariants.

Last, but not least, the AVATAR-to-Petri Net transformation has been programmed in an ad-hoc manner. It could be interesting to describe this transformation with a model transformation language.

6 Related work

Real-time dialects of SysML. The System Modeling Language [18] is a UML profile [19] that may be tailored in turn to fit in with an application domain. Given the concept of "profile of profile" has not been defined by the UML standard, a tailored version of SysML may be termed as a "SysML dialect". Examples of such dialects have been defined for real-time systems. For example, [20] suggests to simplify SysML and to formalize communication ports.AVATAR, which is the subject of the paper, ignores continuous flows and merges the block definition and internal block definition diagrams into one block instance diagram. AVATAR provides a semantics to most SysML state machine elements.

SysML tools. A survey of the literature indicates that SysML tools that target real-time systems have been developed on top of UML tools in the form of SysML add-ons that reuse the capacity of the UML tool to translate a high level model into a formal model that may cater a formal verification tool.

- [21] uses Rhapsoldy and timed automata to formally verify the landing gear of a military aircraft.
- Artisan Studio [2] associates parametric diagrams with solvers such as Matlab or Excel.
- SysMLcompanion [3] translates a SysML model into a VHDL-AMS one.
- OMEGA SysML [4] translate a SysML model into a private intermediate form: IF.
- TOPCASED [5] also translates a SysML model into a private intermediate form: FIACRE.
- TTool [6], which is the subject of the paper, translates an AVATAR model into a timed automata, a pi-calculus specification or a Petri for temporal property verification, security flaw detection, and invariant search, respectively. What makes TTool really user-friendly for people not familiar with formal verification and formal methods in general, is the way the tool drives formal verification at the SysML model level and displays results at the SysML level too. The user of TTool is indeed not obliged to write logic formula to achieve formal verification. Nor he or she is obliged to inspect formal code to understand verification results.

Petri nets and invariants. Invariant search was introduced several decades ago for basic Petri nets. [22] reports a successful experience in applying invariant search to demonstrate token unicity on a local area network. The techniques is still implemented by Petri net tools such as TINA [23]. It has also been extended to search invariants in colored Petri nets [24] .

7 Conclusion

SysML tools that implement reachability techniques face the state explosion problem as far as complex real-time systems verification is at stake. So far, the open-source toolkit TTool has fallen in that category, for it translates a SysML/AVATAR model into timed automata, and model-check the latter using UPPAAL.

By contrast, the paper investigates formal verification of SysML/AVATAR models using a structural approach that does not require generating the state space of the model. The idea is to translate an AVATAR model into a Petri net and to search for invariants by solving an equation system derived from the incidence matrix of the Petri net. TTool implements invariant search algorithms and displays results at the SysML level. It also enables the designer to look for mutually exclusive actions or states in the SysML model. The user-friendliness added to the invariant interpretation phase is a real added value of the tool.

The overall contribution could be adapted to other UML and SysML environments structuring systems with classes / blocks and state machine diagrams.

An education case study of modeling a microwave oven has shown that invariant search usefully complements model checking. Risks of starting the oven before the door is actually closes have been revealed by invariant search only, and a handshake procedure has been added to the model.

References

1. Debbabi, M., Hassane, F., Jarraya, Y., Soeanu, A., Alawneh, L.: Verification and Validation in Systems Engineering: Assessing UML/SysML Design Models. 270 pages, ISBN 978-3-642-15227-6, Springer (2010)
2. Atego ARTiSAN Studio. http://www.atego.com/products/artisan- studio/
3. SysML Companion. http://www.realtimeatwork.com/software/sysml-companion/
4. Dragomir, I., Ober, I., Lesens, D.: A Case Study in Formal System Engineering with SysML. In 17th International Conference on Engineering of Complex Computer Systems (ICECCS 2012), pp.189–198, IEEE Computer Society (2012)
5. TOPCASED, http://www.topcased.org
6. TTool. http://ttool.telecom-paristech.fr
7. Knorreck, D., Apvrille, L., De Saqui-Sannes, P.: TEPE: A SysML Language for Time-Constrained Property Modeling and Formal Verification. In ACM SIGSOFT Software Engineering Notes, Vol. 36(1), pp.1–8, ACM. (2012)
8. Pedroza, G., Knorreck, D, Apvrille, L.: AVATAR: A SysML Environment for the Formal Verification of Safety and Security Properties. In New Technologies of Distributed Systems (NOTERE 2011), pp.1–10.
9. Apvrille L, Becoulet A.: Prototyping an Embedded Automotive System from its UML/SysML Models. Proceedings of Embedded Real Time Systems and Software (ERTSS 2012) www.erts2012.org/Site/0P2RUC89/3C-1.pdf
10. Bengtsson J., Yi. W.: Timed Automata : Semantics, Algorithms and Tools. In Lecture Notes on Concurrency and Petri Nets, pp.87–124, LNCS 3098, Springer (2004)
11. Blanchet, B.: Using Horn Clauses for Analyzing Security Protocols. In Formal Models and Techniques for Analyzing Security Protocols, Cryptology and Information Security Series Vol. 5 pp.86–111, IOS Press (2011)

12. Murata, T.: Petri Nets: Properties, Analysis and Applications. In: Proceedings of the IEEE, Vol. 77(4) pp.541–580 (1989)
13. Diaz, M.: Modeling and analysis of communication and cooperation protocols using petri net based models. Computer Networks Vol. 6(6) pp.419–441, North-Holland (1982)
14. Diaz, M.: Petri Nets : Fundamental Models, Verification and Applications. 768 pages, John Wiley & Sons (2009)
15. Farkas, J.: Theorie den einfachen Ungleichungen. Journal für die reine und angewandte Mathematik (Crelle's Journal), issue 124, pp.1–27 (1902)
16. Colom, J.-M., Silva, M.: Improving the Linearly Based Characterization of P/T Nets. In Advances in Petri Nets, LNCS Vol. 483 pp.113-145, Springer (1991)
17. Kelling, E., Friedewald, M., Leimbach, T., Menzel, M., Séger, P., Seudié, H, Weyl, B.: Specification and evaluation of e-security relevant use cases. Technical Report Deliverable D2.1, EVITA Project (2009)
18. Object Management Group: OMG Systems Modeling Language (OMG SysML^{TM})Version 1.3 http://www.omg.org/spec/SysML/1.3/PDF/
19. Object Management Group: Documents Associated With Unified Modeling Language (UML), V2.4.1 http://www.omg.org/spec/UML/2.4.1/
20. Ober, Il., Ober, Iu., Dragomir, I., Aboussoror, E.A.: UML/SysML semantic tunings. In Innovations in Systems and Software Engineering Vol. 7(4) pp.257–264, Springer (2011).
21. da Silva, E.C., Villani, E.: Integrando sysml e model checking para v&v de software critico espacial. In Brasilian Symposium on Aerospace Engineering and Applications (2009) http://www.cta-dlr2009.ita.br/Proceedings/PDF/59054.pdf
22. Ayache, J.-M., Courtiat, J.-P., Diaz, M: REBUS, A Fault-Tolerant Distributed System for Industrial Real-Time Control. IEEE Transactions on Computers, vol. 31(7) pp.637–647, IEEE (1982)
23. Time Petri Net Analyzer, http://projects.laas.fr/tina/
24. Jensen K.: Coloured Petri Nets and the Invariant Method. Theoretical Computer. Science, Vol. 14(3) pp.317-336, North-Holland (2002)