Adapting models to model checkers, a case study : Analysing AADL using Time or Colored Petri Nets*

Xavier RENAULT, Fabrice KORDON
Université Pierre & Marie Curie,
Laboratoire d'Informatique de Paris 6/MoVe
4, place Jussieu, F-75252 Paris CEDEX 05
xavier.renault@lip6.fr,
fabrice.kordon@lip6.fr

Jérôme HUGUES,
Institut TELECOM, TELECOM ParisTech, LTCI
46, rue Barrault, F-75634 Paris CEDEX 13
jerome.hugues@enst.fr

Abstract

The verification of High-Integrity Real-Time systems combines heterogeneous concerns: preserving timing constraints, ensuring behavioral invariants, or specific execution patterns. Furthermore, each concern requires specific verification techniques; and combining all these techniques require automation to preserve semantics and consistency.

Model-based approaches focus on the definition of representation of a system, and its transformation to equivalent representation for further processing, including verification and are thus good candidates to support such automation.

In this paper, we show there is a strong requirement to automatically map high-level models to abstractions that are dedicated to specific analysis techniques taking full advantage of tools. We discuss this requirement on a case study: validating some aspects of AADL models using both coloured and time Petri Nets.

1 Introduction

The increasing development of distributed real-time embedded (DRE) systems requires the definition of dedicated process to cope with multiple levels of concerns such as: *(i)* resource dimensioning (processor, memory, time, bandwidth, etc.), *(ii)* behavior expectation (causality of events, correct processing of all events, safety, etc.), and *(iii)* cost reduction. Furthermore, current systems now usually require multiple expertise from domain-specific engineers, system integrators, quality and assurance teams.

In this context, Model Driven Engineering (MDE) is a convenient approach to gather those concerns as a set of combined but yet heterogeneous models. Current application of MDE advocate for a clear separation of concerns that are expressed by means of a several of models (from high-level specifications down to implementation models). Although they aim at defining an "ideal process", they seldom address an important issue: how to combine models and tools in order to ensure both consistency between specification levels and the mapping of specifications into refined ones more suitable for analysis.

There is a need for "Verification Driven Engineering" (VDE): the careful choice of design patterns, combined to Validation and Verification tools (V&V). In [15], we note this usually requires engineers to fully understands both benefits and drawbacks of V&V tools. This is crucial to use them in the appropriate conditions and avoid pitfalls like non-applicability of theory (such as schedulability analysis), or combinatorial explosion.

This paper shows how to take advantage of one notation, AADL to support VDE. AADL is a generic modeling framework for designing real-time embedded systems. Our goal is to automate, according to the system property to be verified, the choice of both an appropriate tool and the corresponding mapping to a specification it may process. Such an approach should increase the use of formal specifications in software and thus the quality of produced systems without increasing design time.

In particular, this work focuses on the behavioral analysis of AADL by means of Petri Net models, as an application of VDE. It illustrates how to take advantage of Petri Nets and their extensions for several types of properties to be checked (deadlock detection, schedulability dimensioning). This allows a more efficient prototyping (or MDE) approach in the development of real-time systems.

The next two sections present AADL and Petri Nets. Then, section 4 details how we map an AADL model containing time information into a Time Petri net and how we can verify some important scheduling properties. Section 5 illustrates our approach on a simple example.

*This work has been partially funded by the ANR Flex-eWare project.

2 AADL and V&V

AADL [20] is an architecture description language dedicated to the design of DRE systems standardized by the SAE. AADL is component-centric and allows to specify both software and hardware parts of a systems. It allows one to define consistent block interfaces, and to separate them from block implementation.

An AADL model is made of *components*. Software components (`data`, `thread`, `thread group`, `subprogram`, `process`) are distinguished from execution platform components (`memory`, `bus`, `processor`, `device`) and hybrid components (`system`).

The behavior of a system (e.g. how functional blocks interact) is fully defined in the standard by mean of "properties" (attributes with a dedicated semantics) to progressively refine the semantics of a system, e.g. dispatching invariants, communication patterns. Non-functional properties applied to each model element. Non-functional aspects of components can be described within an AADL model such as thread dispatching condition (periodic or sporadic), interface specifications and how components are interconnected. These have a deep impact on the system's behavior. Functional aspects (algorithmic specifications) are attached separately as source code by means of AADL properties. An introduction to AADL can be found in [6].

AADL provides two major benefits for building DRE systems. First, compared to other modeling languages, AADL defines low-level abstractions including hardware descriptions. These abstractions are more likely to help designing a full system, close to the final product. Second, hybrid system components help to refine the architecture as they can be detailed later in the design process, allowing for successive stages of modeling, from early requirements down to executable models.

AADL tools support schedulability analysis [21], dependability modeling [7] and automatic code generation [13]. Furthermore, numerous works explored the mapping of AADL semantics onto formal specifications for the verification of behavioral properties. This has been performed in multiple experiments from AADL to BIP [5], LOTOS [10] or TLA [19]. However, these works contemplate only a case study, whereas we strive at defining a generic mapping, automated in our OCARINA toolset.

3 Petri Nets and extensions

Petri Nets [9, 14] are a family of formal notations to describe the behavior of distributed and concurrent systems. They offer several facets, we first present basic Petri nets and then two extensions for analyzing AADL models.

Petri nets Petri Nets are composed with *places* that represent resources and *transition* that define constraints between places. Places hold tokens that are consumed or produced by transition according to a *firing rule*.

The firing rule is the following: if all precondition places of a transition hold a sufficient number of tokens, the transition is fired. Corresponding tokens are removed from input places and other tokens are produced in output places.

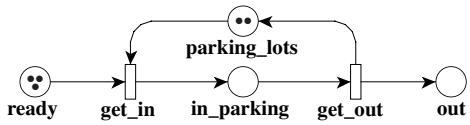

Figure 1. Petri net example

Figure 1 shows a simple Petri net modeling cars in a parking. In the initial state of the system, there are 3 tokens (cars ready to enter the parking) in place *ready* and 2 in place *parking_lots* (2 lots availables).

Transition *get_in* is enabled and may thus be fired (several transition may be enabled at the same time and the interleaving semantics is then used there to show all possible execution of the system). When it fires, it consume one token from input places (ready and parking_lots) and produces one token in *in_parking*. Then, *ready* contains only two tokens, *parking_lots* 1 and *in_parking* 1 token. From this new state, both *get_in* and *get_out* can be fired.

Colored Petri Nets Colored Petri nets are Petri nets in which tokens carry out information. This requires the declaration of "color domains" (data types) to type places and the marking they contain. The firing rule then not only relies on the presence of tokens in input place; it may implies some constraints on the token values.

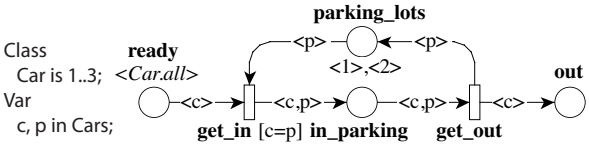

Figure 2. Colored Petri net example

Figure 2 shows a variation of the model of figure 1. There we consider that parking lots (variable *p* in the model) are associated to a given car (variable *c*in the model). Initial marking of place *ready* contains 3 cars with their individual id (<*card.all*> represents tokens 1 to 3 according to the color domain *Car*). There are two parking lots numbered 1 and 2. Then, according to the firing rule of colored Petri nets, only cars 1 and 2 can enter the parking because there is

no lot for cars 3. This behavior is expressed by guard `[c=p]` in transition *get_in*.

We chose symmetric nets[1] [4] (SN) for this work. SN is a class of Colored Petri nets that takes advantages of symmetries in distributed systems.

Time Petri Nets Time Petri nets [2] (TPN) are Petri nets where a time interval $[a,b]$, with $0 \leq a \leq b \leq +\infty$ is associated with each transition. Thus, an implicit clock is associated with transitions : this clock is reset when the transition is newly enabled. A transition cannot be fired until the clock reaches a and must be fired before the clock reaches b. A consequence of this firing rule is that the system reaches a deadlock when time constraints are contradictory.

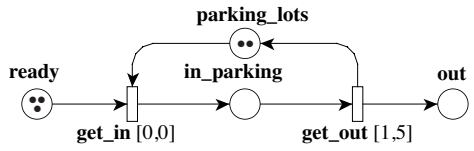

Figure 3. Time Petri net example

Figure 3 shows a new variation of the model of figure 1. as soon as it is enabled, *get_in* must fire immediately while we consider cars to be parked between 1 and 5 time units (the $[0,+\infty]$ interval is used to model Petri net transitions).

Clock values may be discrete or dense (for continuous time). In this paper we only consider discrete time.

Verification Capabilities Petri nets offer both structural analysis (computing from the model structure without having to generate the state space) and model checking (properties computed on the state space generated by "executing" the Petri net). Model checking on Petri nets allows to deduce safety properties (e.g. detection of a state having a given profile) or temporal properties (causal relation between events in the system). For example, a deadlock is a safety property while a livelock is a causal properties.

The advantage of SN is that we can use token values to follow more precisely the execution of the system. In our example, we can identify a given car and thus demonstrate that car 3 cannot fire *get_in*. However, color introduce complexity in the system that can be overcome by various techniques like a set-based representation of states that may reduce complexity by an exponential factor [4].

The advantage of TPN is that time can be inserted in properties to be verified. For example, traditional temporal logic such as CTL and LTL have been extended to consider time. Once again, this capability introduce more complexity that can be overcome by means of techniques like polynomial representation [8] or the region graph [22].

Such properties are of interest to analyze AADL specifications. SN allows to detect properties such as deadlock, livelock while TPN are useful to check for potentially missed deadlines.

[1] *Symmetric Nets* were formerly known as *Well-Formed Nets*, a subclass of *High-level Petri Nets*. The new name was chosen in the context of the ISO standardisation of Petri nets [11].

4 Generic Time Petri Net patterns for AADL

Qualitative properties verified in [18] thanks to SN are of interest to check if the system is safe. However, SN are not suitable for the verification of real-time properties such as deadline missed, or buffers overflow. In this paper, we propose to extend this work using TPN.

In AADL, behavior is mostly represented by Threads and their interactions. This section details the translation pattern of these elements into TPN. It is important to note that, at this stage, we focus on periodic threads in systems that are not preemptive (i.e. the `Preemptive_Scheduling` is set to false in the AADL model).

Exploited AADL properties For each thread, the following properties are analyzed and used as input parameters for the TPN pattern:

- `Period`: used to set *Period_Event* transition interval.
- `Compute_Execution_Time` specifies the amount of time that a thread will execute after a thread has been dispatched, before it begins waiting for another dispatch. It is used to set the Complete transition interval.
- `Compute_Deadline` specifies the maximum amount of time allowed for the execution of a thread's compute sequence. This is used to compute the WCET of a thread, and to set the WCET transition interval.
- `Priority` allows to compute which thread has a lower priority than another: if two threads may be dispatched, the one with the higher priority will be. Those priority relations are translated into priority arcs between Compute_Entrypoint threads transitions.
- `Dispatch_Protocol` decides which pattern is used for the model transformation. In this paper, we only focus on periodic threads.

An AADL specification presenting all these properties can be found in Figure 9.

TPN Pattern for AADL threads Our approach relies on a main pattern representing the thread life-cycle and derived from its modeling in the latest version of the AADL-standard [20]. Figure 4 depicts this pattern, derived from the SN one. We show how to complete it with time.

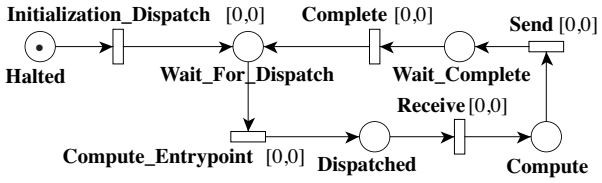

Figure 4. TPN Pattern for Thread Lifecycle

As shown in the figure 4, the thread life-cycle begins in an *Halted* state. Then, initialization can be processed and the thread waits for its dispatch. Since we focus on periodic threads, the dispatch is a clock event. Once dispatched, it collects data from its input ports (if any), processes these data, and potentially generates some event or data to other system thread via its output ports. Finally, once its task completed, the thread waits for another dispatching.

In the pattern, *Compute_Entrypoint* transition is the thread's dispatch interface: any incoming event (clock for example) will be linked to it.

Transitions are immediate (i.e. interval set to $[0,0]$, meaning firing occurs as soon as possible), since related time consumption is not relevant in the model compared to time consumption of the *Send, Compute, Receive* sequence in the thread lifecycle. This sequence could be expanded into subprogram calls.

The pattern In the presented lifecycle, no mention of periodicity is done. To achieve this goal, we need to introduce two places and one time transition in the pattern.

Figure 5 shows the enriched model. Transition *Period_Event* generates a token in place *Clock*: it means that the related thread must be activated.

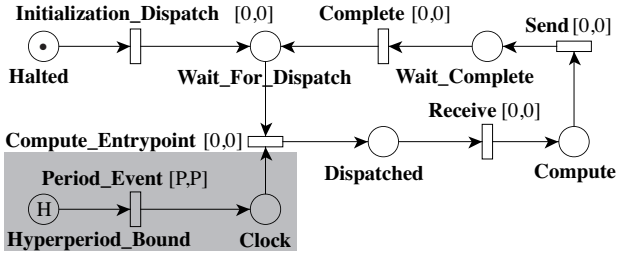

Figure 5. Thread Lifecycle + periodic dispatch

Period_Event is a time transition parameterized from the related thread period. Considering a P periodic thread (i.e. a thread with a P time units period), then *Period_Event* time interval is $[\ P\ ,\ P\ ]$. It means, that, each time the *Period_Event* transition is enabled, it will fire at after exactly P time units. Its firing generates one token in *Clock* and thus enables a dispatch.

H tokens are initially set in *Hyperperiod_Bound* to bound the total number of dispatch and avoid state-space explosion during verification. Value of H is computed in order to let the system work for a complete hyperperiod. For example, in a system with two periodic threads of respectively 5 and 15 time units, H should be set to 3 for the 5-periodic thread and to 1 for the 15-periodic thread. Of course, if only one thread is considered, as in Figure 5, then the marking is 1.

Handling multi-threaded systems Since multi-threaded systems share resources (at least the processor), a dedicated pattern must be added to the previous one.

Let us show this pattern for the processor (figure 6). It is then modeled by means of a place with as many tokens as there are cores. Presence of a token in this place (*Processor* in figure 6) means that the corresponding core is idle.

Then, execution of any actions requires a processor core to be available. Thus, the *Processor* place is a precondition of several transition in the following way. *Compute_Entrypoint* and *Complete* only check availability of the processor. *Receive* get the core for a computation that may take some time until *Send* is fired (when this dispatched event is treated) and release the token in place *Processor*.

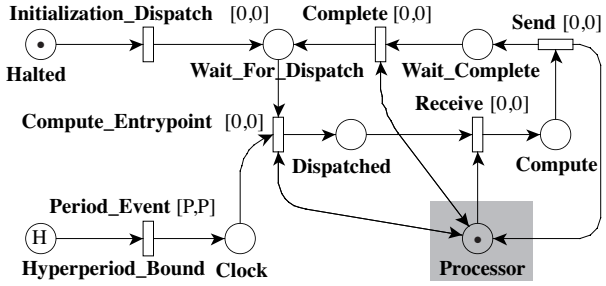

Figure 6. TPN thread pattern and processor

Using priorities between transitions: The TPN formalism allow us to specify priorities between transitions in the model. Then, we can potentially force a firing order when several are simultaneously enabled.

This is of interest for our mapping strategy to model scheduling policies, such as Rate Monotonic Scheduling (RMS): priority is indicated by setting up priorities between each thread *Compute_Entrypoint* transition of the system. Such information is picked up from the AADL model.

Note that we could expand the *Compute* place into more complex subnets, modeling different subprograms calls, consuming the processor resource.

At this stage, our patterns contains all the basic features to model the behavior of an AADL system.

Scheduling properties are verified by means of the observer technique [1]. Let us describe the way we apply this technique for two useful properties in real-time systems.

Checking potentially missed deadlines To detect missed deadlines, the grey subnet is added (Figure 7) to the pattern. The place *Working* gets a token when the corresponding thread is dispatched (i.e. transition *Compute_Enrypoint* is fired). This creates a race between transitions *WCET* and *Complete*. *WCET* time interval is set to $[W,W]$ where W is the worst case execution time.

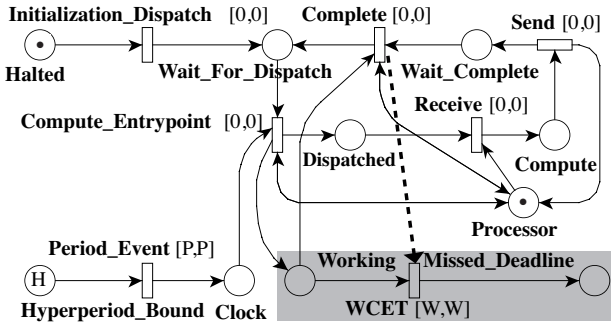

Figure 7. Missed deadline TPN pattern

When the computation meet the thread deadlines, *Complete* fires first, thus consuming the token in *Working*. Otherwise, *WCET* is fired and marks *Missed_Deadline*. Let us note that *Complete* priority must be over *WCET* priority to cope with the case where W time units is reached. This is graphically noted with the dotter arc in figure 7 and an orange arc in figure 9 (following TINA's tool convention).

Thus, if there is no way to have a token in all the *Missed_Deadline* places in the system state space, there is no way for the system to miss any deadline. Note that priority relations are represented with dotted arrows.

Checking potentially missed activations To detect missed activations, we add a place and a transition to the pattern (grey subnet in Figure 8). Transition *Activation_Time* is immediate, and has to be fired if the current thread is still working when a new Clock event is produced (i.e. a token in the Clock place). This transition races with *Compute_Entrypoint* but the later has a higher priority.

Then, when the thread must be dispatched but missed its activation deadline, then *Activation_Time* is fired and *Missed_Activation* is marked. Thus, if there is no way to have a token in all the *Missed_Activation* places in the system state space, then the system miss no activation.

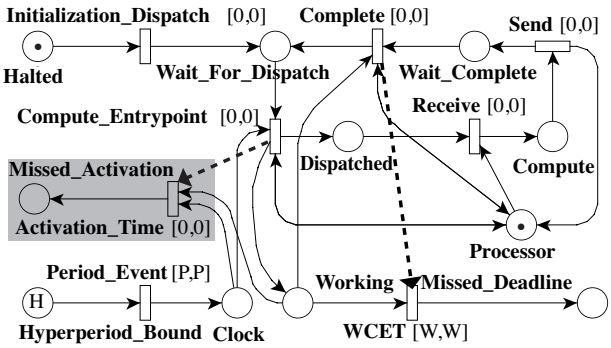

Figure 8. Missed activation TPN pattern

5 A Case Study

This section presents an application of our approach.

The AADL model and its TPN translation In Figure 9, we consider two periodic threads running on a single processor, with a FIFO_Within_Priorities scheduling policy. Those two threads interact through their ports. The sender thread sends a message to the receiver thread each time it is dispatched. This model is simple, yet insightful as it shows how to combine multiple verification.

Analysis on the SN model In [18], we have presented the analysis of SN generated from AADL with CPN-AMI [17]. It shows that typical qualitative analysis such as deadlock detection or evaluation of temporal logic formulae (causal analysis) provides precious information to system designers. We experimented both deadlock and livelock detection, as well as the verification of communication capabilities such as the dimensioning of buffers.

Such analysis is useful when checked properties are verified. Yet, qualitative analysis relies on all possible behavior while sometimes, timing constraints suppress some "pathologic" execution. In other words, the SN model may generate a greater state space than the TPN one for qualitative properties. Thus, such properties may not be verified on the SN model but are, in fact, true on the time model ("pathologic" executions are suppressed due to timing constraints).

So, SN-based analysis is of interest because it does not require timing information and thus provide useful hints to designers at an early stage of system design. Later, when time information is available, TPN-based analysis brings interest: 1) for time properties, 2) for qualitative analysis of properties that could not be verified on the SN model and could be verified on the TPN specification.

Finally, let us note that state-space explosion is also concern since no technique such as symbolic representation of

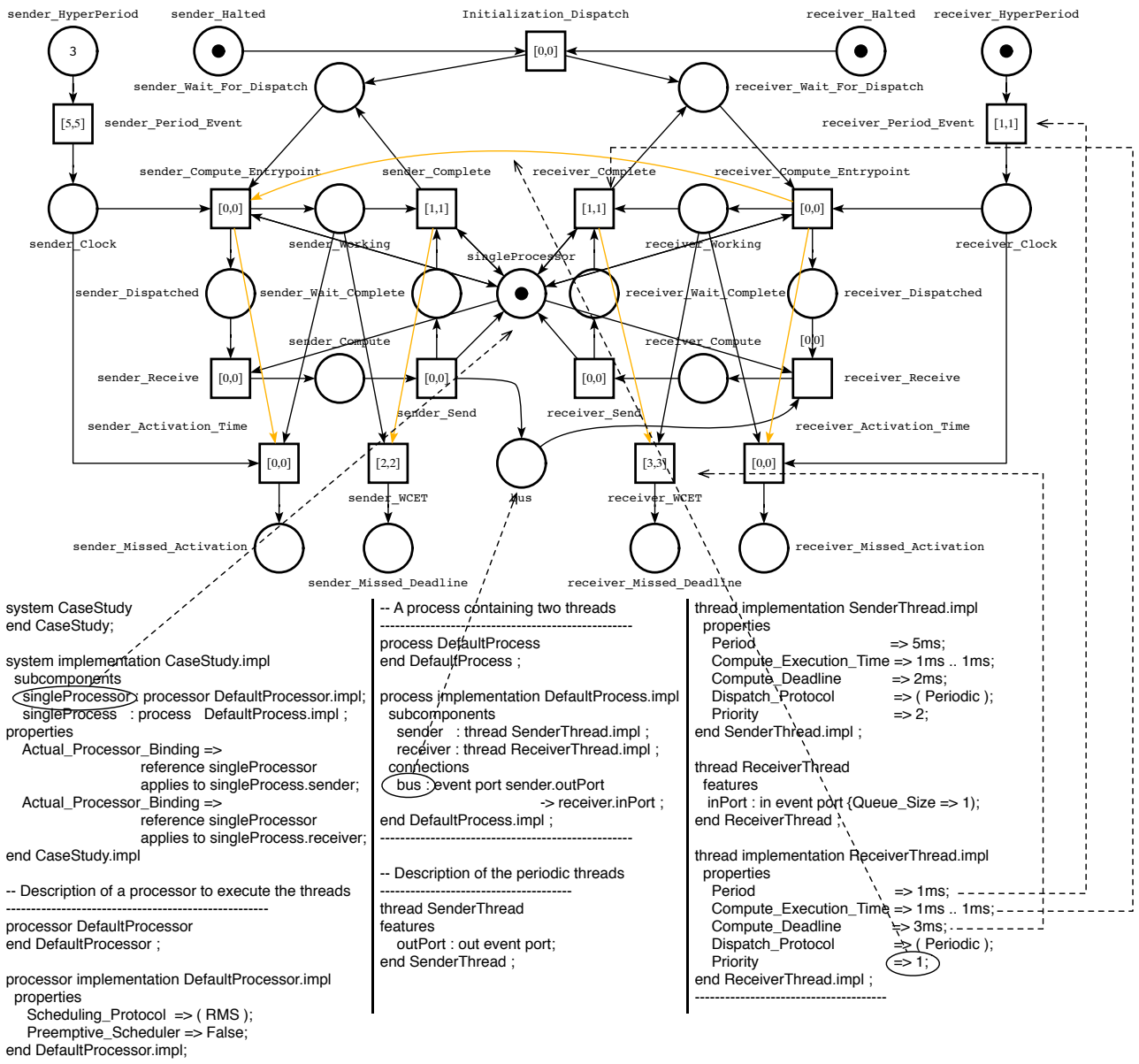

Figure 9. A case study: two periodic threads interacting

states is available for TPN to provide compact state-space storage in memory (gain can be exponential for real systems as shown in [12]). So, when combinatorial explosion cannot be fought for very large systems with TPN, it usually remains at a reasonable level on the SN specification.

Analysis on the TPN model Let us now focus on time analysis with TPN with TINA [3, 16]. It allows to compute the reachability graph but for TPNs. The state space for the model of figure 9 contains 22 states and 21 arcs. It shows that the model contains 4 dead transitions (i.e. transitions which are never fired): *sender_WCET*, *sender_Activation_Time*, *receiver_Complete* and *receiver_Activation_Time*.

The *receiver_Complete* transition belongs to the thread lifecycle. If this transition is dead, then the receiver thread does not achieve its cycle. We detect an error in the scheduling parameters of our system since, as mentioned in the previous section, place *receiver_Missed_Deadline* is marked.

Another dead marking was expected: since the system is bound to one hyperperiod to master complexity of the state space, it corresponds to the end of the hyperperiod.

Checking model resources: The TPN generated models allow to check another interesting property: are the communication channel bounded ? Is any message lost (i.e. the channel has already reached its maximum capacity) ?

To do so, we use the Queue_Size AADL property, applied to the input port of the receiver thread (and, in a general manner, to any input port in an AADL specification).

We then use the following LTL formula: $[](bus \leq X)$, where X is the *Queue_Size value*. In our example, its value is 1, so the formula becomes $[](bus \leq 1)$. Note that $[]$ symbol stands for "always": the formula could be read as "In all execution state, bus marking is always lower than 1".

For the case study, the SELT model-checker in TINA returns FALSE, with a counter example where the *bus* place marking is equal to 2: the queue capacity is not big enough.

Deduced corrections: From those results, the engineer have to correct its specification: the system is not schedulable and threads period or WCET have to be corrected.

For communication channel, changing the *Queue_Size* from 1 to 2 is sufficient: the formula $[](bus \leq 2)$ is TRUE.

6 Conclusion

MDE is now becoming a standard development approach. In that context, it is of interest to exploit models for verification purpose in order to detect early misconceptions as well as violated expected properties. This is called Verification Driven Engineering (VDE) in [15].

This paper extends our previous work on the mapping of AADL onto Symetric Nets [18] to Time Petri nets, allowing the verification of time-based properties such as missed deadline or missed thread activation in a real-time system. SN patterns are revisited and observers are defined to check for such properties. We apply these patterns, and use model checkers to illustrate the benefits of our approach.

This work shows that it is of interest to use a high-level and well defined notation as a basis for the formal verification of various "typical properties" of real-time embedded systems. Since no formal notation fits all needs, each property is verified thanks to the most appropriate transformation (here, SN and TPN).

This is a first and important step to provide automated mechanisms that help designers to verify properties on their systems. With appropriate tools implementing the presented strategies, engineers will not need a deep knowledge of formal methods to formally verify their models. OCARINA already includes such tools as prototypes.

Future work will focus on the generation of LTL formulae from some high-level properties described by the designer at the AADL level.

References

[1] B. Alpern, B. Alpem, F. B. Schneider, and F. B. Schneider. Verifying temporal properties without temporal logic. *ACM Transactions on Programming Languages*, 11:147–167, 1989.

[2] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Trans. Softw. Eng.*, 17(3):259–273, 1991.

[3] B. Berthomieu, P. Ribet, and F. Vernadat. The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *Int. Journal of Production Research*, 42(14):2741–2756, 2004.

[4] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic Well-Formed Colored Nets and Symmetric Modeling Applications. *IEEE Transactions on Computers*, 42(11):1343–1360, 1993.

[5] M. Y. Chkouri, A. Robert, M. Bozga, and J. Sifakis. Translating AADL into BIP - Application to the Verification of Real-time Systems. In *Model Based Architecting and Construction of Embedded Systems*, 2008.

[6] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical report, 2006. CMU/SEI-2006-TN-011.

[7] P. H. Feiler and A. Rugina. Dependability Modeling with the Architecture Analysis & Design Language (AADL). Technical Report CMU/SEI-2007-TN-043, 2007.

[8] G. Gardey, D. Lime, M. Magnin, and O. H. Roux. Romeo: A Tool for Analyzing Time Petri Nets. In *17th International Conference on Computer Aided Verification*, volume 3576 of *LNCS*, pages 418–423. Springer Verlag, 2005.

[9] C. Girault and R. Valk. *Petri Nets for Systems Engineering*. Springer Verlag - ISBN: 3-540-41217-4, 2003.

[10] I. Hamid and E. Najm. Real-Time Connectors for Deterministic Data-flow. In *Proceedings of RTCSA'07*, 2007.

[11] L. Hillah, F. Kordon, L. Petrucci, and N. Trèves. PN standardisation : a survey. In *26th International Conference on Formal Methods for Networked and Distributed Systems (FORTE'06)*, volume 4229 of *LNCS*, pages 307–322, Paris, France, September 2006. Springer Verlag.

[12] J. Hugues, Y. Thierry-Mieg, F. Kordon, L. Pautet, S. Baarir, and T. Vergnaud. On the Formal Verification of Middleware Behavioral Properties. In *Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04)*, Linz, Austria, Sept. 2004. TO BE PUBLISHED.

[13] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. Rapid Prototyping of Distributed Real-Time Embedded Systems Using the AADL and Ocarina. In *Proceedings of the 18th IEEE International Workshop on Rapid System Prototyping (RSP'07)*, pages 106–112, Porto Alegre, Brazil, May 2007. IEEE Computer Society Press.

[14] K. Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use, vol. 1, vol. 2 et vol. 3*. Springer-Verlag, London, UK, 1995.

[15] F. Kordon, J. Hugues, and X. Renault. From model driven engineering to verification driven engineering. In *6th IFIP WG 10.2 International Workshop on Software Technologies*

for Embedded and Ubiquitous Systems, volume 5287 of *LNCS*, pages 381–393. Springer Verlag, 2008.

[16] LAAS. The TINA Home page, url: `http://www.laas.fr/tina/description.php`.

[17] MoVe-Team. The CPN-AMI Home page, url: `http://www.lip6.fr/cpn-ami`.

[18] X. Renault, F. Kordon, and J. Hugues. From AADL architectural models to Petri Nets: Checking model viability. In *12th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'09)*, pages 313–320, Tokyo, Japan, March 2009. IEEE CS.

[19] J.-F. Rolland, J.-P. Bodeveix, D. Chemouil, M. Filali, and D. Thomas. Towards a formal semantics for AADL execution model. In *European Congress on Embedded Real-Time Software (ERTS), Toulouse, 29/01/08-01/02/08*, 2007.

[20] SAE. Architecture Analysis & Design Language V2 (AS5506A), jan 2009. available at `http://www.sae.org`.

[21] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Scheduling and memory requirement analysis with aadl. In A. Press, editor, *proceedings of the ACM SIGADA International Conference*, volume 25, pages 1–10, 2005.

[22] I. Virbitskaite and E. Pokozy. A Partial Order Method for the Verification of Time Petri Nets. In *12th International Symposium on Fundamentals of Computation Theory*, volume 1684 of *LNCS*, pages 547–558. Springer Verlag, 1999.