Specification and Verification of Real-Time Systems using the POLA tool

Florent Peres[1,2], Pierre-Emmanuel Hladik[1,2], François Vernadat[1,2]
[1]CNRS ; LAAS ; 7 avenue du colonel Roche, F-31077 Toulouse, France
[2]Université de Toulouse ; UPS, INSA, INP, INSAE ; LAAS ; F-31077 Toulouse, France
fperes@laas.fr, *pehladik@laas.fr*, *francois@laas.fr*

Abstract

Real-time systems are becoming more and more complex, making it hard for the industrial community to actually use experimental tools which are able to resolve some of the specification and verification issues of such systems. This work presents a tool which automatically transforms a domain specific language for the specification of real-time systems into a time Petri net and some associated logical formulas. The TINA model checking toolbox is then used on the generated model to complete the automatic verification chain. Using this chain, the users can check real-time systems' validity without any prerequisite knowledge on how it actually were accomplished.

Keywords: D.S.L., model checking, real time systems

1. INTRODUCTION

Real-time systems are systems capable of reacting to external events before a certain amount of time specified *a priori*. It is essential to trust them since they are used in human safety critical systems such as aircraft, spacecraft, nuclear plants, etc. This confidence is hard to claim because such systems may possess complex behaviors, such as distributed system with many agents. But even a system as simple as a sensor can become practically impossible to check, due to the dense nature of time. In its essence, there is no atomic grain of time and, for example, certifying communicating systems for which we cannot state any conditions on communication arrival times (because of the absence of atomic time grain) can be a real challenge. One way to gain confidence is to actually see them running and this is why test and simulation are the most common techniques used in industry to certify that a critical system is correct. However, neither simulation nor test can guarantee that a system will effectively work in every possible situations.

Instead of the simulation or test techniques, this work is based on the so-called *model checking* technique, which consists of an exhaustive and automatic exploration of an input model (the specification) as a mean to extract from it interesting properties, expressed by logic formulas.

POLA[1] is a highly declarative domain specific language (DSL). One of its features is the ability to be coupled with a behavioral language whenever a specific behavior is needed that is not expressible using the core declarative part of the language. We currently use Petri nets to describe these specific behaviors, but we built the language such that the behavioral part is orthogonal to the declarative part, thus allowing the use of any language sufficiently expressive (timed automata or any languages with approximately the same expressivity level than Petri nets).

At its earliest stages this tool have been thought of as a means to check schedulability on a very simple type of systems but, as it matured, the specifiable system range increased, as well as the checkable properties number. Those properties can be part of the automatic generation process, as we will see in section 4, or they can be specific to a specification, using an adequate temporal logic formula. Even if the system schedulability check remains one of the most important properties, it is not the only one possible. Once the schedulability is ensured, it is essential to make sure that the system actually do what it is expected to do in spite of the constraints imposed by the

[1]POLA stands for POLicies Analyzer.

scheduler. Moreover, it is not because a system correctly behaves when working with a scheduler that it will remain true for any schedulers. Thus, the specific verification needs are impossible (and especially counterproductive) to take into account in a general *and* automatic way, the designer being the most capable of choosing the most interesting properties to check.

The next section will introduce the real-time system DSL POLA. Then two case studies will be presented: one will show the capabilities of POLA to handle OSEK [3] like systems, and the other to demonstrate the possibilities of this language with regard to the ARINC 653 standard [1]. These two standards are respectively used in automotive and avionic industries. Then, with the help of a simple example, we will present the inner core of the tool, followed by some experiments, to finally conclude this paper with a related works section.

2. THE POLA LANGUAGE: A DOMAIN SPECIFIC LANGUAGE FOR REAL-TIME SYSTEMS

In our model of real-time, we represent task systems in which there are resources and tasks scheduled according to scheduling policies. The scheduler description is split in two parts: the policy and the resource allocation.

Systems. A (task) system is the higher level element of a POLA specification. It allows a better decomposition in functional parts of the global system (parts in which names are protected according to the hierarchy of the POLA elements). A system can be **preemptable**: every clocks within that system are suspended when the system is disabled. In the default case (no **preemptable** keyword) the clocks of such a system are reset when it is disabled. A system includes the definition of tasks, resources, scheduling policies, resources allocations, and a possible *ad-hoc* behavior.

Tasks. Tasks are the operational elements handleable by the scheduler. They can be decomposed in several **actions**, in which case they require the adjunction of a behavioral glue between each of the actions to represent the task behavior; or by a single one, fully representing the computational capacity of the task. A task can have a **period** interval parameter: a point interval represents a periodic task, an unbounded interval represents a sporadic task, etc. A real-time task also have a **deadline** parameter defining the instant after which the task is to be considered faulty. Deadline misses are captured by events which are handleable at the verification phase. A task must specify a scheduling **policy** according to which it will be scheduled. Finally, a task can be declared (resp. **not**) **preemptable** (preemptable is set by default), which tells the scheduler it has (resp. has not) the capacity to interrupt it.

Policies. Policies define tasks priority order. The set of all the tasks is partitioned according to the available policies. To describe policies, we adapted the work in [8], in which the author used a mathematical representation of a task with which he could represent a wide range of policies and be able to reason about them. As the result of this adaptation, a policy is specified by one of the two sort operations: **max** and **min** and by a linear expression on the following characteristics of the tasks: Capacity (C), i.e. the sum of the execution times of the task's actions; Period (P), deadline (D) and user fixed priority level (L). For example:

policy *RateMonotonic* ***is*** *min P* (1)

These are static characteristics. To define Earliest Deadline First (EDF), we must use *dynamic* characteristics written using lower case letters (c, p, d) instead of upper case ones (C, P, D); L has no dynamic counterpart. Here are some examples of how policies can be written in POLA, using a list of commonly used scheduling policies:

- Deadline monotonic $\Longleftrightarrow$ **policy** DeadlineMonotonic **is** min D
- Rate monotonic $\Longleftrightarrow$ **policy** RateMonotonic **is** min P
- User Defined $\Longleftrightarrow$ **policy** UserDefined **is** min L. L is fixed by the user by adding to task declaration, the keyword **level** followed by a natural number.
- Earliest Deadline First $\Longleftrightarrow$ **policy** EarliestDeadlineFirst **is** min (D - d). This policy is not handled by POLA right now. The d means the dynamic clock running as soon as a task is released and stopped and reset as soon as a task is unreleased. D is the deadline, so $D-d$

is the dynamic value that gives the time left before deadline. Taking the minimum of these values is then equivalent to have a E.D.F. policy.

- First come first served (a.k.a FIFO) $\Longleftrightarrow$ **policy** FIFO **is** max p. As previously for d, p is the dynamic clock that starts running as soon as the task is released and reset only on a new period event (when P is reached) or if period generation is deactivated (in which case it is also stopped). Then *max p* correspond to the earliest released task (because it has the highest p), and then it is indeed the first that “came” (and will be the first served).

We recall that the POLA tool currently uses prioritized time Petri nets as its back-end language. Even if we are confident that some of the dynamic policies (like EDF or FIFO) can indeed be translated in prioritized time Petri nets, this is not obvious for the general case and consequently, the implementation of such dynamic policies is left for future works. Furthermore, we think that this high expressivity exigence of dynamic policies’ translation is an issue for many current dense time models (such as timed automata). This may not be the case for hybrid automata, but this formalism is not quite suited for model checking.

Allocations. The scheduling policies, stating tasks execution order, are not sufficient to completely specify a scheduler: it still needs to know the resource(s) a task requires for its execution. This is the role of the allocation elements. It specifies a task list and a resource list where each task of the task list require every of the resources in the resource list to be available to be able to execute. Availability of a resource means that this resource can be preempted, or simply that no task uses it at the moment. Even if an allocation links tasks to resources, it is itself linked to actions. It allows a more dynamic resource allocation of tasks (because allocations can be deactived), while keeping task (instead of actions) as the lowest element handleable by the scheduler, which means a less complex scheduler.

The scheduler is automatically generated from the informations of policies, allocations and preemptability of tasks and resources. The role of the scheduler is to give (resp. take) the resources to (resp. from) the task currently possessing the highest (resp. lower) priority.

Behaviors. This part allows the user to define a specific behaviour (using Petri nets) and binds it to the current POLA specification. The bindings are done using what we call *accessors* which are actually labels. Binding operation is then the composition of the POLA translated net and the given behavior, according to labels.

3. CASE STUDIES

This section presents two case studies that have been successfully handled by POLA, giving a good idea of what it can successfully be applied on.

3.1. OSEK like systems

The OSEK standard [3] defines an operating system adapted to the automotive industry. *Group of tasks* is one of its many features. Every tasks within a group of tasks are considered not preemptable by the other tasks of the same group. This case study is a system that respects OSEK standard and specify two groups of tasks. The first one have a higher priority than the second. Therefore, every tasks in the second group are preemptable by any tasks in the first group, but no tasks within the first group are preemptable by any other tasks. The standard also defines that tasks can indeed be preempted at certain points of their execution. This can be specified using multiple actions: a task is decomposed into two successive actions, the first freeing the resources as soon as it finishes its activity (this is denoted by the **giveback** keyword); the second terminating the task (denoted by the keyword **endoftask**).

In this case study, the important point is the resource management specification. The tasks in the first group have to always be capable of preempting task that belong to the second group. Inside the first group, resource attribution behaves according to the corresponding scheduling policy. To specify this behavior, we use a preemptable resource (*proc* in this example) that every tasks of the system are allowed to use. A consequence of this choice, is that every tasks of the first group must have a higher priority than every tasks of the second group. Because tasks of the first group are not preemptable (even by tasks of the first group) they are declared *not preemptable*. Tasks of the second group behaves in the same way: while a second group task is executing, no second group tasks can interrupt it. But we cannot declare them *not preemptable*, as we did with first group tasks, because this would specify that *no* tasks can interrupt second group tasks, but in our case, we actually wanted first group tasks to be able to interrupt second group tasks. Instead of declaring tasks as *not preemptable*, we use *another* resource (*vproc* in this example) which is declared *not preemptable* and can be considered as a virtual resource. Only second group tasks are allowed to use it, and thus, because they use *two* resources, one stealable by first group tasks and the other not stealable by anyone, we are assured that second group tasks will not be interrupted by any tasks except those of first group.

```
system osek is
    res vproc is not preemptable
    res proc is preemptable
    not preemptable task T1 is
        action act1 in [5,5] with alloc1
        period [31,31]
        deadline 21
        policy RM
    end
    task T2 is
        action act1 in [4,4] with alloc2
        period [73,73]
        offset [7,7]
        deadline 25
        policy RM
    end
    task T3 is
        action act1 in [7,7] with alloc2 giveback
        action act2 in [8,8] with alloc2 endoftask
        period [97,97]
        deadline 45
        policy RM
        behavior is
            tr a1end s1 -> s2
            tr a2end s2 -> s1
            pl s1 (1)
            lb indus.T3.act1 a1end
            lb indus.T3.act2 a2end
    end
    policy RM is min P
    allocation alloc1 is
        resources proc
        tasks T1
    allocation alloc2 is
        resources proc, vproc
        tasks T2, T3
end
```

FIGURE 1: OSEK like task system

In the example of figure 1, the task $T1$ belongs to the first group and $T2$ and $T3$ belong to the second one. To be able to execute, $T1$ only requires the resource *proc*, which is always preemptable (even if it is not really important here as no other task belongs to the first group). Tasks $T2$ and $T3$ require both *proc* **and** *vproc* to execute: *vproc* to express non preemptability and *proc* because first group tasks have to be allowed to preempt them. The allocation parts, *alloc1* and *alloc2*, are responsible for the specification of which tasks require which resources, thus indicating to the scheduler the actions it will have to take. Although in this example it may seem redundant to associate allocation to actions, this is in fact not the case, as a task can have numerous actions (like for the task *T3*) and use a different allocation for each. By associating allocation to actions, tasks are then able to use different resources all along their execution.

Described alone, two actions are considered to run simultaneously as soon as the allocation give to the task all the needed resources. This is not the behavior we want. To make them run one after the other, we will use a specific behavior described using Petri nets (a textual form to be precise). First, we must say that initially the task is in state *s1* (i.e. initially place *s1* have one token, place is specified using **pl** keyword). We then specify that from *s1* we can go to task state *s2* (this is done with the transition *a1end*, transitions are specified using **tr** keyword), and the other way around for transition *a2end*. Then we link those transitions with the actions: *act1* is linked to *a1end* and *act2* to *a2end* (using the **lb** keyword). This means that the firing condition of *a1end* (i.e. there must be a token in *s1*) is added to *act1* (and this is the same for *act2* w.r.t *a2end*). The action *act1* can only be activated when there is a token in *s1*. When it finishes, it removes the token in *s1* and create a new one in *s2*, which activates *act2* in turn.

3.2. ARINC 653 like Partitioning Systems

ARINC 653 [1] is an avionic standard especially thought to ease certification processes as defined in the DO-178 norm. One of the main characteristics of ARINC 653 standard definition is that it uses the so-called *partitions*. A partition regroup tasks operating within the same time frame and within the same memory space. Partitions are useful because they split independent processes, so that they can run on the same hardware, but without even knowing that they are sharing resources. Thus, when a process hangs, the sane parts are not affected (under the assumption that the partition switching process remain sane). Such a system is specified on the figure 2.

```
system arinc653 is
    behavior is
        tr topartition2 [50,50] p1 -> p2
        tr topartition1 [50,50] p2 -> p1
        pl p1 (1)
        lb partition1.active p1
        lb partition2.active p2
end
preemptable system partition1 is
    res proc is preemptable
    task T1 is
        action act1 in [10,10] with alloc
        period [50,50]
        deadline 50
        policy RM
    end
    task T2 is
        action act1 in [20,20] with alloc
        period [50,50]
        deadline 50
        policy RM
    end
    policy RM is min C
    allocation alloc is
        resources proc
        tasks T1, T2
    end

noinit preemptable system partition2 is
    res proc is preemptable
    task T1 is
        action act1 in [10,10] with alloc
        period [50,50]
        deadline 50
        policy RM
    end
    task T2 is
        action act1 in [20,20] with alloc
        period [50,50]
        deadline 50
        policy RM
    end
    policy RM is min C
    allocation alloc is
        resources proc
        tasks T1, T2
    end
```

FIGURE 2: An ARINC 653 like system

A system that respects the ARINC 653 standard comprise several partitions executed sequentially and periodically. When the end of an execution frame of a partition has been reached, it is suspended and the next one is resumed. Even if partitions are independent, the time model is not manipulated as we would expect it to be, that is, local to partitions. Indeed, in the standard definition, time is considered global and it is up to the user to keep into account the duration of the other partitions when building the architecture of a partition. Even if it is expressively powerful, it can be the source of many errors, as, for example, it allows to release a task outside the execution time interval of the partition to which it belongs, which in turn may shift deadline outside the execution frame of its partition. As far as we know, such possible behaviors are avoided by ARINC 653 users by using a coding rule set, which is equivalent to consider time as local to partitions. Those rules form a subset of ARINC 653 , and this is this subset that can be easily specified using POLA.

The example of figure 2 is composed of two preemptable systems named *partition1* and *partition2* and of a controller system *arinc653*. The keyword **preemptable** is used in front of the system declaration to specify that such a system can be suspended. This means that all the clocks of a **preemptable** system are suspended when the system is suspended. Initially the system *partition2* is suspended (declared with **noinit**) whereas *partition1* system is initially running. Both systems are identical: they possess two tasks which are scheduled according to "minimum capacity first" (**min C**), thus task *T1* have a higher priority than *T2*. The resource they use is preemptable, but this is not a very useful information in this system as each task have the same period. The successive execution and suspension of the two partitions is done by the *arinc653* controller system. The behavior is identical to the one presented in the OSEK case study. The accessors used in this example allow the user to change the activity of a system: if a system is inactive, it is suspended (and it executes if the system is active).

4. VERIFICATION

An input POLA model is automatically translated into a time Petri net [7] (extended with priorities, see [2], and stopwatches [12] when needed), which gives an operational model of the system. Along with this Petri net, it also automatically produces a set of temporal logic formula, giving the set of the expected properties. Those two elements are given as input to the TINA toolbox [4] which is in charge of the actual analysis step. The analysis concludes whether the expected properties hold for the checked specification. See figure 3 for a better view of the data flow.

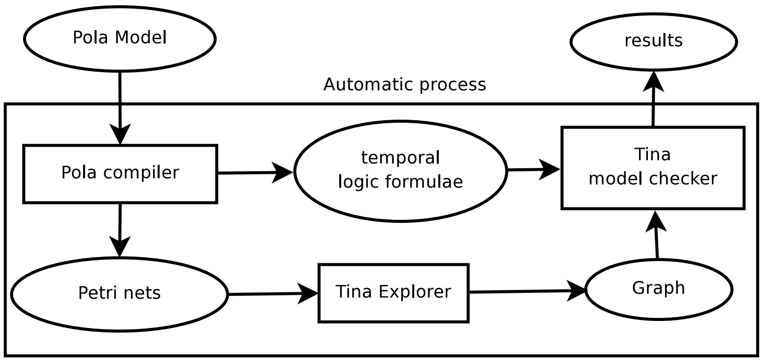

FIGURE 3: The automated verification chain

The verification process of the tool is presented using a very simple POLA model. This simplicity is required to be able to present every step of this verification process. The simple POLA model we chose (see figure 4) is made of a single resource $proc$ and two tasks $T1$ and $T2$. The scheduling policy is Rate Monotonic which implies that $T1$ have the highest priority because of its lowest period.

```
system simple is
res proc is preemptable
policy RM is min P
task T1 is
    action a1 in [1,1] with alloc
    period [2,2]
    deadline 1
    policy RM
end
task T2 is
    action a1 in [1,1] with alloc
    period [3,3]
    deadline 2
    policy RM
end
allocation alloc is
    resources proc
    tasks T1, T2
end
```

FIGURE 4: A very simple POLA model

Even if the goal of this paper is not to present POLA's semantics, we wanted here, using a very simple example, to skim over how the translation, which we recall is automatically done, is accomplished and, as a side effect, what the result looks like.

The decomposition depicted in figures 5, 6, 7 and 8 presents the four patterns needed in this example: actions, period, deadline and allocations. The Time Petri nets presented in this figure use several extensions: read arcs [9] which are drawn using a completely black circle instead of an arrow; they work like normal arcs but when the transition is fired, the tokens are *not* removed. This extension is very useful when used together with time transitions [10]. There are some inhibitor arcs [11], drawn using a black circle filled in white. These arcs prevent the transition from being fired whenever the precondition would be fulfilled with a normal arc, in other words this is a negative of read arcs. And finally stopwatch arcs [12], drawn using a black square. They suspend the transition clock when the stopwatch conditions are not fulfilled.

Let us describe those different patterns:

Common to all Every pattern inside a system will require this system to be active(*simple.active*).

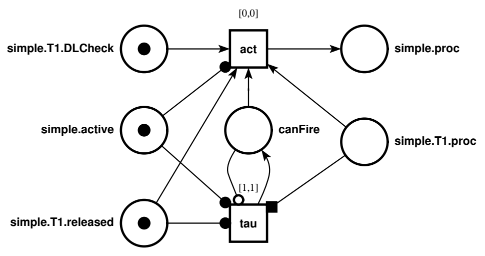

FIGURE 5: Action (T1 case)

Action Because there is a preemptive resource, the actions are split into two transitions: a transition holding the actual time information (transitions tau), and a transition representing the actual action (transition act). An action requires the task to be released (*simple.T1.released*). As this is the only action, it ends the task (coded with a normal arc from *T1.proc* to *Act*). As soon as tau has fired, act can fire and must fire urgently (because of $[0, 0]$ interval). The arc from *canFire* and *tau* is an inhibitor arc that prevents *tau* to be fired when *canFire* have at least one token. Finally, the clock of tau advance when the task possesses the processor (when there is a token in *simple.T1.proc*) and act frees the processor at the end of its execution (in *simple.proc*).

Period When the system is active, the period generator periodically generates a token in *simple.T1.released* indicating the task is released, and in *simple.T1.DLCheck* indicating the deadline checker is turned on. The place *simple.T1.period.active* indicates activity of the period generator. Therefore, the generator can only work if it is activated, and after generating a period it reactivates itself (usefull to be able to interrupt for a time the period generation).

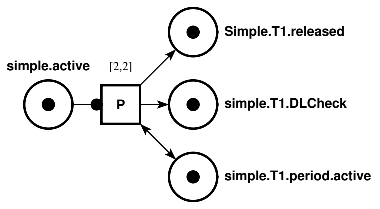

FIGURE 6: Period (T1 case)

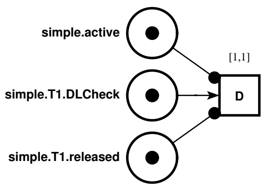

FIGURE 7: Deadline (T1 case)

Deadline The deadline counts the time elapsed since the instant the task have been released (coded using a read arc) and desactives itself as soon it fires (in this model we do not interrupt the behavior on a deadline miss, this is left to the user to accept or not system in which *D* can be fired).

Allocations The Allocation is the decision center of the scheduler. This pattern is in charge of taking/giving ressources from/to tasks. Each transition represents a possible case. Here, because there is very few tasks, the generated allocation pattern is very small, but as the reader might imagine, when there are numerous tasks, the number of possible cases increases greatly. The user can active/deactivate an allocation globaly (no token in *simple.alloc.active*) or for a task (no token in *simple.alloc.T1.active*). To deactivate an allocation for a given task means that the parts of the allocation giving resources to the task are deactivated. A task can only receive a resource when it is released (*simple.T1.released*). There is two possible cases for the task *T1* to have the processor: either the processor is free and *T1* does not possess it, or the processor belongs to *T2* (and then *T1* does not have the processor. This case is usefull when there can be more than one resource (as for multicore processors)). There is only one possible case for *T2*, because it has the smallest priority of the two. For *T2* to have the resource *simple.T2.proc*, it must be free (*simple.proc*).

Priorities are missing on the graphical representations: actions and period have a greater priority than allocations; deadlines are of lower priority lower than actions; allocations are ordered in the priority relation according to the scheduling policy of the tasks participating to the allocation.

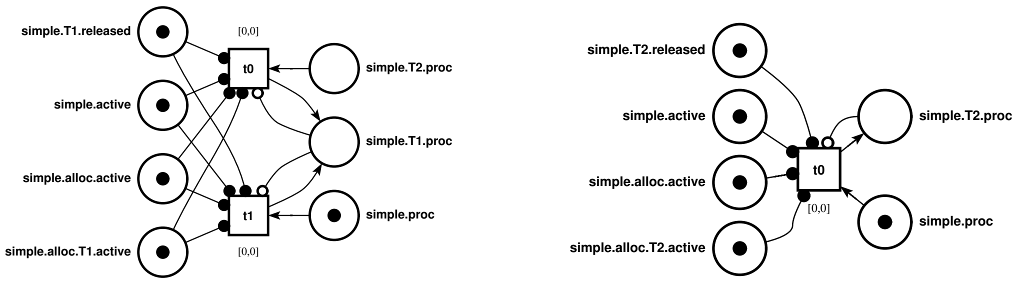

FIGURE 8: Allocation (T1 on the left and T2 on the right)

After the translation of the POLA model into a Petri net, this Petri net is given as an input of the TINAexplorer (see figure 3), which is in charge of the exhaustive behavior exploration of this simple network. The result of the exploration is given by the easy to read Kripke structure of figure 9. This automaton-like representation expresses every possible behaviors of the system starting from state 0. The reader will note that the behavior of this system is extremely cyclic.

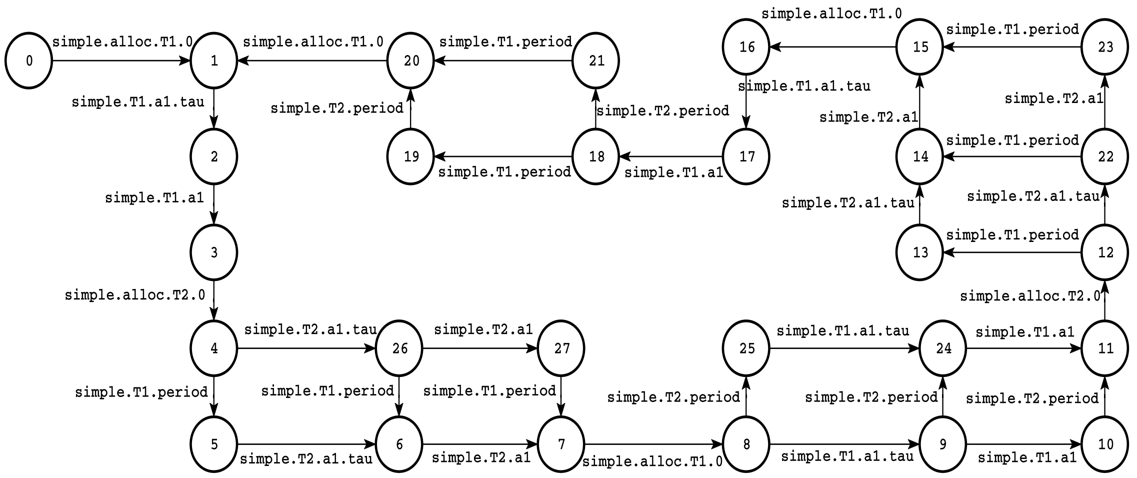

FIGURE 9: The behavior abstraction graph resulting from the analysis done by Tina

This structure is then passed as an input to the model checker along with the three following basic properties:

Does a given task T_i miss its deadline? This is checked by the Linear Time Logic (LTL) formula: $[]\neg DLMiss_{T_i}$, where $DLMiss_{T_i}$ represents the deadline miss event of task T_i.

Does a task execute at least one time each of its actions? This is checked by the LTL formula on every actions: $\diamond Action_j^{T_i}$, where $Action_j^{T_i}$ is the $Action_j$ of T_i.

Does each action execute infinitely often? This is checked by the following LTL formula on every actions: $[] \diamond Action_j^{T_i}$.

The user only see and manipulate its POLA model and eventually get the result from the verification chain (see fig 1), thus demonstrating how far we can go in hiding the verification process to the end user.

From the tool output printed below, there first appears TINAouput which give some informations about the generated kripke structure (stored in a file with *ktz* extension). Then appears the checking step of the three formulas presented earlier. The result is that this model does not miss any deadline, nor have any dead action (i.e. dead code).

```
Generating ktz ...
# net noname, 15 places, 11 transitions                              #
# bounded, not live, possibly reversible                             #
# abstraction        count      props      psets       dead       live #
#      states           27         15         12          0         27 #
# transitions           34         10         11          3          8 #

Begin properties checking ...
**************************
*Deadline Misses Checking*
**************************
Loading graph behavior ... DONE
Is there any deadline miss in the system ?          NO
Does task simple.T1 miss its deadline ?          NO
Does task simple.T2 miss its deadline ?          NO
*******************
*Deadcode Checking*
*******************
Loading graph behavior ... DONE
For all possible execution, will the system execute action simple.T1.a1 ?   YES
For all possible execution, will the system execute action simple.T2.a1 ?   YES
Loading graph behavior ... DONE
Is live simple.T1.a1 ?    TRUE
Is live simple.T2.a1 ?    TRUE
```

TABLE 1: Simple example's POLA output

4.1. Issues

We presented here a very simple example whose Petri net translation is compact, but this not always the case: a model can be very complex and by that we mean that the model can have numerous transitions and places. Complexity of the model comes from major impact features: period/offset which are not interval restricted to a point, allocations, and various minor ones, which we will not list here. The features possessing a major impact on model complexity are explosive according to the number of tasks inside a system. The more the tasks are present, the more complex the model will be.

Another important issue is about the automation of the approach. Even though a lot of practical cases will be efficiently handled automatically, there remains some cases for which this automation will not be possible. Indeed, there are two main issues: *state space explosion*, when the model is so complex that the memory space cannot hold every possible behavior; and *undecidability*, a theoretical limit implying that the model cannot be checked using a general algorithm: it is not generally possible to know whether a time Petri net is bounded whenever its underlying net is unbounded; extending time Petri nets with stopwatches introduces even “more” undecidability: when the underlying net is bounded, the analysis remains undecidable.

One practical problem linked to decidability issues, is that, as we just recalled, boundedness for time Petri net is not decidable. This means that the TINA explorer can never finish its exploration. This is the case when there is a deadline miss but nothing has been added to the POLA specification to prevent the resulting network to be unbounded. Indeed, from the translation you can remark that, on every period event, a token is stacked in the *released* place, but, if the task has not finished its computation yet, then there will be more than one token in the *released* place. For now, it is up to the user to handle this special case, or to check whether the net is bounded by k (k fixed by the user), which is checkable with TINA. Automatically bounding the model can be done in various but incompatible ways. Because of this, we leave this task for future works.

4.2. Experimentations

In this section we present some results obtained with POLA on the ARINC 653 and OSEK examples.

First of all, we will use the OSEK model presented earlier and some of its variants.

- **Osek**: original example model.
- **Osek B**: Like original version, but task T1's period equals 60.
- **Osek C**: Like original version, but task T1's period is reduced to 30 and its deadline to 6. Task T2's deadline is decreased to 17, which means that only T1, T4, and first action of T3 can happen before its execution. Task T3's deadline is decreased to 25. Finally a fourth task is created that belongs the second group, whose capacity is 1 and both period and deadline is 20.
- **Osek D**: Add of a fifth task, without modifying anything else. This task is the highest prioritary task (and belongs to the first group), its period is equal to 15, its capacity and deadline are equal to 1. This example shows that if meta-period is a complexity factor, the number of tasks is too.

In version B, we modified the original model in such a way that its meta-period is doubled. Version C and Version B differs from the number of tasks and in version C, deadlines are tightened (at worst case execution time). In version D, we added yet another task without modifying the meta-period.

The table data are, from left to right, the number of place (pl.) and transitions (tr.) of the generated Petri net, then the number of states and arcs of the behavior graph obtained using Tina. Then the total analysis time of the behavior graph generation added to the logical formula checking's time. The last parameter states the meta-period of the system. The meta-period is the minimal time that must elapse in the system until it returns to its initial configuration. This is then the minimal analysis time period. Here are the results obtained for OSEK:

	nb pl.	nb tr.	nb states	nb arcs	global time	meta period
OSEK	28	25	61,105	62,384	20 s	219,511
OSEK B	29	26	84,934	86,368	28 s	424,860
OSEK B	36	37	211,426	225,486	1m. 50s.	424,860
OSEK C	42	45	365,513	428,463	5m. 25s.	424,860

This example shows that meta-period is a complexity factor and the growing number of tasks induces even more complexity.

Some modification are brought to the ARINC 653 system so that we can see the impact of the different characteristics on the analysis time length and memory consumption. The latter was not directly checked, but it directly follows from the size of the state space. Here are the modifications done to the model:

- **Arinc 653**: original example model.
- **Arinc 653 A**: Partition 1 duration is 1234 time unit and partition 2 is now 6789. Periods of partition 1 and partition 2 are equal to the partitions duration. Partition 1 task 1 capacity and deadline is 555 and task 2 capacity and deadline are set to 666. Partition 2 task 1 capacity and deadline are set to 2222 and task 2 capacity and deadline are set to 3333.

- **Arinc 653 B**: partition 2 is the same as in version A. Partition 1 duration, tasks period and deadlines are all set to 31. Task 1 capacity is now 5 and task 2 capacity is 6.
- **Arinc 653 C**: Everything is like version A, except Partition 1 tasks are modified. Task 1 period and deadline are set to 23, and its capacity to 5. Task 2 period and deadline are set to 31 and its capacity to 6.
- **Arinc 653 D**: Four partitions lasting respectively 50, 100, 160 and 410 time units after their start. In each of them, four identical tasks. The (Capacity,Period) couple for each of them is respectively (5,50), (15,100), (35,160) and (50,410) and Deadline = Period.

In version A, partition duration are greatly increased by a random number. Because tasks period inside their partitions are equal to the partition duration, the meta-period is the sum of the partitions' length. The tasks capacity is also changed to remain in the same order of magnitude as the period increase. In version B, partition 1' time length is drastically reduced, along with its tasks. This is to check the influence of having different order of magnitude. In version C, partition 1 is modified in such a way that its tasks have no more the same period as the partition. This implies that the meta-period of partition 1 is greatly raised (least common multiple = 879,842), implying an even bigger meta-period of the overall system: the value of the partition 1's meta-period does imply that partition 1 is ran 713 times, which in turn means that partition 2 is also executed 713 times before a new meta-period, thus the result of 5,720,399. In version D, 2 more partitions are added, implying a more complex scheduler. The meta-period is kept voluntary low.

	nb pl.	nb tr.	nb states	nb arcs	global time	meta period
ARINC 653	31	24	129	295	$<$ 1 second	100
ARINC 653 A	31	24	141	201	$<$ 1 second	8,023
ARINC 653 B	31	24	141	201	$<$ 1 second	6,820
ARINC 653 C	31	24	1,412,262	1,449,901	8 min. 34 s.	5,720,399
ARINC 653 D	109	84	3,648,001	5,888,004	4h 05 min	720

The meta-period value possesses a real impact on analysis cost. However, this is not the only one: even if meta-period is very low in D version, it was hard to check this model. This is because of the tasks number inducing lots of cases for the scheduler (this is noticable by the bigger size of the generated Petri net).

5. RELATED TOOLS

The tool list presented in this section is not meant to be exhaustive, but these tools are well known and usefull for the real-time community. Therefore, this is necessary to compare POLA to them.

The tool named TIMES [5] is the nearest to our work. Indeed, like POLA it uses model checking to verify properties on the input model. The mathematical task model is roughly the same (it uses capacity, period, deadline and priority) and the language used can be considered high level enough to be a domain specific language (tasks are explicitly defined). Unlike POLA, it does not allow to specify resources; nor can it specify behavior of a given task. In fact, the TIMES task model allows to model tasks with their characteristics but it is left to the user the hard work of implementing more subtle mechanisms. As this tool is based on timed automata, this should be considered possible, but is not something that can be done by a non specialist. The policy is also a divergence point, as it is written out of the model and hardwired in TIMES, instead of inside of the model for POLA. The policies used in TIMES are those presented as an example in section 2.

As for the analytical methods' domain, we wanted to emphasize on Cheddar [6] which is a well known tool that have the ability to compute sufficient conditions for schedulability, or to simulate task systems whenever the former conditions are not sufficient. As simulations cannot guarantee validity of the schedulability in the general case, this can only be used to strengthen the user's intuitions. This kind of tool, in our opinion, can be used as a complement to our: if the sufficient conditions are fulfilled, the answer will be given in no time, thus aleviating the time and resource consuming exact techniques required by our approach; and when the conditions are not, POLA is required to check a more precise schedulability analysis on the system. Unfortunately analytical techniques are not applicable on any system type and this is the case for the rather complex resource manipulation presented in the OSEK example, which is not specifiable using Cheddar.

6. CONCLUSIONS

We presented a new tool which brings new possibilities to the real-time system verification community. Although not perfect (no dynamic policies, for example), it allows, practically speaking, an even wider range of systems to be checked. Moreover, for the first time, users are allowed to integrate sheduler details inside of the model explicitly, without being required to implement it. It also adopts a top down approach rarely seen (if ever) in Domain Specific Language approaches, which allows the use of a high level and easy to use language, and the use of a lower level language used to alleviate some unhandled features (because of its very nature of DSL), and this in an integrated framework.

Because it was not the theme of this paper, the semantics and the translation have not been presented in this paper. We indeed know that the translation efficiency is crucial, as the more the translated Petri net's behavior will be compact the easier the analysis will be. In fact, during this work, it has been observed that another extension is much adapted than time Petri net with priority alone. This extension is being matured and will be the object of a future publication, thus furnishing a sound way to give the language semantics.

Although it is theoretically possible through Petri nets alone, and as a consequence in the behavior part of POLA, we think that it would be interesting for POLA to be capable of handling *naturally* priority ceiling protocols, or more generally priority inheritances protocols. Indeed, within the purely declarative part of POLA it is not possible to specify such a range of protocols.

As future works, dynamic policies are to be integrated since standard like OSEK, or ARINC 653 uses such policies (mainly FIFO and EDF). However, we think that the translation of the general pattern "**min** or **max** of any linear combination of the dynamic clocks p,d or c" will not be achievable.

REFERENCES

[1] Airlines Electronic Engineering Committee, *ARINC Specification 653-1*, October 2003, Aeronautical Radio, Inc., Annapolis, MD

[2] B. Berthomieu, F. Peres, F. Vernadat, *Bridging the gap between bounded time Petri nets and timed automata.*, FORMATS 2006, LNCS 4202

[3] OSEK/VDX Group. *Operating System Specification 2.2.3*, Feb. 2005, www.osek-vdx.org

[4] B. Berthomieu, P.-O. Ribet, and F. Vernadat, *"The tool TINA – construction of abstract state spaces for Petri nets and time Petri nets"* Int. J. of Production Research, vol. 42, no. 14, pp. 2741-2756, 2004

[5] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and Wang Yi, *TIMES - A Tool for Modelling and Implementation of Embedded Systems*, TACAS 2002, LNCS 2280

[6] F. Singhoff, J. Legrand, L. Nana and L. Marcé, *Cheddar : a Flexible Real-time Scheduling Framework* ACM SIGAda Ada Letters, vol.24, number 4, p. 1-8, 2004

[7] P.M. Merlin and D.J. Farber, *"Recoverability of communication protocols: Implications of a theoretical study."* IEEE Tr. Comm., vol. 24, no. 9, pp. 1036-1043, 1976

[8] J. Migge, *L'ordonnancement sous contraintes temps-réel: un modèle à base de trajectoires*, PhD thesis, 1999

[9] S. Christensen and N. D. Hansen, *Coloured petri nets extended with place capacities, test arcs and inhibitor arcs*,ATPN, volume 691 of LNCS, 1993, p. 186–205

[10] W. Vogler, *Efficiency of asynchronous systems and read arcs in petri nets*, ICALP'97, LNCS 1256, p. 538-548, 1997

[11] T. Agerwala and M. Flynn, *Comments on capabilities, limitations and "correctness" of Petri nets*, SIGARCH Comput. Archit. News, vol. 2,num. 4, 1973,p. 81–86, ACM

[12] B. Berthomieu, D. Lime, O.H. Roux, F. Vernadat, *Problèmes d'accessibilité et espaces d'états abstraits des réseaux de Petri temporels à chronomètres*, MSR'2005, JESA, Vol. 39