Plan validation via Petri Nets in the Real-Time Performers
Java framework

Andrea Trentini
Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano
Via Comelico 39, Milano, Italy
trentini@dico.unimi.it

Daniela Micucci
Dipartimento di Informatica, Sistemistica e Comunicazione,
Università degli Studi di Milano-Bicocca
Via Bicocca degli Arcimboldi 8, Milano, Italy
micucci@disco.unimib.it

ABSTRACT
The Real-Time Performers (RTP) architecture is a framework to design distributed soft real-time systems based on timed plans. Timed plans - they define system workflow - contain actions to be executed by distributed components at specified times. An RTP system is controlled by a strategist that may modify plans to adapt system behavior to environment conditions. When a timed plan is changed it may be checked for coherency. This paper presents techniques used in RTP to check timed plans validity through Petri Nets (PNs). These techniques are based on the definition of mappings from RTP topology and plans to PNs. Once PNs are generated, structural and behavioural properties can be calculated and mapped back to the RTP system.

General Terms
Performance, design, verification

Keywords
Validation, soft real-time, planning

1. INTRODUCTION
A time-sensitive system needs an underlying software architecture capable of capturing temporal aspects of the system itself. This kind of system needs to execute various activities with dynamic and inter-dependent temporal requirements. Moreover it may happen that such kind of system needs to dynamically change activities temporal requirements.

Real-Time Performers [8] (RTP) is an architectural reflection-based framework to monitor and control time related behavioural aspects of systems built upon it. RTP is based on the following concepts: the system runs temporally planned actions; actions are planned for execution by placing them in timelines (defining overall system behaviour); actions are executed by distributed components connected one another; a timeline is “ticked” by a virtual clock; a virtual clock may be tuned to modify execution rates of actions it controls; a strategist may control the system by tuning virtual clocks an by changing timelines content (i.e., adding, removing, and modifying actions). A system timeline is also called “timed plan”; it models overall system workflow in terms of: What to do (action to be performed); When to do (execution time interval); Who (which component has to execute a command).

An RTP timed plan may be runtime modified to adapt a running system to new environment conditions. Changes are in the form of planned interval modifications, insert/removal of requests, etc. Any applied change must keep the system in a safe state. The term safe has more than one meaning in time-sensitive systems, since both functional and non-functional requirements must be fulfilled. Briefly, functional requirements are fulfilled when a system does what is expected, whereas non-functional ones when it does what is expected on time. In RTP, functional requirements depend on system topology, i.e., on the network of components, whereas non-functional ones are tweaked by carefully plan requests on a timeline. Thus, an important aspect of RTP system management is plan tweaking, but every change must keep system workflow inside the set of valid workflows. So we need a mechanism to decide if a plan change “breaks” system behavior. We devised a validation technique based on Petri Nets (PN) mappings.

2. RTP ARCHITECTURE
To explicitly deal with non-functional requirements, an application program needs abstractions modeling these requirements, so that they can be directly observed and controlled. In this view, a time-sensitive system must explicitly deal with all the aspects concerning its temporal behaviour. RTP tries to capture and model temporal abstractions into a reference software architecture for the design of modular, distributed, and real-time systems.

Since RTP may help in building dynamic and self-adaptive systems, it is based upon reflective mechanisms. We recall that computational reflection is defined as the activity performed by an agent when doing computations about itself [13]. Whereas, architectural reflection [5] is the computation performed by a system about its own internal architecture. We are interested in the second kind of reflection: it reifies architectural features as meta-objects that the application
exploits at run-time to observe and control concrete architecture features. In this view, the use of architectural reflection helps bringing visibility over the computation performed by system components at the programming level. In RTP we exploit, as well architectural reflection, a new kind of reflection termed temporal reflection [7] enabling the system to reify, observe, and control its own temporal behaviour. RTP raises (by reflection) at the application programming level: strategies definition (action choice on behalf of events); timing tuning issues (speed-up/slow-down); component behaviour definition; system topology definition (adding/removing connected components).

According to the definitions in [12] and in [6], RTP software architecture is defined in terms of components and connectors: an RTP-based system is made up by computational components; computational components exchange information via alignment components (connectors); both computational and alignment components are activated and controlled by supervising components.

In this view, RTP defines the following components: Performer, Projector, and Strategist. Performer is the entity designed to perform data elaboration, Projector is the entity designed to project (distribute) data between Performers, and Strategist is the entity designed to use plans or stratagems to achieve system goals by planning activations. The identification of these well-distinguished components emphasises the “separation of concerns” between information processing, information alignment, and their activation.

A Performer is a passive entity that performs its activities only upon triggered commands. It is unaware about both the surrounding environment (in terms of topology) and system behaviour. It acquires and provides information through its visible variables, arranged in two subsets: exported visible variables, that are writeable by the Performer and readable from outside (i.e., which can be assigned from inside and observed from outside); imported visible variables, that are readable by the Performer and writeable from outside (i.e., which can be observed from inside and assigned from outside).

This formal separation is important because it allows a Performer not to be aware of information source and target respectively: a Performer may only publish data on exported visible variables and read data from imported visible variables. Thus, the set of visible variables represents the only “port” to the outside world of the Performer: communication between the Performer and the rest of the system must always pass through this set. This must be regarded as an advantage because it makes Performers completely unaware of their environment. This means that such a component may be reused in different topologies. A Performer may require data produced by other Performers to carry out its activity. Moreover, a Performer may produce data useful to other Performers at the end of its computation. In RTP, information exchange between Performers is possible only if data is projected between Performers. In other words, a Performer can observe exported visible variables of another Performer only if its exported visible variables are projected into its imported visible variables by some Projector.

Strategies definition implies the definition of planned actions. An action specifies the activity that may be executed by a Performer or a Projector. An action must be fulfilled within temporal constraints, i.e., the time interval in which it should be executed. The Strategist plans system temporal behaviour by adding temporal actions on a timeline.

A timeline models overall system temporal behaviour. It is modelled as a monotonic sequence of time instants characterised by the non-decreasing value of the current time. Current time splits a timeline into past and future. When an action is performed, it is moved in the past timeline and it is filled with additional information about when (actual time) it has been performed.

In RTP concrete architecture, a Strategist defines system behaviour by manipulating TimedTraces. A TimedTrace defines system behaviour in terms of a partially ordered set of TimedRequests. A TimedRequest models an action; it is defined as (recipient, command_to_perform, planned_time), where the recipient is a Performer or a Projector, command_to_perform is the command that the recipient has to execute, and planned_time is the time interval in which the command must be delivered and executed (planned).

A TimedTrace is aware of current time by means of a VirtualClock, an active entity that is in charge of advancing current time. When its next() method is invoked, it returns a TimedRequest whose planned TimeInterval is valid with respect to current time.

What is important is that this architecture model allows dynamic strategy changes during system lifecycle. In fact, since Strategies can be planned at application domain level, system behaviour may be easily changed to respond to specific requirements by changing, adding, and removing Requests inside a Trace. Moreover, a Strategist may change, at the application level, system dynamics in terms of timing issues, e.g., it may accelerate or decelerate some activities by tuning virtual clocks (Figure 1).

### 3. MAPPING RTP ON PETRI NETS

The core of our validation technique is the ability to map an RTP system onto PNs. RTP architecture is quite naturally PN-mappable. After creation of PN maps (every RTP aspect is mapped onto different PNs) it is possible to use currently available tools (see [1] for a fairly complete list) to check PN properties that can be related to expected features (such as deadlock-freedom) of an RTP system. Before describing our validation technique, we should now explain how to generate a PN from an RTP instantiated topology. We first recall some basic concepts concerning PNs.

A simple Petri Net [10], also called Basic Net, is a tuple: (P, T, Pre, Post) where: P is a set of Places (symbol: circle); T is a set of Transitions (symbol: square); Pre: P x T → N (defines input arcs); Post: T x P → N (defines output arcs). With P ∩ T = ∅ and P ∪ T ≠ ∅.
A transition is enabled when all of its input places contain at least one token. When a transition triggers, it moves tokens from input places to output places (the number of moved tokens depends on the type of transition/net).

A “marking” of a net is the distribution of tokens at any given time, usually described by a vector \( m : P \rightarrow N \) (every \( n \) in \( m \) represents the number of tokens in a place). The marking of a net is very important since it determines the current state of a net and many properties are related to PN states (such as the liveness [2]). Given a marking \( m_0 \), when the transition \( t \) triggers, it is possible to calculate the new marking by using the \( m = m_0 + C_t \) formula, where \( C_t \) is the characteristic vector of \( T \) (value =1 only for the triggered transition, the rest is =0). Moreover, \( m = m_0 + C\sigma \) is the state equation formula, where \( \sigma \) is the trigger vector: it contains transition trigger occurrences (order info is lost). The marking of \( \omega \) because the symbol means “a number of tokens”. An example of a tool-generated graph is shown below (order info is lost).

The coverage graph is a concise representation of states of a net, every node is labelled by a marking in the form of \( (n_1, n_2, ..., n_k) \) where \( n_j \) is the number of tokens in place \( P_j \). When a marking contains the \( \omega \) symbol instead of a number, it no longer represents a single marking but a set of them, because the \( \omega \) symbol means “a number of tokens”. An example of a tool-generated graph is shown below (order info is lost).

An RTP topology is mapped by: RTP Performer/Projector \( \Leftrightarrow \) PN Transition; RTP Visible \( \Leftrightarrow \) PN Place; RTP Data (moved around) \( \Leftrightarrow \) PN Tokens.

Even if RTP data is typed the generated PN will drop details about data type since our only goal is to reason about data flow. The resulting PN topology is very similar to the original RTP topology. The only notable differences are that transitions (Performers) do not perform computation, they just pass data around, and tokens represent data but they do not have attributes.

The idea behind this mapping is that since a Performer performs its functions using available data in input Visible and puts results in output Visible, by dropping the notions of computation (thus yielding pure data transfer) and of data type (thus yielding pure data tokens) an RTP topology becomes a PN.

To be more formal let us have a generic Performer (it may also be a Projector, since it extends a Performer) \( P_j \) with its \( V_l[1-N] \) input Visible and its \( V_0[1-M] \) output Visible. PN generation is detailed as following: for every Performer \( P_j \) create a Transition \( T_{P_j} \); for every \( P_j \) input Visible create/link an input place for \( T_{P_j} \), labelled \( P_{P_jV_l[1-N]} \); for every \( P_j \) output Visible create/link an output place for \( T_{P_j} \), labelled \( P_{P_jV_0[1-M]} \).

Thus, obtaining the PN shown in Figure 2. Decision between “create” and “link” in the above procedure is taken depending on the type of Performer under analysis: in case of a Projector there is no need to create new places since a Projector links output and input Visible from other Performers. So that, for example, a situation like the following: \( P_1 \), Performer, with one output Visible \( P_1V_1 \); \( P_2 \), Projector, with one input Visible \( P_2V_1 (=P_1V_1) \) and one output Visible \( P_2V_0 (=P_3V_1) \); \( P_3 \), Performer, with one \( P_3V_1 \), will be mapped with the following PN: \( T_1 \), with one output place \( P_{P_1V_1} \); \( T_2 \), with one input place \( P_{P_2V_1 (=P_1V_1)} \) and one output place \( P_{P_2V_0 (=P_3V_1)} \); \( T_3 \), with one input place \( P_{P_3V_1} \).

Figure 3 shows an example RTP topology: \( W_1 \) and \( W_2 \) are webcams capturing still images; \( C_1 \) is a format converter (we assume that \( W_1 \) and \( W_2 \) generate different formats); \( S_1 \) is some form of storage; \( D_1 \) is a performer that compares two frames; \( P_1 \) is a projector that creates frame copies for the storage; Coloured circles before \( W_1 \) and \( W_2 \) represent input from system environment (they will be marked as “initial places”); Coloured circles after \( D_1 \) and \( S_1 \) represent output to system environment (they will be marked as “final places”).

In RTP, runtime behaviour of any system topology must be controlled by timed plans. A timed plan is a set of timed
(thus ordered) requests that trigger\(^1\) Performers by sending them commands. PN mapping of controlled transitions is shown in Figure 4: a new input place \(T_{\text{ex}}\) (Transition enabler) is added to every Transition in the generated PN topology. These places inhibit firing of Transitions unless they are filled with some token, of course these special places will be filled by a PN mapped plan. Applying this method, the final generated PN is shown in Figure 5.

A plan is represented with a simple PN by dropping time information and yielding an ordered sequence of "activations". The actual procedure is just an iteration performed on the timed trace. Given a generic timed trace with requests of the form \((pB-pE, aB-aE, \text{performer}, \text{command})\) with \(pB = \text{planned Begin}, pE = \text{planned End}, aB = \text{actual Begin}, aE = \text{actual End}.\) The \(pB\) value is used to order requests, and the \text{performer} value to link plan to topology.

### 4. VALIDATION TECHNIQUES

By "validation" we mean the ability to calculate properties of an instance of RTP (topology+plan) by computing properties on mapped PNs representing aspects of the system under analysis. PN properties are computable using known algorithms [9]. Since many tools [1] implement the gcc compiler. Validation checks may be done both offline (e.g., topology design verification) and online (plan validation). Of course, online verification actions should be taken only in case of low computational and delay impacts, i.e., when validation is significantly faster than action/plan execution (every example shown below was analysed offline).

What is "validation"? In general, we can say that a topology or a plan are valid if they let the system reach some final configuration. Final configurations are decided at design time. So, before applying PN analysis, there is only one thing left to do: the definition of initial, final markings and error conditions for every PN mapped aspect of RTP. Plan validation, only two places are labelled: \(P_{\text{bp}}\) (= Place "begin plan") and \(P_{\text{ep}}\) (= Place "end plan"). Initial marking is \((P_{\text{bp}}=1, 0, 0, ..., P_{\text{ep}}=0)\), final marking \((P_{\text{bp}}=0, 0, 0, ..., P_{\text{ep}}=1)\). If at the end of a PN run we end up with a marking such as \((P_{\text{bp}}=0, 0, ..., 1, ..., 0, ..., P_{\text{ep}}=1)\) it means that some triggering has no been consumed and it must be treated as an error. Topology validation, initial marking is \((P_{\text{bp}}\geq1, P_{\text{b2}}\geq1, ..., P_{x}=0, ..., P_{e1}=0, P_{e2}=0)\), i.e., every initial place is marked with one token, the others with 0 tokens. Final marking is \((P_{b1}=0, P_{b2}=0, ..., P_{x}=0, ..., P_{e1}\geq1, P_{e2}\geq1)\), i.e., every final place is marked with one or more tokens, the others with 0 tokens. Any other configuration must be treated as an error since it means that some data has not been consumed completely.

The procedure to check Topology validity can be sketched as the following: 1) build PN topology map; 2) set initial marking; 3) compute coverage graph; 4) check that terminal nodes of the graph are of the type \((P_{b1}=0, ..., P_{x}=0, P_{e1}\geq1, ..., P_{eN}\geq1)\), see for example in Figure 6, the small ellipse emphasising a final marking that satisfies this check (p6 and p9 are final places, this tool does not allow relabelling of PN nodes).

Topology check "per se" is useful because if a topology is not valid without a hooked plan, there will be no plan that can make it valid because a plan is only a trigger for "natural" net behaviour.

There is almost no sense in checking the validity of an unhooke standalone Plan since it will inevitably reach \((P_{b1}=0, ..., 1, ..., P_{ep}=1)\). The procedure to check the validity of a hooked Plan (Figure 5) can be sketched as the following: 1) build PN topology map; 2) add "hook" places; 3) build PN plan map; 4) hook plan to topology; 5) set initial marking; 6) compute coverage graph; 7) check that terminal nodes of the graph are of the type \((P_{b1}=0, ..., P_{x}=0, P_{e1}\geq1, ..., P_{eN}\geq1)\). Figure 7 shows an example system with a wrong plan, in this case a final configuration has been reached, but not every token in the net has been consumed: place p8 still contains 1 token but it is not in the set of final places. Figure 8 shows the corrected plan (an activation was added) with the right result.

Finally a remark on the limit of this technique: our map-
Figure 8: (after plan correction) Finished AND exhausted: right plan!

Figure 9: Plan validation not always possible

ping assumes that every Performer uses (needs) data from every input Visible and outputs results on every output Visible. This assumption is not always valid and it could break the mapping. In case of such system, the mapping procedure must be adapted. A Performer is able to perform various actions using data from subsets of Visibles (Figure 9 top) so that a situation like the one shown in Figure 9 and a plan without action detail info avoids invalidation (PlanB is not valid but it cannot be spotted). We need more information at architectural level, a more detailed reification is needed. Every Performer should advertise its needs (input/output) on a “per action” basis. With this kind of information available, topology mapping can be “exploded” by generating a PN with “action” granularity.

5. CONCLUSIONS

We presented a technique to validate topology and timed plan coherency for our Real-Time Performers architecture. Timed plans define system behaviour in terms of computation activations planned in time. In RTP systems, behaviour tuning can be done by rearranging requests on the future part of a plan. To validate both topology and plan we devised a technique based on mapping an RTP system onto Petri Nets. The generated PN can be fed into available tools to compute reachability of a desired (and final) state. This technique is currently semi-automatic and needs human intervention so it is only usable in systems where time quantums are human comparable. In the future we plan to fully integrate PN tools into RTP to achievea fully automated validation process at runtime.

6. REFERENCES