UNIVERSITY OF OSLO
Department of Informatics

Implementing timed extensions of Petri nets in Real-Time Maude

Research Report No. 408

Krzysztof Michał Majewski

ISBN 82-7368-370-2
ISSN 0806-3036

August 10, 2011

Abstract

We study three timed extensions of Petri nets. We demonstrate methods for transforming these types of Petri nets into Real-Time Maude programs. This lays the groundwork for a timed Petri net analysis tool in which these transformations will be automated. The programs we present elucidate the differences between the timed Petri net variants. This work provides further evidence that Real-Time Maude can be used to naturally express different models of concurrent real-time systems.

1 Introduction

This report describes ongoing work on timed extensions of Petri nets. We have implemented several dominant timed Petri net (henceforth *TPN*) variants in Real-Time Maude[6] (henceforth *RTMaude*), a timed extension of Maude[16]. The Maude system is a declarative language and accompanying toolkit (model checker, etc.) based on rewriting logic[11].

1.1 Contributions

Our contributions are threefold:

1. We express the semantics of three timed Petri net variants from the literature (two popular and one less popular) as real-time rewrite theories[8]. More specifically, these are RTMaude programs which are executable and can be analyzed by the tools provided by RTMaude. We show how one might go about proving the correctness of these programs.
2. We lay the groundwork for a RTMaude-based tool for the analysis of timed Petri nets. Our programs demonstrate strategies for the transformation of TPNs into RTMaude code, with a view to automating this transformation.
3. We lend further support to the claim, advanced in e.g. [8], that RTMaude can be used to naturally various models of concurrent real-time systems.

This report describes work in progress. It concludes the first chapter of this work: the experimental phase, consisting of RTMaude hacking and a review of the literature.

1.2 Petri nets

The term *Petri nets* designates a family of models used in the study of concurrent systems. For an introduction to Petri nets, see e.g. [13].

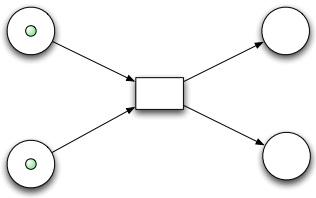

Figure 1: A Petri net with four places, one transition, and two tokens

Briefly, a Petri net is a graph in which some nodes are called *places* and others *transitions*. Directed edges or *arcs* connect a place to a transition (*P-T arcs*) or a transition to a place (*T-P arcs*). The *preset* of a transition is the set of places adjacent to it along P-T arcs. Analogously, the set of places adjacent to a transition along T-P arcs is called the *postset*.

Places may contain *tokens*. An assignment of tokens to places is called a *marking*. A transition consumes tokens from its *preset*. It then produces some tokens, which appear in the postset. When all the required tokens are available for consumption, the transition is said to be *enabled*. The consumption and subsequent production of tokens by a transition is called a *firing*.

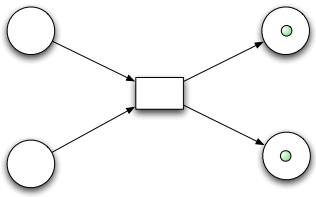

Figure 2: The transition has fired

The Petri net idiom is sufficiently general to be suitable for modeling a wide range of problems. For example, one can imagine the tokens to represent resources which are consumed and produced by some actions (the transitions). Since two or more transitions may fire concurrently, Petri nets lend themselves naturally to the modeling of concurrent systems.

Many extensions of the basic Petri net paradigm exist, for example *Colored* Petri nets[5] in which the tokens may be used to represent structured data. In what follows, we consider one specific family of extensions: those which add time constraints to the basic model.

1.3 Timed extensions

Various timed extensions of Petri nets have been proposed for modeling real-time systems. Most of these variants can be seen as deriving from one of the two earliest proposals, namely that of Ramchandani[12] and that of Merlin and Farber[10]. The former considers time constraints which represent the *duration* of a firing, whereas the latter constrain *when* a transition may fire.

The many timed extensions which have since sprung up may sometimes be hard to distinguish at first, but on closer inspection reveal fundamental differences. In particular, as shown e.g. by Cerone et al.[3] and by Boyer and Roux[2], some of these extensions are more expressive than others. Moreover, some pairs of extensions are incomparable in terms of expressiveness. In cases where a more expressive extension can be used to express a less expressive one, the transformation required may be too difficult for a human, and computationally expensive for a machine.

Some of the parameters on which the various timed extensions may differ are:

- Which elements of the net (places, transitions, or arcs) are labelled with time constraints;
- Whether the time constraint is a scalar (e.g. a delay), an interval, or a tuple;
- Whether the time constraint denotes the duration of a firing, or the time(s) at which a firing is allowed to occur ("how long" vs. "when");
- Whether the time constraints themselves are required to be *static* (e.g. a fixed time interval), or whether they can be a function of the current state of the net,
- Whether the number of tokens in a place is bounded (in particular, bounded by 1);
- Whether the time constraints are strict, or whether a transition whose constraints are satisfied may fail to fire.

In the literature, the terminology used by the various timed Petri net proposals can be confusing. For example, Ramchandani[12] called his seminal model "*Timed* Petri nets", whereas Merlin and Farber[10] use the term "*Time* Petri nets" (note the missing *d*). Hanisch[4] labels P-T arcs with time intervals constraining when the transition may fire and calls this "arc-timed Petri nets". Ruiz[14] describes a similar model, based on that of Bolognesi et al.[1], and calls it "timed-arc Petri nets".

The timed Petri net variants have certain features in common: they are based on the same underlying untimed Petri net concept, to which they all add time constraints. However, they differ in subtle but fundamental ways, some of which were listed above. Cerone et al.[3] give a classification of some common variants along these lines. That paper also presents an operational semantics which can be used to express all of the timed Petri net variants they discuss.

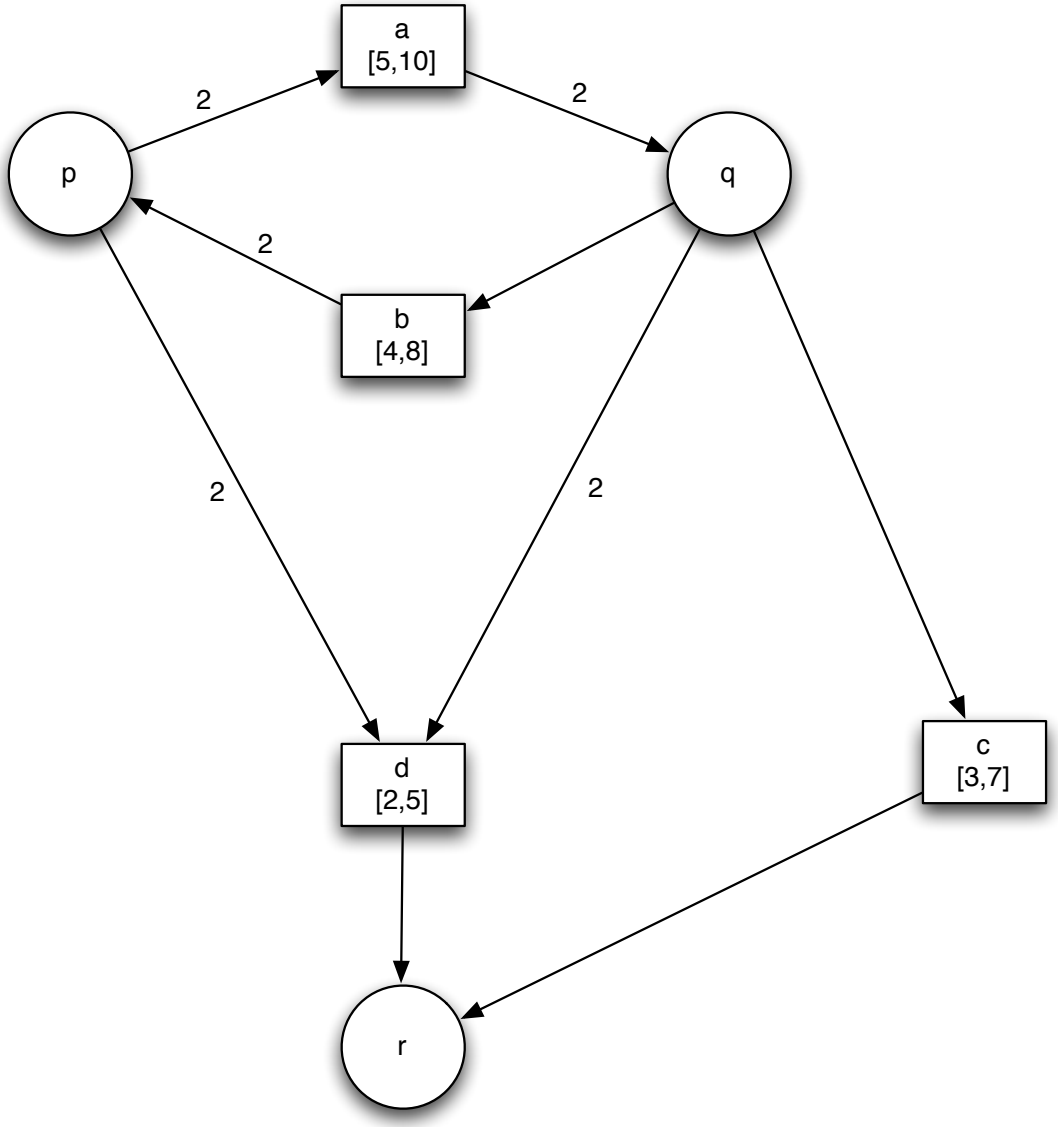

Figure 3: The transition-TPN example from Appendix C. A label on an arc indicates the arc's *weight*. For example, the arc from p to a has weight 2. This means that transition a consumes 2 tokens from place p. Arcs of weight 1 are not labelled. The interval on a transition constrains when the transition may fire, as described in Section 2.4.1.

2 Three timed Petri net variants

The following subsections present three TPN variants. The first, *interval timed Petri nets*, has been previously studied in the context of RTMaude[8]. The remaining two, *timed-arc Petri nets* and *transition time Petri nets*, are popular variants found in the literature. For each variant, we describe its semantics as established by the literature, and discuss its implementation in RTMaude.

2.1 Preliminaries

A Maude program partitions the problem to be implemented into a set of *equations* and a set of *rewrite rules*. The application of the rewrite rules models how a system's state evolves. The reduction of equations takes care of everything else. That is, the equations express those computations which are necessary for the functioning of the model, but which we do not wish to have contribute to the state space which will be explored in subsequent analyses of the model. RTMaude adds to Maude the concept of *tick* rules, which express a state transition in which time elapses. The traditional Maude rewrite rules are then called *instantaneous* rewrite rules.

Thus, in order to implement a TPN variant in RTMaude, we must partition the dynamics of the TPN into *timed* state transitions and *instantaneous* state transitions. The timed and instantaneous transitions correspond to, respectively, tick rules and instantaneous rules. This partitioning is not obvious, and its correctness remains to be proved. An incomplete proof sketch is given in Section 2.4.3. One implication of such a partitioning is that the following two assumptions (enforced by RTMaude) hold:

1. No time elapses while an instantaneous transition is taking place;
2. No instantaneous transitions take place while time is elapsing.

We now give some definitions that will be shared by the following subsections.

A *marking* is, as before, an assignment of tokens to places. However, in some timed semantics, a token may have a *clock* associated with it. We overload the term *marking* to denote not only the locations of the tokens, but also the states of their clocks, should they have any. (In that case, we can also use the term *timed* marking.)

Given a marking M and some $M' \subseteq M$, we call M' a *consumable* with respect to transition T iff a firing of T can consume M'[1]. If M is a timed marking, the "consumability" of M' may depend on the values of its clocks (see the individual semantics, below, for details). Note that a transition may have more than one consumable at any given moment.

We use the term *deadline* to denote the latest possible time that an event may occur. For example, the *deadline* of a transition is equivalent to its latest firing time. Deadlines are relative to the current time: a deadline of τ means that, starting now, at most τ time units may elapse before the event happens.

A Petri net is *alive* when, given the current marking, some transition can fire, now or in the future.

A marking M is *reachable* from the current marking if there exists some (timed) sequence of transition firings which leads to M.

2.2 Interval timed Petri nets

2.2.1 Semantics

Our starting point was the implementation of the timed Petri net semantics described in [8]. The authors of that paper give a pseudocode implementation of timed Petri nets based on an earlier version of RTMaude. The earlier version differs from the current version in at least one way: it allows rewrite rules to be marked `lazy` or `eager`, which changes their semantics in the expected way: `eager` rules must be applied before `lazy` rules. (This feature was subsequently considered unnecessary and has since been abandoned.) To our knowledge, the pseudocode given in [8] had never been implemented as an executable RTMaude program[7].

[1] The noun becomes an adjective in the expected way: the *consumables* are exactly those sets which are *consumable*. We can abbreviate "with respect to" as "of": M' is a consumable *of* T.

The TPN semantics in [8] is based on the *interval timed colored Petri nets* of van der Aalst[19], with the *colored* part abstracted away; thus, these Petri nets are called *ITPN*s. Moreover, ITPNs allow the concurrent firing of multisets of transitions.

A key feature of this timed Petri net semantics is that the time intervals ascribed to the transitions represent, as per Ramchandani[12], the *duration* of a transition's firing. This is in contrast to the time[2] Petri net model of Merlin and Farber[10], in which the intervals represent the time *when* an enabled transition may be taken.

Another feature is that a place may hold arbitrarily many tokens. Petri nets of with this feature are called *unbounded*. Boundedness is especially relevant because of its implications on computability: in general, unbounded nets are Turing-complete.

In contrast, the A-TPN semantics given in [2] is not only bounded but *1-safe* (or just *safe*), meaning that a place may contain at most one token. Discussions of 1-safe Petri nets may conflate consumables with presets, which in some cases[3] can lead to confusion.

An ITPN transition fires as soon as it becomes enabled. When a transition fires, some consumable is consumed (note non-determinism). The firing continues for some (again, note non-determinism) amount of time within the transition's interval. When the firing terminates, the transition produces tokens into its postset.

Note that at any given moment, two or more transitions may be firing concurrently. Note also that the only "clocked" components in an ITPN are the transitions – or, more correctly, the firings (since there may be several concurrent firings of the same transition). There are no timestamps or clocks ascribed to places, arcs, or tokens. This is in contrast to the two other timed Petri net models we will describe.

2.2.2 Implementation

The implementation can be found in Appendix A. In this implementation, as in the two subsequent ones, we use the RTMaude programming paradigms described in [9]. In particular, we partition state transitions in our model into *timed* and *instantaneous* transitions. The timed transitions are implemented via a single `tick` rule. The instantaneous transitions are implemented via the instantaneous rules.

The function `delta : GlobalSystem Time -> GlobalSystem` takes the model from its current state to its state after the time given by the second argument has elapsed. This time is chosen non-deterministically when the tick rule is applied, subject to the constraints of that rule and the time sampling strategy (e.g. `set tick def 1`) specified.

The function `mte : GlobalSystem -> Time` gives the maximum time that may elapse in the application of a tick rule to the current state of the model. After this amount of time has elapsed,

This implementation (as well as the subsequent ones) consists of two Maude (timed) modules: a *generic* module and an *instance* module. The generic module contains code common to all timed Petri nets with the given semantics (in this implementation, ITPN semantics). The instance module is specific to a given input net.

A term of sort `PetriNet` is a multiset of *Locations* and *Firings*, where a Location corresponds to a single token in a single place, and a Firing corresponds to a transition being fired. Each rewrite rule in the instance module – call it a *firing rule* – corresponds to a transition (i.e. a transition node) in the ITPN. Thus, the left-hand side of the rule represents the tokens consumed by the transition. The right-hand side is the Firing, which encapsulates four terms: one multiset of Locations and three `Time` values. The multiset represents the tokens produced by the transition. The first two time values denote the interval ascribed to the transition; that is, the lower and upper bounds on the duration of the transition's firing. The third and final time value is initially zero, and represents the amount of time elapsed since this firing started. It is incremented by the amount of time that elapses in the application of the tick rule, and may not exceed the upper bound (second time value).

The predicate `stable : PetriNet -> Bool` is true exactly in those cases when no transition is enabled. By using this predicate as a condition in our tick rule, we ensure that time may not advance when a transition is enabled. Thus, transitions fire as soon as possible.

[2] As opposed to time**d**.

[3] To wit: ours.

The function `mte` takes the minimum, over all firings currently in progress, of the maximum time remaining in each firing (i.e. the difference between the fourth and third terms encapsulated by the corresponding `Firing`). Thus, a single application of the tick rule will not allow time to elapse past the point where some firing must complete.

Define a firing to be *mature* if enough time has elapsed for it to complete; that is, the fourth argument of the corresponding `Firing` meets or exceeds the second argument. A mature firing *may* (again, note non-determinism) be rewritten, by an application of the instantaneous rule `finish_firing`, to the multiset of tokens produced by the corresponding transition, as specified by the first argument to the Firing.

The application of instantaneous rules can cause the tick rule to become re-enabled. Consider a state of the system in which `mte` evaluates to 0. In this case the tick rule cannot be applied (due to the non-Zeno requirement of RTMaude). Now, suppose the system contains a mature firing. The rule `finish_firing` can now be applied. This can lead to the enabling of a transition which wasn't previously enabled. This, in turn, can result in the application of a firing rule. The firing rule introduces a new term of sort `Firing` into the system. Now the function `mte`, when evaluated on the system, will[4] yield a non-zero value, and thus the tick rule is again enabled.

2.3 Timed-arc Petri nets

Next, we considered a more popular variant, *timed-arc Petri nets*. These are also known as *arc-time Petri nets* or *A-TPN*.

2.3.1 Semantics

The next timed Petri net semantics we considered[14] differs fundamentally from the first. The time intervals are, in this model, ascribed to each arc leading from a place to a transition, rather than to the transition itself. There may be at most one arc leading from a place to a transition. Thus, when a transition fires, it consumes exactly one token from each place in its preset. However, a place may contain more than one token.

The intervals constrain *when* a transition may fire (as opposed to *how long* it takes to fire, as in the previous model) in the following way.

Each token has a location and a clock. The location is, as before, the place where the token resides. The clock measures how long the token has been in its location. We call this the *age* of the token. Initially (as when a token is produced in a location) it has age 0.

Consider a token in place P with age ν. In order for this token to be consumed by a transition T, ν must be within the time interval ascribed to the arc from P to T. Thus, for a set of tokens S to be a consumable of T, all the tokens in S must satisfy this time constraint.

As before, a transition T may fire when the current marking contains a consumable with respect to T (the transition is then said to be *fireable*). In the more popular *weak* semantics, which we treat here, a fireable transition is not obliged to fire. Indeed, a token may age until it can no longer be consumed by *any* transition. That is, given a token in place P with age ν, the value of ν may exceed the upper bounds of the time intervals on *all* the P-T arcs leaving P. If that happens, the token is called *dead*. Interestingly, the "liveness" of a token may be undecideable[14]. Thus, we do not model dead tokens. However, there are some contexts in which the liveness of a token *is* decideable. One possible optimization would be to remove dead tokens from the system in those contexts (in the interests of reducing the state space of the model).

2.3.2 Implementation

The implementation can be found in Appendix B. It follows the pattern we used for ITPNs: a generic module containing code shared by all A-TPN instances, and an instance module representing a particular instance.

However, each token has a clock, as explained above. The computation of `mte` is now more complex. We seek the maximum, over all transitions in the net, of the latest possible firing of each transition. Thus, for each transition T, we must iterate over all the consumables with respect to T.

[4] assuming a non-zero time interval on the transition

For a consumable C of T, let f be the function which determines the amount of time that must elapse before C is no longer consumable. That is, f returns the *minimum* amount of time that must elapse before *some* token in C exceeds its time constraint. Let S be a set of tokens (T again a transition). The function `mte-consumable` (defined in the instance module) takes S and T and returns the value of f if S is consumable by T, and `INF` (i.e. ∞) otherwise.[5]

The function f must take into account the time constraints on all the relevant arcs. In the example shown in the code, transition `t2` is the destination of two P-T arcs whose time intervals are disjoint: $[3, 4]$ and $[5, 6]$. Given a putative consumable `token(p2,R) token(p3,R')`, it is not enough to compute `min(4 monus R, 6 monus R')`[6]. That is to say, the deadline imposed by `t2` is not simply the minimum of the deadlines imposed by each arc. For example, consider the case where `R` and `R'` are both 0. The naive expression above would yield $min(6, 4) = 4$. However, ageing the two tokens by 4 results in the term

```
token(p2,4) token(p3,4)
```

which is not consumable by `t2`, since it violates the constraint on the arc from `p3` to `t2`.

2.4 Transition time Petri nets

2.4.1 Semantics

This is the original *time Petri net* model of Merlin and Farber[10]. This variant (henceforth *T-TPN*) ascribes a time interval $[lb, ub]$ to each transition T. The transition may fire if it has been enabled for no less than lb and no more than ub time units. After firing, its clock is reset. In the (more common) *strong* semantics, which we treat here, T *must* fire when it has been enabled for ub time units. This last requirement is waived in the event that T is disabled by the firing of some other transition.

2.4.2 Implementation

The implementation can be found in Appendix C. Again, the firing of each transition is modeled by an instantaneous rewrite rule in the instance module. Every transition has a clock (but tokens do not have clocks). The function `mte` takes the minimum, over all enabled transitions, of the time remaining until the transition must fire. The function `delta` ensures that, if transition T is disabled by the firing of some other transition T', T's clock is reset.

2.4.3 Correspondence between semantics and implementation

We now sketch a proof that the RTMaude program in Appendix C correctly implements (strong) Transition-TPN semantics as defined in [10].

We assume the correctness of RTMaude, Maude, the C++ compiler used to compile Maude, the operating system, and the hardware. We also assume that the time sampling strategy and time domain we have chosen in our RTMaude program is appropriate (more on this below).

Note that our program implements a single instance of a T-TPN, namely, that expressed in its instance module. We claim that this generalizes to *all* instances, because the T-TPN instance given in the example illustrates a strategy for translating any T-TPN instance to RTMaude code. To have a proof, this would need to be formalized.

Our implementation partitions the dynamics of a T-TPN into timed and untimed state transitions (tick rules and instantaneous rewrite rules). The challenge is to show that this partitioning is correct. Similar work has been done by Boyer and Roux[2], who give a semantics of T-TPNs in terms of *timed transition systems*. However, their work considers only 1-safe nets, whereas we treat unbounded nets.

To avoid confusion between transition nodes in the TPN and state transitions in the system, we refer to the former as *TPN-transitions*. The proof sketch consists of the following steps:

1. Demonstrating a correspondence between states in the program and states in the T-TPN.

[5] In our code, the function f is defined implicitly in `mte-consumable`. We do not declare any function called `f`.
[6] `monus` returns either 0 or the difference of its arguments, whichever is greater.

2. Showing that state transitions in the T-TPN can be partitioned into timed and untimed state transitions.
3. Showing correspondence between untimed state transitions:
   (a) Any untimed state transition that the program can take corresponds to an untimed transition that the T-TPN can take (correctness).
   (b) Any untimed transition the T-TPN can take corresponds to an untimed transition that the program can take (completeness).
4. Showing correspondence between timed state transitions:
   (a) Any timed transition the program can take corresponds to a timed transition that the TPN can take (correctness).
   (b) Any timed transition the TPN can take corresponds to a timed transition that the program can take (completeness).

We now sketch a proof of each step.

1. Our program defines the state of the system (i.e., a ground term of sort `PetriNet`) as a multiset of `Tokens` and `Transitions`. Each term of the form `token(P::Place)` corresponds to a token in the T-TPN: `P` is the location of the token. Each term of the form `trans(S::String, R::Time)` gives the amount of time `R` that has elapsed since the transition labelled by string `S` became enabled. The state of a T-TPN is fully expressed by the current (untimed) marking of the net, and the state of each transition's clock. Since every ground term of sort `PetriNet` contains exactly this information, it corresponds exactly to the state of the T-TPN.
2. The firing of a TPN-transition corresponds to the application of an instantaneous rewrite rule in our program. These are the untimed state transitions.

   The elapsing of time in the absence of any transition firings corresponds to the application of the tick rule. These are the timed state transitions. Since the firing of a TPN transition in this semantics is instantaneous, the timed and untimed transitions are disjoint.
3. We need to show that the application of instantaneous rewrite rules in our program corresponds to the taking of a discrete transition in the timed transition system.

   Consider for instance the module `TRANSITION-TIME_EXAMPLE`. Here there are four instantaneous rewrite rules, labelled a, b, c, d. Each such rule corresponds to the firing of a TPN-transition: the current marking is rewritten to another marking, subject to the time constraints ascribed to the transition. There are no other instantaneous rewrite rules in our program.
4. Our program contains a single tick rule, modeling the ageing of tokens in the absence of any TPN-transition firing. The application of a tick rule causes the TPN-transitions' clocks to advance in accordance with the RTMaude time domain and sampling strategy we have chosen (more on this below). The tick rules are *non-deterministic*, so the transitions can age any amount, up to the point where one of them must fire (as defined in Section 2.4.1). Thus, any untimed transition in the TPN (modulo time domain and sampling strategy) can be modelled by the application of the tick rule. Conversely, any application of the tick rule corresponds to a possible timed state transition in the net.

The preceding sketch gives a proof strategy. To formalize it, we could express our implementation and the T-TPN semantics in terms of timed transition systems, as in [2] (see their Definition 3.3). On the other hand, one could argue that RTMaude is as good a formalism as timed transition systems, and that our code in fact gives an operational semantics of T-TPNs.

The correctness of our program depends on an appropriate choice of time domain and time sampling strategy[6]. In our experiments, we used the `POSRAT-TIME-DOMAIN-WITH-INF` time domain and the default time sampling strategy with increment `1` (`set tick def 1`). Since our Petri net instances contained only integer time constraints, this seems to be a reasonable choice. (In fact, the positive rationals are more than we need: an integer time domain should be sufficient.)

3 Related work

An attempt at expressing timed Petri nets in RTMaude is made in [8], and was our starting point. Steggles[15] uses a competing rewriting logic based tool to model the time Petri nets of Merlin and Farber[10].

Tina[18] and Romeo[17] are the most popular[7] timed Petri net analysis tools. These two tools boast an impressive feature set, including various types of model checking. We tried both of the tools, but more work needs to be done to properly evaluate them. The Romeo package (for both Linux and OS X) is so buggy that we were unable to run any analyses, or even reliably re-open a previously created net. Tina seems more robust, but its user interface is Spartan at best.

4 Future work

This report documents work in progress. The following work remains:

- Formalize the correspondence proof for T-TPN, and construct similar proofs for other variants;
- Prove that the transformations shown here are computable (and feasible) for any reasonable input net;
- Do a case study, using the strategies shown in this report to implement e.g. an industrial problem as a timed Petri net in RTMaude, and analyzing the model with the RTMaude tools;
- Build a tool that takes a TPN description as input and generates RTMaude code which can subsequently be analyzed using RTMaude's toolkit (timed model checker, etc.), using the implementation strategies shown in this report;
- Compare the performance of this tool to that of the existing tools;
- Since Petri nets have gained popularity partly because they are naturally expressed graphically, it might be desirable to add a graphical user interface (GUI). One possibility is to reuse the GUI from an existing tool like Romeo[17].

5 Conclusions

We have considered three timed Petri net variants from the literature and implemented their semantics in RTMaude. To our knowledge, this has not been done before.

The last two variants, *timed-arc* and *transition time* Petri nets, seem to be the most popular variants in the recent literature. In particular, transition time Petri net semantics is used by the tools Tina[18] and Romeo[17].

The expressiveness comparisons in [3] and [2] show that the timed-arc semantics is the most expressive of the three considered here. This is not entirely surprising: the ages of *all* the consumable tokens must be compared with the constraints on *all* of the relevant arcs. Moreover, the strong variant of this semantics is more expressive than the weak one. We only experimented with the weak semantics. Perhaps an implementation of the strong timed-arc semantics could be used as a canonical representation for TPNs. However, it would have to be shown that such a transformation is computationally feasible. That is, the increase in size from some input TPN to the canonical TPN would have to be acceptable.

We have learned much from the work presented here. Perhaps the most important lesson is that implementing a formalism from the literature is tremendously helpful in understanding it. Descriptions of the semantics found in the literature can be unclear. We suggest that published real-time system semantics could take the form of RTMaude code. This would provide a standardized, precise notation, robust to varying English writing skills. Moreover, such a semantics would actually be *executable* and *analyzable*.

Finally, formal models are motivated by systems that exist in the physical world. Although some of the timed Petri net variants in the literature are more popular than others, there is no

[7] Or perhaps the only!

timed Petri net variant which is "the best one": the suitability of a particular semantics depends on the problem domain. Even Merlin and Farber's seminal paper[10] introducing the transition time Petri net model devotes most of its length to a specific problem in communication protocols. It is our view that the study of timed Petri nets in the academic community has become too far removed from reality. There are so many time Petri net variants that an exhaustive study, for example with respect to computability and complexity questions, has itself become intractable. Our experimentation in RTMaude relies on the construction of example timed Petri net instances (such as the one in Figure 3). We suggest that a case study would provide us with the opportunity to perform more realistic experiments. We believe that this would shed more light not only on the specific problem posed, but also on the nature of timed Petri nets.

A Implementation of ITPN

```
load real-time-maude.maude

***
*** Interval Timed Petri Nets as per Olveczky and Meseguer.
***

  (tmod PETRI is
    protecting STRING .
    protecting POSRAT-TIME-DOMAIN-WITH-INF .

    sort Location . --- a single token at a single place
    sort Firing .
    sort PetriNet .
    subsort Location < PetriNet .
    subsort Firing < PetriNet .
    subsort PetriNet < System .

    vars R LB UB TimeElapsed : Time .
    var F : Firing .

    var Outputs : PetriNet .
    var L : Location .
    var PN : PetriNet .
    var SYSTEM : System .

    op emptyPN : -> PetriNet [ctor] .
    op __ : PetriNet PetriNet -> PetriNet [ctor assoc comm id: emptyPN] .

    --- input marking, lower bound, upper bound, time elapsed
    op firing : PetriNet Time Time Time -> Firing [frozen(1) ctor] .

    op delta : System Time -> System [frozen(1)] .
    eq delta(emptyPN, R) = emptyPN .
    eq delta(L PN, R) = L delta(PN, R) .
    eq delta(PN firing(Outputs, LB, UB, TimeElapsed), R) =
        delta(PN, R) firing(Outputs, LB, UB, TimeElapsed + R) .

    op mte : System -> Time [frozen] .
    eq mte(emptyPN) = INF .
    eq mte(L PN) = mte(PN) .
    eq mte(PN firing(Outputs,LB,UB,TimeElapsed)) =
        min(mte(PN), UB monus TimeElapsed) .

    op stable : System -> Bool [frozen] .
```

```
    eq stable(emptyPN) = true .

    --- a firing transition completes
    crl [finish_firing] : PN firing(Outputs,LB,UB,TimeElapsed) =>
        PN Outputs if TimeElapsed >= LB .

    --- currently firing transitions proceed
    crl [tick] : {SYSTEM} =>
        {delta(SYSTEM,R)} in time R if
          R le mte(SYSTEM) /\ stable(SYSTEM) [nonexec] .

endtm)

(tmod PETRI_EXAMPLE is
    including PETRI .

    ops p q r : -> Location [ctor] .

    op initNet : -> PetriNet .

    var PN : PetriNet .

    rl [a] : p p => firing(q q,5,10,0) .
    rl [b] : q => firing(p,4,8,0) .
    rl [c] : q => firing(r,3,7,0) .
    rl [d] : p q => firing(r,2,5,0) .

    eq stable(PN p p) = false .
    eq stable(PN q) = false .
    eq stable(PN p q) = false . --- redundant!
    eq stable(PN) = true [owise] .

    eq initNet = p p .
endtm)

(tmod MODEL-CHECK-PETRI is
    including TIMED-MODEL-CHECKER .
    including PETRI_EXAMPLE .

    op alive : -> Prop [ctor] .

    var PN : PetriNet .

    ceq {PN}
        |= alive = true if mte(PN) < INF .

    eq {PN}
        |= alive = false [owise] .

endtm)

(set tick def 1 .)
--- Will transition 'c' start firing within 100 time units?
(tsearch [1] {initNet} =>*
  {PN firing(r,3,7,DURATION::Time)} in time <= 100 .)
--- Deadlock (no transition can ever fire) reachable in 15 time steps:
(mc {initNet} |=t []<> alive in time <= 15 .)
```

B Implementation of Arc-TPN

```
load real-time-maude.maude

***
*** Arc Time Petri Nets as per Ruiz et al.
***
*** Weak semantics: allows tokens to age past the upper bound of any arc.
*** The question of whether a token is "dead" may be undecideable;
***   we do not model token liveness here.
***

--- TODO: remove tokens which are obviously "dead"

  (tmod ARC-TIME-WEAK is
    protecting STRING .
    protecting POSRAT-TIME-DOMAIN-WITH-INF .

    sort Place .
    sort Token .
    sort Transition .
    sort TransitionSet .
    sort PetriNet .
    sort Consumable .
    sort ConsumableSet .
    subsort Consumable < ConsumableSet .
    subsort Transition < TransitionSet .
    subsort Token < PetriNet .
    subsort PetriNet < System .

    var K : Token .
    vars PN PN' : PetriNet .
    var SYSTEM : System .
    vars R R' R'' : Time .
    var T : Transition .
    vars TS TS' : TransitionSet .
    vars P P' : Place .
    vars C C' : Consumable .
    vars CS CS' : ConsumableSet .

    op emptyCS : -> ConsumableSet [ctor] .
    op _##_ : ConsumableSet ConsumableSet ->
        ConsumableSet [ctor assoc comm id: emptyCS] .
    eq C ## C = C .

    op consumable : PetriNet -> Consumable [ctor] .
    op net : Consumable -> PetriNet .
    eq net(consumable(PN)) = PN .

    op emptyPN : -> PetriNet [ctor] .
    op __ : PetriNet PetriNet -> PetriNet [ctor assoc comm id: emptyPN] .

    op emptyTS : -> TransitionSet [ctor] .
    op _;_ : TransitionSet TransitionSet ->
        TransitionSet [ctor assoc comm id: emptyTS] .
    eq T ; T = T .

    op token : Place Time -> Token [ctor] .
```

```
    op transitions : -> TransitionSet .

    op _between_and_ : Time Time Time -> Bool .
    ceq R between R' and R'' = true if R >= R' /\ R <= R'' .
    eq R between R' and R'' = false [owise] .

    op delta : System Time -> System [frozen] .
    eq delta(emptyPN, R) = emptyPN .
    eq delta(token(P,R) PN, R') = token(P,R + R') delta(PN, R') .

    op mte : System -> Time [frozen] .
--- the following function is defined in the next module:
    op mte : PetriNet TransitionSet -> Time [frozen] .
    op mte-trans : PetriNet Transition -> Time [frozen] .
    op mte-cs : ConsumableSet Transition -> Time [frozen] .
    op mte-consumable : Consumable Transition -> Time [frozen] .
    op powerset : Consumable Transition -> ConsumableSet [frozen] .
    op map : ConsumableSet Token Transition -> ConsumableSet .

    eq mte(PN) = mte(PN, transitions) .

--- takes the max over all transitions T of the latest firing of T
    eq mte(PN, T) = mte-trans(PN,T) .
    ceq mte(PN, TS ; T) = max(mte(PN,TS), mte(PN,T)) if
          TS =/= emptyTS .

--- latest firing of T
    eq mte-trans(PN, T) = mte-cs(powerset(consumable(PN), T), T) .

--- returns INF if T cannot fire
    eq mte-cs(C, T) = mte-consumable(C, T) .
    ceq mte-cs(C ## CS, T) = max(mte-cs(C,T), mte-cs(CS,T)) if CS =/= emptyCS .

--- this might be further optimized?
    eq powerset(consumable(emptyPN), T) = emptyCS .
    ceq powerset(consumable(K PN), T) = map(CS, K, T) ## CS if
          CS := powerset(consumable(PN), T) .

--- adds second argument to every element of first argument
--- prunes irrelevant consumables
    eq map(emptyCS, K, T) = consumable(K) .
    ceq map(C ## CS, K, T) = C' ## map(CS,K,T) if
          C' := consumable(K net(C)) /\ mte-consumable(C',T) < INF .
    eq map(C ## CS, K, T) = map(CS,K,T) [owise] .

--- tokens age
    crl [tick] : {SYSTEM} =>
        {delta(SYSTEM,R)} in time R if R le mte(SYSTEM) [nonexec] .

endtm)

(tmod ARC-TIME-WEAK_EXAMPLE is
    including ARC-TIME-WEAK .

    ops p1 p2 p3 p4 : -> Place .
    ops t1 t2 t3 : -> Transition .
    op initNet : -> PetriNet .
```

```
    vars DELTA R R' : Time .
    var T : Transition .
    var C : Consumable .

--- TODO: enforce single P-T arcs by using Consumable?
---   (this would involve making Consumable more restrictive)
    crl [t1] : token(p1,R) => token(p2,0) token(p3,0) if R between 2 and 4 .

    crl [t2] : token(p2,R) token(p3,R') => token(p1,0) if
          R between 3 and 4 /\ R' between 5 and 6 .

    crl [t3] : token(p3,R) => token(p4,0) token(p3,0) if R between 2 and 8 .

    eq transitions = t1 ; t2 ; t3 .

--- what is the latest that a transition can fire?
    ceq mte-consumable(consumable(token(p1,R)), t1) =
        4 monus R if R <= 4 [label mte-c-t1] .
    ceq mte-consumable(consumable(token(p2,R) token(p3,R')), t2) = DELTA if
          DELTA := min(4 monus R, 6 monus R') /\
          R <= 4 /\
          R' <= 6 /\
          R' + DELTA between 5 and 6 /\
          R + DELTA between 2 and 4 [label mte-c-t2] .
    ceq mte-consumable(consumable(token(p3,R)), t3) =
        8 monus R if R <= 8 [label mte-c-t3] .
    eq mte-consumable(C,T) = INF [owise] .

    eq initNet = token(p1,0) token(p1,1) token(p1,2) .

endtm)

(tmod MODEL-CHECK-PETRI is
    including TIMED-MODEL-CHECKER .
    including ARC-TIME-WEAK .
    including ARC-TIME-WEAK_EXAMPLE .

    op alive : -> Prop [ctor] .

    var PN : PetriNet .

    ceq {PN}
        |= alive = true if mte(PN) < INF .
    eq {PN} |= alive = false [owise] .

endtm)

(set tick def 1 .)
--- can a token age past the deadline of one transition if another is
---   still fireable?
(tsearch [1] {token(p1,0) token(p1,0)} =>* {PN token(p1,5)} in time <= 10 .)
--- Deadlock (no transition can ever fire) reachable in 15 time steps:
(mc {initNet} |=t []<> alive in time <= 15 .)
--- Given 3 tokens of different ages, is our algorithm non-deterministic w.r.t.
---   which token is chosen for consumption? (Yes, it is!)
--- oldest token consumed first:
(tsearch [1] {initNet} =>*
   {token(p2,0) token(p3,0) token(p1,0) token(p1,1)} in time <= 0 .)
```

```
--- middle token consumed first:
(tsearch [1] {initNet} =>*
   {token(p2,0) token(p3,0) token(p1,1) token(p1,3)} in time <= 1 .)
--- newest token consumed first:
(tsearch [1] {initNet} =>*
   {token(p2,0) token(p3,0) token(p1,3) token(p1,4)} in time <= 2 .)
```

C Implementation of Transition-TPN

```
load real-time-maude.maude

***
*** Transition Time Petri Nets as per Merlin & Farber
*** Strong semantics: an enabled transition _must_ eventually fire
***

  (tmod TRANSITION-TPN is
    protecting STRING .
    protecting POSRAT-TIME-DOMAIN-WITH-INF .

    sort Place .
    sort Token .
    sort Transition .
    sort PetriNet .
    subsort Token < PetriNet .
    subsort Transition < PetriNet .
    subsort PetriNet < System .

    var PN : PetriNet .
    var SYSTEM : System .
    vars R R' R'' : Time .
    var P : Place .
    var T : Transition .
    var S : String .

    op emptyPN : -> PetriNet [ctor] .
    op _;_ : PetriNet PetriNet -> PetriNet [ctor assoc comm id: emptyPN] .

    op token : Place -> Token [ctor] .

    op trans : String Time -> Transition [ctor] .

    op _between_and_ : Time Time Time -> Bool .
    ceq R between R' and R'' = true if R >= R' /\ R <= R'' .
    eq R between R' and R'' = false [owise] .

    op delta : System Time -> System [frozen(1)] .
    ceq delta(trans(S,R) ; PN, R') = trans(S,R + R') ; delta(PN, R') if
          enabled(PN,trans(S,R)) .
    ceq delta(trans(S,R) ; PN, R') = trans(S,0) ; delta(PN,R')  if
      not enabled(PN,trans(S,R)) .
    eq delta(PN,R) = PN [owise] .

    op mte : System -> Time [frozen] .
    op mte : PetriNet Transition -> Time [frozen] .
    op enabled : PetriNet Transition -> Bool [frozen] .
```

```
    ceq mte(T ; PN) = min(mte(PN,T), mte(PN)) if enabled(PN,T) .
    eq mte(PN) = INF [owise] .

    --- tokens age
    crl [tick] : {SYSTEM} =>
        {delta(SYSTEM,R)} in time R if
          R le mte(SYSTEM) [nonexec] .

endtm)

(tmod TRANSITION-TPN_EXAMPLE is
    including TRANSITION-TPN .

    ops p q r : -> Place .
    op initNet : -> PetriNet .

    var R  : Time .
    var PN : PetriNet .
    var T : Transition .

    crl [a] : trans("a",R) ; token(p) ; token(p) =>
        trans("a",0) ; token(q) ; token(q) if R between 5 and 10 .

    crl [b] : trans("b",R) ; token(q) =>
        trans("b",0) ; token(p) ; token(p) if R between 4 and 8 .

    crl [c] : trans("c",R) ; token(q) =>
        trans("c",0) ; token(r) if R between 3 and 7 .

    crl [d] : trans("d",R) ; token(p) ; token(p) ; token(q) ; token(q) =>
        trans("d",0) ; token(r) if R between 2 and 5 .

    eq mte(PN, trans("a",R)) = 10 - R .
    eq mte(PN, trans("b",R)) = 8 - R .
    eq mte(PN, trans("c",R)) = 7 - R .
    eq mte(PN, trans("d",R)) = 5 - R .

    eq enabled(PN ; token(p) ; token(p), trans("a",R)) = true .
    eq enabled(PN ; token(q), trans("b",R)) = true .
    eq enabled(PN ; token(q), trans("c",R)) = true .
    eq enabled(PN ; token(p) ; token(p) ; token(q) ; token(q),
          trans("d",R)) = true .
    eq enabled(PN, T) = false [owise] .

    eq initNet = token(p) ; token(p) ; token(p) ;
        trans("a",0) ; trans("b",0) ; trans("c",0) ; trans("d",0) .

endtm)

(tmod MODEL-CHECK-PETRI is
    including TIMED-MODEL-CHECKER .
    including TRANSITION-TPN_EXAMPLE .

    op alive : -> Prop [ctor] .

    var PN : PetriNet .

    ceq {PN}
```

```
        |= alive = true if mte(PN) < INF .

    eq {PN}
        |= alive = false [owise] .

endtm)

(set tick def 1 .)
--- Will there be a token in place 'q' within 100 time units?
(tsearch [1] {initNet} =>*
   {PN ; token(q)} in time <= 100 .)
--- How soon can there be a token in place 'r'?
(find earliest {initNet} =>* {PN ; token(r)} .)
--- How late can there be a token in place 'r'?
(find latest {initNet} =>* {PN ; token(r)} in time <= 100 .)
--- Deadlock (no transition can ever fire) reachable in 15 time steps:
(mc {initNet} |=t [] <> alive in time <= 15 .)
```

References

[1] Tommaso Bolognesi, Ferdinando Lucidi, and Sebastiano Trigila. From timed Petri nets to timed LOTOS. In *Proceedings of the IFIP WG6.1 Tenth International Symposium on Protocol Specification, Testing and Verification*, pages 395–408, Amsterdam, The Netherlands, 1990. North-Holland Publishing Co.

[2] Marc Boyer and Olivier H. Roux. On the compared expressiveness of arc, place and transition time Petri nets. *Fundam. Inf.*, 88(3):225–249, 2008.

[3] Antonio Cerone and Andrea Maggiolo-Schettini. Time-based expressivity of time Petri nets for system specification. *Theoretical Computer Science*, 216(1-2):1 – 53, 1999.

[4] Hans-Michael Hanisch. Analysis of place/transition nets with timed arcs and its application to batch process control. In *Proceedings of the 14th International Conference on Application and Theory of Petri Nets*, pages 282–299, London, UK, 1993. Springer-Verlag.

[5] Kurt Jensen. An introduction to the theoretical aspects of coloured Petri nets. In *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*, pages 230–272, London, UK, 1994. Springer-Verlag.

[6] Peter Csaba Ölveczky. Real-Time Maude. World Wide Web: `http://www.ifi.uio.no/RealTimeMaude/`.

[7] Peter Csaba Ölveczky. Personal communication, 2010.

[8] Peter Csaba Ölveczky and José Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285(2):359 – 405, 2002.

[9] Peter Csaba Ölveczky and José Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher Order Symbol. Comput.*, 20(1-2):161–196, 2007.

[10] P. M. Merlin and D. J. Farber. Recoverability of communication protocols: Implications of a theoretical study. *IEEE Trans. Comm.*, 24(9):1036–1043, 1976.

[11] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73 – 155, 1992.

[12] C. Ramchandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*. PhD thesis, MIT, 1973.

[13] W. Reisig. *Petri nets: An introduction*. Springer, Berlin, 1985.

[14] Valentin Valero Ruiz, Fernando Cuartero Gomez, and David de Frutos Escrig. On non-decidability of reachability for timed-arc Petri nets. In *PNPM '99: Proceedings of the The 8th International Workshop on Petri Nets and Performance Models*, page 188, Washington, DC, USA, 1999. IEEE Computer Society.

[15] L. J. Steggles. Rewriting logic and Elan: Prototyping tools for Petri nets with time. In *ICATPN*, pages 363–381, 2001.

[16] The Maude Team. Maude. World Wide Web: `http://maude.cs.uiuc.edu/`.

[17] The Romeo Team. Romeo. World Wide Web: `http://romeo.rts-software.org/`.

[18] The Tina Team. Tina. World Wide Web: `http://homepages.laas.fr/bernard/tina/`.

[19] W.M.P. van der Aalst. Interval timed coloured Petri nets and their analysis, 1993.