RIPN system construction rules. v1.0

6th May 2025   *Pedro Sánchez Palma*

pedro.sanchez@upct.es

Universidad Politécnica de Cartagena, Spain

For a full understanding of the approach around RIPN, we encourage you to read the following paper

Summary:

RIPN is a Java-based tool that allows for the integration of a specific type of interpreted Petri net with teleo-reactive (TR) programs. The Petri net is interconnected with the environment such that the occurrence of events in said environment serves as input for triggering transitions, which we call "inputs." Transitions can also have conditions defined on a set of variables (integers or real-world variables) and/or facts, all managed in a BeliefStore shared with the TR program. A RIPN system can be defined by a TR program, a Petri net, or a combination of both, such that changes in the BeliefStore are shared between both subsystems. Thus, the Petri net influences the TR program and vice versa.

The system allows for the execution of discrete or durative actions, both in the Petri net and in the TR program. In the TR program, these actions are executed in the rules. In the Petri net, they are executed only in the locations when they acquire the marking. Changes to the BeliefStore (changing variables or adding/deleting facts) can be made in both locations and transitions. Actions and facts can have parameters. Timers can be added and manipulated (paused, continued, stopped). The end of a timer is automatically added to the BeliefStore.

The semantics associated with durative and discrete actions are the same for both the TR program and the Petri net. The only consideration is that for the Petri net, the reference will be the marking of a previously unmarked location, while for the TR program, the reference will be the activation of a rule not active in the immediately preceding state.

The system allows you to leave traces of everything that happens in different files with time stamps to facilitate debugging.

Concerning the TR programs

INT or REAL variables are considered. A section is included first, before the rules, where the following statements are made. All sections must be included, even if nothing is specified in some of them:

- **FACTS:** statement of facts, indicating whether they have parameters or not, e.g., open_clamp, see_robot(INT, INT), all separated by ';'. They shall be added using remember, e.g. remember(see_robot(5,6)), and deleted using forget, e.g. forget(open_clamp).
- **VARSINT:** declaration of integer variables separated by ';'.
- **VARSREAL:** declaration of real variables separated by ';'.
- **DISCRETE:** discrete actions, with or without parameters, e.g. open(INT, INT), close().
- **DURATIVE:** durative actions, with or without parameters, same syntax as discrete, e.g. rotate(), move(REAL).
- **TIMERS:** declaration of timers, e.g. timer1.
- **INIT:** initial values to integer or real variables using the symbol ":=". For example: x:=1, y:=5.66. If not specified, a value of 0 is assumed.

This is followed by the block delimited by the "<TR>" mark where the rules are specified, from highest priority (top) to lowest. The rules are constructed with the format:

```
<TR>
condition -> action(s) [changes in the BeliefStore, separated by ';']
```

Conditions are logical expressions with the usual operators (&&, ||, !) using BeliefStore variables and facts.

Each rule can include none, one or more actions, whether durative or discrete, using in its arguments values from the BeliefStore variables or arithmetic expressions with the usual operators (+, -, *, /).

Changes in the BeliefStore can be changes to variables with the usual arithmetic expressions (+,-,*,/) and/or adding or deleting facts (remember/forget), with or without parameters. It is possible to use the wildcard character "_" in the facts both in the conditions and in the "forget" operation (which would delete all occurrences matching the indicated pattern).

Declared timers can be started ("start", with seconds in brackets), paused ("pause"), continued ("continue") or stopped ("stop"). When a timer ends, the fact ".end" is automatically added to the BeliefStore, can be used in the conditions and can be removed manually, "forget(t1.end)" in the case of a timer named "t1".

For example:
```
FACTS: seen(INT); done
VARSINT: x; y;
VARSREAL: z
DISCRETE: open(INT, REAL)
DURATIVE:  rotate(INT); close()  TIMERS:
t1
INIT: x:= 0; z:=4.5

<TR>
seen(3) && x==2 && t1.end-> open(2, 4.5); rotate() [forget(seen(_)]) seen(_) &&
x==4 -> open(3, 6.6)
y<5 -> t1.continue(); close() [x:=3]; y<5 -> t1.continue(); close() [x:=3]. x> 1 ->
rotate(3); close() [y:=4; remember(seen(3))]
x==0 -> t1.start(4)
True -> act1() [y:=2; z:=z+1].
```

The system is continuously evaluating the highest priority rule to be executed. There must always be at least one that meets the condition (hence the lowest one is usually set to True).

Durative actions are running if the rule that initiated them is active. Discrete actions are triggered upon each activation of the rule. But it requires another rule to gain control and the first rule to re-activate for the execution of the discrete action to be triggered again. Operations on timers are considered discrete. Starting a timer that is already running removes the previous instance of the timer.

Another example:

```
FACTS: veo(INT); touch
VARSINT: x; y; z
VARSREAL:
DISCRETE: act1(); act2()
DURATIVE: actFIN()
TIMERS: t1
INIT: x:=0

<TR>
t1.end -> actFIN()
x==2 -> act2() [remember(touch)]
True -> act1(); t1.start(5) [y:=2; z:=55; x:=2; remember(veo(8))]
```

In this TR program, one fact is declared with arguments and another ("touch") without arguments (note that parentheses are not included, unlike actions, where they are included by convention). The rule below activates a 5-second timer and executes the discrete action "act1()". In addition, several changes are made to the BeliefStore. The change in "x" implies that the rule with the discrete action "act2()" is executed, and the fact "touch" is remembered. After 5 seconds have elapsed, the durative action "actFIN()" is executed. The trace generated from this execution is as follows:

2025-05-07 10:51:15.936]

◆ Current BeliefStore state:

  Active facts without parameters: []

  Active facts with parameters: {}

  Integer variables: {x=0, y=0, z=0}

  Real variables: {}

[2025-05-07 10:51:15.951] 🔄 Executing rule with condition: True

[2025-05-07 10:51:15.952] ⏩ Executing discrete action: act1 with parameters: []

[2025-05-07 10:51:15.952] ⏩ Executing discrete action: act1 with parameters: []

[2025-05-07 10:51:15.957] ⏩ Executing discrete action: t1.start with parameters: [5.0]

[2025-05-07 10:51:15.958] 🛠 Extracted timer command: start for timer: t1

[2025-05-07 10:51:15.958] ⏳ Timer started: t1 for 5 seconds

[2025-05-07 10:51:16.072]

🔷 Current BeliefStore state:

  Active facts without parameters: []

  Active facts with parameters: {veo=[[8]], }

  Integer variables: {x=2, y=2, z=55}

  Real variables: {}

[2025-05-07 10:51:16.076] 🔄 Executing rule with condition: x==2

[2025-05-07 10:51:16.076] ⏩ Executing discrete action: act2 with parameters: []

[2025-05-07 10:51:16.077] ⏩ Executing discrete action: act2 with parameters: []

[2025-05-07 10:51:16.177]

🔷 Current BeliefStore state:

  Active facts without parameters: [touch]

  Active facts with parameters: {veo=[[8]], }

  Integer variables: {x=2, y=2, z=55}

  Real variables: {}

[2025-05-07 10:51:16.282]

🔷 Current BeliefStore state:

  Active facts without parameters: [touch]

  Active facts with parameters: {veo=[[8]], }

  Integer variables: {x=2, y=2, z=55}

  Real variables: {}

The output shown in the console (which acts as an observer object for the TR program outputs) is as follows:

Initiating tr program...

[2025-05-07 10:51:15.955] Observer: Executing discrete action: act1 with parameters: []

[2025-05-07 10:51:16.077] Observer: Executing discrete action: act2 with parameters: []

[2025-05-07 10:51:21.075] Observer: Starting durative action: actFIN() with parameters: []

Concerning Petri Nets

The initial declaration section is the same as for the TR program. Each part of the system (Petri net or TR program) must declare what it uses in its context. If a variable/fact/action/timer is declared in both the Petri net and the TR program, it is because it is the same variable/fact/action/timer, so that changes made by one part of the system are visible from the other part.

After the part of the basic declarations, a Petri net part is included where places, transitions, arcs and initial marking are indicated. Places, transitions and arcs are separated by ";". An arc is indicated by "place->transition". If the arc is inhibiting, "-0>" is the syntax. The initial marking must give a value (0 or 1) to each place.

```
FACTS:
VARSINT: x; y VARSREAL:
INIT:
DISCRETE: open(INT, REAL) DURATIVE:
TIMERS: t1

PLACES:  p0;  p1  TRANSITIONS:
t0; t1
ARCS: p0->t0; t0->p1; p1->t1; t1->p0

INITMARKING: (1,0)

<PN>
p0: [open(3,4); t1.start(8); x:=1]. t0: []
if(y==2)
p1: [x:=2]
```

After the declarations, a block delimited by the "<PN>" mark specifies the behaviour. The changes that are carried out in the BeliefStore are indicated for the appropriate places with the same syntax that we discussed for the TR programs but also adding the actions (durative or discrete) and the handling of the timers. The conditions follow the same considerations as for TR programs:

```
place_name: [operations on the BeliefStore or durative/discrete actions/timers, separated
by ";"] if(condition on the BeliefStore)
```

Not all places need to be listed in this section, only those where changes are made to the BeliefStore and/or actions are initiated. Durative actions are running while the place is marked. Discrete actions only when the place loses the marking and regains it (same analogy as with the TR programme rules).

The transitions we call of type "input" are also specified:

```
transition_name:  [operations  on  BeliefStore,  separated  by  ";"]  if(condition  on
BeliefStore) when(event_name)
```

Transitions that are not listed in the section or don't have the "when" part are considered of type **immediate** and have priority over incoming calls. If the system has not triggered all the immediate ones enabled to do so (their entry places are marked, and the exit places have no tokens) it will continue to evolve the net. Once all the immediate transitions have been triggered, the system is ready to evaluate the triggering of the input transitions, which are those listed in the <PN> block. The system will fire the transitions that have a condition that is met given the state of the BeliefStore. In these transitions only changes can be performed on

the BeliefStore, no actions can be performed, and no timers can be managed.

Let's look at an example.

FACTS: see(INT)
VARSINT: x; y; p1; p2
VARSREAL:
INIT: x:=3
DISCRETE:
DURATIVE: act1(INT)
TIMERS: t1

PLACES: p0; p1; p2
TRANSITIONS: t0; t1; t2; t3
ARCS: p0->t0; t0->p1; p1->t1; t1->p0; p1->t2; t2->p2; p2->t3; t3->p0
INITMARKING: (1,0,0)
EVENTS: ev1(0, INT); ev2(2, INT, INT)

<PN>
p0: [t1.start(5); act1(88)]
t0: [x:=p1+y-p2; remember(see(x+p1))] if (t1.end)
t2: [y:=p1+p2] when (ev2(p1, p2))
p1: [x:=x+1]

When place 'p0' receives the tag, it starts a 5-second timer and initiates the "act1(88)" lasting action. Once the timer has elapsed, the "t1.end" condition becomes true, so transition "t1" can be triggered, having as changes in the BeliefStore the change of variable "x" and remembering the fact "see(x+p1)". Transition "t2" has an associated event, but since transition "t1" is immediate (it is not listed in the <PN> section), it has priority over "t2", so it is never executed. The execution trace can be seen below.

[2025-05-07 09:59:02.815] ⏳ Timer started: t1 for 5 seconds
[2025-05-07 09:59:02.816] 🕐 Timer t1 started for 5 seconds
[2025-05-07 09:59:02.820] Observer: Starting durative action: act1 with parameters: [88.0]
[2025-05-07 09:59:02.822] Current state of the Petri Net:
p0: ●
p1: ○
p2: ○
[2025-05-07 09:59:02.822]
🔷 Current BeliefStore state:
  Active facts without parameters: []
  Active facts with parameters: {}
  Integer variables: {p1=0, p2=0, x=3, y=0}
  Real variables: {}

[2025-05-07 09:59:03.139] 🚫 Skipped firing transition t0 (Condition not met: t1.end)
[2025-05-07 09:59:03.139] Current state of the Petri Net:
p0: ●
p1: ○
p2: ○
[2025-05-07 09:59:03.139]
🔷 Current BeliefStore state:
  Active facts without parameters: []
  Active facts with parameters: {}
  Integer variables: {p1=0, p2=0, x=3, y=0}
  Real variables: {}

[2025-05-07 09:59:05.144] 🚫 Skipped firing transition t0 (Condition not met: t1.end)
[2025-05-07 09:59:05.145] Current state of the Petri Net:
p0: ●
p1: ○

p2: ○
[2025-05-07 09:59:05.145]
🔷 Current BeliefStore state:
  Active facts without parameters: []
  Active facts with parameters: {}
  Integer variables: {p1=0, p2=0, x=3, y=0}
  Real variables: {}

[2025-05-07 09:59:07.149] 🚫 Skipped firing transition t0 (Condition not met: t1.end)
[2025-05-07 09:59:07.149] Current state of the Petri Net:
p0: ●
p1: ○
p2: ○
[2025-05-07 09:59:07.149]
🔷 Current BeliefStore state:
  Active facts without parameters: []
  Active facts with parameters: {}
  Integer variables: {p1=0, p2=0, x=3, y=0}
  Real variables: {}

[2025-05-07 09:59:09.155] 🚫 Skipped firing transition t0 (Condition not met: t1.end)
[2025-05-07 09:59:09.155] ✅ Timer expired: t1_end activated
[2025-05-07 09:59:09.155] 🔴 Timer fully removed: t1
[2025-05-07 09:59:09.155] Current state of the Petri Net:
p0: ●
p1: ○
p2: ○
[2025-05-07 09:59:09.155]
🔷 Current BeliefStore state:
  Active facts without parameters: [t1_end]
  Active facts with parameters: {}
  Integer variables: {p1=0, p2=0, x=3, y=0}
  Real variables: {}

Another example:

FACTS: see(INT)
VARSINT: x; y; p1; p2
VARSREAL:
INIT: x:=3; p1:=5
DISCRETE: act2(INT)
DURATIVE: act1(INT)
TIMERS: t1

PLACES: p0; p1; p2
TRANSITIONS: t0; t1; t2; t3
ARCS: p0->t0; t0->p1; p1->t1; t1->p0; p1->t2; t2->p2; p2->t3; t3->p0
INITMARKING: (1,0,0)
EVENTS: ev1(0, INT); ev2(2, INT, INT)

<PN>
p0: [y:=6; act2(p1)]
t0: [x:=p1; remember(see(y+p1))] if (p1>100) when(ev1(p1))
t2: [y:=33] when (ev2(p1, p2))
p1: [x:=x+1]

It can be seen that with the initial marking of the net, the discrete action "act2(5)" is executed, since "p1" is equal to 5. The system waits for the event "ev1" to occur in order to trigger the transition "t0", which will have an INT value as an argument that is stored in the variable "p1" and will be used to evaluate the condition and, if it is met, carry out the changes in the BeliefStore (in this case, assign the value of "p1" to "x" and store the fact "see" with the value of the argument after performing the sum). The trace of this execution is the following

after injecting the event "ev1(200)":

[2025-05-07 10:34:20.978] Observer: Executing discrete action: act2 with parameters: [5.0]
[2025-05-07 10:34:20.984] Current state of the Petri Net:
p0: ●
p1: ○
p2: ○
[2025-05-07 10:34:20.985]
🔷 Current BeliefStore state:
  Active facts without parameters: []
  Active facts with parameters: {}
  Integer variables: {p1=5, p2=0, x=3, y=6}
  Real variables: {}

⏸ No non-immediate transitions enabled at this time. Waiting...
⏸ No non-immediate transitions enabled at this time. Waiting...
⏸ No non-immediate transitions enabled at this time. Waiting...
⏸ No non-immediate transitions enabled at this time. Waiting...
⏸ No non-immediate transitions enabled at this time. Waiting...
⏸ No non-immediate transitions enabled at this time. Waiting...
⏸ No non-immediate transitions enabled at this time. Waiting...
⏸ No non-immediate transitions enabled at this time. Waiting...
[2025-05-07 10:34:37.236] Current state of the Petri Net:
p0: ○
p1: ●
p2: ○
[2025-05-07 10:34:37.236]
🔷 Current BeliefStore state:
  Active facts without parameters: []
  Active facts with parameters: {see=[[206]], }
  Integer variables: {p1=200, p2=0, x=200, y=6}
  Real variables: {}

[2025-05-07 10:34:39.241] Observer: Executing discrete action: act2 with parameters: [200.0]
[2025-05-07 10:34:39.241] Current state of the Petri Net:
p0: ●
p1: ○
p2: ○
[2025-05-07 10:34:39.241]
🔷 Current BeliefStore state:
  Active facts without parameters: []
  Active facts with parameters: {see=[[206]], }
  Integer variables: {p1=200, p2=0, x=200, y=6}
  Real variables: {}

⏸ No non-immediate transitions enabled at this time. Waiting...
⏸ No non-immediate transitions enabled at this time. Waiting...

Concerning Events

From the Java Swing interface, we can inject events into the system as simulations. Events are declared in the corresponding section:

EVENTS: ev1(0, INT); ev2(2, INT, INT)

The first parameter indicates how many seconds the event will be active in the queue of events waiting to be served. A '0' indicates that there is no time limit. Once the time has passed, if the event has not been consumed since any Petri net transition, it is removed from the pool. The history of occurrences is kept in the trace. If an event is received with an incorrect number of parameters, it is rejected and recorded in the trace. A graphical interface is provided to manage event injection. In a real deployment, it would be replaced by the corresponding Java module to receive events from the environment.

A possible trace could be the following where it can be observed that the ev2 expires after 2 seconds and that it was received at the end with an incorrect number of parameters.

```
[LOG STARTED] 2025-05-07 09:55:44.081
[2025-05-07 09:55:52.174] Event added: ev1
Active Events:
    ev1 [100.0] expires in 9223370290250223633 ms
Unattended Events:

[2025-05-07 09:55:58.747] Event added: ev2
Active Events:
    ev2 [200.0, 100.0] expires in 1999 ms
    ev1 [100.0] expires in 9223370290250217060 ms
Unattended Events:

[2025-05-07 09:56:02.348] Expired events cleaned
Active Events:
    ev1 [100.0] expires in 9223370290250213459 ms
Unattended Events:

[2025-05-07 09:56:02.351] Event wrong arity: ev2
Active Events:
    ev1 [100.0] expires in 9223370290250213456 ms
Unattended Events:
    ev2[200.0] (wrong arity)
```

Regarding the outputs

The system provides for three output files whose names can be specified in the class included in MainUnified.java:

•   One corresponding to a dump of the state of the BeliefStore by the Petri net and the TR program, each at the completion of each execution cycle.

•   One corresponding to the execution trace of the TR program, indicating the rules that are being activated and possible errors detected both in the loading of the TR program from the input file and in the execution.

•   One corresponding to the evolution of the Petri net marking and possible incidents that may occur during implementation.

It is possible to have all outputs displayed by indicating it in the main program.

It is also possible to run only the Petri net or only the TR program for debugging purposes with outputs in the console, for this purpose, we have included two files with the main method for each part.

By default, it is in main where the outputs resulting from the execution as a RIPN system are shown: discrete actions triggered and durative actions started or stopped. All other things that happen in the system are plotted in the files mentioned above because they do not correspond to the effect of the system execution. For a real system deployment, those messages that are received in main as the observer object that it is, must be passed on to the corresponding physical system devices and actuators, making the appropriate modifications to the code. Having decoupled it following the observer pattern, the change is as simple as passing a different observer to both the network and the TR program instead of being the object included in the main method.

Architecture of the RIPN framework in Java

The program has been developed in Java and can be extended as desired. The packages are shown below, with details of the role played by each of the included Java classes. The PN subpackage for Petri nets includes the MainPN.java class, which includes the main method for running a Petri net separately. The same applies to the TR package, which allows a TR program to be run in isolation. This can facilitate testing before integrating the entire system.

- ∨ 🗄 src
  - ∨ ⊞ both
    - 〉 🗐 LoggerManager.java
    - 〉 🗐 MainUnified.java
  - ∨ ⊞ bs
    - 〉 🗐 BeliefStore.java
    - 〉 🗐 Observer.java
  - ∨ ⊞ guiEvents
    - 〉 🗐 EventPool.java
    - 〉 🗐 GUIEvents.java
  - ∨ ⊞ pn
    - 〉 🗐 Arc.java
    - 〉 🗐 BeliefStoreLoader.java
    - 〉 🗐 ExpressionEvaluatorPN.java
    - 〉 🗐 MainPN.java
    - 〉 🗐 PetriNet.java
    - 〉 🗐 PetriNetAnimator.java
    - 〉 🗐 PetriNetLoader.java
    - 〉 🗐 Place.java
    - 〉 🗐 Transition.java
  - ∨ ⊞ tr
    - 〉 🗐 Action.java
    - 〉 🗐 DiscreteAction.java
    - 〉 🗐 DurativeAction.java
    - 〉 🗐 ExpressionEvaluator.java
    - 〉 🗐 MainTR.java
    - 〉 🗐 ParameterizedAction.java
    - 〉 🗐 TRParser.java
    - 〉 🗐 TRProgram.java
    - 〉 🗐 TRRule.java
- 〉 📚 Referenced Libraries
- 📄 log_BS.txt
- 📄 log_PN.txt
- 📄 log_TR.txt
- 📄 RIPN_PN.txt
- 📄 RIPN_TR.txt

Java Class Responsibilities in the TR Program Execution Framework

The TR (Teleo-Reactive) program execution system is composed of several coordinated Java classes, each responsible for a specific part of the program parsing, rule evaluation, action execution, and belief state management. The following describes the main roles of each class in the system:

1. MainTR.java
This class serves as the entry point for executing a TR program. It is responsible for initializing the BeliefStore, parsing the input TR file, configuring the cycle delay, and starting the main execution thread of the TRProgram. It ensures the system remains active, allowing the TR program to continuously evaluate and respond to

changes in the belief state.

2. TRParser.java

The TRParser class is responsible for parsing the TR program file. It reads the declaration blocks (facts, variables, actions, timers, and initial values) and constructs the corresponding internal representations. Additionally, it extracts the TR rules and builds a TRProgram instance that contains all parsed rules and metadata.

3. TRProgram.java

This class encapsulates the TR program logic. It holds the list of rules (TRRule instances), the belief store, timer management, and the execution cycle. The run() method in this class defines the reactive loop: it continuously evaluates the rules based on the current belief state and executes the first applicable rule, updating the state accordingly.

4. TRRule.java

Each TR rule is represented as an instance of the TRRule class. It stores:

- A logical condition (evaluated against the belief state),
- A list of discrete actions to be executed when the rule fires,
- An optional list of updates to be applied to the BeliefStore. This class provides methods for evaluating whether the rule is applicable and for executing its associated actions.

5. BeliefStore.java

The BeliefStore represents the dynamic state of the system. It maintains the currently active facts (both parameterized and unparameterized), as well as declared integer and real variables. It provides methods for:

- Declaring and checking facts or variables,
- Retrieving and setting variable values,
- Remembering or forgetting facts,
- Dumping the current belief state for logging purposes.

6. Action.java *(abstract class)*

This is the base class for all action types. It defines the common structure for actions and includes utility methods such as parameter parsing and logging. Subclasses inherit from it to implement specific behavior.

7. DiscreteAction.java

This class handles discrete (instantaneous) actions. It evaluates and executes actions declared in the TR program, which are triggered when a rule fires. Execution may include logging, variable manipulation, or symbolic communication with external systems.

8. DurativeAction.java

This class models durative actions, which remain active over time. It manages the start and stop of such actions depending on the changes in the belief state (e.g., whether the condition that activates the action remains true or not).

9. ParameterizedAction.java

This class generalizes actions that take typed parameters. It is used internally by both DiscreteAction and DurativeAction to manage action declarations, parameter arity, and consistency checks with the action invocation syntax.

10. ExpressionEvaluator.java

This utility class handles the evaluation of logical expressions and assignments used in rules. It relies on a dynamic expression language (e.g., MVEL) to parse and evaluate rule conditions and variable updates at runtime.

Petri Net Java class roles

1. MainPN.java

This is the entry point for executing the Petri Net system in standalone mode. It:

- Loads a Petri Net specification from a file.
- Initializes the associated BeliefStore and loads variables, facts, and actions.
- Starts a PetriNetAnimator thread to execute the simulation cycle.

2. PetriNet.java

This class represents the entire Petri Net model. It contains:

- All Place and Transition objects, along with arcs.
- Logic for checking if transitions can fire, executing transitions, and updating the state (especially marking and variable changes).
- Handling of durative and discrete actions, including interactions with Observer and BeliefStore.

3. PetriNetLoader.java

Parses the input Petri Net file to:

- Create places, transitions, arcs, and initial marking.
- Associate logic and actions (e.g., conditionals, when event triggers) with places and transitions.
- Register declared events and their specifications in EventPool.

4. PetriNetAnimator.java

Controls the execution loop:

- Repeatedly checks for enabled transitions and fires them according to priority (immediate first).
- Evaluates transition conditions (if) and event triggers (when) using data from BeliefStore and EventPool.
- Coordinates with the Observer to handle actions and logs the system state periodically.

5. BeliefStoreLoader.java

Handles loading of:

- Declared facts and variables (integers and reals).
- Initial variable assignments.
- Action declarations (discrete and durative).
- Event specifications for integration with the EventPool.

6. ExpressionEvaluatorPN.java

Evaluates logical conditions and arithmetic expressions in the context of a Petri Net:

- Used to compute expressions inside transition if and action [...] blocks.
- Supports parameter substitutions from events or variable lookups in BeliefStore.

7. Place.java

Represents a place in the Petri Net:

- Holds a token state (true or false).
- Used to determine the enabled state of transitions and to track marking.

8. Transition.java

Represents a transition:

- Stores its name, associated event triggers (when) and parameter variable bindings.
- Maintains a list of variables that are populated when an event with parameters is consumed.
- Used for checking eligibility for firing in PetriNetAnimator.

9. Arc.java

Represents a directed arc:

- Links places to transitions and vice versa.
- Supports topology validation and firing logic.

BS package class roles

BeliefStore.java

The BeliefStore class functions as the central knowledge base for both the Petri net and the Teleo-Reactive (TR) program. It maintains the current state of symbolic and numeric information in the system. Key responsibilities include:

- **Variable Management**: Supports declaration and dynamic assignment of integer (int) and real (double) variables. Each variable is stored with its current value and can be queried or updated.
- **Fact Management**: Manages both parameterless facts (simple flags) and parameterized facts (e.g., see(2)), allowing logic-driven conditions to be evaluated against the system state.
- **Action Registration**: Differentiates between discrete and durative actions, storing which actions are declared and available in the system.
- **Timer Support**: Enables declaration, activation, and expiration tracking of timers, which can be queried in logical expressions (e.g., t1.end).
- **State Dumping**: Provides a dumpState() method for outputting the current internal state, used for debugging or trace logging.
- **Thread-Safety**: Synchronization is used where necessary to support concurrent access from multiple threads (e.g., the Petri net thread and the TR thread).

Observer.java

The Observer interface defines a contract for external components to receive notifications about action executions during simulation. It allows the system to be extended or monitored in real time. The methods include:

- onDiscreteActionExecuted(String actionName, double[] parameters): Called when a discrete action is triggered.
- onDurativeActionStarted(String actionName, double[] parameters): Called when a

durative action starts execution.

- onDurativeActionStopped(String actionName): Called when a durative action is stopped due to a place losing its marking.

This interface decouples the core logic from the external response to actions, allowing flexible extensions such as GUI updates, logging, or integration with physical systems.

GuiEvents package class roles

Class: EventPool

The EventPool class provides a centralized, thread-safe mechanism for managing event instances within the system. It implements the singleton design pattern to ensure only one global instance is accessible throughout the application. Events are first registered with metadata (duration and parameter types) and can later be added from external sources like the graphical interface.

Each active event is stored with its parameters and an expiration time, depending on its declared time-to-live (TTL). The pool also supports event consumption (with automatic removal), removal of expired events, and tracking of invalid or unrecognized events (e.g., due to undeclared names, wrong arity, or type mismatches). All significant interactions, including additions, deletions, and expirations, are logged to a file (log_EV.txt) with timestamps.

The pool also offers maintenance features such as clearing all events or unattended ones and printing the list of declared events for debugging or audit purposes. Two nested classes, EventSpec and EventInstance, encapsulate the specification and runtime instance details respectively.

Class: GUIEvents

The GUIEvents class implements a simple Java Swing-based graphical user interface (GUI) for injecting external events into the system at runtime. It creates a windowed form that allows the user to specify an event name and up to three optional parameters (of numeric type), and to send the event to the system.

Each row in the grid layout contains fields for event name and parameters, along with a "Send" button. When pressed, the button parses the input values and passes the resulting data to the EventPool. There is also a dedicated "Delete All Events" button for clearing the entire pool.

This interface is useful for simulating interactions or conditions during execution and facilitates manual testing and validation of event-driven behavior in the system.

Both package class roles

LoggerManager.java

Purpose:
LoggerManager is a centralized utility class responsible for managing the system's logging behavior. It supports flexible output to either a file or the console and ensures timestamps are consistently included when required.

Key Features:

- **Dual Output Modes:**
  The logger can write to either a file or to the screen, depending on the file_or_screen flag provided during instantiation.

- **Timestamped Logging:**
  Messages can be prepended with precise timestamps (to the millisecond) using Java's DateTimeFormatter, which is useful for tracking system behavior over time.

- **Thread Safety:**
  The log method is synchronized to ensure that concurrent threads (e.g., from the Petri Net and TR program) do not interfere with each other's log entries.

- **Immediate Flushing:**
  The logger flushes after every write when outputting to a file, ensuring that logs are always up-to-date even in case of system crashes.

- **Error-Resilient Initialization:**
  If a log file fails to open, the constructor reports the error to the standard error stream without crashing the system.

  Use Case: Each subsystem (Petri Net, TR Program, BeliefStore) uses its own instance of LoggerManager, ensuring separation of concerns and traceability across different log files such as log_PN.txt, log_TR.txt, and log_BS.txt.

MainUnified.java

Purpose:
MainUnified acts as the orchestrator of the complete RIPN system. It is responsible for launching and coordinating both the **Petri Net system** and the **Teleo-Reactive (TR) program**, and it serves as a unified **observer** for action monitoring.

Key Responsibilities:

- **Startup Configuration:**
  Using ENABLE_PN and ENABLE_TR flags, users can specify which parts of the system should be launched (Petri Net, TR program, or both).

- **Shared BeliefStore Setup:**
  A single BeliefStore is shared across both subsystems to facilitate synchronized state management.

- **Logger Injection:**
  Three distinct LoggerManager instances (loggerPN, loggerTR, loggerBS) are created and passed to the corresponding subsystems for file-based logging of trace data.

- **Component Launch:**

  o The Petri Net is loaded from a file, its places, transitions, conditions, and actions

are configured, and a GUI is launched for manual event injection.

- The TR Program is parsed from a file and begins execution in a dedicated thread, looping indefinitely with a delay between cycles.

- **Action Observation:**
Implements the Observer interface, allowing it to receive and print notifications when discrete or durative actions are executed or stopped. These notifications are shown on the console using timestamps.

- **Error Management:**
Any exceptions during the startup of either system are logged appropriately, and a clear termination message is logged if either system encounters a fatal issue.

Work in progress

You can cite RIPN framework by using the DOI 10.5281/zenodo.15088988.

If you have any comments or questions, please write to **pedro.sanchez@upct.es**, Universidad Politécnica de Cartagena, Spain.