Uppaal. The Model Checker.

Patryk Kiepas

March 21, 2016

Uppaal. What is it?

Uppaal is a model checker for real-time systems (in mind of embedded systems). What we can do with it?

1. Modeling
2. Simulation
3. Verification

Internal representation of model consists of:

- Network of timed automata
- Extended with data types

Where to use?

"Any system can be analyzed using a model checker, as long as it has *states* and *transitions* between states" (from Chapter 1: A First Introduction to Uppaal by Frits Vaandrager)

Reactive systems such as:

- Hardware components
- Embedded controllers
- Network protocols
- Others...

Whenever there is need to handle real-time issues (the timing of transitions).

Brief history

Uppaal was started by Uppsala University, Sweden and Aalborg University, Denmark.
Timeline of development:

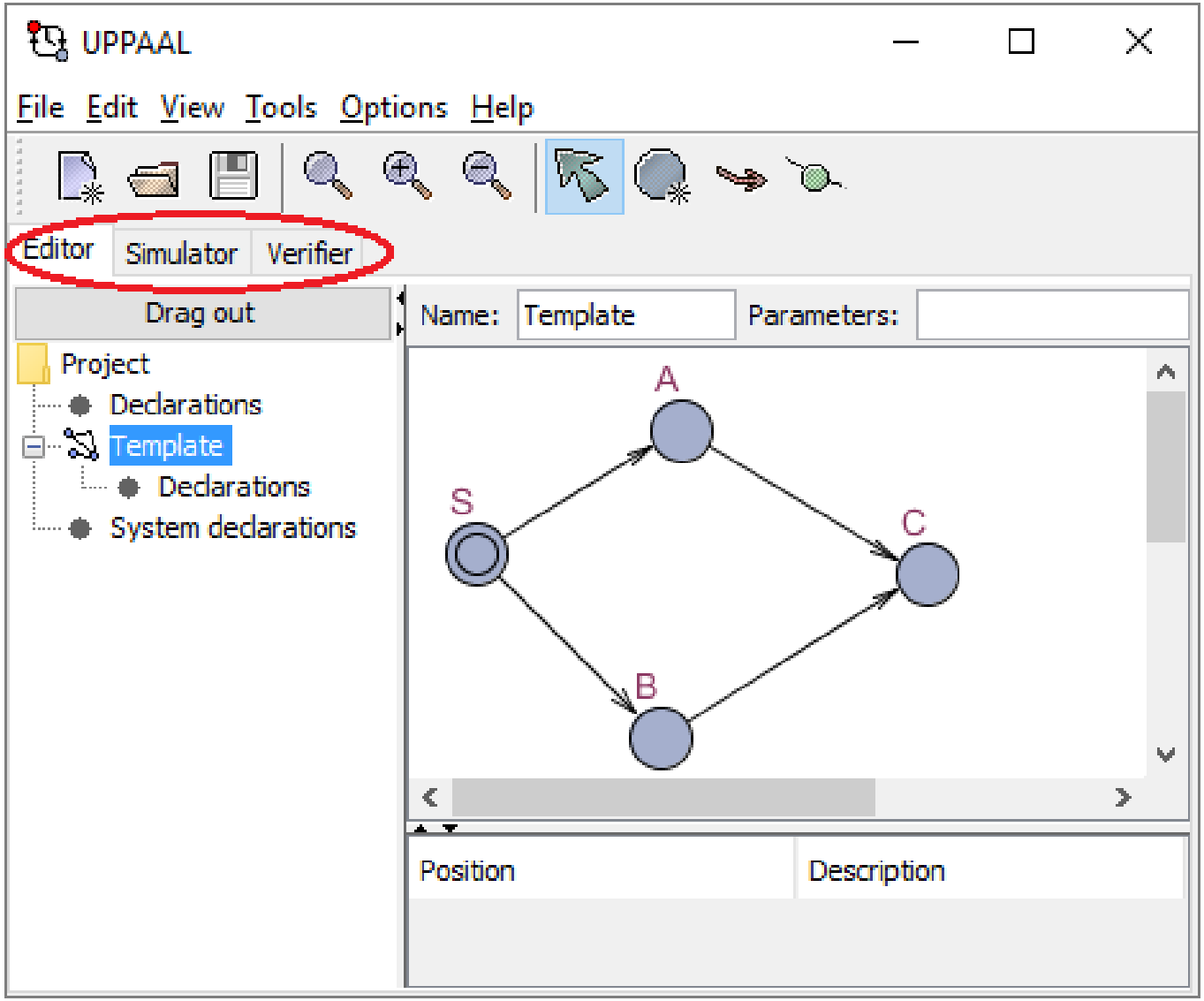
- 1995 - project started
- 1999 - first beta
- 1999/2000 - first stable release (v 3.0.X)
- September 27, 2010 - latest stable release (v 4.0.13)
- July 1, 2014 - preview release (v 4.1.19)

Uppaal variations

Versions: Windows, Mac, Linux, 32/64 bits
Available licenses:

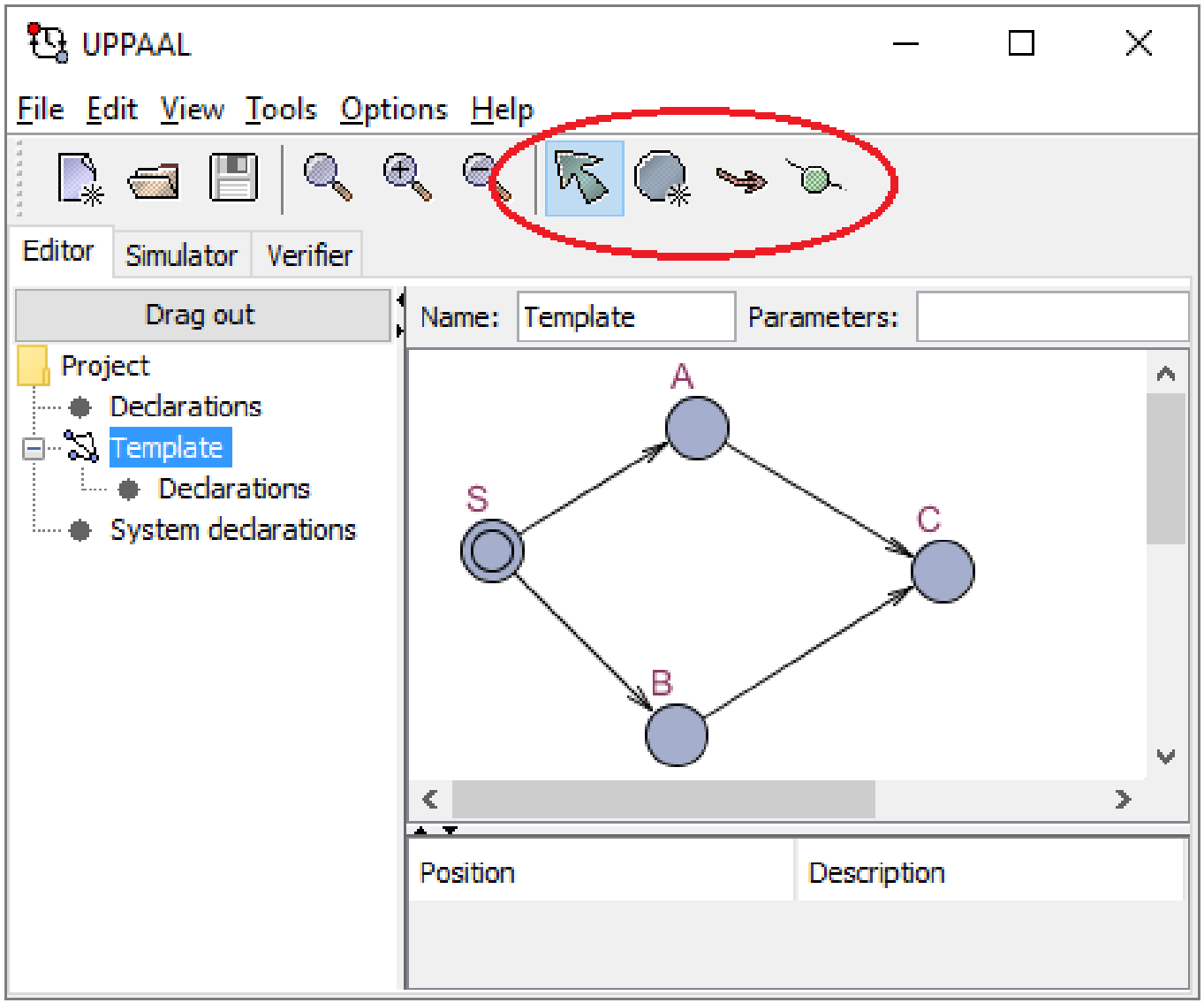
- Academic use (more info: http://www.uppaal.org/)
- Commercial use (more info: http://www.uppaal.com/)

Uppaal GUI - Main parts



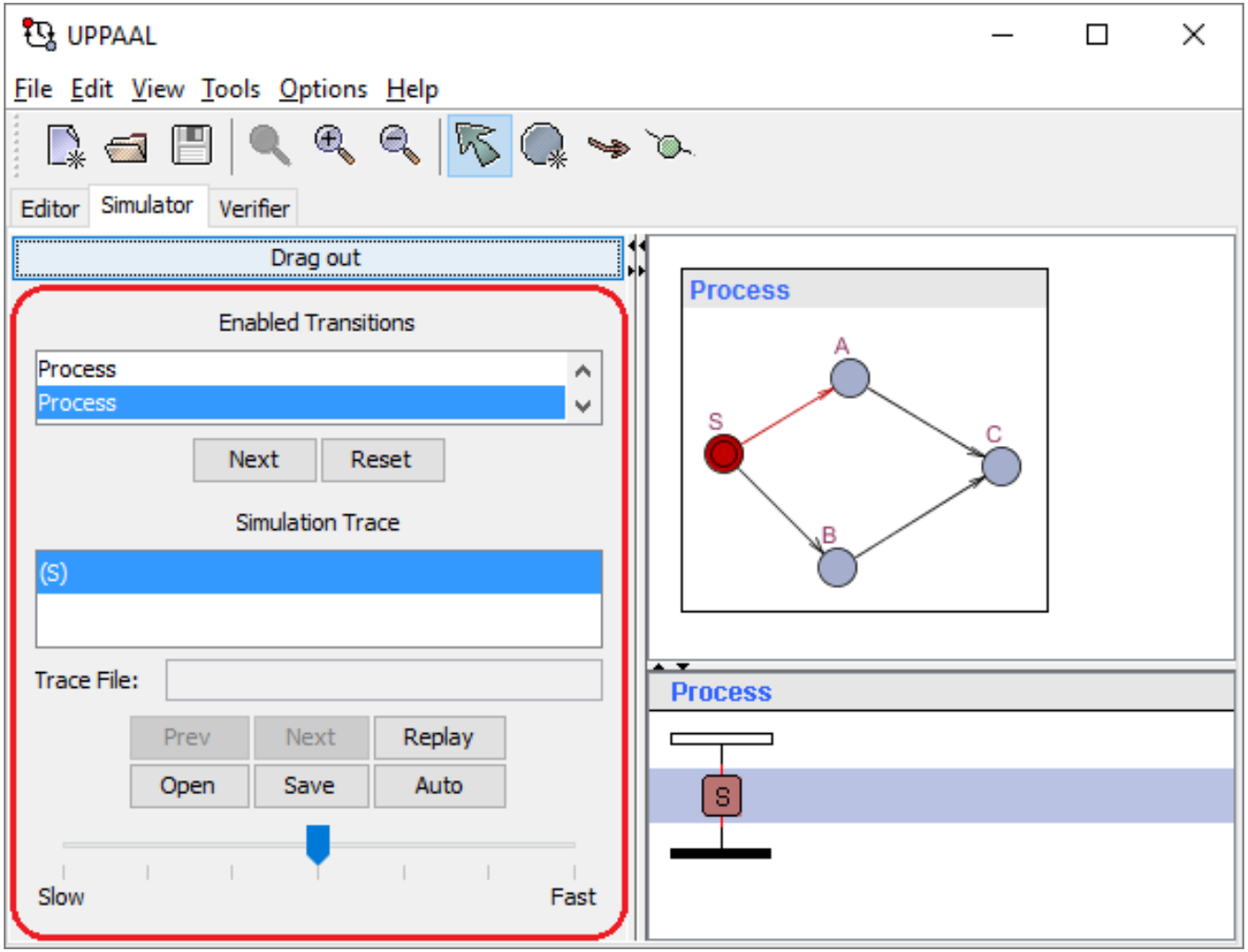
- System editor
- Simulator
- Verifier

Uppaal GUI - System editor



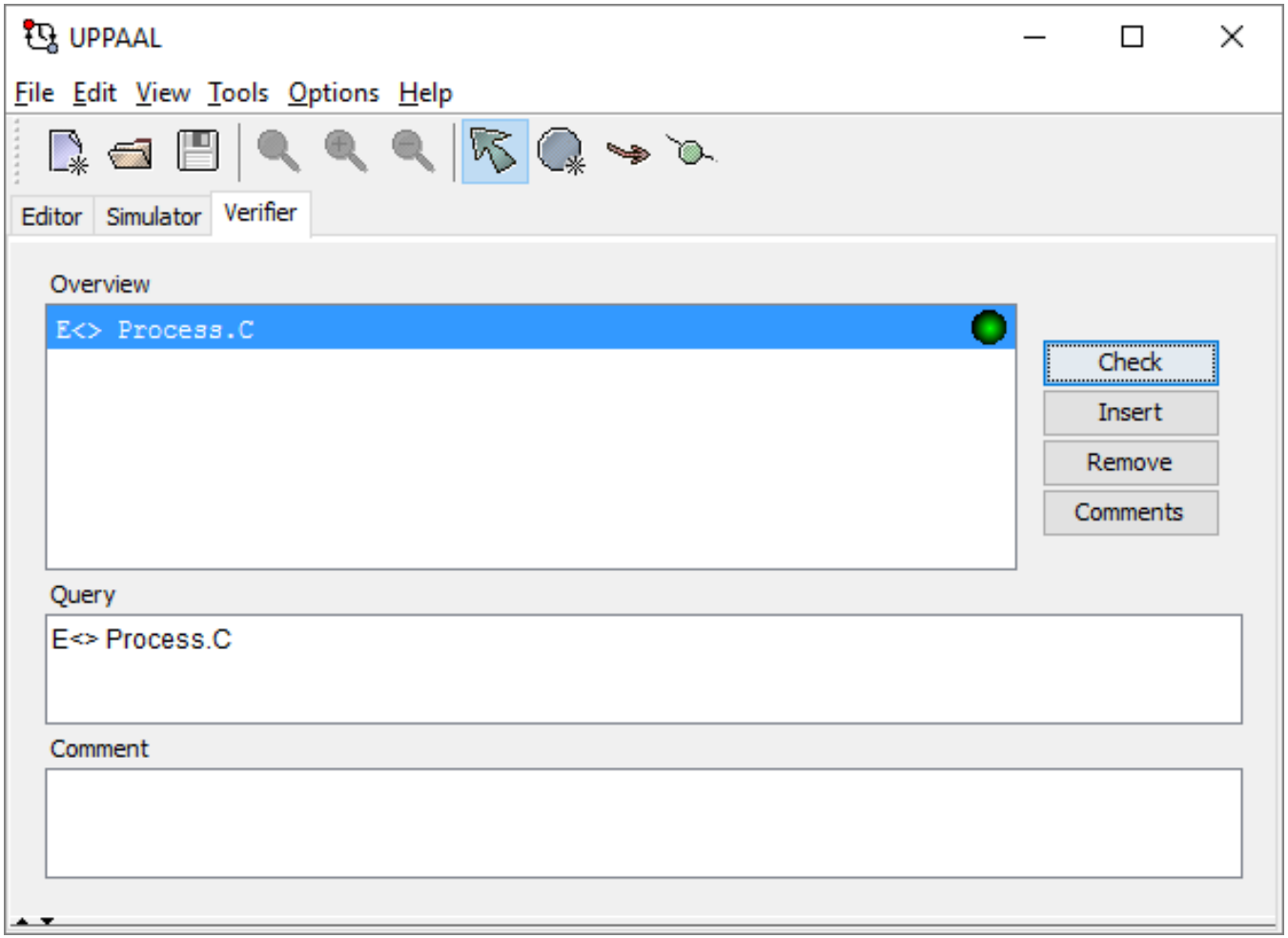
- Name: Template (defualt)
- Select
- Add location
- Add edge
- Add nail
- Syntax check (for global, local, system declarations)

Uppaal GUI - Simulator



- Select transition
- Track simulation
- Control (Prev/Next/Auto)
- Visualization

Uppaal GUI - Verifier



- Query editor
- Check query
- Overview
- Save/load

Project

All projects consist of:

- Global declaration for whole project
- Processes modelled using timed automata
- Local declaration for each process
- System description

Model/system

A Uppaal model (called system) is defined as a composition of a number of basic components (called automata or processes). Automata are diagrams with states (called locations) and transitions between states (called edges).

Process/automata

Each automaton has at most one initial location, marked by a double circle.

Location/state

System declaration

Edge/transition

Choices for which the model does not specify how they are resolved are called nondeterministic.

Channels

In order to model the synchronization between tools and jobbers, we use the notion of (synchronization) channels from Uppaal. Once a has been declared as a channel, transitions can be labeled with either a! or a?. This can be done by double clicking (within the Editor) transitions with the Select tool, and then writing a! or a? in the Sync field. When two automata synchronize on channel a, this means that an a! transition of one automaton occurs simultaneously with an a? transition of another automaton. An a! or a? transition can never occur on its own: a! always has to synchronize with a?, and vice versa. If there is one automaton S that can do an a! but two automata R1 and R2 that can do an a?, then there is a nondeterministic choice and S can synchronize with either R1 or R2. The other automaton has do something else or has to wait until the next a! synchronization will be offered.

Variables

Verification

Queries (part I)

Query is a property that may or may not hold for a given model. The Verifier can establish whether a Query is **satisfied** or **not**.
Notation:

- $A[]$ - for all reachable states...
- $E <>$ - there exists a reachable state...

Simple queries are in form of $A[]e$ and $E <> e$ where e is an expression build from **boolean combination** of **atomic propositions**.

Queries (part II)

Atomic propositions can be of the form $A.c$, for A an automaton and c a location. Such a proposition is true in a global state of the model if in this state automaton A is in location c.

Expression	**Name**	**True when...**
e && f	and	e and f evaluate to **true**
e || f	or	e or f evaluate to **true**
e == f	equality	e and f evaluate to the same value
e *imply* f	implication	e evaluate to **false** or f evaluate to **true**
e *not* f	negation	e evaluates to **false**

Query examples

- $A[]$ not deadlock
- $E <>$ Process_1.C
- $E <>$ (Process_1.C && Process_2.C)

Reasoning

(!)The tool just uses brute force to explore all the reachable global states of the model and to check for each of these states whether both jobbers are working on a hard job.

Diagnostic traces

(!)Uppaal can also provide a concrete example that illustrates why the property holds (for $E <>$ properties) or not (for $A[]$ properties, so called: *counterexample*).

(!)In the case of $E <>$ properties that do not hold, or $A[]$ properties that hold, Uppaal can only report that it exhaustively checked all the reachable states of the model and didnt find anything.

(!)We choose under *Options* the entry *Diagnostic Trace* and then select the option *Shortest*. Simulator will then show found diagnostic trace.

Saving data

We can save various data:

- Model/system - **.xml* file
- Queries - **.q* file
- Traces (binary) - **.xtr* file