CEDRIC Laboratory
Conservatoire National des Arts et Métiers
http://quasar.cnam.fr/

QUASAR

A Tool For Automatic Verification Of Concurrent Programs

Team :
Olivier Alzeari
Sami Evangelista
Claude Kaiser
Christophe Pajault
Jean-François Pradat-Peyre
Pierre Rousseau
Nicolas Treves

USER–VIEW

INTERNAL–VIEW

Concurrent Software Model-Checking

Quasar is an **automatic concurrent software analysis tool** based on this promising method that uses the application source code for generating and validating a semantic model (a **high-level colored Petri net**).

Quasar is presented to the user as a simple tool although it is in reality a composition of tools that can be used independently. Quasar takes as input a **concurrent program** with a **property specification**. If the property is not verified by the program, Quasar displays a **faulty sequence of actions** leading to the property violation. Note that no knowledge about Petri nets is required by the user.

Quasar follows a four step process :

❶ Slicing: the aim of this step is to remove parts of the source code which are not related to the investigated property.

❷ Modeling: translation of the sliced program into a high-level Petri net.

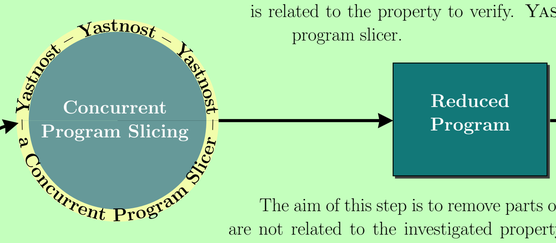
❸ Model-Checking: Analysis of the model by combining structural techniques (like Petri nets reductions) and finite state verification methods (like temporal logic formula verification).

❹ Error-reporting: when the target property is not verified, the state in which the application is faulty is displayed and a report indicates the sequence of program actions that ends by invalidating the property.

Thus, if the specified property is violated, a graphical error-report with the sequence of actions that leads to this violation is displayed. Our graphical error-report is based on the **XUL** language.

Concurrent Program Slicing With Yastnost

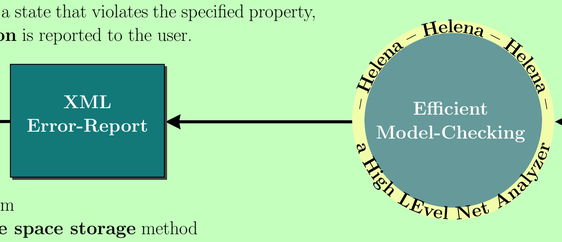
The **slicing** step consists in an **automatic extraction** of the input source code which is related to the property to verify. Yastnost is a concurrent program slicer.



The aim of this step is to remove parts of the source code which are not related to the investigated property. This contributes to **generate a smaller model** compared to the one corresponding to the whole program, but which has the **same behavior according to the investigated property**.

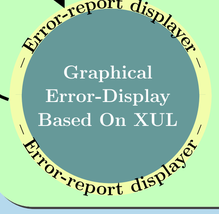
Model-Checking With Helena

Model checking is an automatic technique to **verify properties of finite state systems** by inspecting all the possible configurations of the system. In the current version, Helena can be used for the verification of **state properties** and **deadlock freeness**.
When Helena finds a state that violates the specified property, the **faulty execution** is reported to the user.



Main features:
- High level formalism
- **Optimized state space storage** method
- The **stubborn set** method (since version 1.0.1)
- **Distributed-memory** Model-Checking
- Implementation of **structural abstractions** techniques (transitions agglomerations)

Graphical Error-Report Displaying

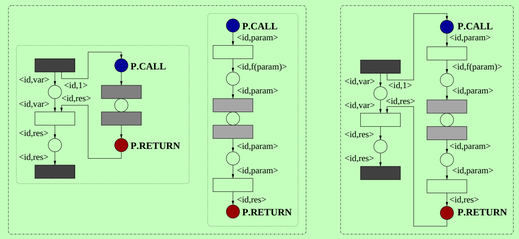


Once the XML error-report is built, our graphical error-report displayer.
- **XML**-based error report input
- Easy-to-read graphical display with **XUL**

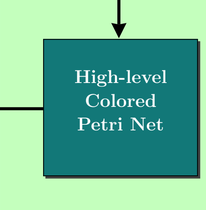
Program Translation Into A High-Level Colored Petri Net

This step translates the simplified (sliced) program into a **formal model**. The target formalism used is **colored Petri nets** because their analysis may combine several techniques that are supported by experienced tools and that we continue to develop successfully.

For translating a program into a Petri net we use **patterns**. Each element of the language has a corresponding pattern (or **sub-net**). These sub-nets allow a **hierarchical construction** since meta-nets can be used to abstract other (list of) sub-net(s).



Example of meta-net composition.



The single final net corresponding to the whole program is produced using two basic constructors : the **substitution** (that substitutes a meta-net by a concrete sub-net) and the **composition** (that merges two different sub-nets into an unique one).

This is the pattern for a *loop* statement. Transitions *Then* and *Else* are protected by a guard meaning that they cannot be fired unless the condition is true. Statements of the loop are abstracted by a **meta-net**.