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

Concurrent Software Model-Checking

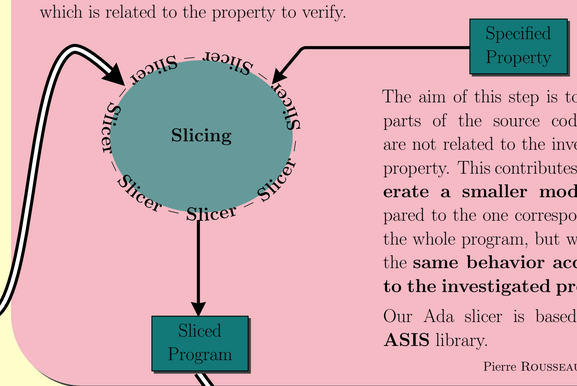
Quasar is an **automatic concurrent program analysis tool** based on this promising method that uses the application source code for generating and validating a semantic model (a **High-level colored Petri net**). Quasar follows a four step process :
❶ **Slicing**: the aim of this step is to remove parts of the source code which are not related to the investigated property.
❷ **Modelling**: 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.

Quasar is currently able to cover most of the concurrency features and a large part of the **Ada** language. It shows good performances when analyzing the concurrency part of non-trivial application programs. This reflects the fact that, although intrinsically more complex, unfamiliar and prone to errors than most other aspects of programming, the part of an application program which is concerned by concurrency remains hopefully small. The slicing step of Quasar (the first step) eliminates this large part of the application program which is not concerned with concurrency. And moreover the efficiency of the reduction techniques and state verification methods of the third step of Quasar succeeds in restraining the combinatorial explosion of states.

Ada has been chosen as targuet language since it is normalized, its concurrency semantic is well defined, many decisions are taken at compile time (this reduces useless combinatorics), it allows a priori and static tuning decisions which are required in reliable software applications. The results and techniques gained with Ada will allow developping similar verifications tools for other concurrent languages as **Java** or concurrent standards as **Posix**.

Efficient program slicing

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



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**.

Our Ada slicer is based on the **ASIS** library.

Pierre Rousseau

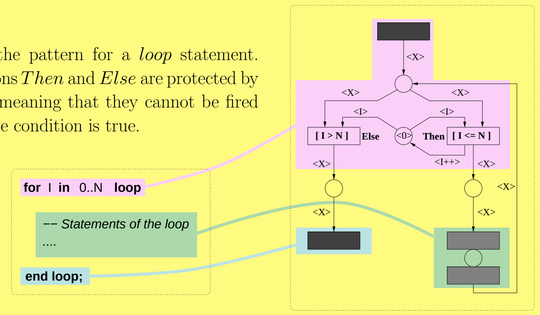
Translation into a high-level colored Petri net

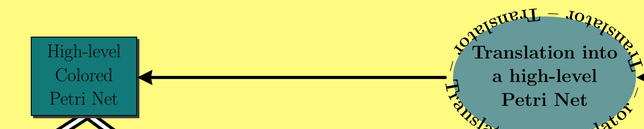
This step translates the simplified (sliced) program into a **formal model**. The target formalism used is **colored Petri nets** because their analysis may combine several techniques that are supported by experienced tools and that we continue to develop successfully. For translating a program into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

For example, the sub-net of the loop statement contains a meta-net abstracting the statements of the loop. The final net corresponding to the whole (simplified) 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 final single Petri net can then be exported to several tools such as Prod, Maria or our specialized model-checker Helena.

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.



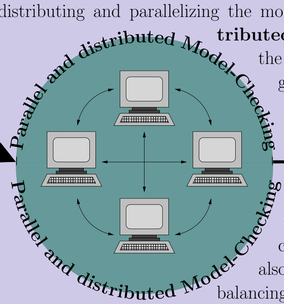


❷ Translation

❸ Model-Checking

Parallel and distributed Model-Checking*

In order to cope with the combinatory explosion problem, we are working on distributing and parallelizing the model-checking process. Based on an **distributed memory architecture**, each node of the network builds a part of the reachability graph and communicates with other nodes *via* **message exchange**.



Our tool allows **load-balancing** to increase efficiency. **Multi-threading** can also be used to increase reactivity during load-balancing or speed-up the verification process of very-high level Petri nets (ie. with complex firing rules).

Christophe Pajault

❹ Error-reporting

Graphical error-report displaying*

- **XML**-based error report input
- Easy-to-read graphical display with **XUL**

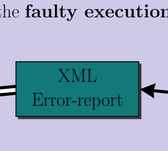
Olivier Alzeari

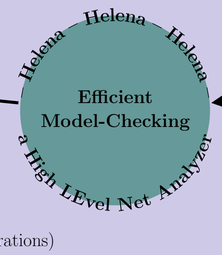
❸ Model-Checking

Model-Checking with Helena

a High LEvel Net Analyzer

Model checking is an automatic technique to **verify properties of finite state systems** by inspecting all the possible configurations (or states) of the system. In Helena, this system is described as a high level Petri net. 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
- Efficient firing rule
- Code generation to speed up the analysis
- Optimized state space storage method
- Implementation of structural abstractions techniques (transitions agglomerations)
- The stubborn set method (since version 1.0.1)

Sami Evangelista

* Work in progress