Combined Static and Runtime Verification of Distributed Objects

Wytse Oortwijn[1] and Wolfgang Ahrendt[2]

[1] University of Twente, the Netherlands - w.h.m.oortwijn@utwente.nl
[2] Chalmers University of Technology, Sweden - ahrendt@chalmers.se

TOOLCHAIN INPUT

Our approach targets real industrial distributed Java programs

The toolchain takes distributed Java programs as input, like cloud- and web services, written with the Active Objects design pattern using the ProActive programming platform. In addition, JML-like contracts can be specified for verifying data-flow properties, and runtime monitors described as automata can be specified for verifying control-flow properties.

STATIC VERIFICATION

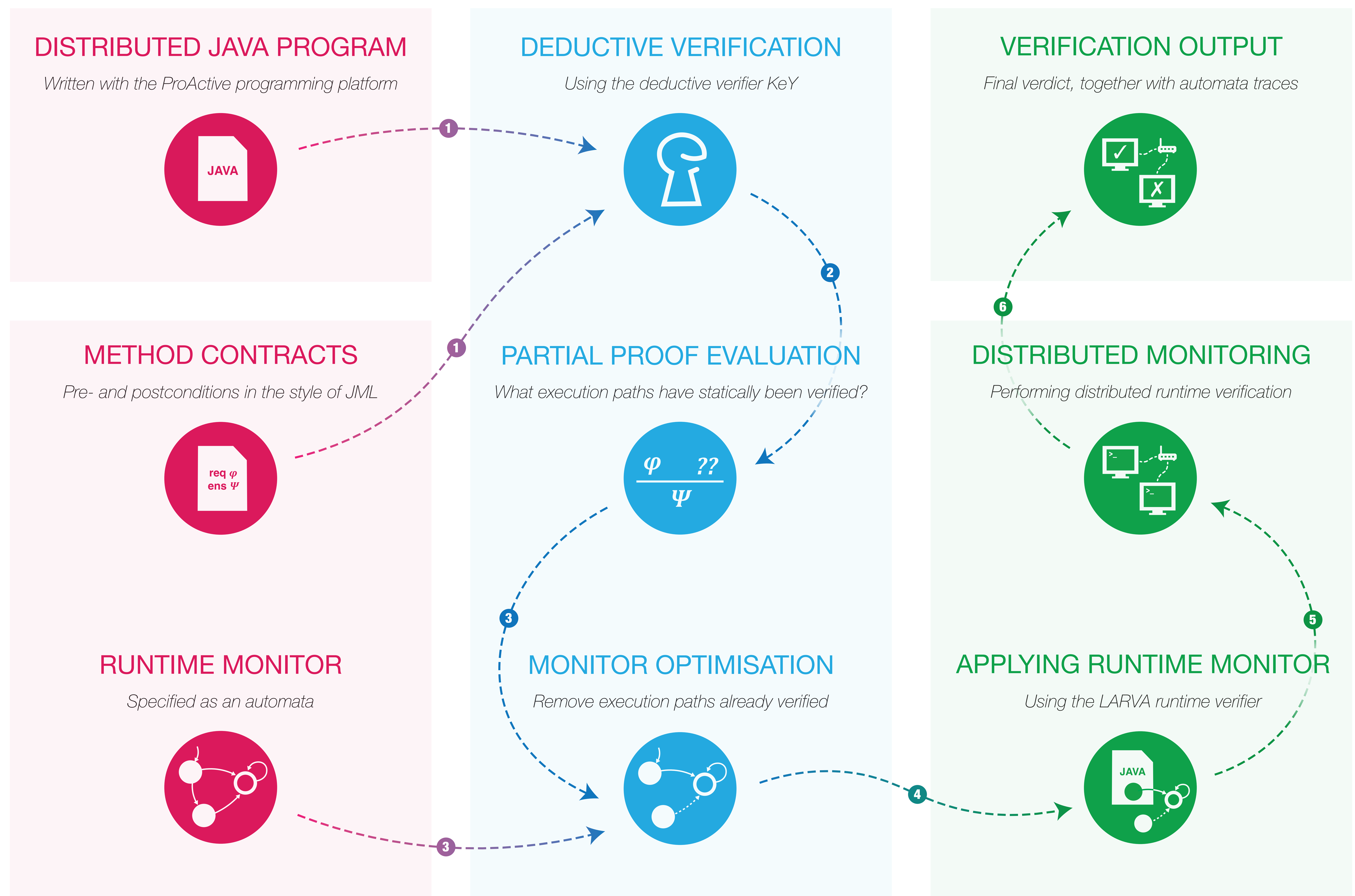
Statically verify data-flow properties and thereby optimise dynamic verification

Users may put any desirable effort into statically verifying data-flow properties, which are specified as Hoare triples in the style of JML. The resulting (partial) proofs are used to optimise the distributed runtime monitors, thereby making dynamic verification more efficient.

RUNTIME VERIFICATION

Dynamically verify control-flow properties of the executions not already proven statically

Dynamic verification of control-flow properties is performed by monitoring each participating node using a separate runtime monitor. Assumption-guarantee reasoning is used in the monitors to efficiently distinguish between violations from the environment and violations from the implementation being monitored, which makes the approach modular and efficient.



1 Deductive verification

Before applying runtime verification, first deductive verification is applied to optimise the runtime monitors. The deductive verifier KeY is used for this, which takes Java source code as input, together with JML-like specifications.

2 Partial proofs

Some Hoare triples may be fully proven by KeY, while others may only partially be proven. In the latter case, the verification attempt may result in partial, non-closed proofs.

3 Partial specification evaluation

The (partial) proofs are evaluated to generate conditions that describe the execution paths that have not yet been statically verified. These conditions are used to optimise distributed runtime verification.

4 Optimising the runtime monitor

The generated conditions are used to optimise and simplify the runtime monitors, by cutting all transitions in the underlying automata that have already been verified statically.

5 Applying the runtime monitor

For runtime verification we use the Larva runtime verifier, which outputs a monitored program by compiling the Java source code and weaving-in the (optimised) runtime monitors via aspects.

6 Distributed runtime verification

The monitored program is executed over a network of nodes, each having a separate runtime monitor. Several monitoring configurations are supported, including distributed monitoring with assumption-guarantee reasoning, but also centralised and orchestrated monitoring.

NWO Netherlands Organisation for Scientific Research
UNIVERSITY OF TWENTE.
CHALMERS UNIVERSITY OF TECHNOLOGY