



Runtime Verification - Monitor-oriented Programming - Monitor-based Runtime Reflection

Martin Leucker

Technische Universität München

(joint work with Andreas Bauer,
Christian Schallhart et. al)

FLACOS

October 10, 2007



Contents

- **Runtime Verification**
 - description, applications
 - theoretical background
- **Monitor-oriented Programming**
 - description, applications
 - theoretical background
- **Monitor-based Runtime Reflection**
 - description, applications
 - theoretical background



Runtime Verification



Never trust a running system



Definition

Verification

Verification comprises all techniques for showing that a system satisfies its specification.

Runtime Verification

Runtime verification deals with those techniques that allow checking whether a system under scrutiny satisfies or violates a given safety property.



Characterization

- It aims to be *lightweight* verification technique
- complementing verification techniques such as
 - model checking
 - testing



Formally

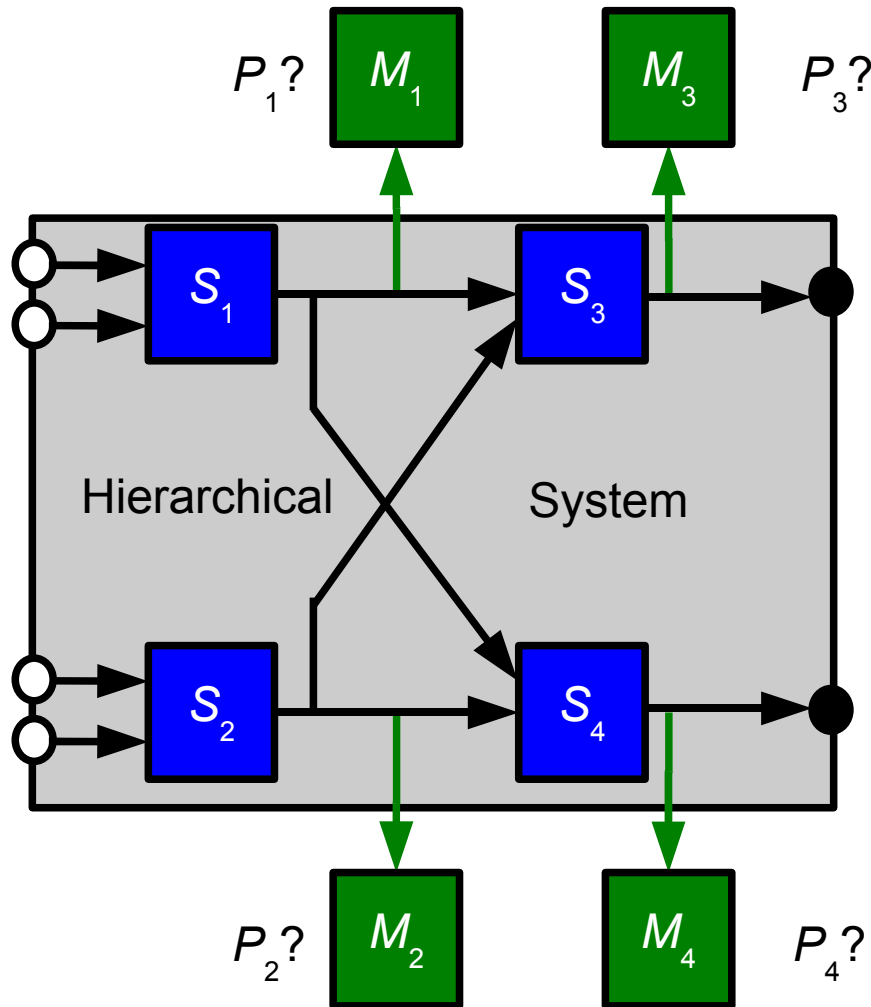
- Let φ be a correctness property
- Let $[[\varphi]]$ denote the set of words conforming to
- Let w be an execution – a word
- In its mathematical essence, runtime verification is answering whether

$$w \stackrel{?}{\in} [[\varphi]]$$

thus the **word problem**.



How it works – Online Monitoring



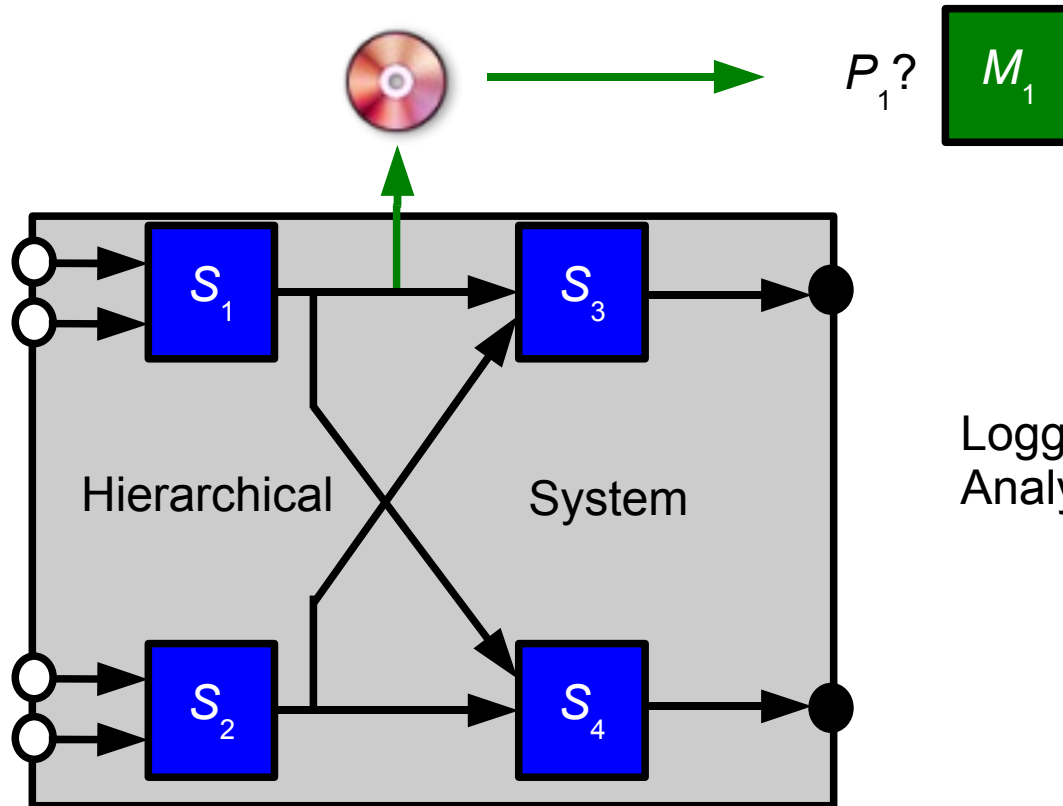
Four monitors,
 M_1, M_2, M_3, M_4 ,
checking independently,
properties
 P_1, P_2, P_3, P_4

S – Components,
Procedures, ...

Properties typically given
in some variant of
Lineartime Temporal Logic
(LTL)



How it works – Offline Monitoring



Logging on Disk
Analysis by Monitor offline



Model Checking

- Formally, model checking asks

$$[[S]] \stackrel{?}{\subseteq} [[\varphi]]$$

which is the language *inclusion problem* and often of much higher complexity

- Model checking consider (usually) *infinite traces*, runtime verification necessarily *finite traces*
- finite but *continuously longer* traces, asks for monitors working in an incremental fashion



Model Checking II

- LTL *semantics for finite traces* needed
- Complexity of monitor (automata) generation not important, but *complexity of monitor*
- Model checking only applicable to white-box systems
Runtime verification *ideal* also *for black-box systems*



Testing (I)

- Testing is usually *incomplete* – like runtime verification
- Test case: finite sequence of input/output actions
- Test suite: finite collection of test cases
- Test execution: check whether output is as expected when input sequence is given to the system



Testing II

- Test case: only (finite) sequence of input actions
- Test suite: finite collection of test cases
- **Test oracle**: monitor checking behavior of the system
- Test execution: give test cases (input sequences) to system, check whether monitor complains

- sounds like runtime verification!



Testing II (cont.)

- Oracle defined manual
Monitor derived from formal specification (LTL formula)
- Research issue in testing:
How to find *good* test suites?
- Research issue in runtime verification:
How to derive *good* monitors?



Runtime verification for LTL

- Syntax of LTL

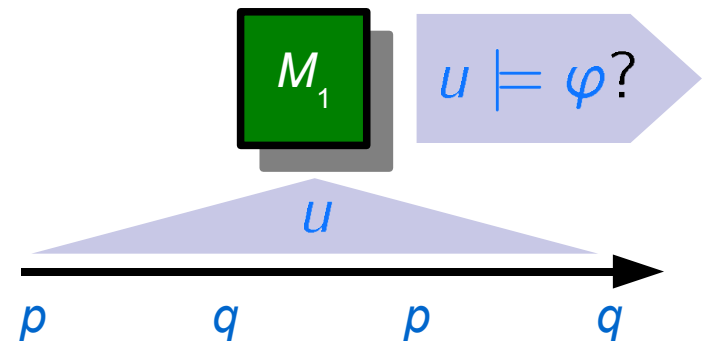
$$\varphi ::= true \mid p \mid \neg\varphi \mid \varphi \text{ op } \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi \quad (p \in AP)$$

- Abbreviations: $F\varphi \equiv true \mathbf{U} \varphi$ $G\varphi \equiv \neg F\neg\varphi$

- Some examples: $\mathbf{G}p$, $\mathbf{G}(\neg p \rightarrow \mathbf{X}p)$, $\mathbf{G}\neg(p \wedge q)$

- Interpretation of φ over sequence of system events (behaviours), $u \in (2^{AP})^*$

- Automatic monitor generation: “Inspired” by translation of LTL to Büchi-automata



$$\varphi \rightarrow BA_\varphi \text{ s.t. } L(BA_\varphi) = L(\varphi)$$

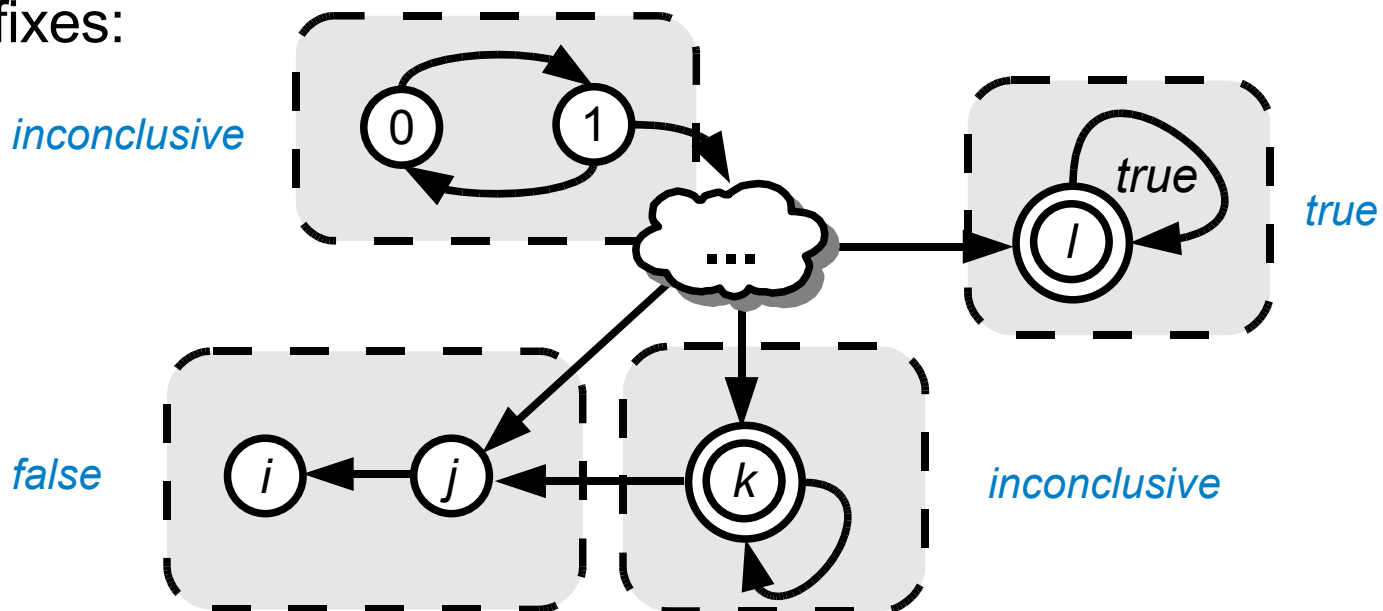


LTL over finite traces

- Introduce 3-valued semantics for LTL (resembling the 2-valued one):

$$[u \models \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\ \perp & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\ ? & \text{otherwise} \end{cases}$$

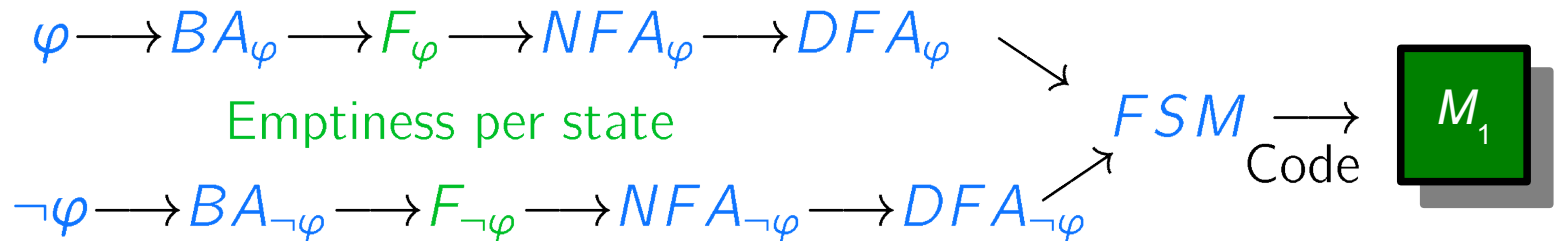
- The goal – finite-state machines for detecting *minimal* bad prefixes:





How it actually works

- Use emptiness-per-state as alternative acceptance condition:

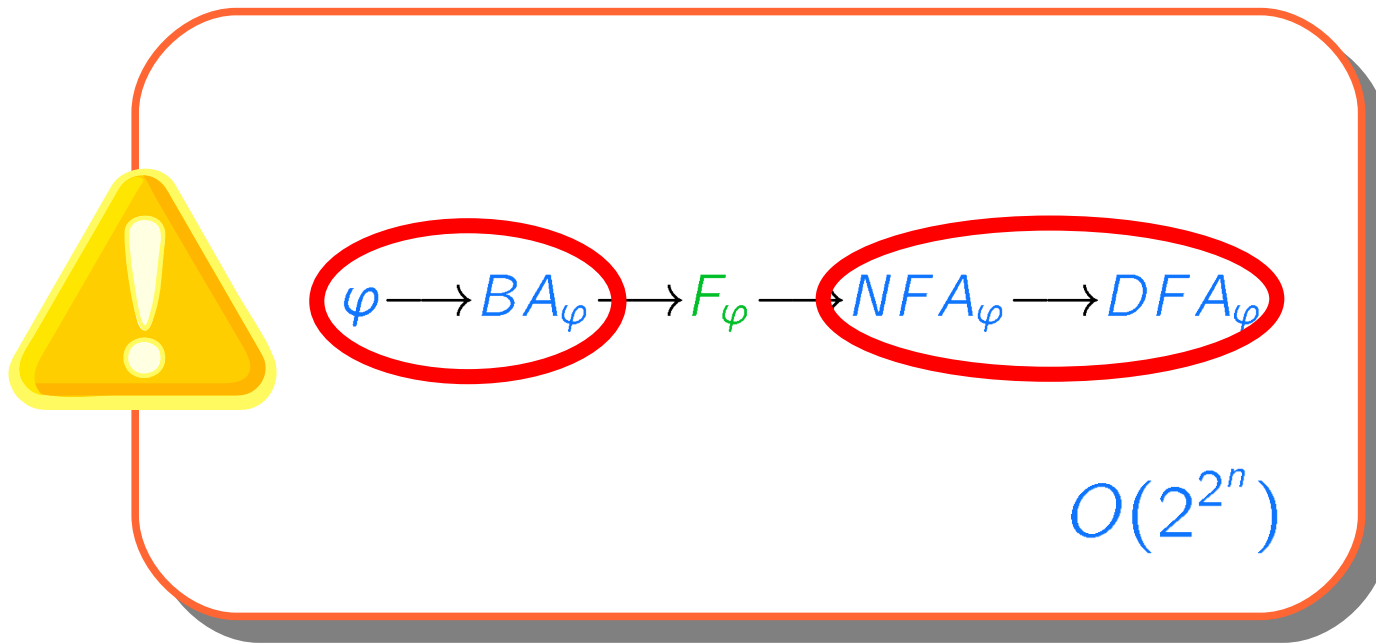


- Emptiness only says whether language is non-empty, not whether all future traces are accepting or not!
- Now we have a formal decision procedure:

$$[u \models \varphi] = \begin{cases} \top & \text{if } u \notin L(NFA_{\neg\varphi}) \\ \perp & \text{if } u \notin L(NFA_{\varphi}) \\ ? & \text{if } u \in L(NFA_{\varphi}) \text{ and } u \in L(NFA_{\neg\varphi}) \end{cases}$$



Complexity





But optimal solution...

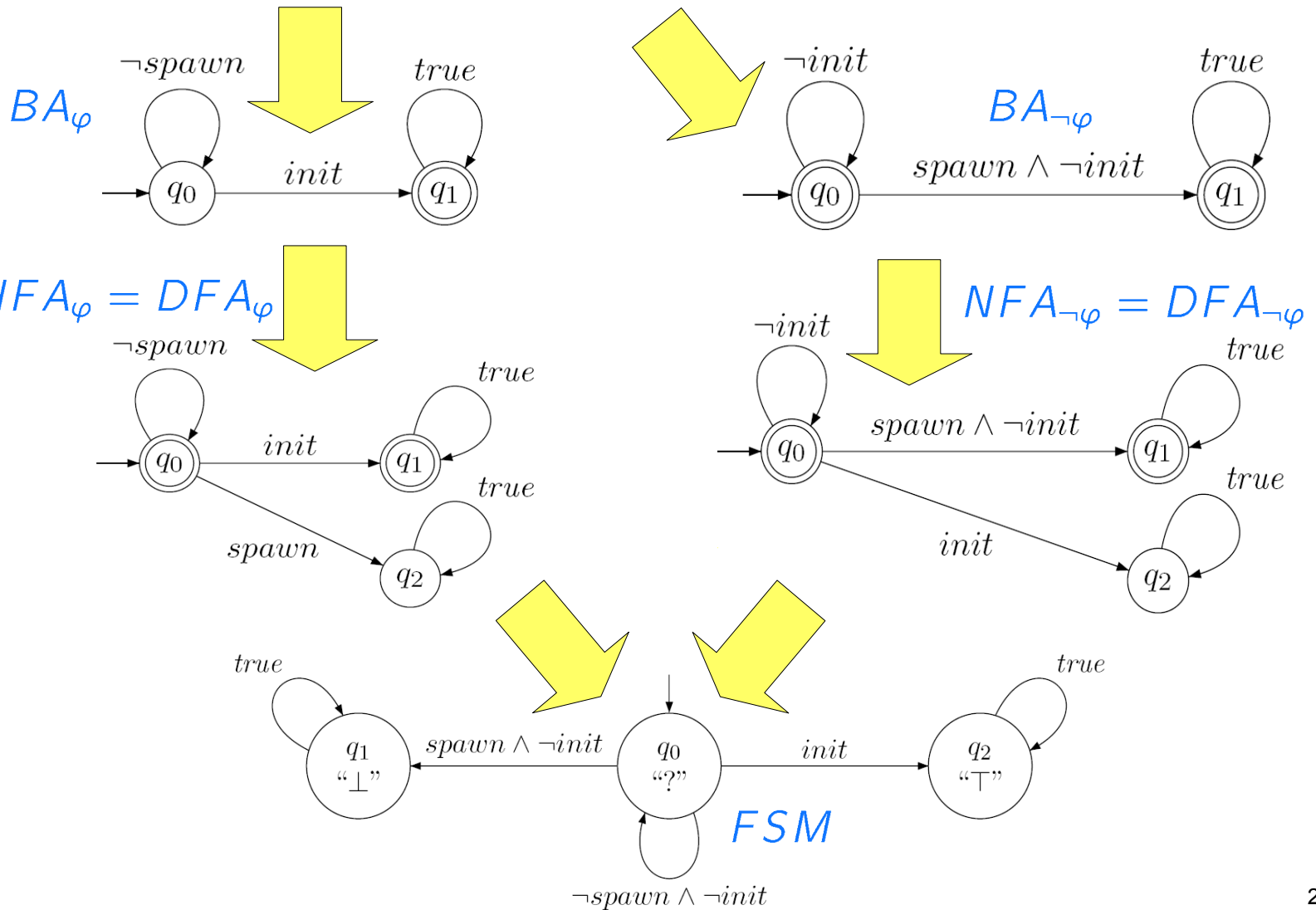
- Consider this:
 - FSMs can be minimised (Myhill-Nerode equivalence)
 - As such, any smaller automaton does not accept the same language \Rightarrow Monitor generation optimal
- Moreover, procedure detects *all minimal* bad prefixes by definition of 3-valued semantics:

$$[u \models \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\ \perp & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\ ? & \text{otherwise} \end{cases}$$



Static initialisation order fiasco

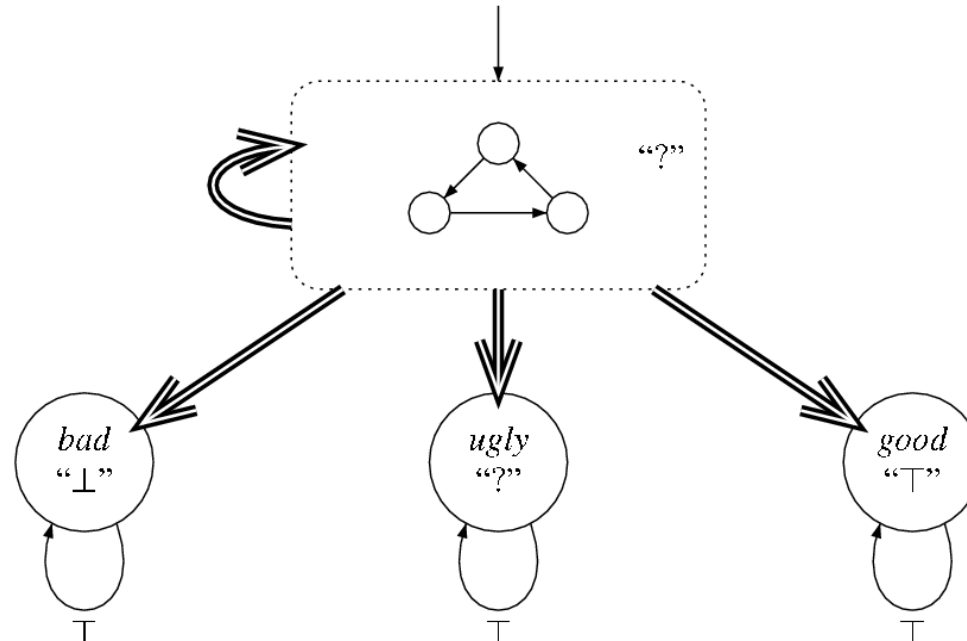
$$\varphi \equiv \neg\text{spawn} \text{ U } \text{init} \quad \neg\varphi \equiv \neg(\neg\text{spawn} \text{ U } \text{init})$$





Monitors revisited

- The general structure



- Bad prefixes/Good prefixes/Ugly prefixes
- Monitorable (after u)
- Safety/co-safety properties



Works also for timed systems!

- Principle stays the same, but we use
 - Timed words $(a_0, t_0)(a_1, t_1) \dots$ ($t \in \mathbb{R}$)
 - TLTL aka state-clock logic for LTL
(FO fragment of MSO interpreted over timed words)
 $\varphi ::= true \mid a \mid \triangleleft_a \in I \mid \triangleright_a \in I \mid \neg\varphi \mid \varphi \text{ op } \varphi \mid \varphi \mathbf{U}\varphi \mid \mathbf{X}\varphi$
for each symbol we have two clock variables

Useful for time-bounded response properties!

- a *predicting* clock
 - for translation timed (event-clock) automata with runs of the form: $\theta : (q_0, \gamma_0) \xrightarrow{d_1} \xrightarrow{a_1} (q_1, \gamma_1) \xrightarrow{d_2} \xrightarrow{a_2} (q_2, \gamma_2) \xrightarrow{d_3} \xrightarrow{a_3} \dots$
 - Obviously, this execution scheme poses a problem!



Observing – Not Acting!

- Runtime verification does not influence/modify the running system
- at least tries to – it might consume resources of the monitored system



Monitor-oriented Programming (Grigore Rosu et. al. - thanks for Slides)



But first

A commercial...



Main Ideas

- Use monitors to check behavior of program
- *Influence behavior* of program when monitor either confirms that a correctness property is satisfied or falsified
- annotate code (e.g. Java code) by
 - monitor description, and
 - code to be triggered by monitor



Enforce authentication before use

```
class Resource {
  /*@
  scope = class
  logic = PTLTL
  {
    Event authenticate: end(exec(* authenticate()));
    Event use: begin(exec(* access()));
    Formula : use -> <*> authenticate
  }
  violation Handler {
    @this.authenticate();
  }
  @*/
  void authenticate() {...}
  void access() {...}
  ...
}
```

Where

How

What

What if

→ scope = class

→ logic = PTLTL

→ {
Event authenticate: end(exec(* authenticate()));
Event use: begin(exec(* access()));
Formula : use -> <*> authenticate
}

→ {
@this.authenticate();
}



Example

```
... (Java code A) ...

/*@ FTLTL
  Predicate red : tlc.state.getColor() == 1;
  Predicate green : tlc.state.getColor() == 2;
  Predicate yellow : tlc.state.getColor() == 3;
  // yellow after green
  Formula : [] (green -> (! red U yellow));
  Violation handler : ... (Java "recovery" code) ...
@*/

... (Java code B) ...
```

- Programming methodology
- some ideas similar as in aspect-oriented programming



MOP key features

- **Extensible** logic framework
 - “Plug and use” user-defined logics using **logic plugins**
 - Supported logics: ERE (Extended Regular Expression), PtLTL (Past Time Linear Temporal Logic), FtLTL (Future Time Linear Temporal Logic), ATL (Allen Temporal Logic)
- **Generic** and **efficient** support for universal parameters
 - Allow monitor instances per groups of objects
 - Faster than logic-specific monitoring techniques, e.g., Tracematches and PQL. **[OOPSLA'07]**
- **Configurable** monitors
 - **Running mode**: inline/outline, online/offline
 - **Working scope**: method, class, global, ...
 - User-provided actions for violations and validations

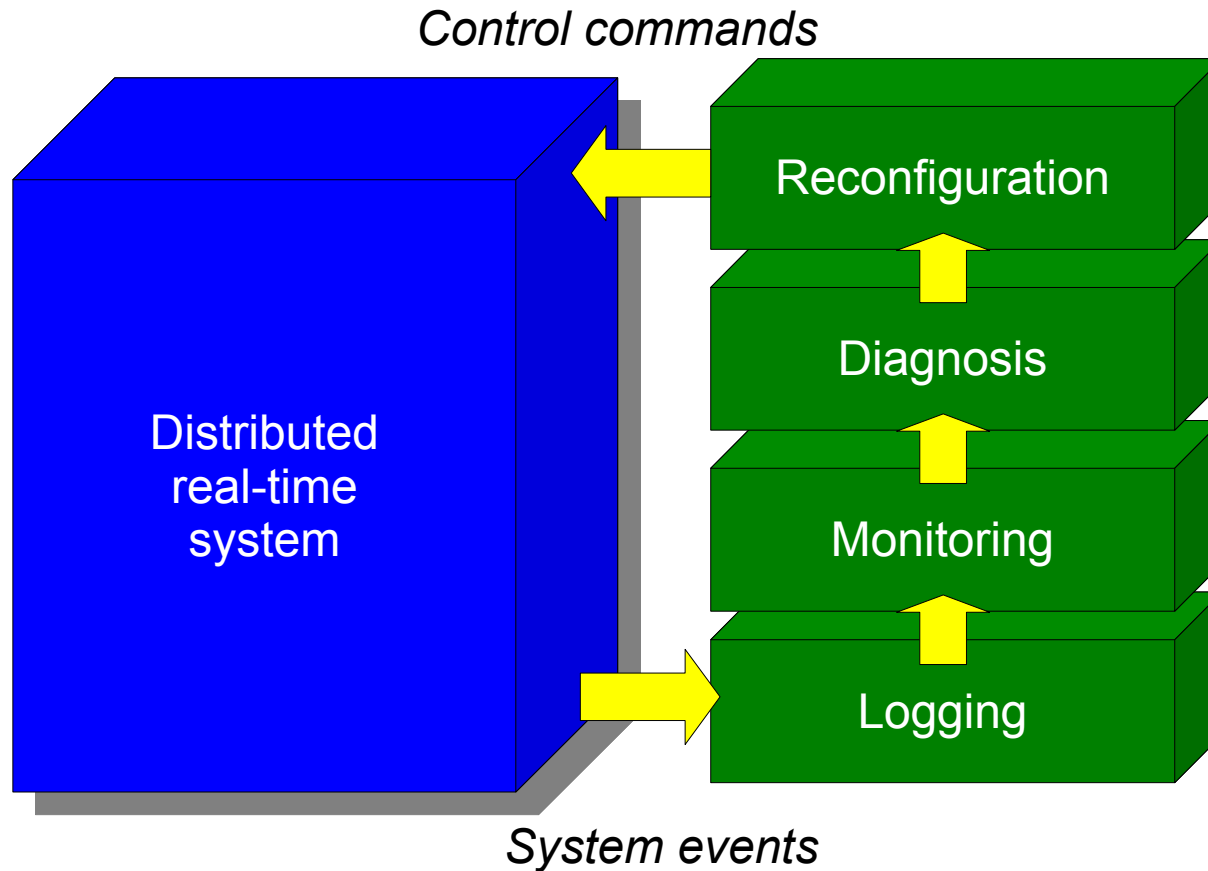


Monitor-oriented Runtime Reflection



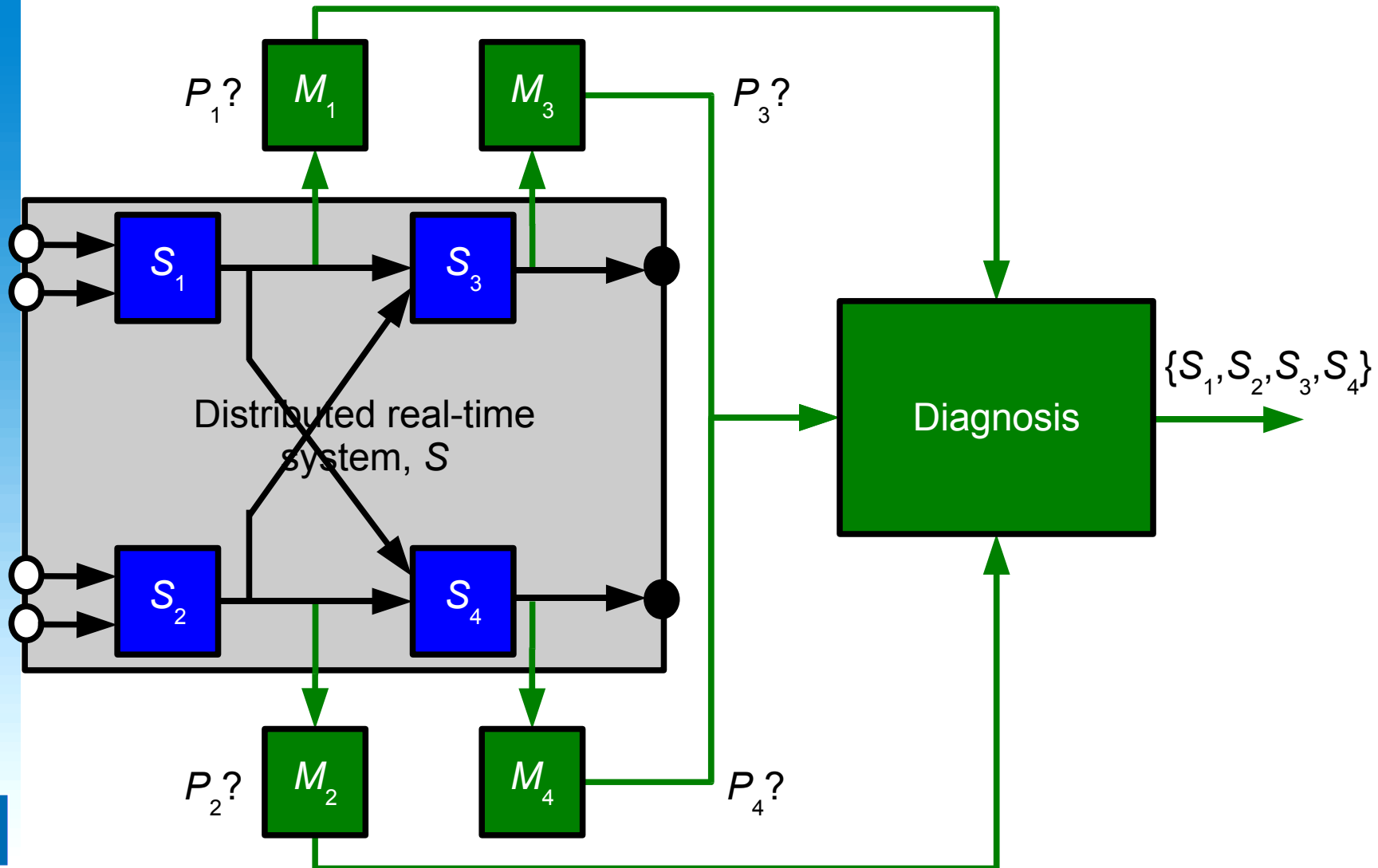
Runtime reflection

- An architecture for model-based runtime analysis:



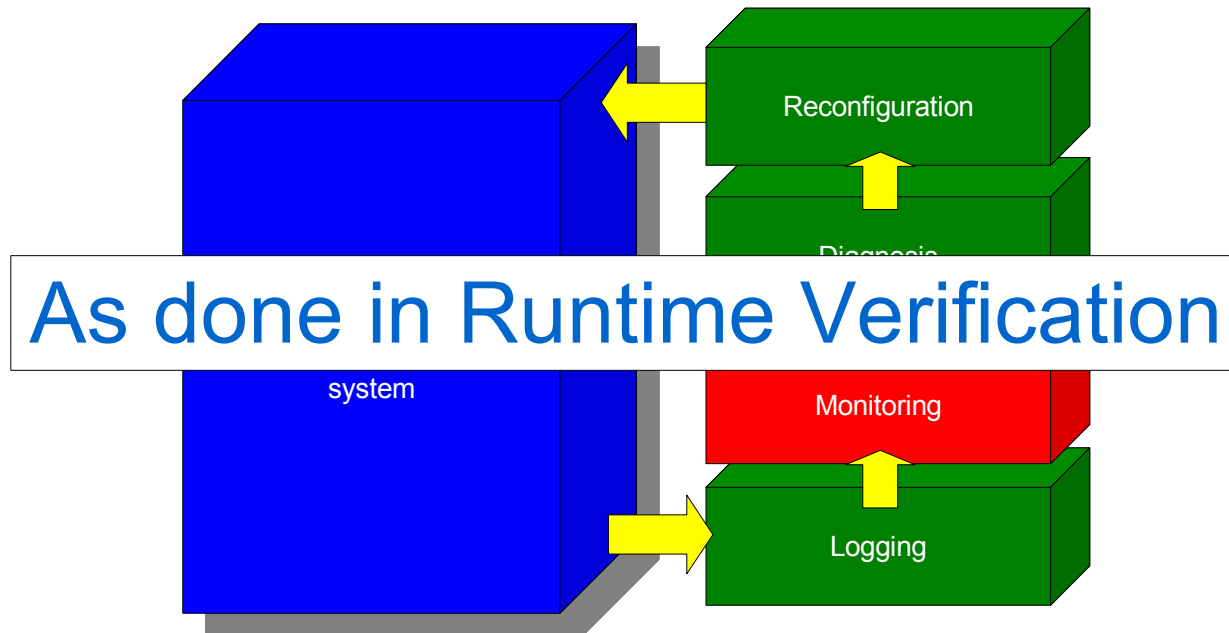


How it works – A functional overview



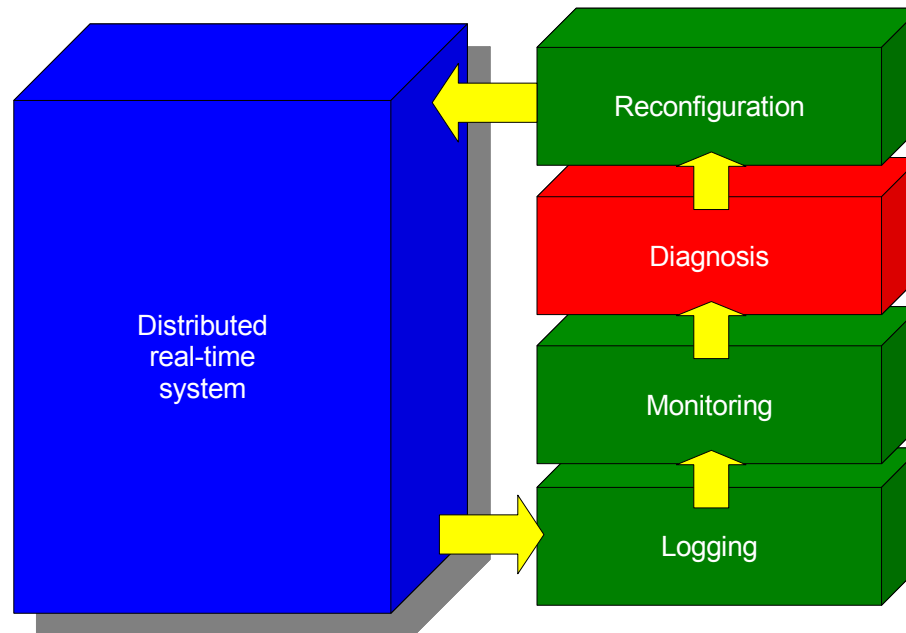


Part - Monitoring



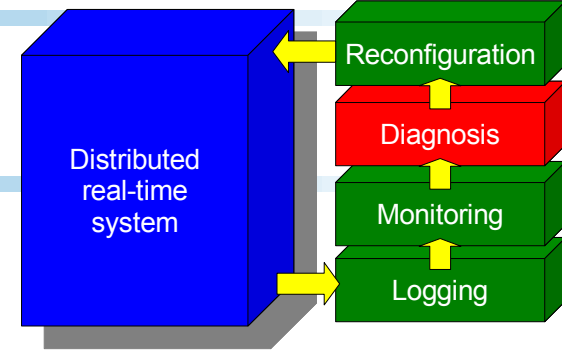


Part - Diagnosis





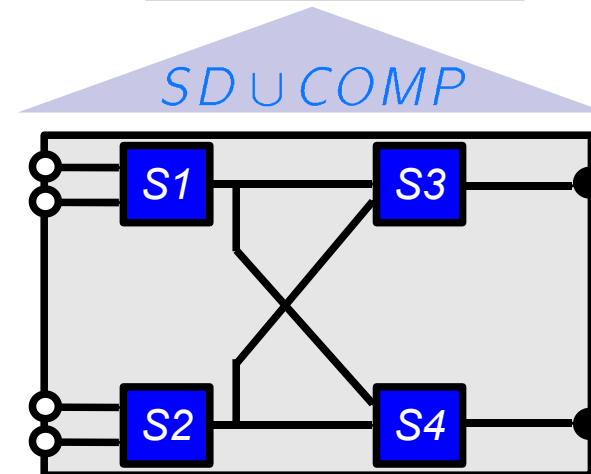
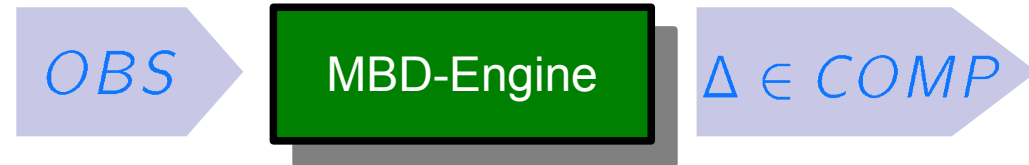
Part – Diagnosis



- System model:** set of components, $COMP$, system description, SD , specified in terms of first-order sentences, e.g.,

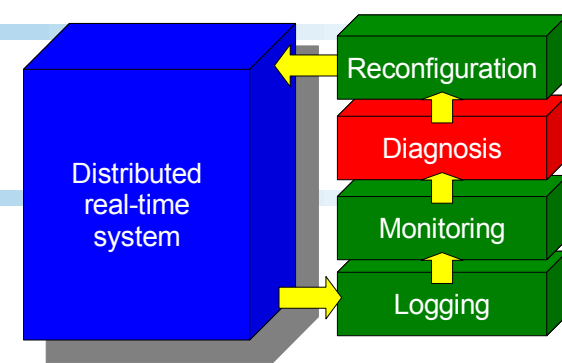
$$SD = \{mult(X) \wedge \neg AB(X) \rightarrow (out(X) = in1(X) * in2(X)), mult(S1), \dots\}$$

- Diagnostic task:** given a set of observations, OBS , find *all minimal diagnoses* $\Delta \in COMP$ s.t. $SD \cup COMP \cup OBS$ is explicable, i.e., is consistent





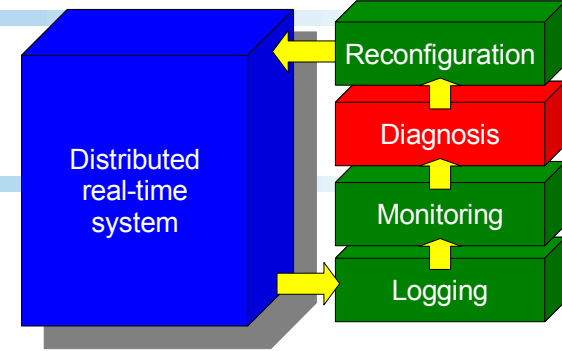
But what about...



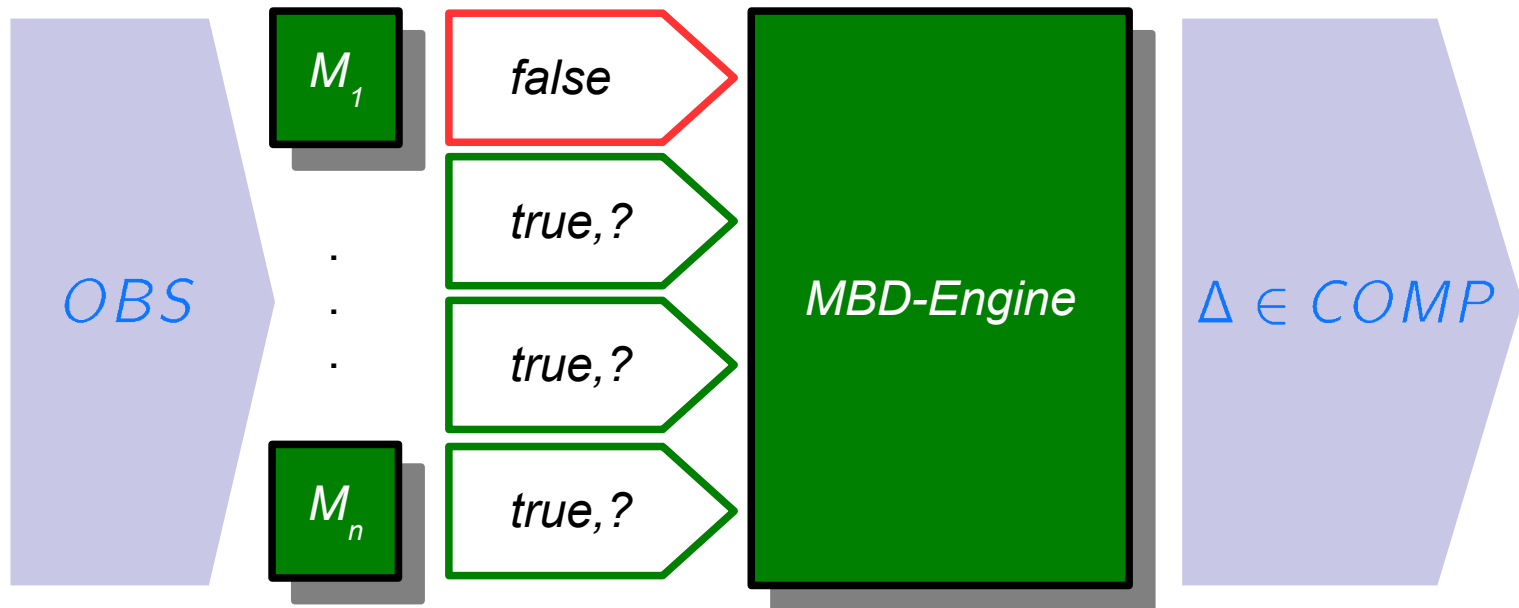
- But, first-order diagnosis is hard!
 - Automatic conflict set determination may not always be possible (e.g., non-termination, HS-problem is NP-hard)
 - Although tailored for it, introduction of “*unknowns*” may result in exponential blowup of conflict search space
- Not laid out for real-time / event-triggered systems
- Ultimately, hard to think of doing all this *continuously* at *runtime*!



Combined analysis



- Diagnosis-on-demand, using the automatically generated monitors as inputs as inputs

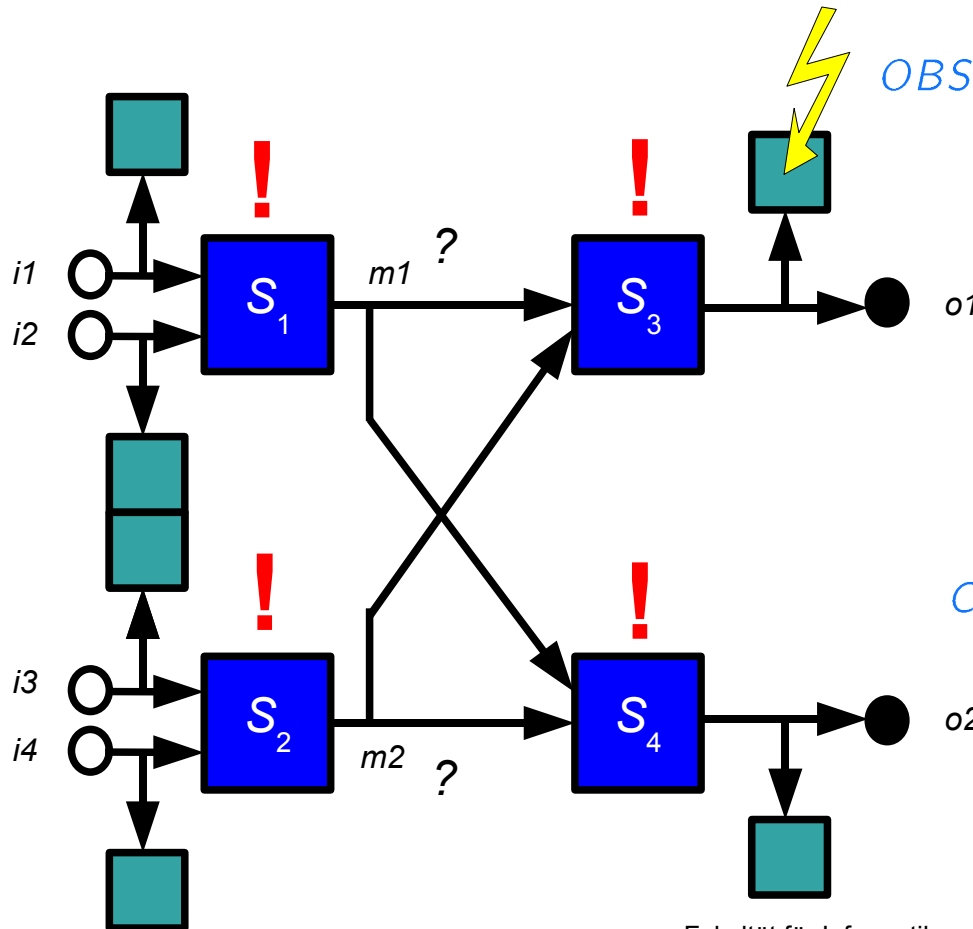


- Propositional system model, $SD: ok(M1), ok(M2), \dots$
- Problem still in NP, but powerful algorithms for computing conflicts



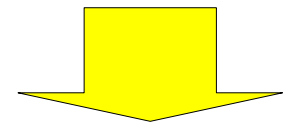
Example: where is the fault?

$$SD \in MF(SD) = \left\{ \begin{array}{l} ok(i_1) \wedge \neg ok(i_2) \wedge \neg AB(S_1) (\neg S_1) \wedge k(m_1), \\ ok(i_3) \wedge \neg ok(i_4) \wedge \neg AB(S_2) (\neg S_2) \wedge k(m_2), \\ ok(m_1) \wedge ok(m_2) \wedge \neg AB(S_3) (\neg S_3) \wedge k(o_1) \\ ok(m_1) \wedge ok(m_2) \wedge \neg AB(S_4) (\neg S_4) \wedge k(o_2) \end{array} \right\}$$



$OBS = \{i_1, i_2, i_3, i_4, \neg o_1, o_2\}$

$SD \cup COMP \cup OBS$

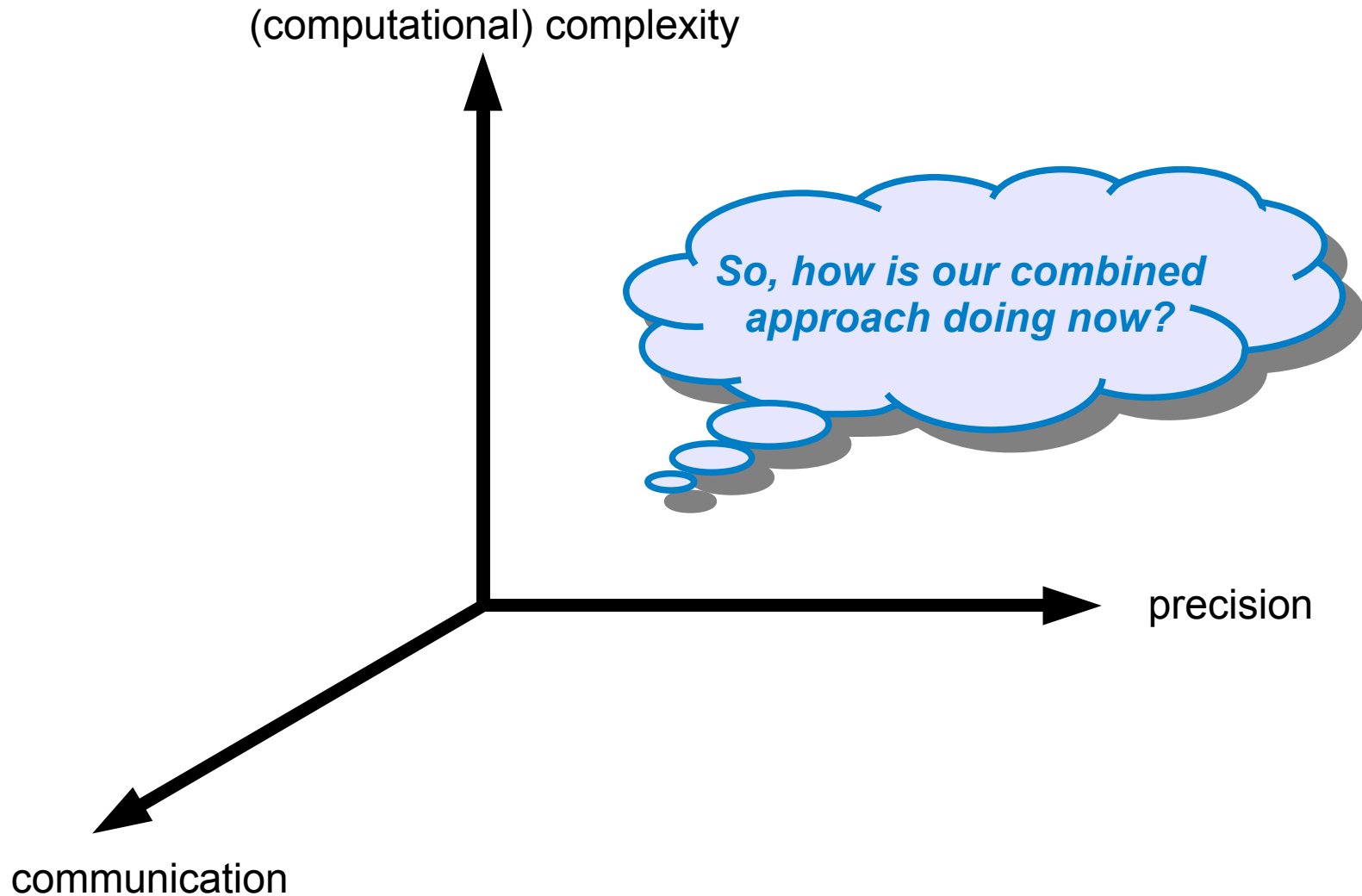


$$Conf = \left\{ \begin{array}{l} \{S_1, S_2, S_3, \neg S_4\}, \\ \{S_1, S_2, \neg S_3, \neg S_4\}, \\ \{S_1, \neg S_2, S_3, \neg S_4\}, \\ \{S_1, \neg S_2, \neg S_3, \neg S_4\}, \\ \{\neg S_1, S_2, S_3, \neg S_4\}, \\ \{\neg S_1, S_2, \neg S_3, \neg S_4\}, \\ \{\neg S_1, \neg S_2, S_3, \neg S_4\} \end{array} \right\}$$



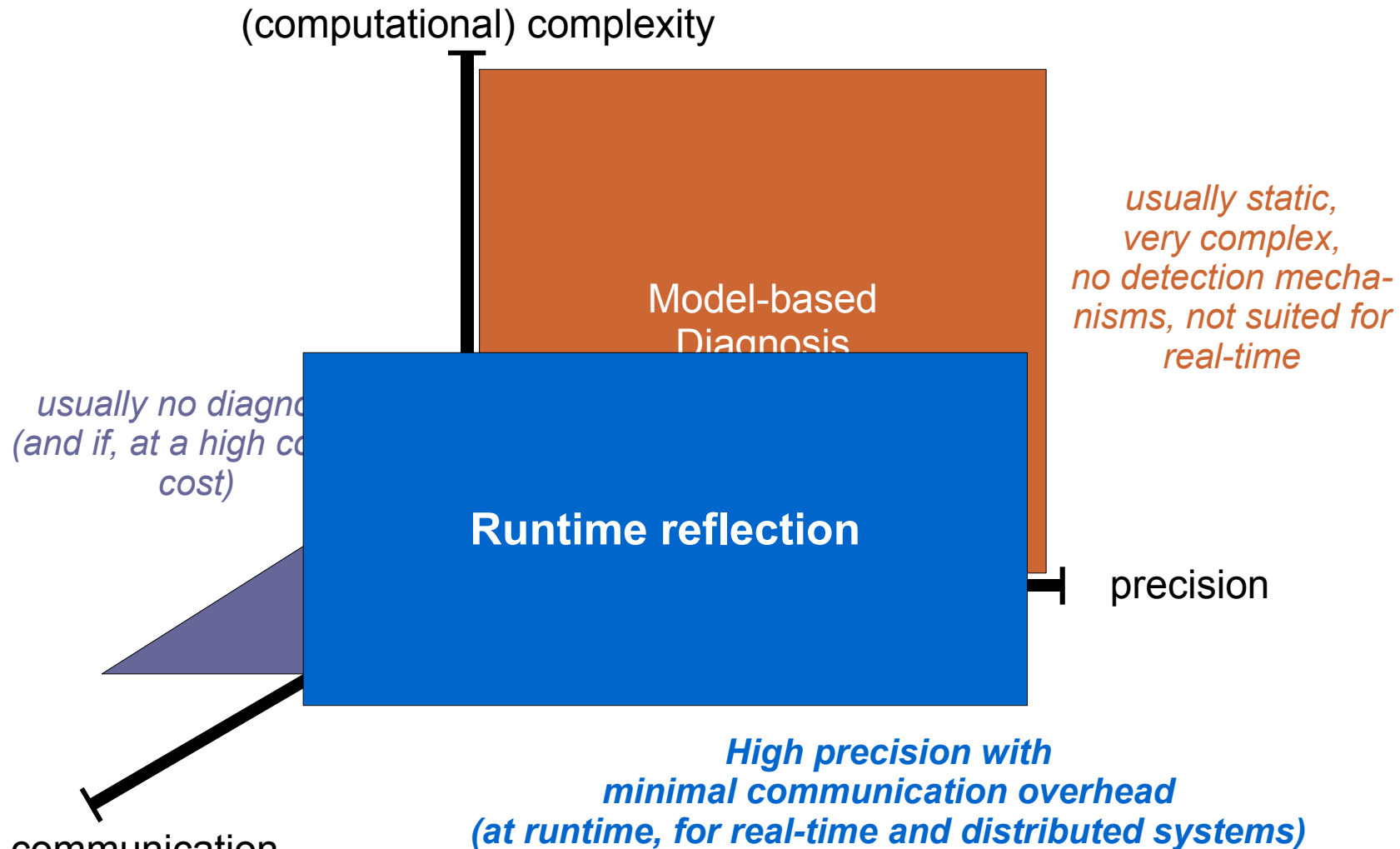
Let's compare!

The three dimensions of runtime analysis





Problems and strengths of methods



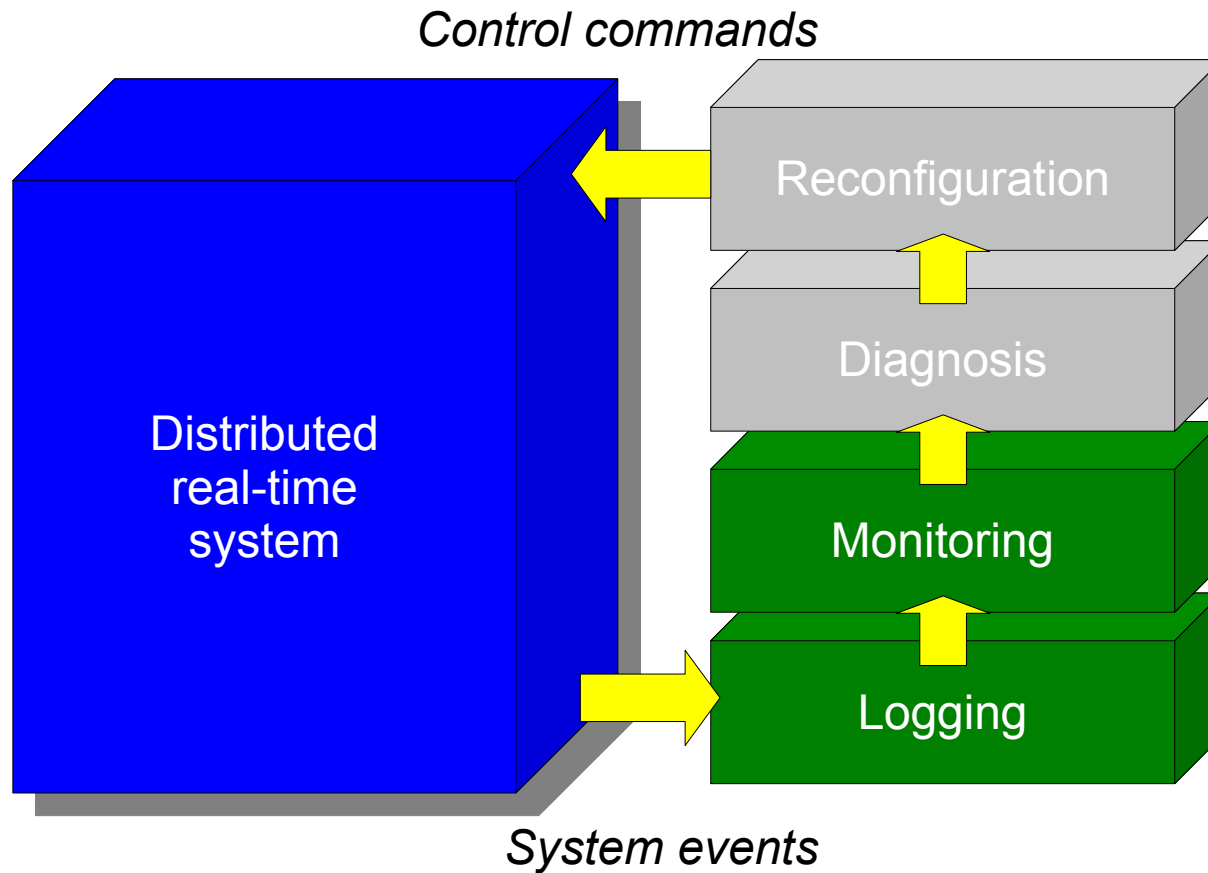


Towards conclusion

Let's wrap up!



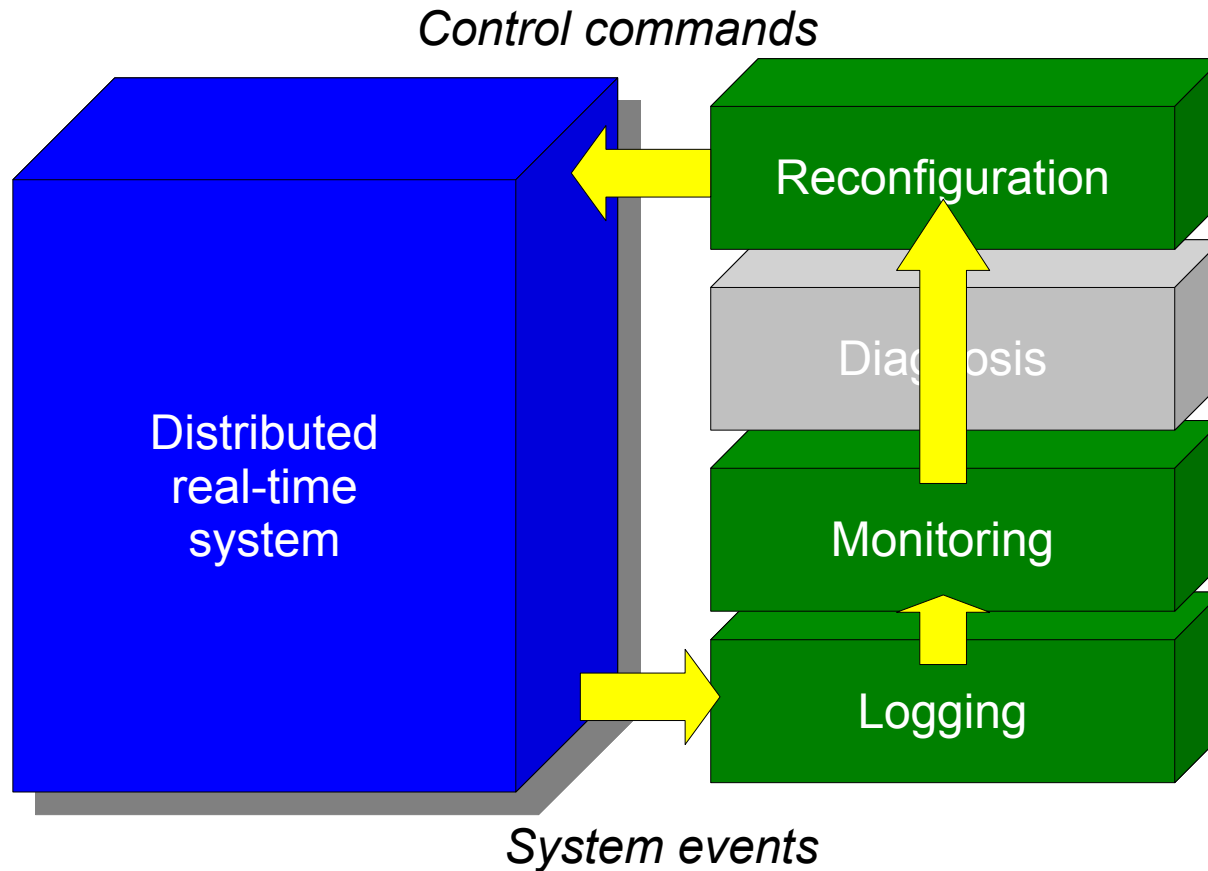
Runtime verification





Monitor-oriented programming

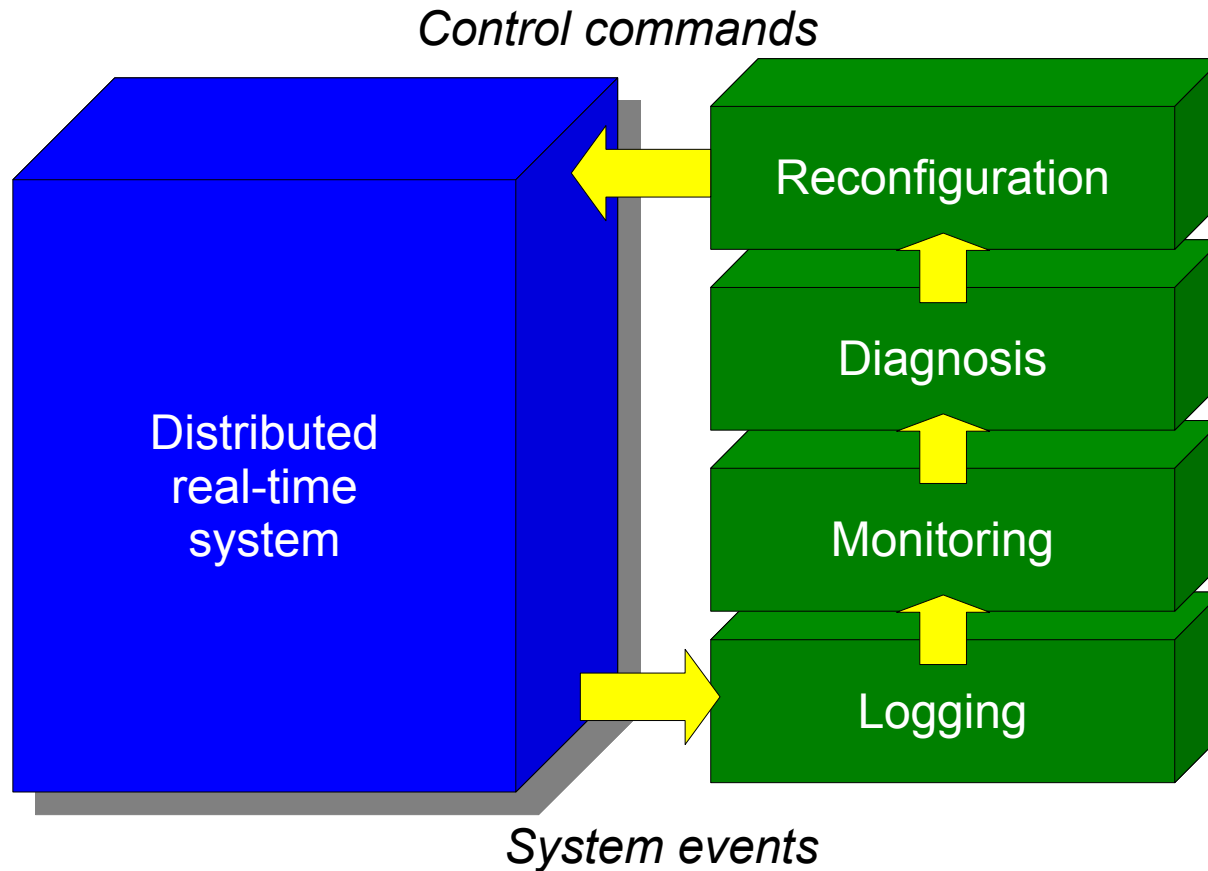
- Combination of Runtime Verification with Diagnosis





Runtime reflection

- Combination of Runtime Verification with Diagnosis





Thank you!

- `http://runtime.in.tum.de/`

File Edit View Web Go Bookmarks Tabs Help

Back Forward Stop Refresh New Home 100 http://runtime.in.tum.de/

Google Dictionary WWiki debian CS

(Home)
(News)
(Documentation)
(Download)
(Support)
(Contact)

runtime reflection

Copyright © 2005 The Runtime Reflection Project Team <runtime@in.tum.de>

General Information:

The runtime reflection project establishes methods to dynamically analyse reactive distributed systems at runtime. The approach is layered and modular in that we provide the means for first detecting failures of a system by means of runtime verification and secondly, provide means for identifying their causes by means of a detailed diagnosis. As such, diagnoses can be subsequently used in order to trigger recovery measures, or to store detailed log-files for off-line analysis.

In principle, runtime reflection spans from the early development phases, e.g., systems design and specification to their actual implementation and analysis at runtime; thus, ultimately at more fault-tolerant systems.

On this project page we provide the tools which make up our framework to accomplish the above objectives. In particular, we currently offer a tool for automatically generating so-called runtime monitors from a given LTL-specification which can be used to monitor arbitrary C++-programs. The tool allows to annotate C++-code in order to generate log-records for events such as function entries and exits, unexpected exceptions, violated assertions, and passing of simple trace points. The resulting traces of a program execution can then be analysed at runtime for erroneous behaviour by the generated monitor. The tool will report a violation of a given property immediately when it is clear that it will eventually happen.

Additionally, we provide an implementation to perform the main diagnostic task; that is, based on the results of the monitors, find causes for occurred errors. The implementation is currently based on a highly optimised SAT-solver, which computes for a number of given symptoms possible conflict sets, hence diagnoses, that explain the undesired behaviour. As such, we are able to differentiate between the symptoms of a fault, and the actual fault itself.

License:

All tools, part of the runtime reflection framework, are released under the terms of the GNU General Public License. For more information, see the according README and LICENSE documents provided with the source codes.

Current Status:

Oct-25-2005: Released 0.0.1 lsat_diagnosis - Development Version
Oct-18-2005: Released runtime_verification 0.0.1 - Development version

Last updated on October 21, 2005 by The Runtime Reflection Project Team.
Copyright © 2005 The Runtime Reflection Project Team. All rights reserved.

W3C HTML 4.0