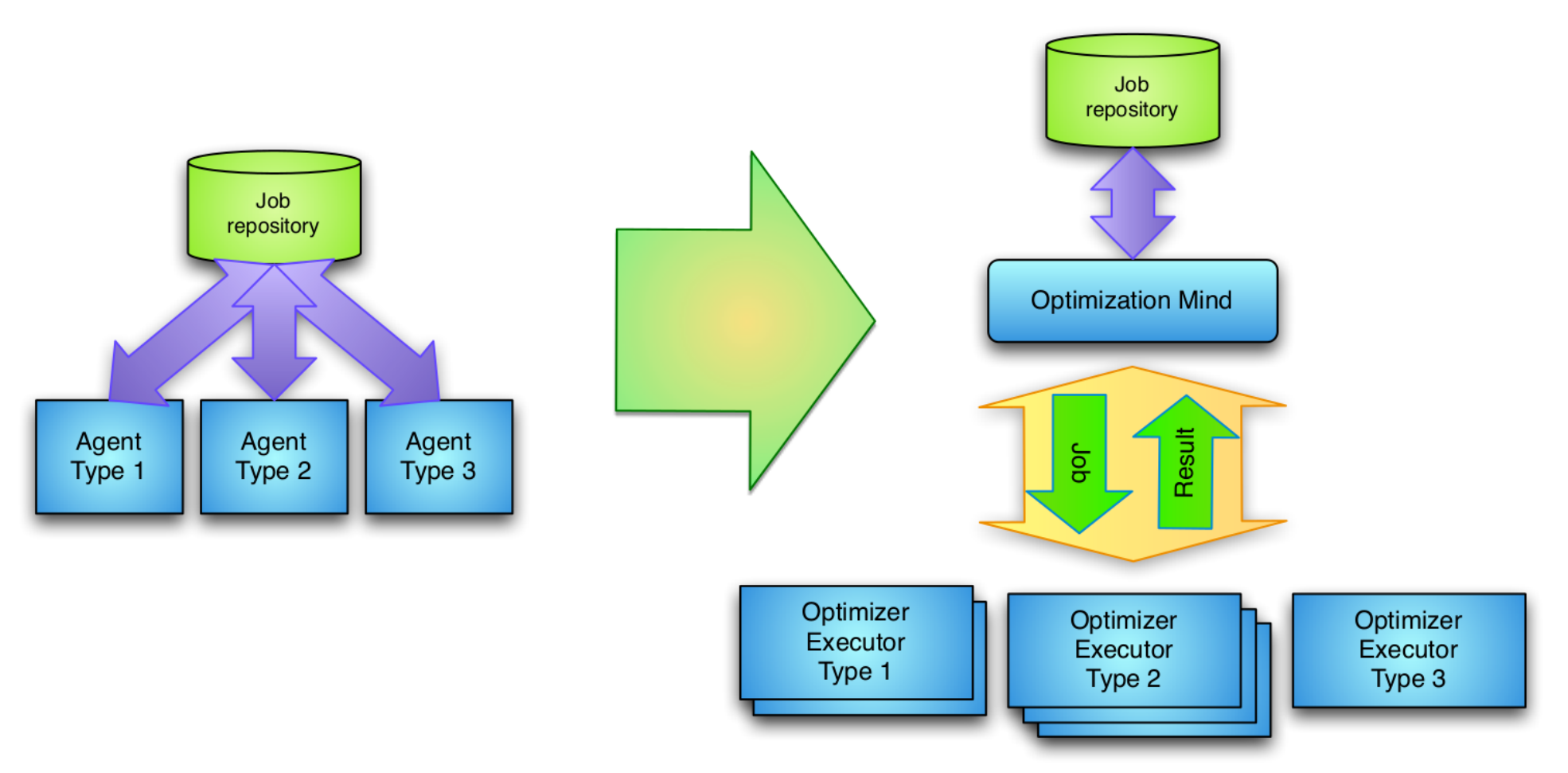
Round-tripping DIRAC: Automated Model-Checking of Concurrent Software Design Artifacts

Daniela Remenska - Jeff Templon - Tim Willemse - Henri Bal - Kees Verstoep - Adrian Casajous - Philip Homburg

Motivation

DIRAC WMS background



■ **Executors Framework**: Set of omponents responsible for orchestrating the WMS before the jobs run on the Grid: *OptimizationMind* and *Executors*

■ The *Executors* process any job sent to them by the *OptimizationMind*, each one being responsible for a different step in the handling of jobs (e.g., resolving the input data for a job)

■ The *ExecutorDispatcher* persists the state of the jobs and distributes them among the *Executors*, based on the requirements of each job.

■ Certain problems manifested in the new components: occasionally, jobs submitted in the system would not get dispatched, despite the fact that their responsible Executors were idle at the moment

■ Difficult to trace the root cause of such faulty behavior
many scenarios due to concurrent components

■ Attempts to identify the problem: test with different workload scenarios, analyse the generated logs

There are formal or systematic approaches to tackle this!

Why Formal Methods?

⊞ Based on process algebra laws
no ambiguity

⊞ Mature model checking tools
Full control over the execution of concurrent processes.
This way one gains more insight into the system behavior.

⊞ Automatically explore all possible system behaviors and check if some interesting properties hold
Tools report problems for you, just ask

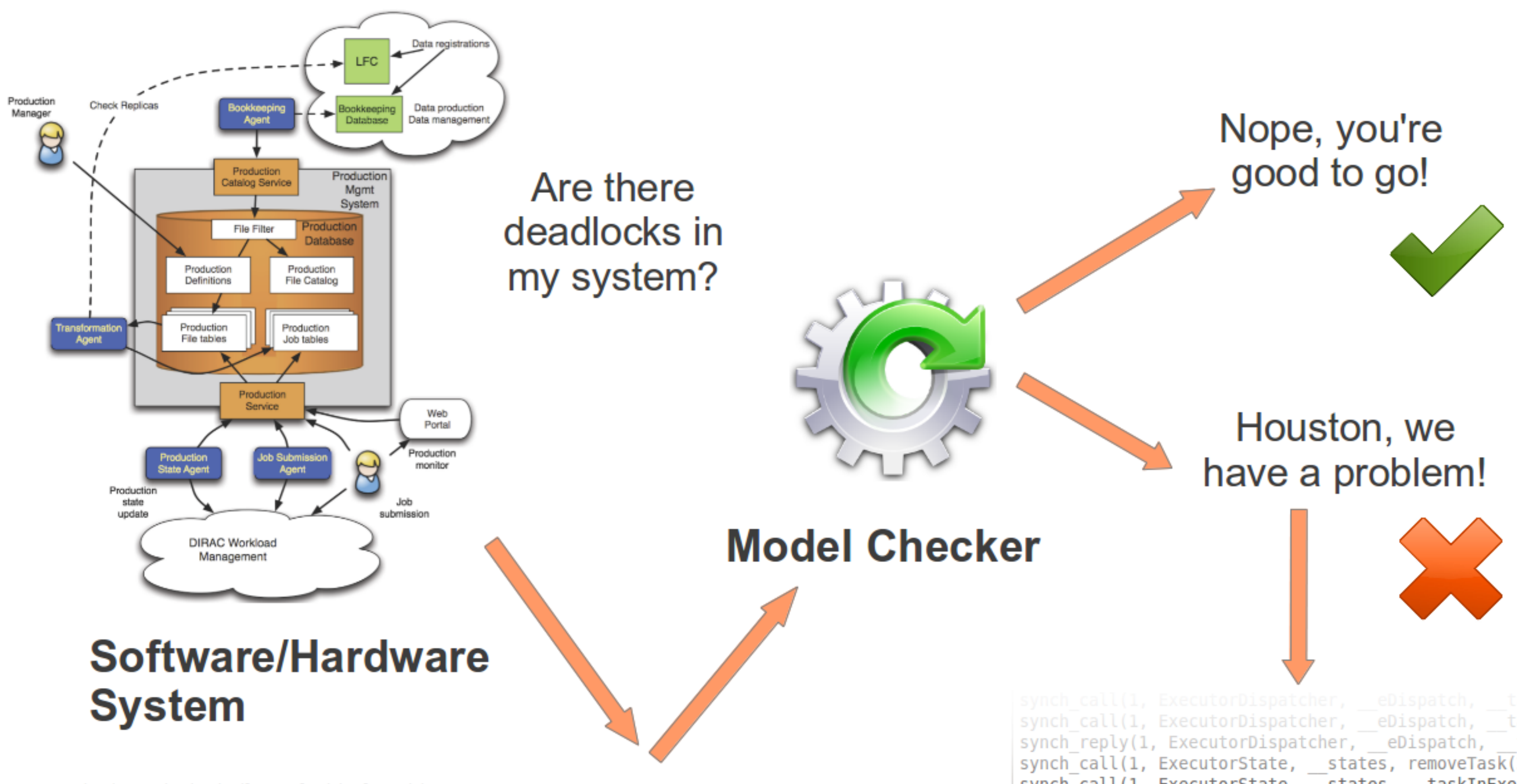
⊞ Stronger than testing a limited amount of scenarios

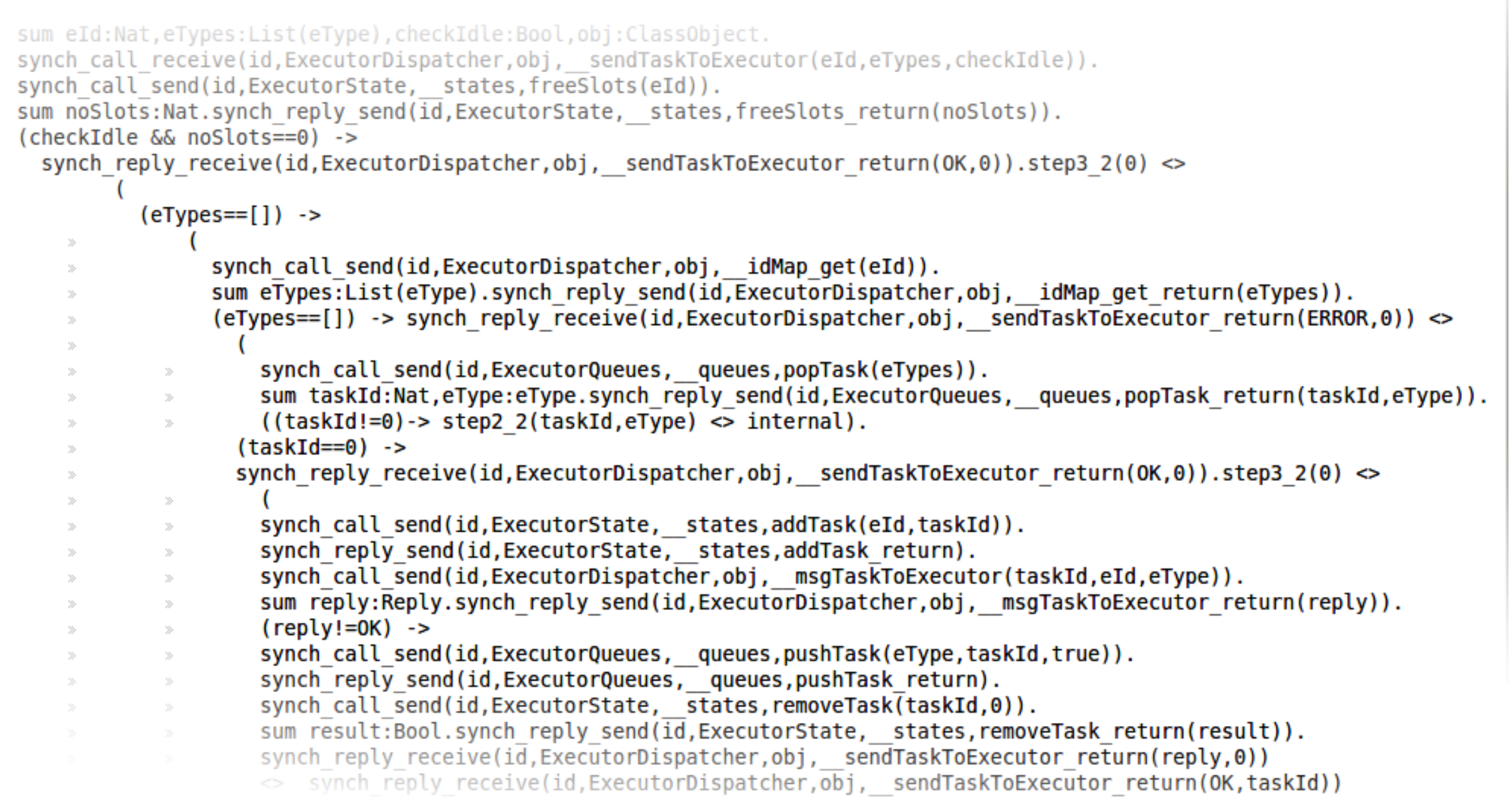
BUT....! ☹

One must build/write a sound model in the language of process algebra, as an abstraction of the real behavior

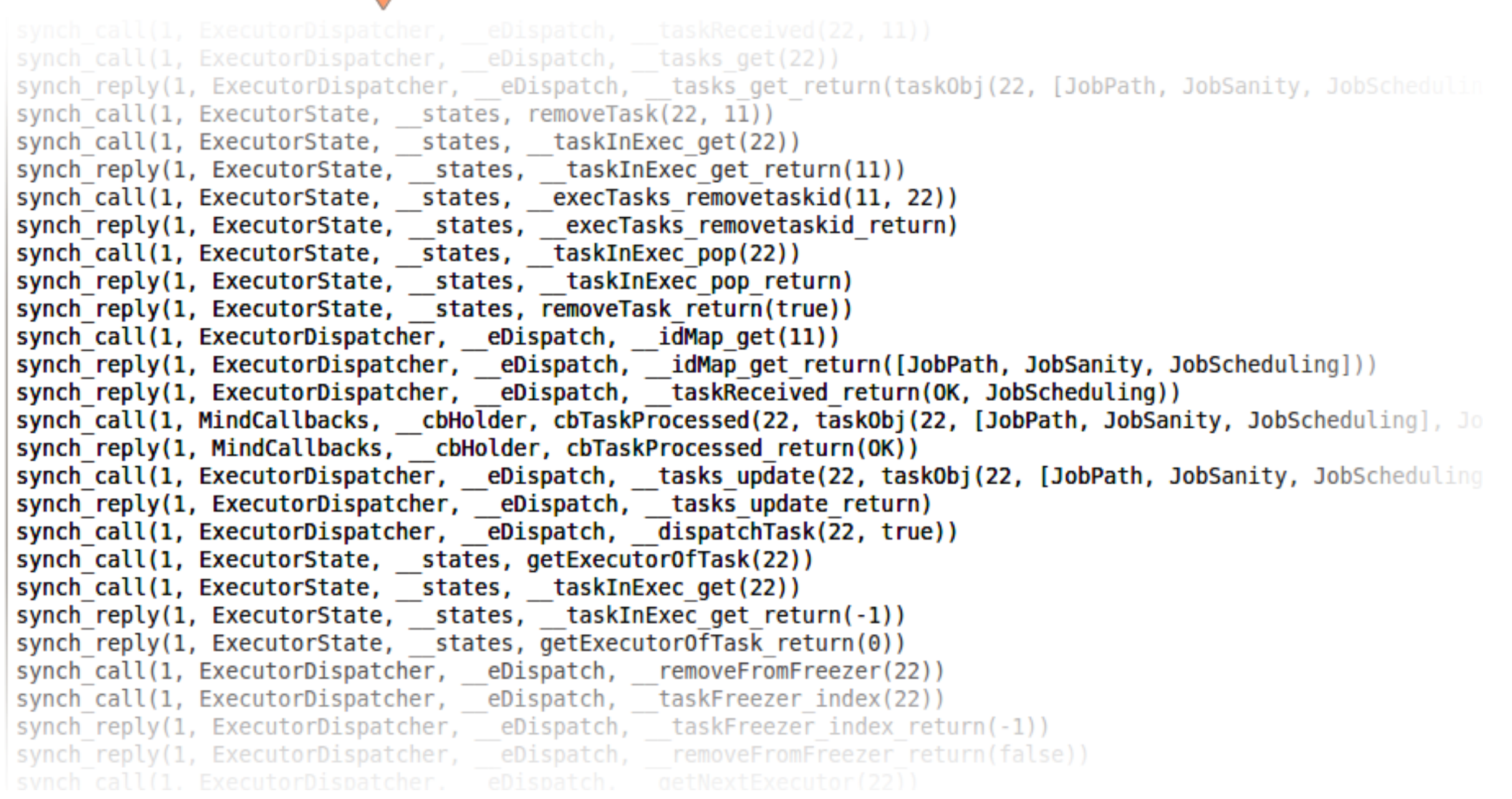
Software engineers lack expertise in formal methods

Investing time to learn process algebra and model checking concepts is mostly a no-no!





```
sum eId:Nat,eTypes:List(eType),checkIdle:Bool,obj:ClassObject.
synch_call_receive(id,ExecutorDispatcher,obj,__sendTaskToExecutor(eId,eTypes,checkIdle)).
synch_call_send(id,ExecutorState,__states,freeSlots(eId)).
sum noSlots:Nat.synch_reply_send(id,ExecutorState,__states,freeSlots_return(noSlots)).
(checkIdle && noSlots==0) ->
  synch_reply_receive(id,ExecutorDispatcher,obj,__sendTaskToExecutor_return(OK,0)).step3_2(0) <>
        (
          (eTypes==[]) ->
            (
              synch_call_send(id,ExecutorDispatcher,obj,__idMap_get(eId)).
              sum eTypes:List(eType).synch_reply_send(id,ExecutorDispatcher,obj,__idMap_get_return(eTypes)).
              (eTypes==[]) -> synch_reply_receive(id,ExecutorDispatcher,obj,__sendTaskToExecutor_return(ERROR,0)) <>
                (
                  synch_call_send(id,ExecutorQueues,__queues,popTask(eTypes)).
                  sum taskId:Nat,eType:eType.synch_reply_send(id,ExecutorQueues,__queues,popTask_return(taskId,eType)).
                  ((taskId!=0)-> step2_2(taskId,eType) <> internal).
                (taskId==0) ->
                synch_reply_receive(id,ExecutorDispatcher,obj,__sendTaskToExecutor_return(OK,0)).step3_2(0) <>
                  (
                  synch_call_send(id,ExecutorState,__states,addTask(eId,taskId)).
                  synch_reply_send(id,ExecutorState,__states,addTask_return).
                  synch_call_send(id,ExecutorDispatcher,obj,__msgTaskToExecutor(taskId,eId,eType)).
                  sum reply:Reply.synch_reply_send(id,ExecutorDispatcher,obj,__msgTaskToExecutor_return(reply)).
                  (reply!=OK) ->
                  synch_call_send(id,ExecutorQueues,__queues,pushTask(eType,taskId,true)).
                  synch_reply_send(id,ExecutorQueues,__queues,pushTask_return).
                  synch_call_send(id,ExecutorState,__states,removeTask(taskId,0)).
                  sum result:Bool.synch_reply_send(id,ExecutorState,__states,removeTask_return(result)).
                  synch_reply_receive(id,ExecutorDispatcher,obj,__sendTaskToExecutor_return(reply,0))
                  <>  synch_reply_receive(id,ExecutorDispatcher,obj,__sendTaskToExecutor_return(OK,taskId))
```

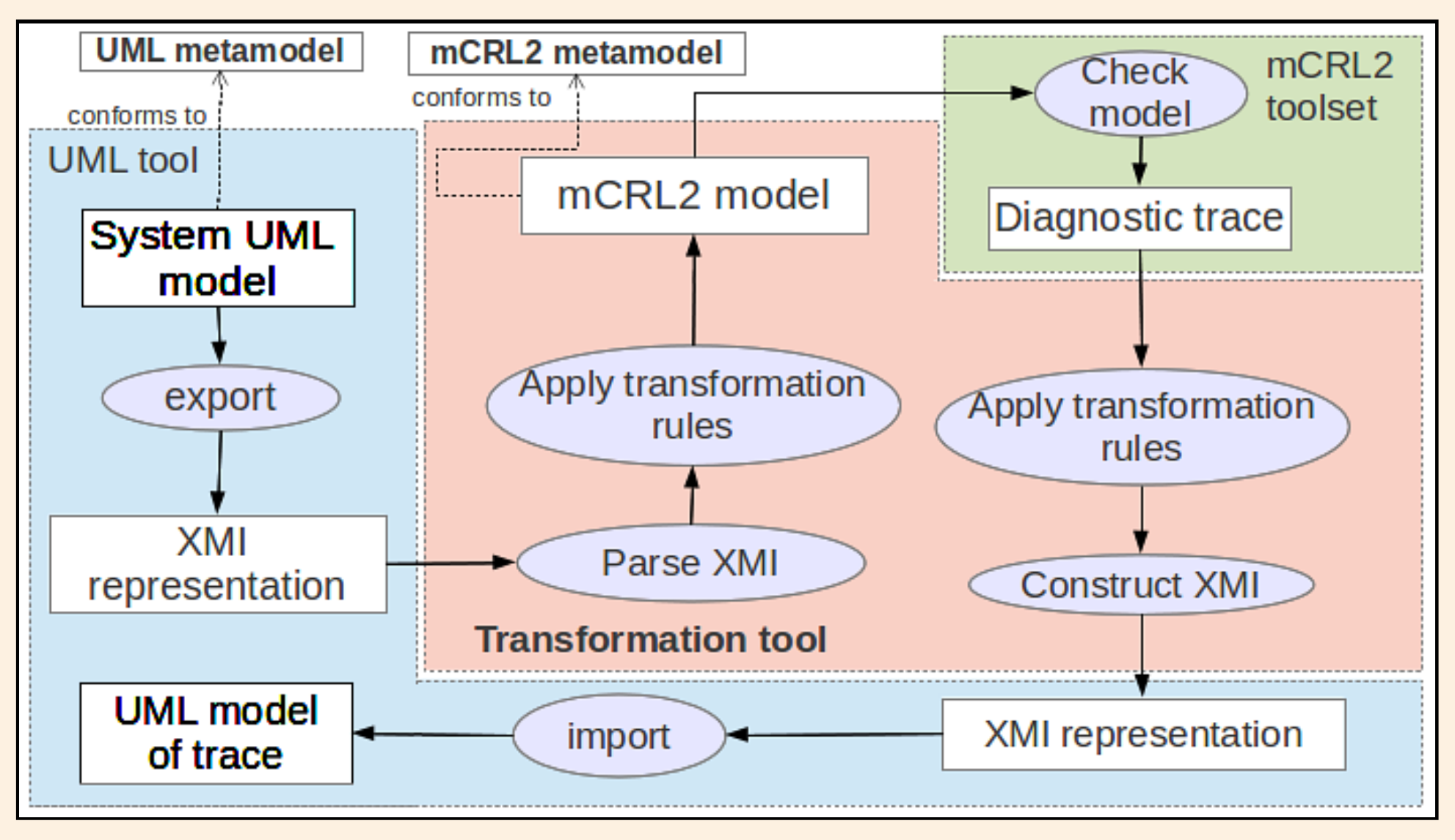


```
synch_call(1, ExecutorDispatcher, __eDispatch, __taskReceived(22, 11))
synch_call(1, ExecutorDispatcher, __eDispatch, __tasks_get(22))
synch_reply(1, ExecutorDispatcher, __eDispatch, __tasks_get_return(taskObj(22, [JobPath, JobSanity, JobSchedulin
synch_call(1, ExecutorState, __states, removeTask(22, 11))
synch_call(1, ExecutorState, __states, __taskInExec_get(22))
synch_reply(1, ExecutorState, __states, __taskInExec_get_return(11))
synch_call(1, ExecutorState, __states, __execTasks_removetaskid(11, 22))
synch_reply(1, ExecutorState, __states, __execTasks_removetaskid_return)
synch_call(1, ExecutorState, __states, __taskInExec_pop(22))
synch_reply(1, ExecutorState, __states, __taskInExec_pop_return)
synch_reply(1, ExecutorState, __states, removeTask_return(true))
synch_call(1, ExecutorDispatcher, __eDispatch, __idMap_get(11))
synch_reply(1, ExecutorDispatcher, __eDispatch, __idMap_get_return([JobPath, JobSanity, JobScheduling]))
synch_reply(1, ExecutorDispatcher, __eDispatch, __taskReceived_return(OK, JobScheduling))
synch_call(1, MindCallbacks, __cbHolder, cbTaskProcessed(22, taskObj(22, [JobPath, JobSanity, JobScheduling], Jo
synch_reply(1, MindCallbacks, __cbHolder, cbTaskProcessed_return(OK))
synch_call(1, ExecutorDispatcher, __eDispatch, __tasks_update(22, taskObj(22, [JobPath, JobSanity, JobScheduling
synch_reply(1, ExecutorDispatcher, __eDispatch, __tasks_update_return)
synch_call(1, ExecutorDispatcher, __eDispatch, __dispatchTask(22, true))
synch_call(1, ExecutorState, __states, getExecutorOfTask(22))
synch_call(1, ExecutorState, __states, __taskInExec_get(22))
synch_reply(1, ExecutorState, __states, __taskInExec_get_return(-1))
synch_reply(1, ExecutorState, __states, getExecutorOfTask_return(0))
synch_call(1, ExecutorDispatcher, __eDispatch, __removeFromFreezer(22))
synch_call(1, ExecutorDispatcher, __eDispatch, __taskFreezer_index(22))
synch_reply(1, ExecutorDispatcher, __eDispatch, __taskFreezer_index_return(-1))
synch_reply(1, ExecutorDispatcher, __eDispatch, __removeFromFreezer_return(false))
synch_call(1, ExecutorDispatcher, __eDispatch, __getNextExecutor(22))
```

Can we decode this trace easily?

Can we automatically generate this formal model?

Methodology



UML is the lingua-franca for software engineers, so we use UML designs as a starting point. Behavioral views such as Sequence Diagrams express the system dynamics, which is necessary for analysis. However, UML tools do not provide exhausive verification capabilities.

To bridge the gap between UML software designs and formal analysis & verification with model checking tools, we designed a transformation methodology to obtain a process algebra model automatically.

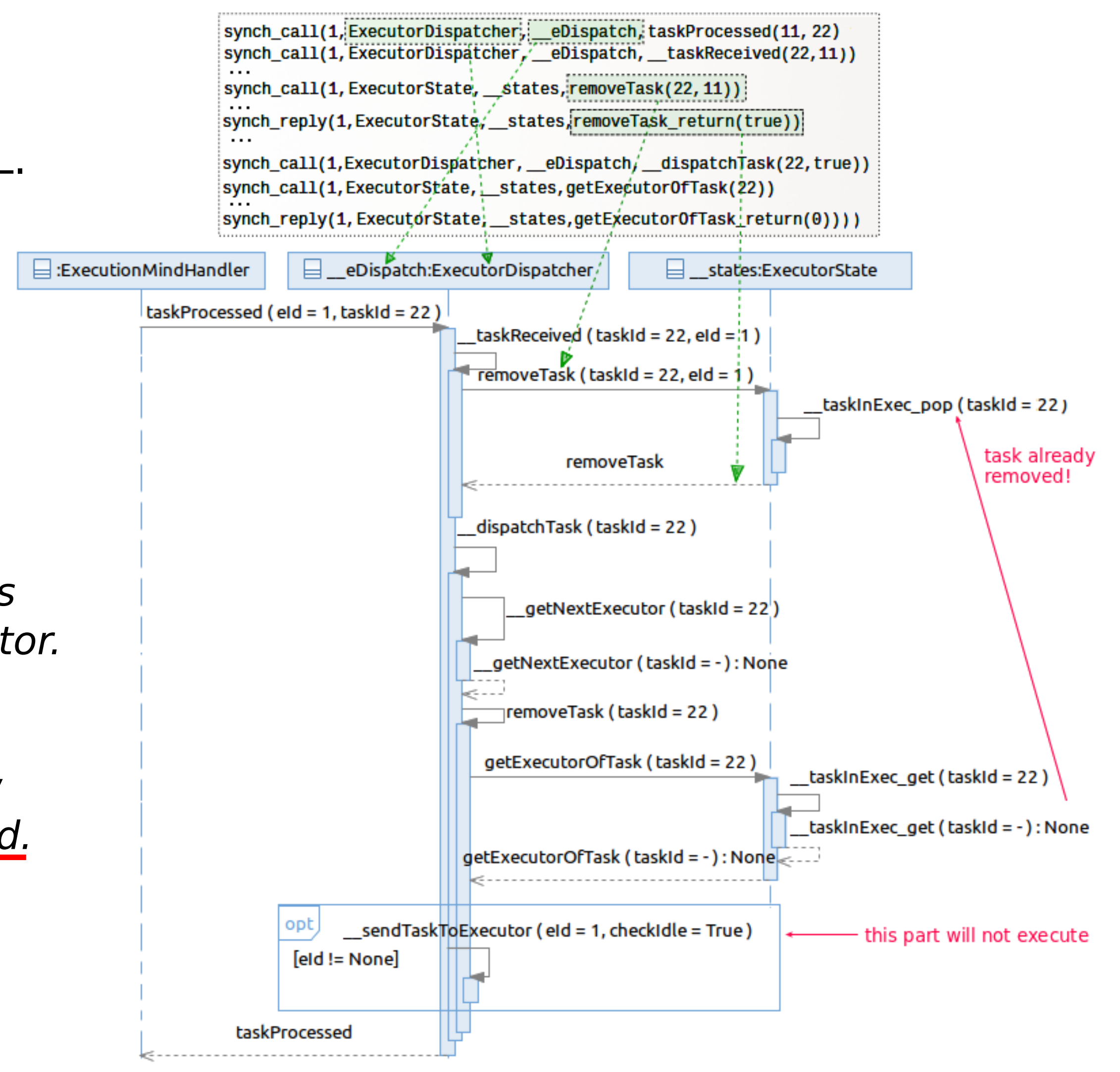
Our methodology allows counter-example traces from the model checking tool to be conveniently displayed back in any UML design tool, allowing the software designer to stay in his familiar environment.

Analysis & Issues

We used the toolset to generate a formal model of the Executor framework, based on Sequence Diagrams in UML. We formulated a property to express that any task pushed in the OptimizationMind's queue should be processed, i.e., eventually removed from the queue, before declaring that there are no more tasks to process.

"*Whenever a task has been processed by some Executor, the ExecutorDispatcher is notified, and this removes it from its list of processing tasks. To further dispatch the task to another Executor, this task is removed from the dispatcher's memory followed by retrieval of the next responsible Executor. In case it was actually processed by the last Executor in the chain. The dispatcher attempts to retrieve its last Executor, so that more tasks can be dispatched to this (now free) Executor. However, this information is already removed. As a result, the opt fragment will not be executed, and no further tasks waiting for this Executor will be dispatched.*"

The bug was reported and fixed.



```
synch_call(1,ExecutorDispatcher,__eDispatch,taskProcessed(11, 22)
synch_call(1,ExecutorDispatcher,__eDispatch,__taskReceived(22,11))
...
synch_call(1,ExecutorState,__states,removeTask(22,11))
...
synch_reply(1,ExecutorState,__states,removeTask_return(true))
...
synch_call(1,ExecutorDispatcher,__eDispatch,__dispatchTask(22,true))
synch_call(1,ExecutorState,__states,getExecutorOfTask(22))
...
synch_reply(1,ExecutorState,__states,getExecutorOfTask_return(0))))
```

:ExecutionMindHandler
__eDispatch:ExecutorDispatcher
__states:ExecutorState
taskProcessed (eId = 1, taskId = 22)
__taskReceived (taskId = 22, eId = 1)
removeTask (taskId = 22, eId = 1)
__taskInExec_pop (taskId = 22)
task already removed!
removeTask
__dispatchTask (taskId = 22)
__getNextExecutor (taskId = 22)
__getNextExecutor (taskId = -) : None
removeTask (taskId = 22)
getExecutorOfTask (taskId = 22)
__taskInExec_get (taskId = 22)
__taskInExec_get (taskId = -) : None
getExecutorOfTask (taskId = -) : None
opt
__sendTaskToExecutor (eId = 1, checkIdle = True)
[eId != None]
this part will not execute
taskProcessed

Conclusions

We developed a toolset for verification of UML models, based on dynamic views of the system

The methodology is based on transforming the UML model into process algebra one, amenable to model checking

The transformation preserves the object-oriented view of the system, making the model easier to understand

The model checking counter-example traces can be fed back into any UML tool for analysis

Future Work

Model checking for application-specific properties still requires the use of temporal logic formulas. Expressing such properties as sequence diagrams of accept/reject scenarios is part of our future work

DIRAC THE INTERWARE
VU VRIJE UNIVERSITEIT AMSTERDAM
TU/e
NIKHEF
UNIVERSITAT DE BARCELONA