Verification of a Wireless ATM Medium-Access Protocol

Natalia Sidorova[1] and Martin Steffen[2]

[1] Dept. of Math. and Comp. Science,
Eindhoven University of Technology,
The Netherlands
n.sidorova@tue.nl

[2] Inst. für angewandte Mathematik und Informatik
Christian-Albrechts-Universität
Kiel, Germany
ms@informatik.uni-kiel.de

Contents

- *Motivation*
- *Mascara protocol*
- *Verification methodology*
- *Results of the protocol verification*
- *Conclusions*

Model Checking in Practice

Model checking problem:
Does system S satisfy property f ?

Method: *state-space exploration*

⇓

Pros: *a "push-button technology"*
and
Cons: *applicability to relatively small finite state systems only*

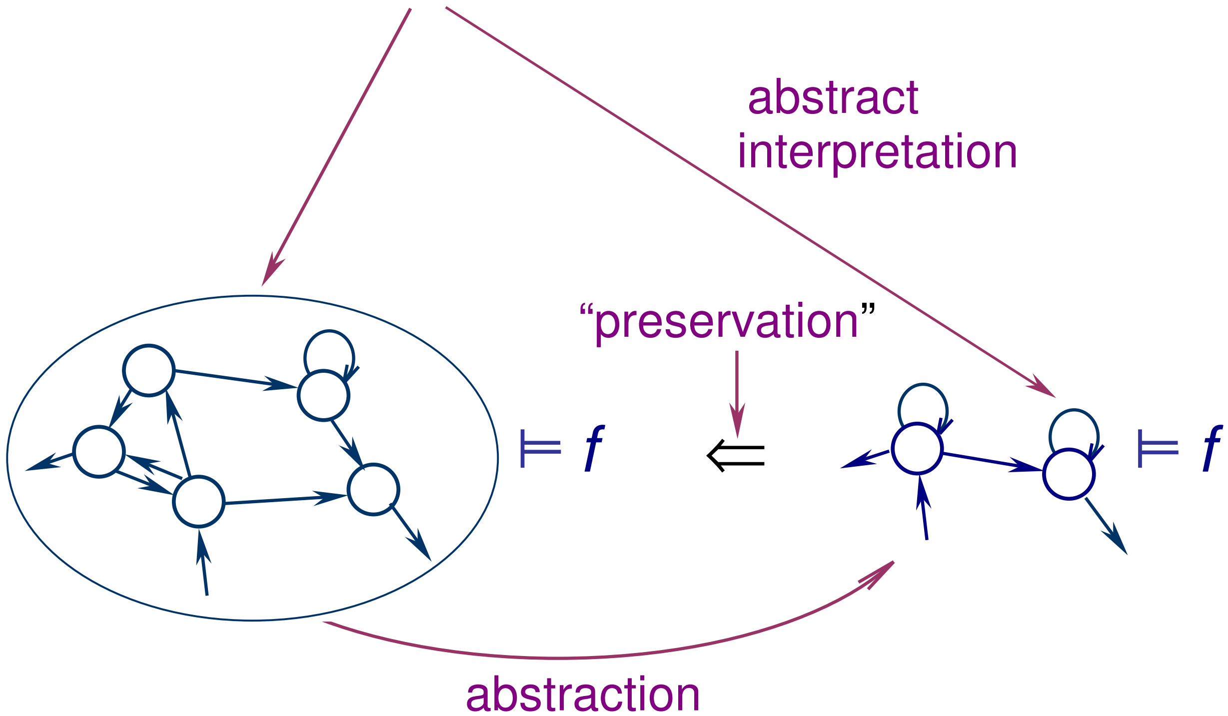
Task: *to verify (model-check) the control layer of the wireless ATM communication protocol Mascara*

?????

Abstraction and Compositional Techniques

Abstraction:

Does system S satisfy property f ?



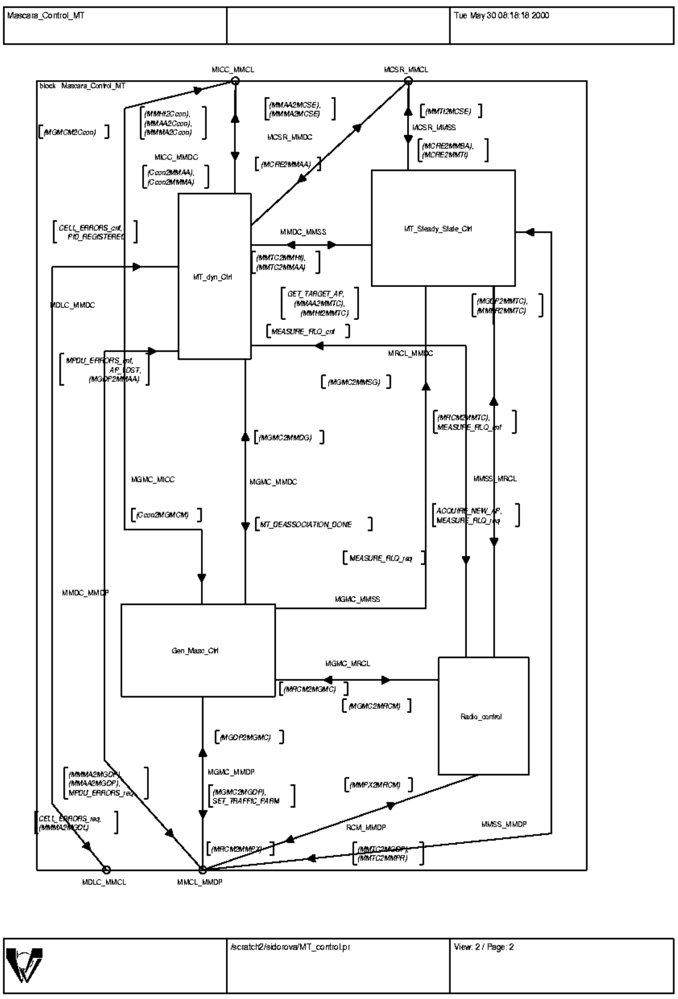
Safe abstractions:

Every property checked to be true on the abstract model holds for the concrete one as well

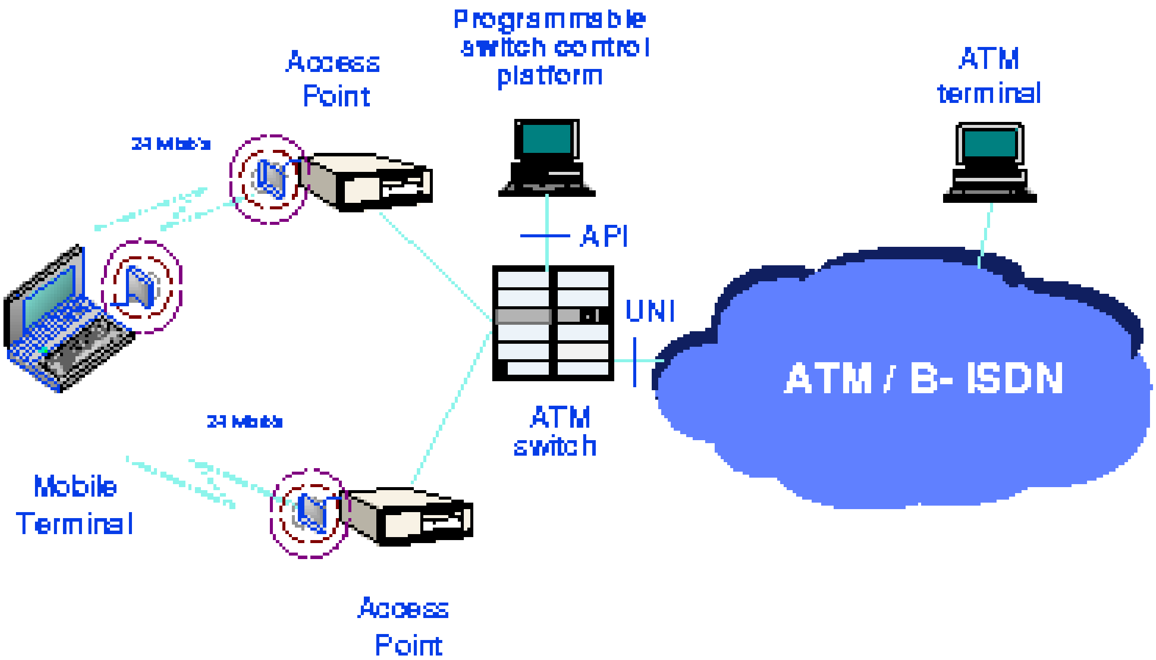
Abstraction and Compositional Techniques

Compositional approach:

Divide et impera!



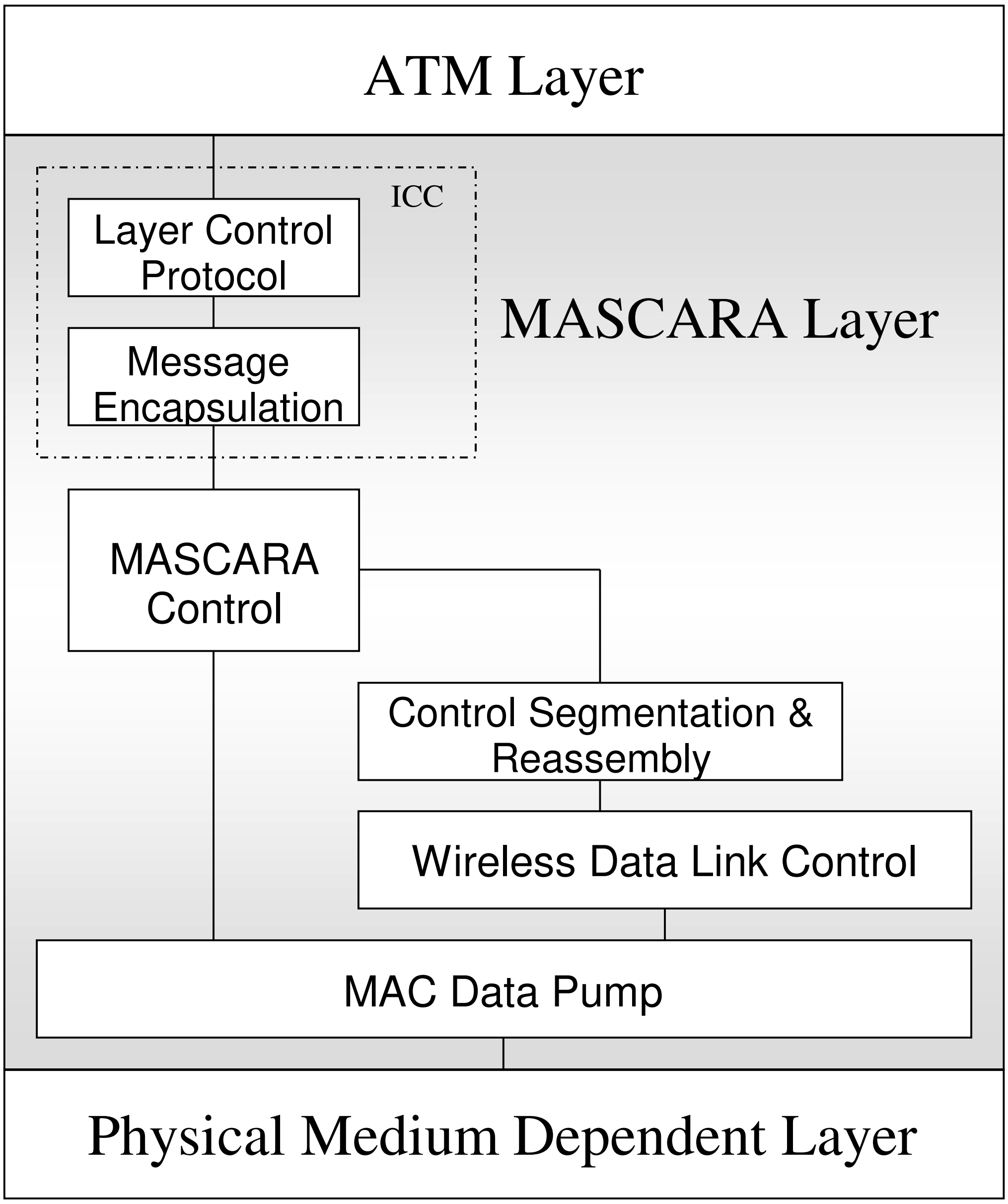
Mascara Protocol



A medium-access layer for wireless ATM communication in local area networks, developed within the WAND project (Intracom)

- *Cell-delineation*
- *Transmission frame adaptation*
- *Header error control*
- *Cell-rate decoupling*
- *Operating over radio links*
- *Mobility features*

Mascara Protocol (continued)



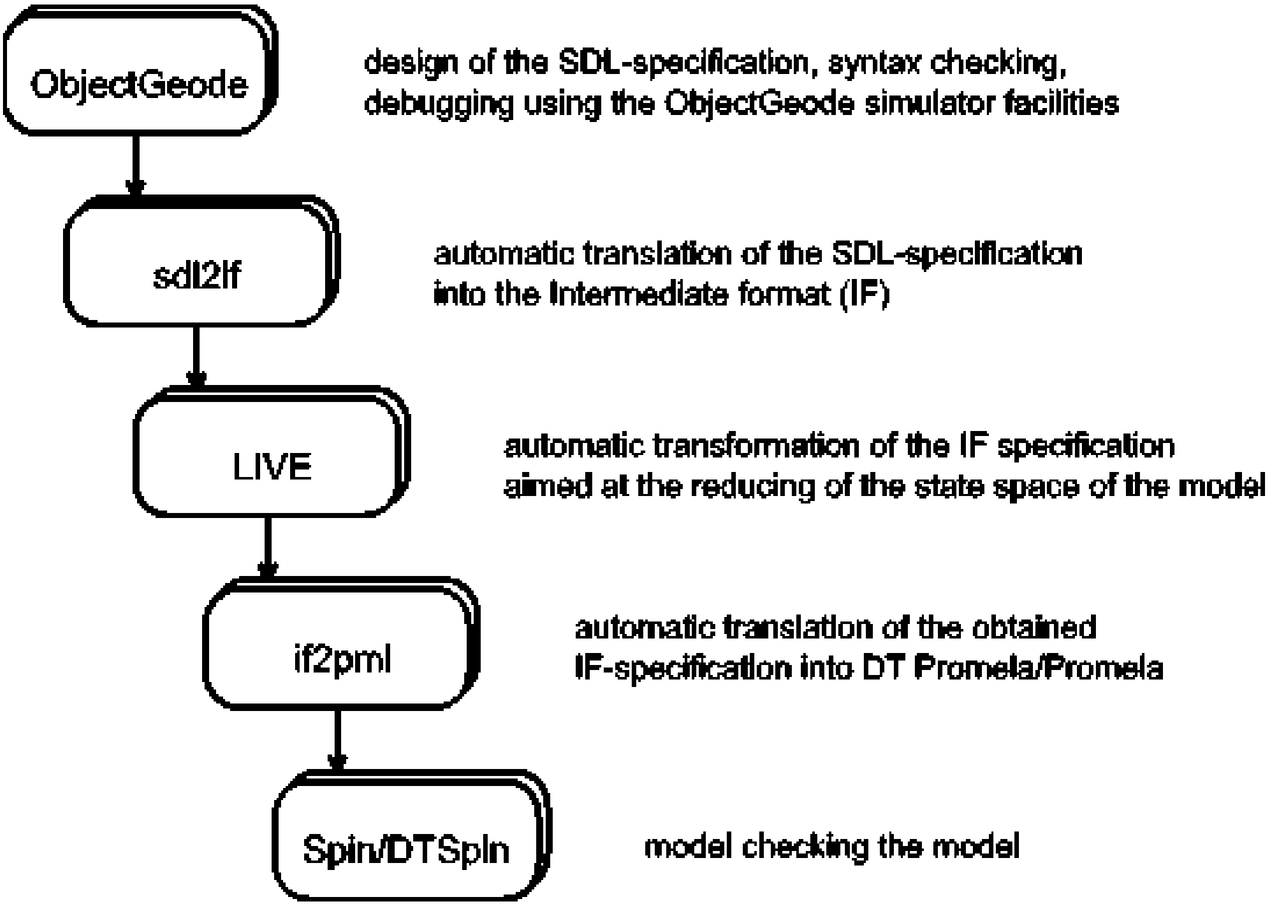
Mascara Control

Function: *to maintain and to manage associations and connections*

Sub-entities:

- *Dynamic Control*
  sets up and tears down associations and connections, performs address management and resource allocation
- *Steady-State Control*
  monitors current associations and the quality of radio environment, initiates in time handovers (change of associations)
- *Radio Control*
  upon a request instructs the radio modem to tune into a specific radio frequency and reports back whether it succeeded or not
- *Generic Mascara Control*
  brings into operation and terminates the entire MAC layer.

Model-checking Environment



Bottom-up Verification

1. Splitting the system

global specification $\Rightarrow$

extract a block of processes to verify
from the global specification $\Rightarrow$

open specifications of sub-blocks $\Rightarrow$

close the specification
by adding an environment $\Rightarrow$

closed specification of sub-blocks

Bottom-up Verification
(continued)

2. Verification of subsystems

specification of a block $\Rightarrow$

formulate the requirements, formalize them;
construct abstractions;

model-check the abstract model ; $\Rightarrow$

verification results $\Rightarrow$

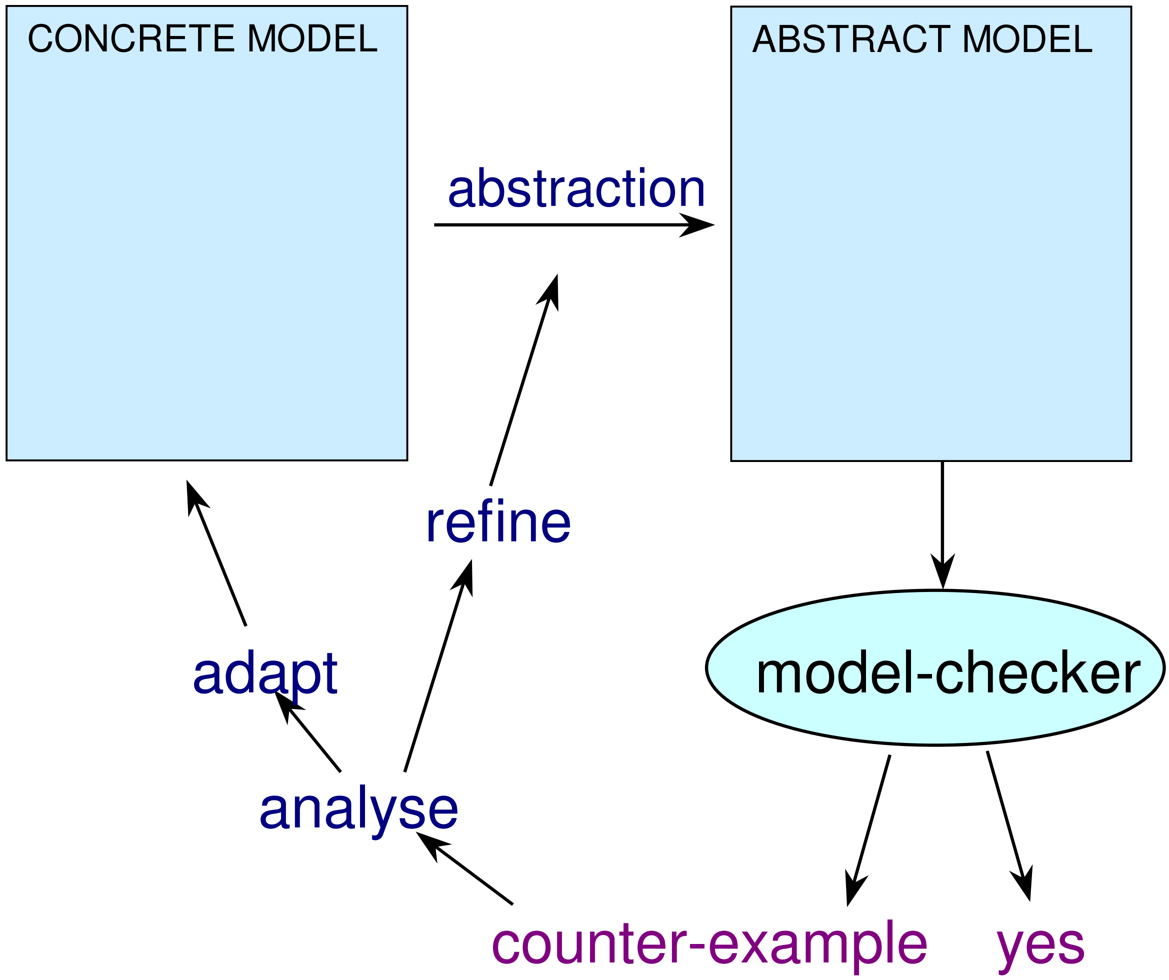
interpret the results/debug…

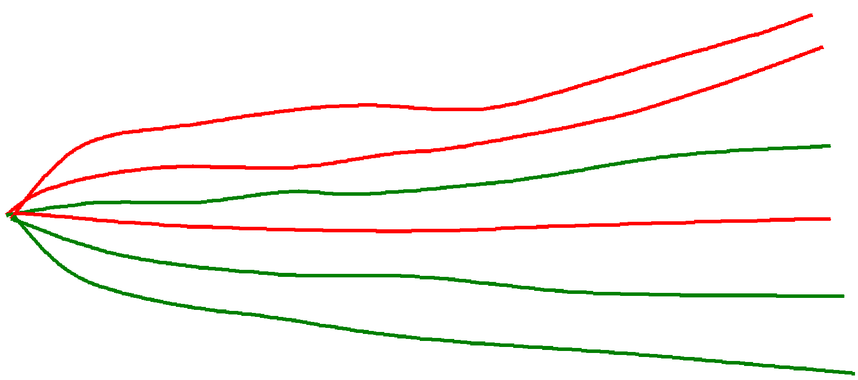
3. Going up

basing on the results of step 2,
construct the abstractions of the components;
build a system consisting of several sub-blocks and verify it.

Proceed going a level up every time.

Applying Safe Abstractions





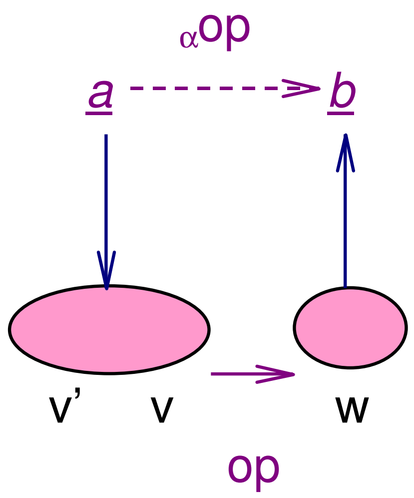
Data Abstractions

<u>Idea:</u>

Replace data values by descriptions (abstract values) and "mimic" operations. Then every universal property checked to be true on the abstract system holds for the concrete system as well.

<u>a</u>, <u>b</u> $\in$ Val
${}_{\alpha}$op $\subseteq$ Val $\times$ Val

γ: ${}_{\alpha}$Val $\rightarrow$ P(Val)
concretisation function



γ: P(Val) $\rightarrow$ ${}_{\alpha}$ Val
abstraction function

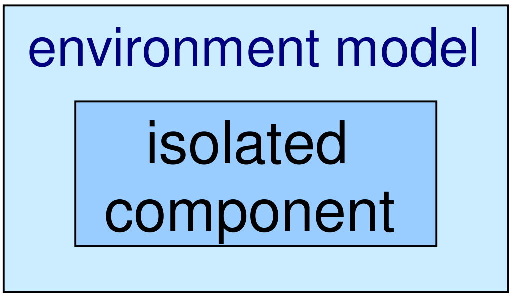
v, v', w $\in$ Val
op $\subseteq$ Val $\times$ Val

(α, γ) Galois Insertion

[Cousot & Cousot 77
Clarke, Grumberg & Long 92
Graf & Loiseaux 92
Dams, Gerth & Grumberg 94]

Modelling Environment

(control abstractions)



1. *Chaotic environment*
   - *all traces are allowed ⇒ abstraction is safe*
   - *can be constructed fast and routinely*
   - *the structure of the entity under investigation is left untouched*
   - *can cause "false-negatives"*
   - *redundant behaviour can increase the state space*

2. *Embedded chaotic environment*
   - *signals from the environment "coming" only when the modelled system ready to get them*

3. *Designer-constructed environment*
   - *models the assumptions under which the component was designed*
   - *soundness of the verification results depends on the quality of the model*

Time abstraction

2-values abstraction:

off $\to$ *off*

0, 1, 2, … $\to$ *on*

3-values abstraction:

off $\to$ *off*

0, 1, 2, … $\to$ *0*

1, 2, 3, … $\to$ *1*

Spin ? DT Spin

Time-dependant property $\Rightarrow$ *verification of the concrete model in DT Spin*

Property that should hold for all timer settings $\Rightarrow$ *verification of abstract model in Spin*

No clear dependence on time $\Rightarrow$ *???*

Time abstraction (continued)

No clear dependence on time ⇒
try to verify the abstract model first;
if the property is proved stop
otherwise switch to the concrete model

Reason:

An abstraction adds some behaviour ;

if the property is disproved, it should be checked whether the erroneous trace given by Spin is a real error or a false error caused by adding behaviour.

DT Spin guarantees that timers expire in the correct order ⇒ less "false-negatives"

An Example: Abstracting Radio Control

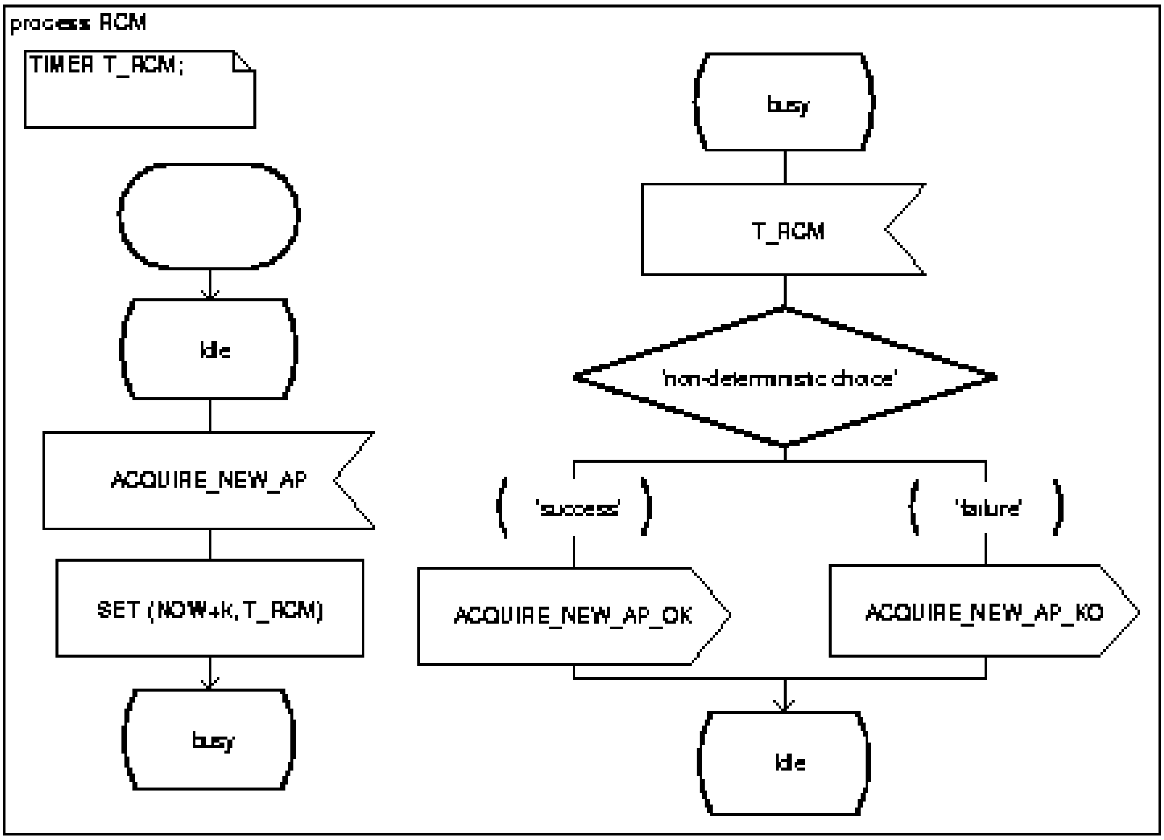
Property (formalized in LTL)

" Whenever, after initialisation, the radio control manager receives a request *AcquireNewAP(newchannel)*, the RCM-process responds either positively or negatively (*AcquireNewAPok* or *AcquireNewAPko*).

Moreover, the answer is sent in a given amount of time after getting the request."

was proved for the component closed in a chaotic environment with the only restriction on the number of signals that can be send per time unit.

An Example: Abstracting Radio Control
(continued)



Abstract Radio Control Manager

Verifying Mascara

- Reachability checks: *"A distinguished state will be eventually reached".*
  - Analysis of the unreachable code detected by Spin
  - Assertion violations
- Safety properties: *"Nothing bad may happen".*
  - Variables are not out of range
- Liveness properties: *"Something good has to happen".*
- Response properties: *"Every request is eventually confirmed" and "Every acknowledgment is caused by a previous request.*
  - Bounded response properties: *The confirmation comes within some defined amount of time.*

Errors found

- programming errors
  - Forgotten branches in case distinctions
  - Mal-considered limit cases in loops
  - …
- race conditions
- ambiguous receiver
- unspecified reception
- variables out of range
- components waiting for a reception confirmation that does not come

A Time-Dependent Safety Property
(how far we could get)

Safety requirement:

"never the access point relinquishes an association before the mobile terminal does"

In *LTL:* $\Box(\varphi_{\text{mt-lost}} \rightarrow \varphi_{\text{ap-lost}})$

$\varphi_{\text{mt-lost}}$ – the *access point* gives up the association sending the signal *MT_Lost.*

$\varphi_{\text{ap-lost}}$ – the *mobile terminal* gives up the association.

The property holds if $min(\tau_{AP}) > max(\tau_{MT})$

$$\tau_{AP} \geq (Max\_Time\_periods + 1) * T_{iaa\_poll} + (IAA\_Max - 1) * T_{frame\_start}$$

$$\tau_{MT} \leq (Max\_Cellerrors) * T_{GDP\_period} + (Max\_AP\_Index + 1) * T_{RCM}$$

Conclusions

- Model-checking is an effective way of *debugging.*
- At the *first stage* of the debugging, it is easier and better to use *simulation*, not model-checking.
- Reports of a model-checker about *unreachable code* are a cheap and easy way for the following debugging.
- *Shortest trail* option greatly simplifies the analysis of the cause of the error.
- Verifying simple properties is very fruitful for finding errors.
- *Tools supporting abstractions* are needed! Just with applying the *Live* tool, the state space is in average reduced by one order of magnitude.