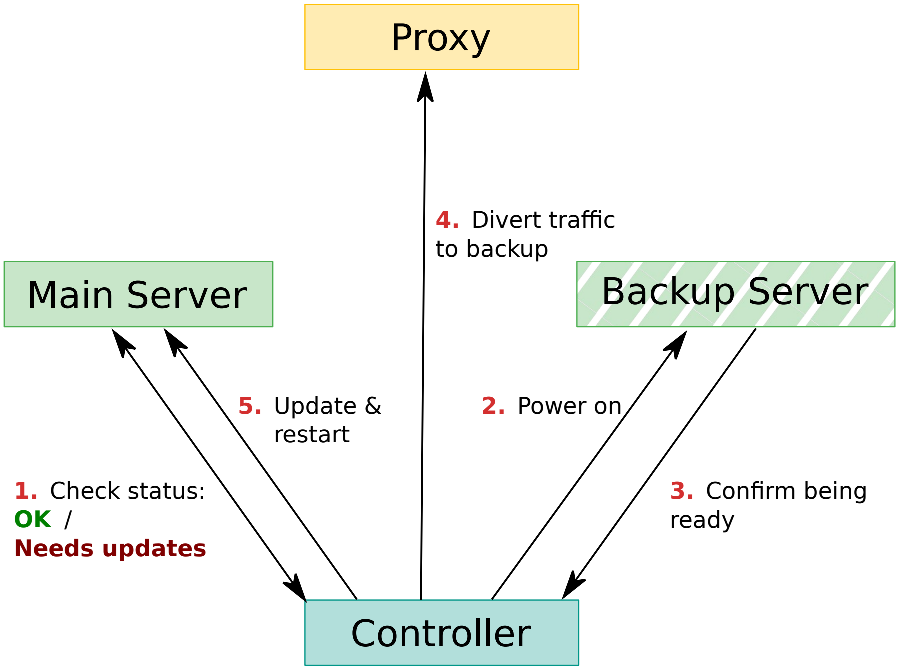
Semi-Dynamic Session Types for ABS

Student: Anton Haubner
Examiner: Prof. Dr. rer. nat. Reiner Hähnle
Supervisor: Eduard Kamburjan, M.Sc.

Motivation - Protocols

- Interactions within a distributed system may require a certain order
- Example: Swapping out a web server for maintenance must adhere to this protocol



Motivation - Protocols

Manually ensuring compliance with protocol is…

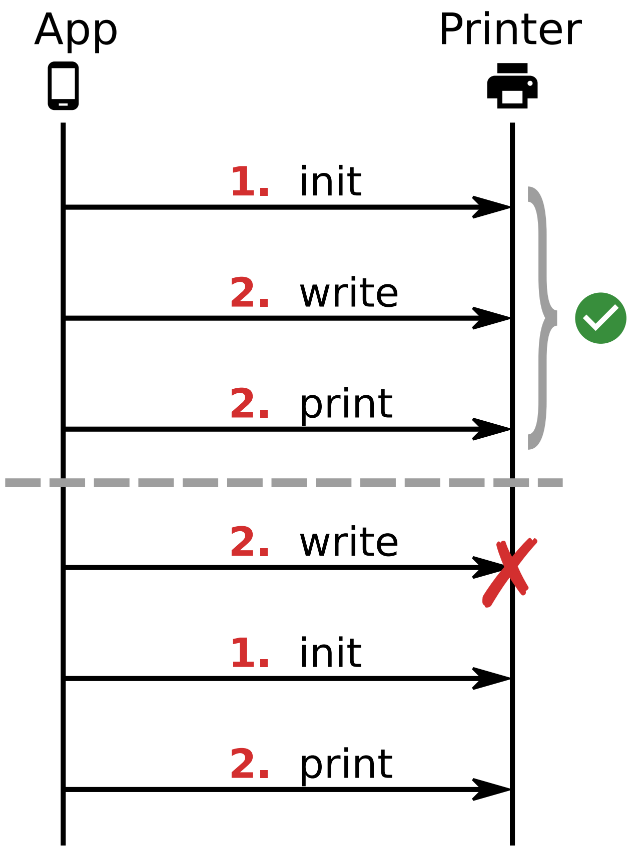
- susceptible to human error
- Labor-intensive
- Impossible for large systems

Formal specification
+ automatic verification / enforcement can help

Motivation – Enforcing Scheduling

- Messages sent in the right order must also processed in that order
- Issue if messages can arrive disordered or endpoints may process them in any order
- Static verification can not prevent this

⇒ processing order must be enforced / corrected at runtime



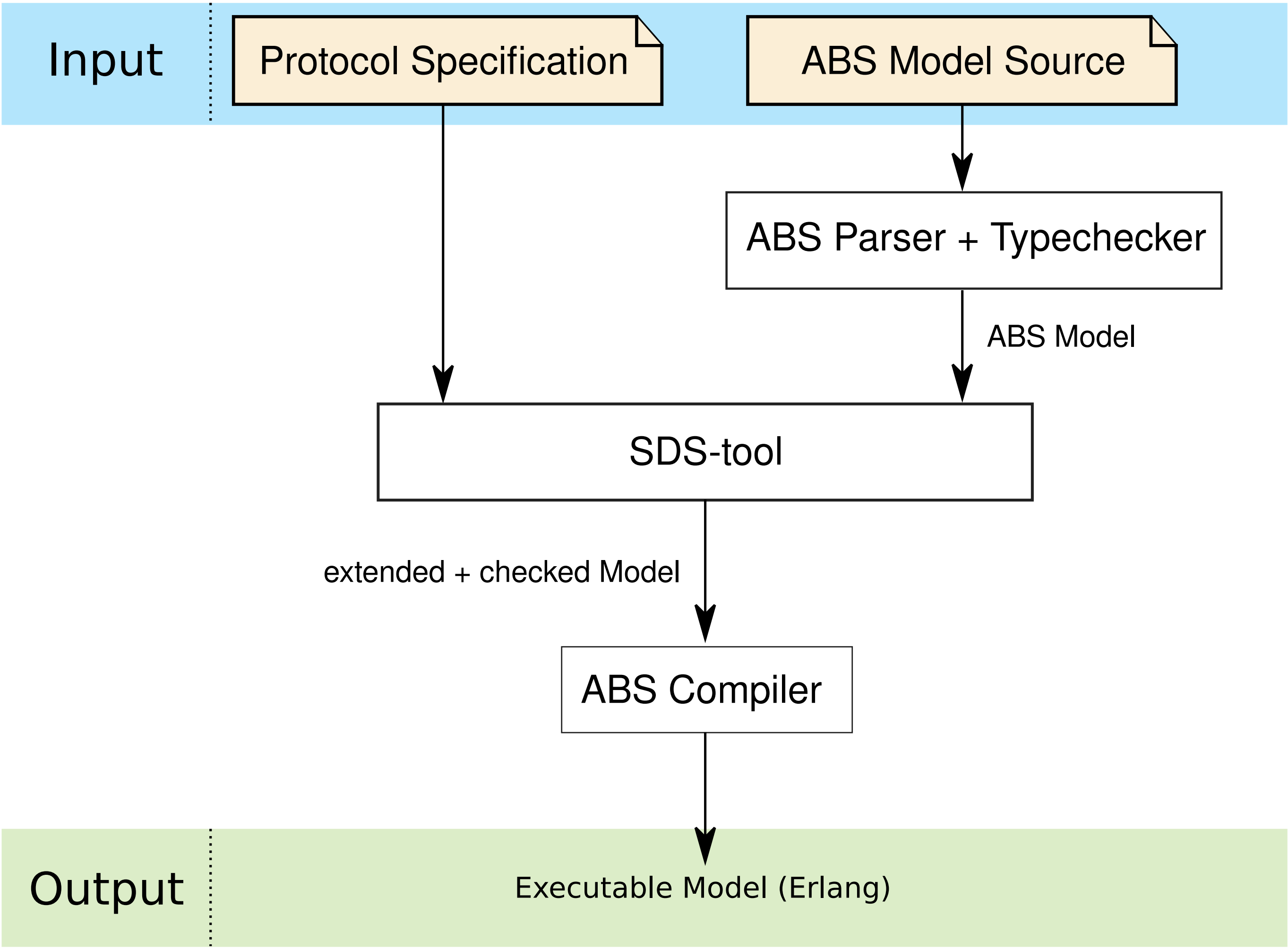
Solution: Combine static & dynamic approaches

- We need a specification language
  → Session Types
- We to **verify** implementations against a protocol
  → Static verification of call orders etc.
- We need to **enforce** message processing orders at runtime
  → Schedulers

Implementation Environment: ABS Language

- ABS is intended for modeling „distributed, object-oriented“ software
- designed to facilitate its verification
- ABS allows defining schedulers

Workflow Overview

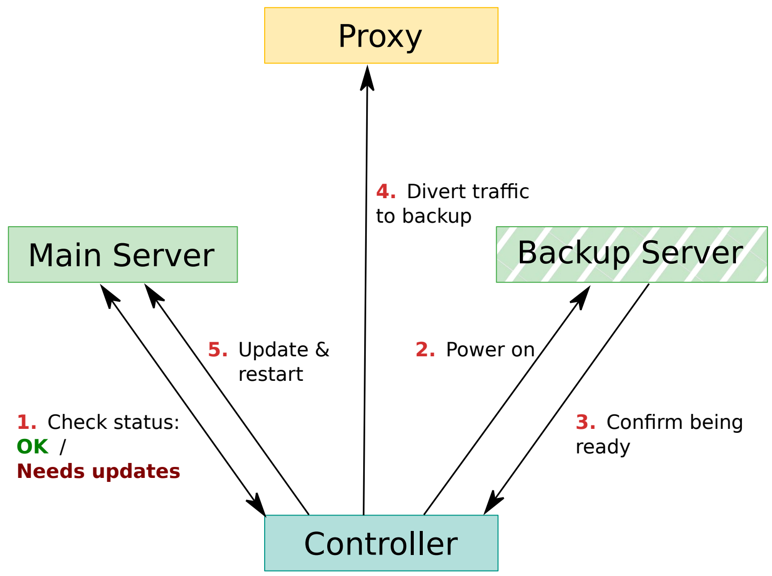


Structure

- Specification Language
- Static Verification
- Dynamic Enforcement
- Evaluation

Specification Language: Session Types

$$
\begin{aligned}
&\text{Controller} \xrightarrow{f} \mathit{MainServer}:\ \mathit{checkStatus}.\\
&\mathit{MainServer}\{\\
&\quad \mathit{MainServer} \downarrow f\ (\mathsf{Ok})\ ,\\
&\quad \mathit{MainServer} \downarrow f\ (\mathsf{NeedsUpdate})\\
&\qquad \ldots\\
&\qquad \mathit{Controller} \xrightarrow{f'} \mathit{Proxy}:\ \mathit{divert}\\
&\qquad \ldots\\
&\}\\
&\ldots
\end{aligned}
$$



Session Type Capabilities

- express interaction (calls, retrieving values)
- Branchings & loops
- ABS cooperative scheduling (suspension, reactivation)

Specification Sanity Checks

Specification language allows expressing impossible protocols

- Releasing control on non-active participants
- Fetching results from unresolved futures
- …

$$A \xrightarrow{f} B : m. A \uparrow f$$

⇒ Some basic sanity checks are necessary

Specification Sanity Checks – Analysis by Execution

$$A \xrightarrow{f} B : m.\, A \uparrow f$$

„A calls m on B and fetches result (before B computes it!)“

$$\emptyset \overset{A \xrightarrow{f} B:m}{\rightsquigarrow} \{(f : \text{unresolved})\}$$

$$\overset{A \uparrow f}{\rightsquigarrow} \quad \text{error!}$$

abstract state (captures used futures etc.)

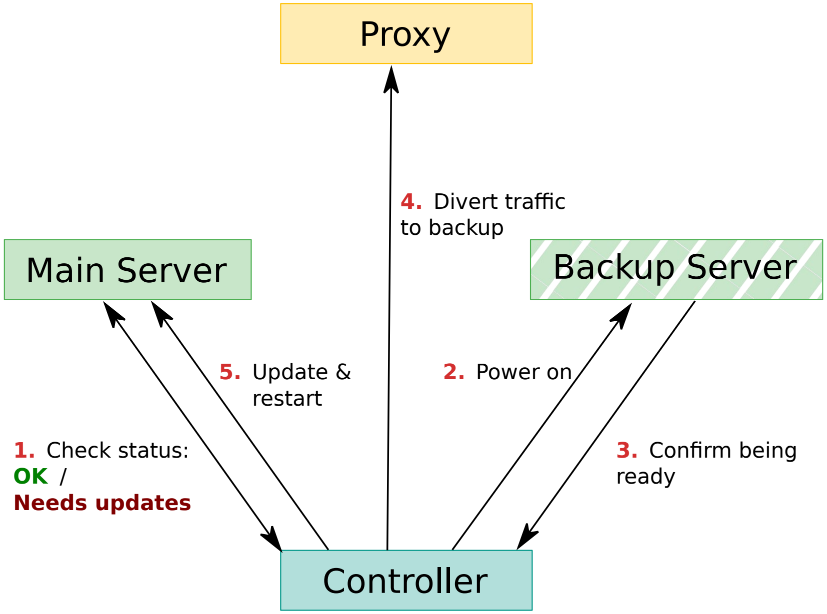
type components used as instructions of a program

analysis by transition relation

Structure

- Specification Language
- Static Verification
- Dynamic Enforcement
- Evaluation

Static Verification – Compositional Approach



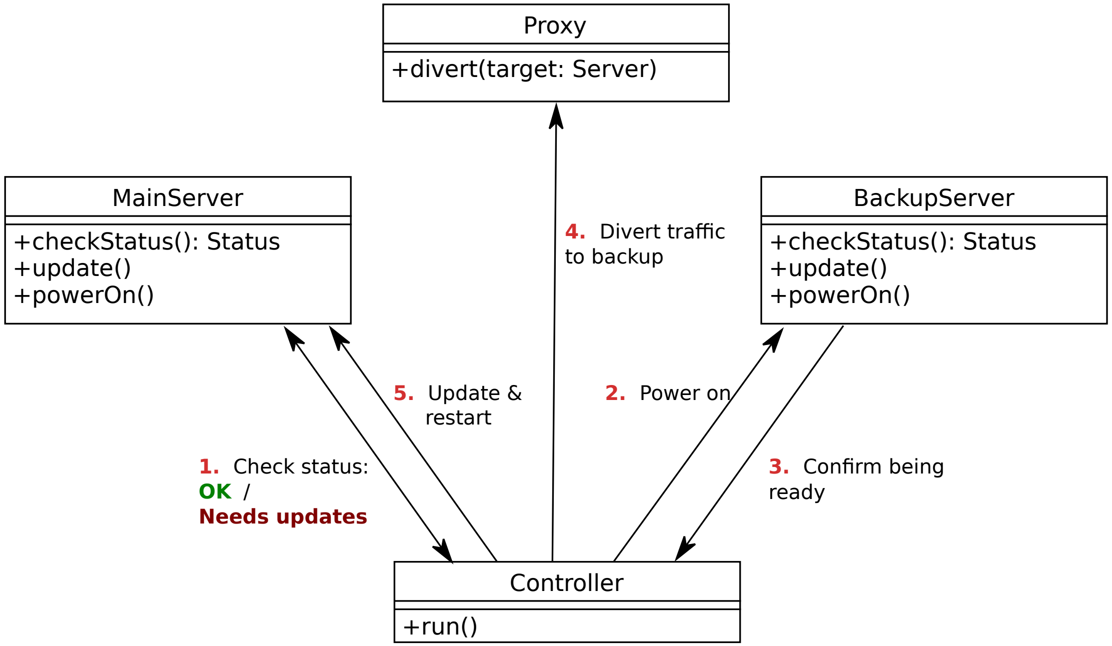
Static Verification – Compositional Approach

TECHNISCHE UNIVERSITÄT DARMSTADT

- every method is verified individually against the session type
- a specification of a method's behavior is extracted from a global protocol
- Verification of method AST against local type

⇒ Projection

⇒ Type System



Projection - Implementation

- Function
  $project : Session\ Type \times \mathbb{S} \times Participant \rightarrow Local\ Session\ Type$

- Extracts parts of behavior relevant to the participant
- Using a set of rules and information from abstract state of validity analysis

Projection: Extraction of a localized specification

Global Session Type

$$
\begin{aligned}
&0 \xrightarrow{f_0} A\colon init.A \xrightarrow{f_1} B\colon m_1.\\
&\quad B \xrightarrow{f_c} C\colon m_c.C \downarrow f_c.B \uparrow f_c.B \downarrow f_1.\\
&A \xrightarrow{f_2} B\colon m_2.\\
&\quad B \to D\colon m_d.\ldots
\end{aligned}
$$

Projection: Extraction of a localized specification

Global Session Type

$$0 \xrightarrow{f_0} A\colon init.\ A \xrightarrow{f_1} B\colon m_1\ .$$
$$B \xrightarrow{f_c} C\colon m_c\ .C \downarrow f_c.\ B \uparrow f_c.B \downarrow f_1.$$
$$A \xrightarrow{f_2} B\colon m_2.$$
$$B \rightarrow D\colon m_d\ .\ldots$$

Projection on **object** B

Object Local Type for B

$$A?_{f_1} m_1.$$
$$C!_{f_c} m_c.\,\mathrm{Get}\ f_c.\,\mathrm{Put}\ f_1.$$
$$A?_{f_2} m_2.$$
$$D! m_d.\ldots$$

Projection: Extraction of a localized specification

TECHNISCHE UNIVERSITÄT DARMSTADT

Global Session Type

$0 \xrightarrow{f_0} A\colon init.\ \boxed{A \xrightarrow{f_1} B\colon m_1}\ .$

$\boxed{B \xrightarrow{f_c} C\colon m_c}\ .C \downarrow f_c.\ \boxed{B \uparrow f_c.B \downarrow f_1.}$

$\boxed{A \xrightarrow{f_2} B\colon m_2.}$

$\boxed{B \rightarrow D\colon m_d}\ \ldots.$

Projection on **object** B

Object Local Type for B

$\boxed{A?_{f_1} m_1.}$

$\boxed{C!_{f_c} m_c.\,\mathrm{Get}\ f_c.\,\mathrm{Put}\ f_1}\ .$

$A?_{f_2} m_2.$

$D! m_d.\ldots.$

Projection on **method** B::m_1 for future f_1

Method Local Type for B::m_1

C $!_{f_c} m_c.\,\mathrm{Get}\ f_c.\,\mathrm{Put}\ f_1$

Verification using a Type System

```
String m1() {
  fc = c!mc();

  String result = fc.get;
  print(result);

  return result;
}
```

$C\ !_{f_c} m_c.$

$\text{Get } f_c.$

$\text{Put } f_1$

$$\frac{\Delta(\mathit{interface}(e_p)) = q \qquad \Gamma; F; \Delta \vdash \overline{s} : M}{\Gamma; F; \Delta \vdash \texttt{this}.f\ \texttt{=}\ e_p\texttt{!}m(\overline{e_p})\texttt{;}\ \overline{s} : q!_f m.M}\ \textsc{Call}$$

$$\frac{\forall f'. v \notin \Gamma(f') \qquad \Gamma[f := \Gamma(f) \cup \{v\}]; F; \Delta \vdash \overline{s} : M}{\Gamma; F; \Delta \vdash [T]\ v\ \texttt{=}\ \texttt{this}.f\texttt{.get;}\ \overline{s} : \text{Get } f\,[(C)]\,.M}\ \textsc{Get}$$

...

$$\frac{\mathit{commInert}_{\Gamma;F;\Delta}(e)}{\Gamma; F; \Delta \vdash \texttt{return}\ e\texttt{;} : \text{Put } f}\ \textsc{ReturnNoMsg}$$

Now, method bodies can be verified against local type using a *type system*

- Specified actions must each be implemented by a corresponding statement
- Control structures must match type branchings / loops
- Correct usage of variables, futures etc. is checked
- Data type checking is left to the ABS compiler
- ...

Verification using a Type System

```
String m1() {
  fc = c!mc();

  String result = fc.get;
  print(result);

  return result;
}
```

$\leftarrow$ C $!_{f_c} m_c.$

$\leftarrow$ Get $f_c.$

$\leftarrow$ Put f_1

$$\frac{\Delta(\mathit{interface}(e_p)) = q \quad \Gamma; F; \Delta \vdash \overline{s} : M}{\Gamma; F; \Delta \vdash \texttt{this}.f\ \texttt{=}\ e_p\texttt{!}m(\overline{e_p})\texttt{;}\ \overline{s} : q!_f m.M}\ \textsc{Call}$$

$$\frac{\forall f'. v \notin \Gamma(f') \quad \Gamma[f := \Gamma(f) \cup \{v\}]; F; \Delta \vdash \overline{s} : M}{\Gamma; F; \Delta \vdash [T]\ v\ \texttt{=}\ \texttt{this}.f\texttt{.get;}\ \overline{s} : \text{Get}\ f\ [(C)]\ .M}\ \textsc{Get}$$

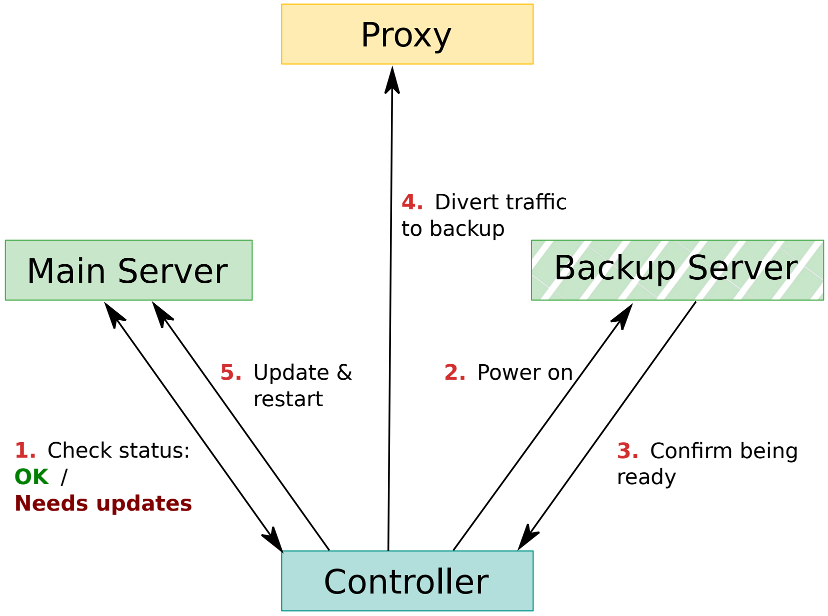
...

$$\frac{\mathit{commInert}_{\Gamma;F;\Delta}(e)}{\Gamma; F; \Delta \vdash \texttt{return}\ e\texttt{;} : \text{Put}\ f}\ \textsc{ReturnNoMsg}$$

Design Goals

- Soundness
- Support current implementation of ABS
- Leave developers as much freedom as possible

  i. e. allow all additional code that does not influence session


Proxy
4. Divert traffic
to backup
Main Server
Backup Server
5. Update &
restart
2. Power on
1. Check status:
OK /
Needs updates
3. Confirm being
ready
Controller

Demo

Structure

- Specification Language
- Static Verification
- Dynamic Enforcement
- Evaluation

Dynamic enforcement of protocols

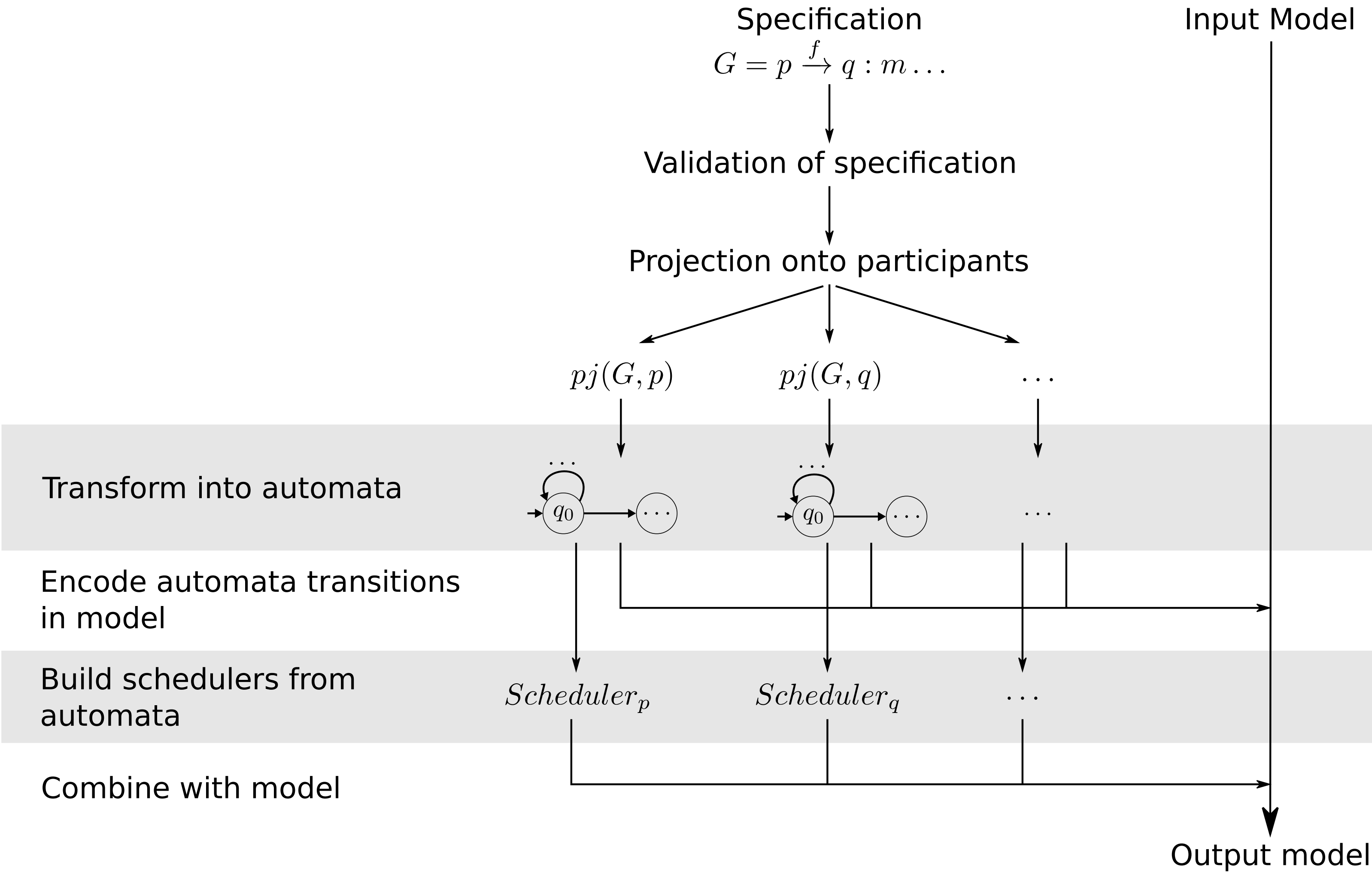
TECHNISCHE UNIVERSITÄT DARMSTADT

What does it take to enforce a protocol?

At every point of execution, we need to know…

- How far we have progressed in the protocol
- Which method activations are allowed next
- For every participant

From Specification to Scheduler

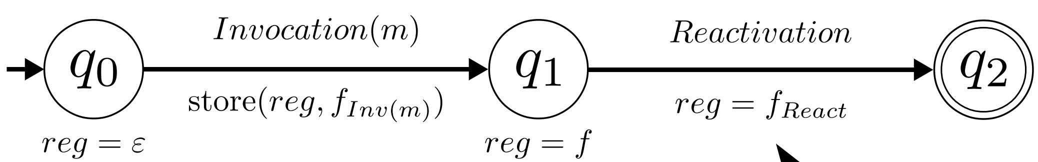


Session Automata – Example

$$p?_f : m.Await(f, f').React(f)$$

„Receive call on m as future f, then suspend, then reactivate f“

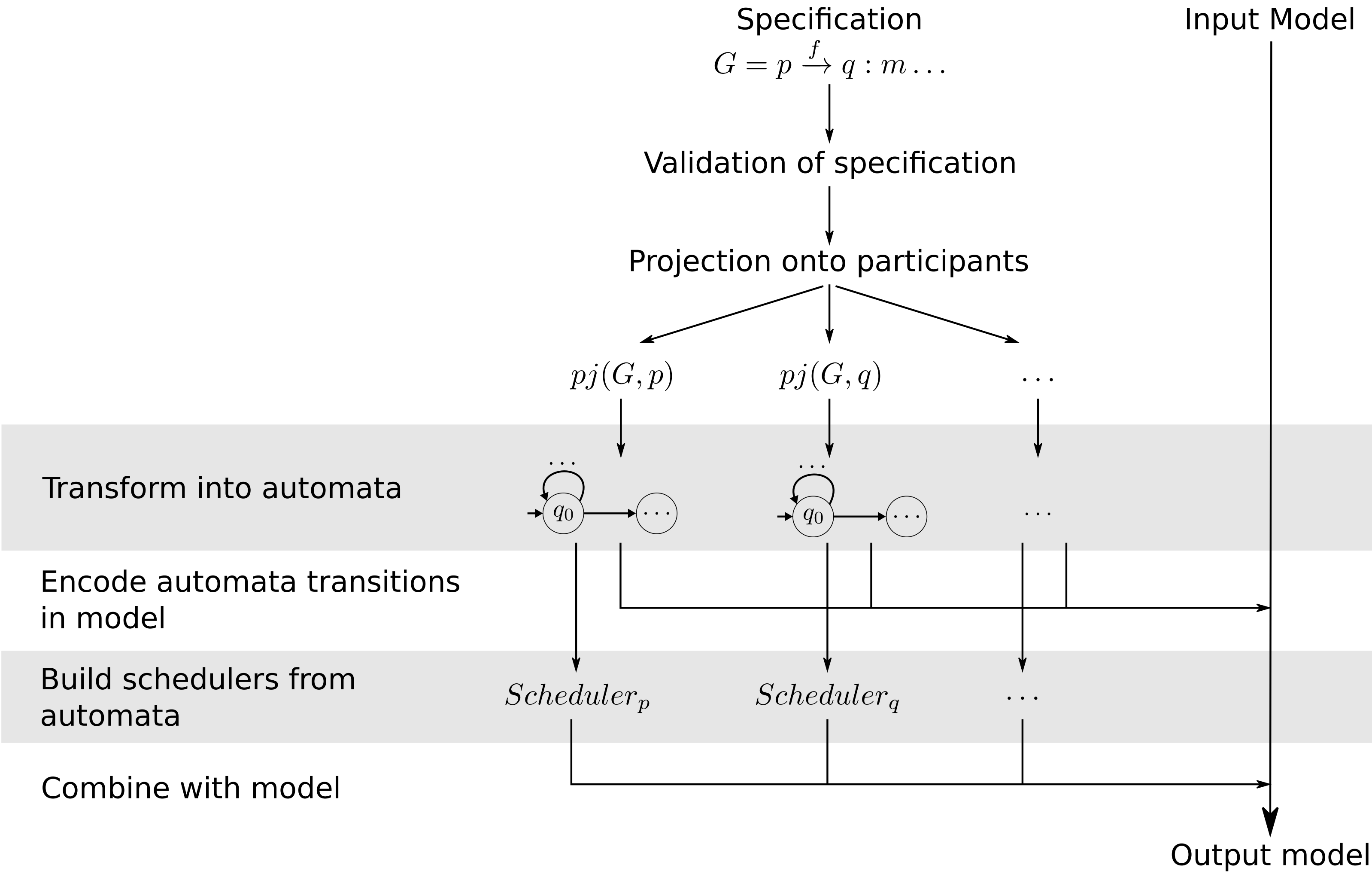
$$\Downarrow$$



Automaton state contains registers for storing futures

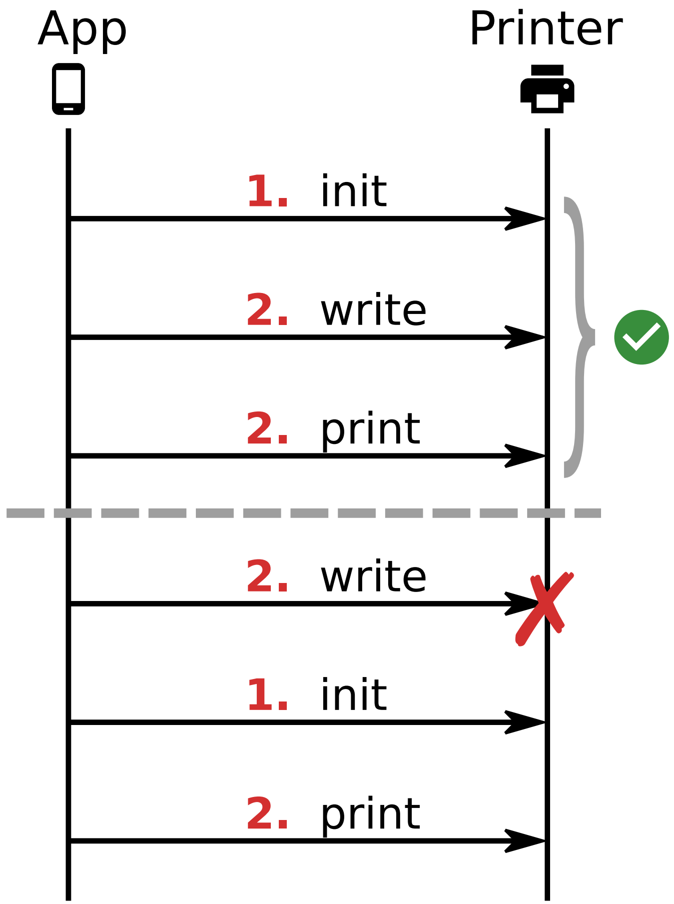
automaton allows reactivation only, if future to be reactivated belongs to original invocation

From Specification to Scheduler



Scheduling Functions

- Input
  - possible (re-)activations
  - automaton state
- Ouput
  - selected (re-)activation
  - Or nothing → pauses
- Called upon
  - invocations, reactivations, ...



Demo

Runtime Verification of Method Side Effects

- Specification so far describes only structure of interactions
- but not side-effects of method calls etc.
- Session types are extended with post-conditions for interactions, e. g.

$$p \xrightarrow{f} q : m.\, q \downarrow f \, \langle q.\mathrm{attrib} = 42 \rangle \ldots$$

Runtime Verification of Method Side Effects

TECHNISCHE UNIVERSITÄT DARMSTADT

- Hard to verify statically
- Easy to check at runtime
  - especially since AST is being modified anyway
- By introducing assertions :

$$p \xrightarrow{f} q : m.\, q \downarrow f \,\langle q.\mathrm{attrib} = 42 \rangle \ldots$$

$$\Rightarrow$$

```
Unit m() {
   …
   if (automatonState == 2)
     assert(attrib == 42);
}
```

Structure

- Specification Language
- Static Verification
- Dynamic Enforcement
- Evaluation

Evaluation – Main Questions

Static Verification

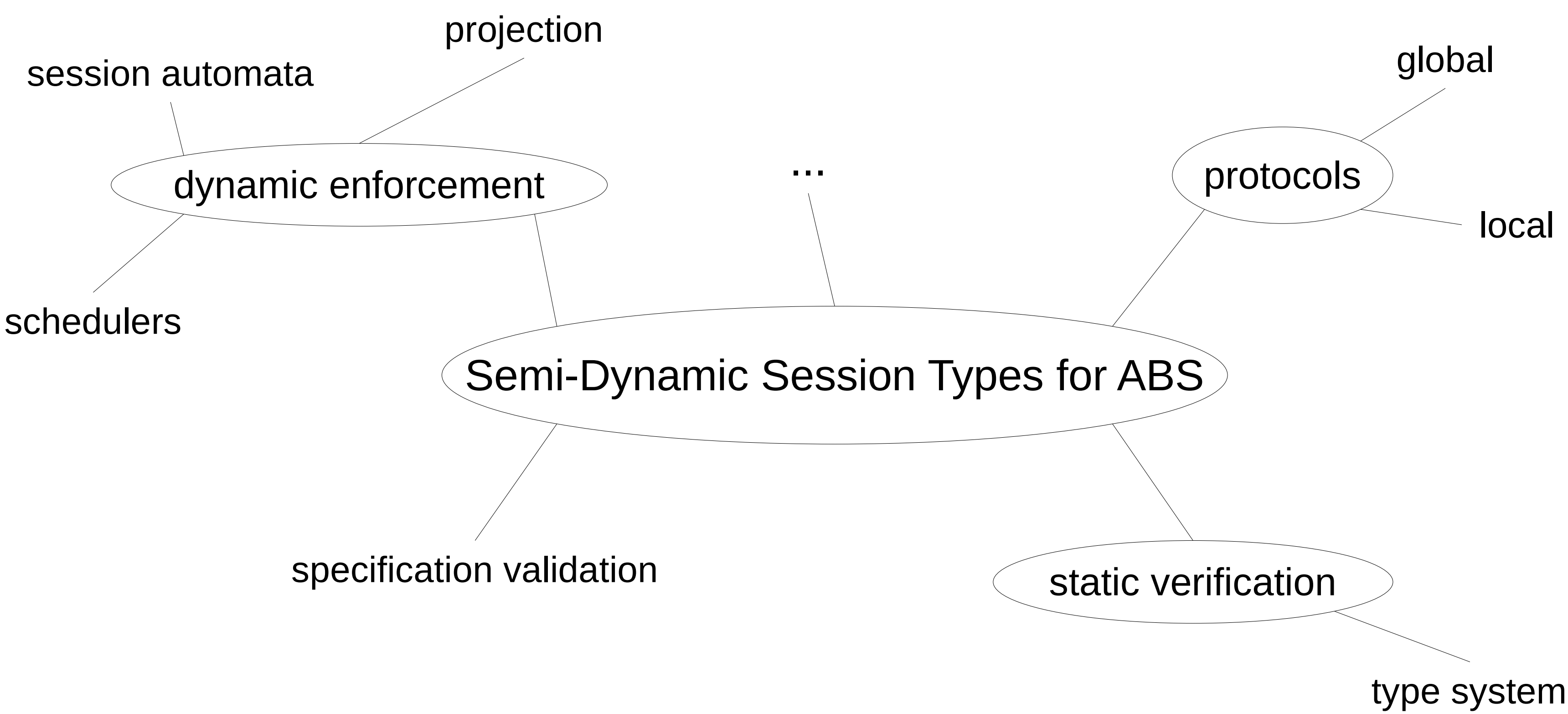
- Does the type system meet design goals?
- Soundness?
- Modeling freedom?

Dynamic Enforcement

- Is method activation order reliably ensured?
- How do schedulers affect execution performance?

- TODO

Thank you for listening



based on

- „Session-based compositional analysis for actor-based languages using futures."
- „Stateful Behavioral Types for ABS“

by Kamburjan, E., Din, C. C., Chen, T.

Image sources

- Google Material Design Icons have been used in this presentation
- Source: https://github.com/google/material-design-icons
- License: Apache 2.0