Computer Supported Modeling and Reasoning

David Basin, Achim D. Brucker, Jan-Georg Smaus, and Burkhart Wolff

April 2005

http://www.infsec.ethz.ch/education/permanent/csmr/

Higher-Order Logic Applications: Refinements

Burkhart Wolff

Overview

In previous weeks, we saw various embeddedings in HOL:

- Imperative languages
- Functional languages
- Fragments of Specification Lanuages (HOL, Z)

Can we apply these theories to development methods such as

Refinement ?

Can we apply HOL to prove the relations between functions, operations, processes, architectures?

Rough Overview

Various Refinement Methods are described in the literature:

- Observational/Behavioural Equivalence
- Forget/Restrict/Identify-Method
- Operation Refinement, Data Refinement [Spi92]
- Refinement Calculus
- Process Refinement (CSP [A.W97])
- Machine Refinement (B-Method [Abr96])
- ...

(thousands of articles and many books on the subject. Arbitrary selection by the author).

Common Formal Method Classification

One destinguishes:

- Data-Oriented Modelling Techniques:

  one system step involving
  complex transformation of
  input, output and state data,

- Behavioral Modelling:

  sequences of system steps considering
  the evolution of input, output and states.

Common Formal Method Classification

One destinguishes:

- Data-Oriented Modelling Techniques:

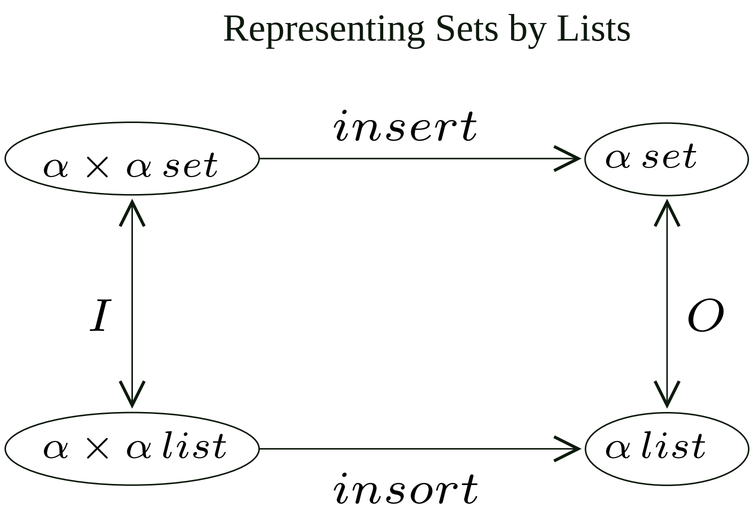
  data refinement(Z, KIV, B),
  algebraic specification techniques
  (Behavioural Equivalence),
  Hoare-like calculi (Morgan, Back/Wright)

- Behavioral Modelling:

  process algebras (CSP,CCS,. . .),
  temporal logics

Data Refinement for a Function

A simple example for refining a function:

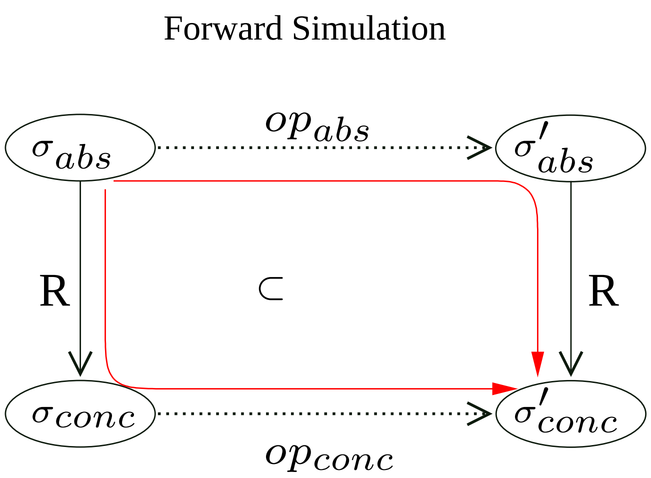


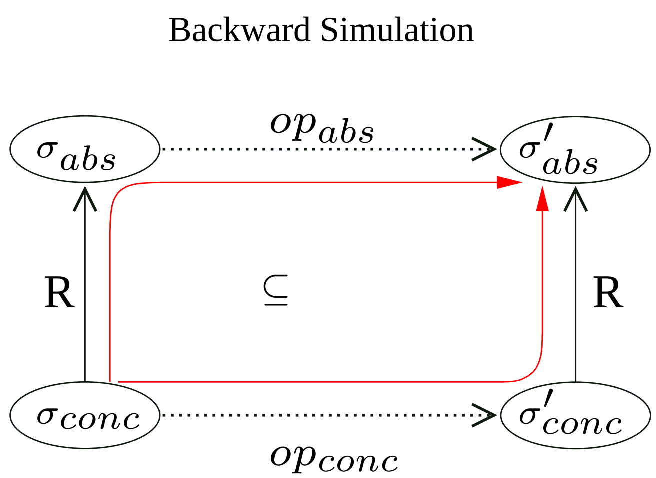
Can this be generalized to operations (i.e. "procedures" with input, output, and an implicit state transition) ?

Data Refinement

Principles of Data-Refinement

- Forward Simulation
- Backward Simulation

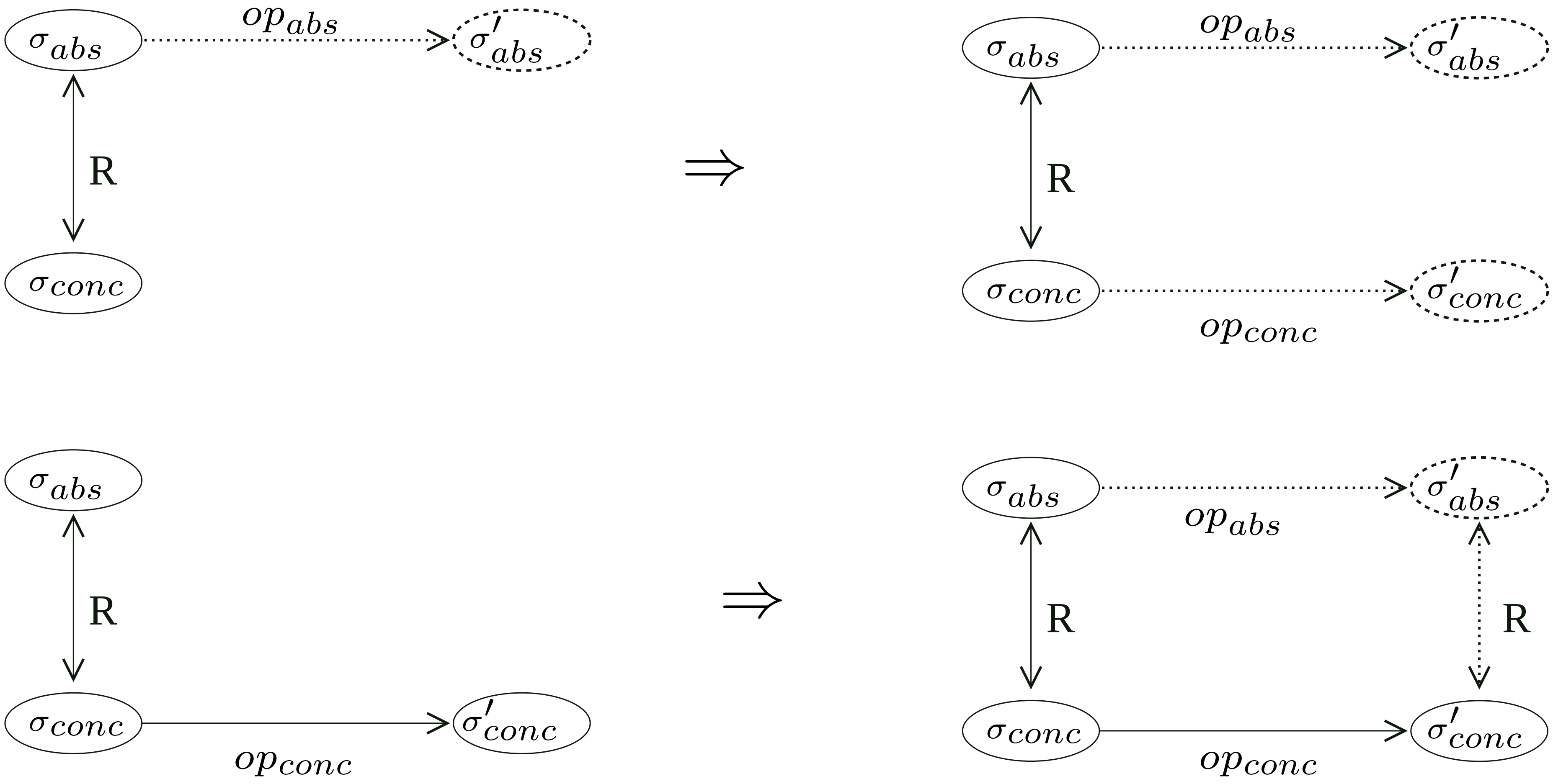




See also [Spi92] and [WD96]!

Data Refinement

Forward Simulation



Data Refinement

Can we

- represent refinement in Isabelle ?
- verify and compare refinement notions ?
- integrate refinement for functions and operations?

YES! In the following, we present a theory of Abstract IOS Specifications and a forward simulation refinement on it. (backward refinement is analogously)

IOS-Forward Simulation

An abstract system IOS-step has the type:

types ('i, 'o, 's) ios_rel = "(('i × 's) × ('o × 's))set"

An Abstract IOS Specification is:
(closely related to a Z operation schema):

```
record ('i,'o,'s) spec =
   init :: "'s set"
   inv  :: "'s set"
   opn  :: "('i, 'o, 's) ios_rel"
```

IOS-Forward Simulation

The generalized abstraction relation on abstract IOS specifications looks as follows:

```
record ('i,'i','o,'o','s,'s') abs_rel =
  i    :: "('i × 'i') set"
  o    :: "('o × 'o') set"
  abs  :: "('s × 's') set"
```

The relation is just a triple of relations on input data, output data and states.

IOS-Forward Simulation

We define a FS-refinement on IOS specifications by its three "proof obligations":

constdefs

```
FS_refine  :: " [('i ,'o,'s) spec,
                 ('i ,'i ','o,'o ','s,'s') abs_rel ,
                 ('i ','o ','s') spec] ⇒ bool"
A \<sqsubseteq>R C ≡
FS_init A R C ∧FS_corr1 A R C ∧FS_corr2 A R C
```

In conceptual notation, we will also write : $A \sqsubseteq_R^{fs} B$ for forward simulation (resp. $A \sqsubseteq_R^{bs} B$ for backward simulation).

IOS-Forward Simulation

The three conditions are:

- FS_init : The set of initial states must be compatible,
- FS_corr2: When an abstract state transition is possible, then a corresponding concrete state transition must be possible,
- FS_corr1: When a concrete operation reaches a target state, then the corresponding abstract must exist.

(Terminology follows [WD96]).

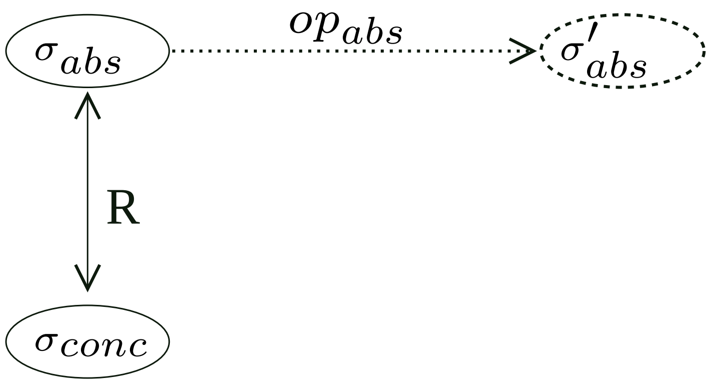
IOS-Forward Simulation

The proof-obligation FS_init

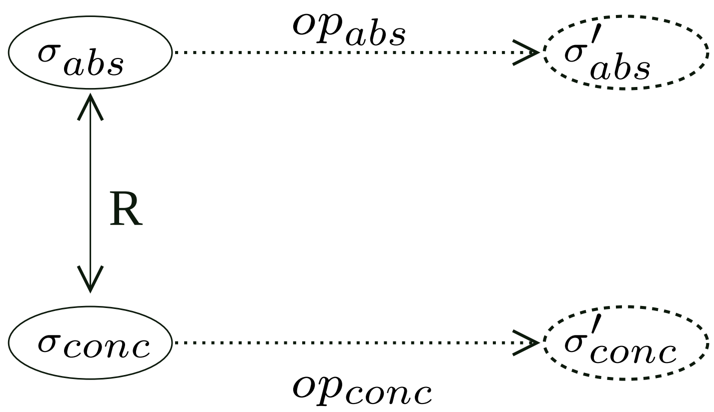
$$
\begin{aligned}
&\text{FS\_init}\ \ A\ R\ C\ \equiv\\
&\quad \forall\ cs{\in}(\text{inv}\ \ C).\ \ cs{\in}(\text{init}\ \ C) \longrightarrow\\
&\qquad\qquad\qquad\qquad \exists\ as{\in}(\text{inv}\ \ A).\ \ as{\in}(\text{init}\ \ A) \wedge (as, cs){\in}\text{abs}\ R
\end{aligned}
$$

IOS-Forward Simulation

Recall the diagrams for FS_corr2



$\Rightarrow$



IOS-Forward Simulation

The formalization for FS_corr2

FS_corr2 A R C ≡

∀as∈(inv A). ∀cs∈(inv C).

∀inp∈(Domain(i R)). ∀inp'∈(Range(i R)).

((inp, as)∈ Domain (opn A) ∧

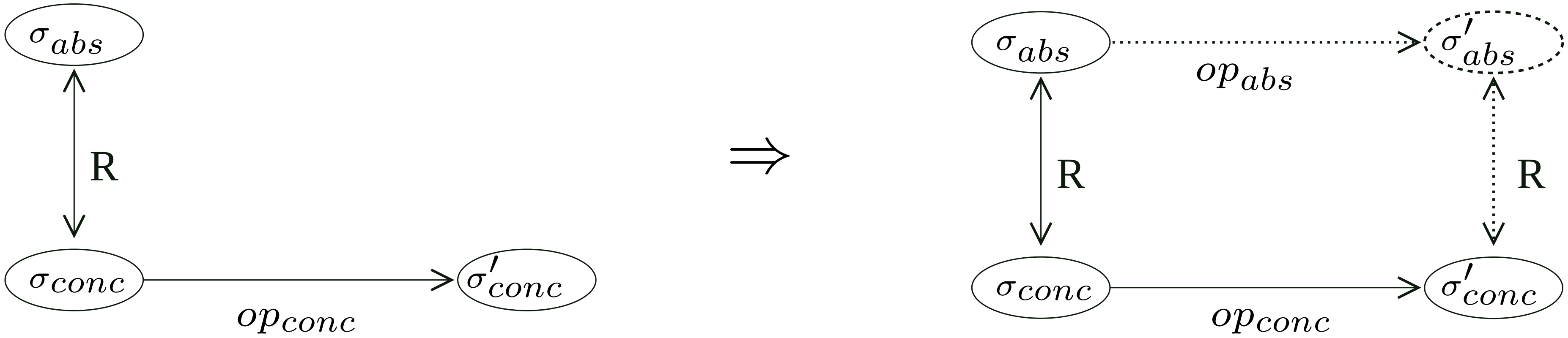
(as, cs) ∈ abs R ∧ (inp, inp')∈ i R)

⟶

(inp', cs) ∈ Domain(opn C)

IOS-Forward Simulation

Recall the diagrams for FS_corr1



IOS-Forward Simulation

Recall the diagrams for FS_corr1

FS_corr1 A R C ≡

$\forall$ as$\in$(inv A). $\forall$ cs$\in$(inv C). $\forall$ cs'$\in$(inv C).
$\forall$ inp$\in$(Domain(i R)). $\forall$ inp'$\in$(Range(i R)). $\forall$ out'$\in$(Range(o R)).
((inp, as) $\in$ Domain(opn A) $\wedge$
(as, cs)$\in$ abs R $\wedge$ (inp, inp')$\in$ i R $\wedge$
((inp', cs), (out', cs'))$\in$ opn C)

$\longrightarrow$

($\exists$ as'$\in$(inv A). $\exists$ out$\in$(Domain(o R)).
(as', cs')$\in$ abs R $\wedge$ (out, out')$\in$ o R $\wedge$
((inp, as), (out, as'))$\in$ opn A)

Tayloring IOS-Forward Simulation (1)

Tayloring forward simulations for functions : Prerequisite:
We embed functions as abstract specifications:

constdefs

```
fun2op :: "['i set, 'i ⇒ 'o] ⇒ ('i,'o,unit) spec"
"fun2op precond F ≡(| init = {()}, inv = {()},
                      opn = {(a,b). ∃ x∈precond. a=(x,()) ∧
                                               b=(F x,())}|)"
```

procond serves as an additional means to formalize preconditions, under which the refinement is supposed to hold.

Tayloring IOS-Forward Simulation (1)

. . . derive the specialized version FS_refine_fun :

$[\![$ R = (| i = RI, o = RO, abs = Id |);

∀ inp ∈ pa. A inp ∈ Domain RO;
∀ inp ∈ pa. ∀ inp'. (inp,inp')∈RI ⟶ inp'∈pc;
∀ inp ∈ pa. ∀ inp'∈ pc. (A inp, C inp') ∈ RO $]\!]$

⟹ (fun2op pa A) \<sqsubseteq>R (fun2op pc C)"

Note that the first assumption constrains the structure of the generalized abstraction to default values on dummy states . . .

Tayloring IOS-Forward Simulation (1)

A (standard) example. We assume the usual:

consts

```
insort    :: "['a::order, 'a list] => 'a list"
is_sorted :: "['a list] => bool"
```

... and set up the refinement relation as:

consts

```
data_R     :: "('a::order set × 'a list)set"
set_list_R :: "('a::order × 'a set,'a × 'a list,
               'a set,'a list,
               unit,unit) abs_rel"
```

defs

```
data_R_def:   "data_R ≡{(x,y). x=set y ∧ is_sorted  y}"
set_list_R_def :
              " set_list_R  ≡(|i = {(x,y). fst x = fst y ∧
                                    (snd x,snd y)∈data_R},
                         o = data_R, abs = Id|)"
```

Tayloring IOS-Forward Simulation (1)

A refinement proof is started:

lemma insert_insort_refine_FS :
  "(fun2op {λ(x,S). finite S} (λ(x,S). *insert* x S))
  \<sqsubseteq>set_list_R
  (fun2op {λ(x,S). is_sorted S} (λ(x,S). insort x S))"

. . . and, after applying FS_refine_fun as introduction rule, we derive the proof obligations:

1. ∀a b. finite b ⟶ (∃y. *insert* a b = set y ∧ is_sorted y)
2. ∀a b. finite b ⟶
   (∀aa ba.
   is_sorted ba ⟶ *insert* a b = set (insort aa ba) ∧
   is_sorted (insort aa ba))

Tayloring IOS-Forward Simulation (2)

. . . derive FS_refine_opn_Z for operations

```
⟦ R = (|i = Id, o = Id, abs = Abs|);
  ∀cs∈(inv C). cs∈(init C) ⟶
                 ∃ as∈(inv A). as∈(init A) ∧ (as,cs)∈Abs;
  ∀as∈(inv A). ∀cs∈(inv C). ∀inp∈(Domain(i R)).
                 ( pre(opn A)(inp,as) ∧ (as,cs) ∈(abs R)) ⟶
                 pre(opn C)(inp,cs);
  ∀as∈(inv A). ∀cs∈(inv C). ∀cs'∈(inv C). ∀inp. ∀out.
                 ( pre(opn A)(inp,as)  ∧
                   (as,cs)∈Abs ∧((inp,cs),(out,cs'))∈opn C)⟶
                 ∃ as'∈(inv A). (as',cs')∈ Abs ∧
                                ((inp,as),(out,as'))∈(opn A))⟧
  ⟹ A \<sqsubseteq>R C
```

Tayloring IOS-Forward Simulation (2)

Do you recognize the pattern? :This represents forward simulation a la [Spi92] and [WD96]):

$$\forall Cstate \bullet Cinit \longrightarrow (\exists Astate \bullet Abs \wedge Ainit)$$

$$\forall Astate\, Cstate\, Cstate'\, x?\, y! \bullet$$
$$pre\, Aop \wedge Abs \wedge Cop \longrightarrow (\exists Astate' \bullet Abs' \wedge Aop)$$
$$\forall Astate\, Cstate\, x? \bullet pre\, Aop \wedge Abs \longrightarrow pre\, Cop$$

Note that in this refinement notion, input $x?$ and output $y!$ are identical!

Example: BirthdayBook Refinement

A (standard) example: Spivey's Birthdaybook[Spi92]:The states of the two systems are:

```
record BirthdayBook =
  birthday  ::  "Name ~=> Date"
  known     ::  "Name set"

record BirthdayBook1 =
  dates     ::  "(nat ~=> Date)"
  hwm       ::  nat
  names     ::  "nat ~=> Name"
```

(The invariant states that known is equal to the domain of birthday).

Example: BirthdayBook Refinement

The two operation schemas are immediately represented as abstract IOS specifications:

consts

AddBirthday :: "((Name $\times$Date), unit, BirthdayBook) spec"
AddBirthday1:: "((Name $\times$Date), unit, BirthdayBook1) spec"
. . .

Example: BirthdayBook Refinement

The abstraction relation between the underlying states is:

constdefs

Abs :: "(BirthdayBook × BirthdayBook1) set"

"Abs ≡ {(x,y).((known x) = {n. ∃ i∈{1..(hwm y)}.
n = the (names y i)}) ∧
(∀ i∈{1..(hwm y)}. birthday x (the(names y i))
= dates y (the(names y i)))}"

. . . which is generalized to:

constdefs

gen_Abs :: "('a,'a,'b,'b,BirthdayBook,BirthdayBook1) abs_rel"

"gen_Abs ≡ (|i = Id, o = Id, abs = Abs|)"

Example: BirthdayBook Refinement

The question to be asked:

lemma AddBrithday__FS_refine :

"AddBirthday \<sqsubseteq>gen_Abs AddBirthday1"

Example: BirthdayBook Refinement

Applying FS_refine_opn_Z yields:

1. ∀cs∈spec.inv AddBirthday1.
   cs ∈ init AddBirthday1 ⟶
   (∃as∈inv AddBirthday. as∈init AddBirthday ∧(as, cs)∈Abs)
2. ∀as∈inv AddBirthday. ∀cs∈inv AddBirthday1. ∀inp.
   pre(opn AddBirthday)(inp,as) ∧ (as, cs)∈Abs ⟶
   pre(opn AddBirthday1)(inp,cs)
3. ∀as∈inv AddBirthday. ∀cs∈inv AddBirthday1.
   ∀cs'∈inv AddBirthday1. ∀inp out.
   pre(opn AddBirthday)(inp,as) ∧
   (as, cs)∈Abs ∧((inp, cs), out, cs')∈opn AddBirthday1 ⟶
   ∃as'∈inv AddBirthday.
   (as', cs')∈Abs ∧((inp, as), out, as')∈opn AddBirthday

(see [Spi92] and the HOL-Z-disribution [BRW03]!)

Connection to Behavioral Refinement(1)

- How do abstract IOS specifications relate to behavioral models?
- Can we extend reasoning over refinements of individual system steps to sequences of steps ?
- How do established notions of behavioral specification relate to forward/backward simulation ?

Partial Answer: abstract IOS specifications generate behavioral notions like Kripke-Structures, (Event) Traces and (Event) Failures. The former talks about states, the latter two over "observable input/output"(=Events)

Connection to Behavioral Refinement(1)

State Projection into Kripke Structures :

```
types
  's trace = "nat ⇒ 's"

record 's kripke =
  init   :: "'s set"
  step   :: "('s × 's) set"

constdefs
  state_projection :: "('i,'o,'s) spec ⇒ 's kripke"
  "state_projection A ≡
   (|kripke.init = spec.init A,
     kripke.step = {(s1,s2). ∃ i' o'.((i',s1),(o',s2))∈spec.opn A}|)"
```

Connection to Behavioral Refinement(1)

constdefs

```
  is_trace        :: "['s kripke, 's trace] => bool"
" is_trace K t ≡ t 0 ∈ kripke.init K ∧
                 (∀i. (t i, t (Suc i)) ∈ kripke.step K)"
  traces          :: "'s kripke => 's trace set"
" traces K        ≡ { t. is_trace K t }"
```

And now, a standard temporal logics $K \models \phi$ can be defined on top of the Kripke structure K. Open problem: Under which conditions can a forward refinement allow for system abstractions?

```
 ⟦ A \<sqsubseteq>R C; kripke_projection A |= phi ⟧
 ⟹   kripke_projection  C |= phi
```

Paves the way for temporal abstractions and model-checking.

Connection to Behavioral Refinement(2)

Event Projection of ('i,'o,'s)spec's to event traces:

constdefs

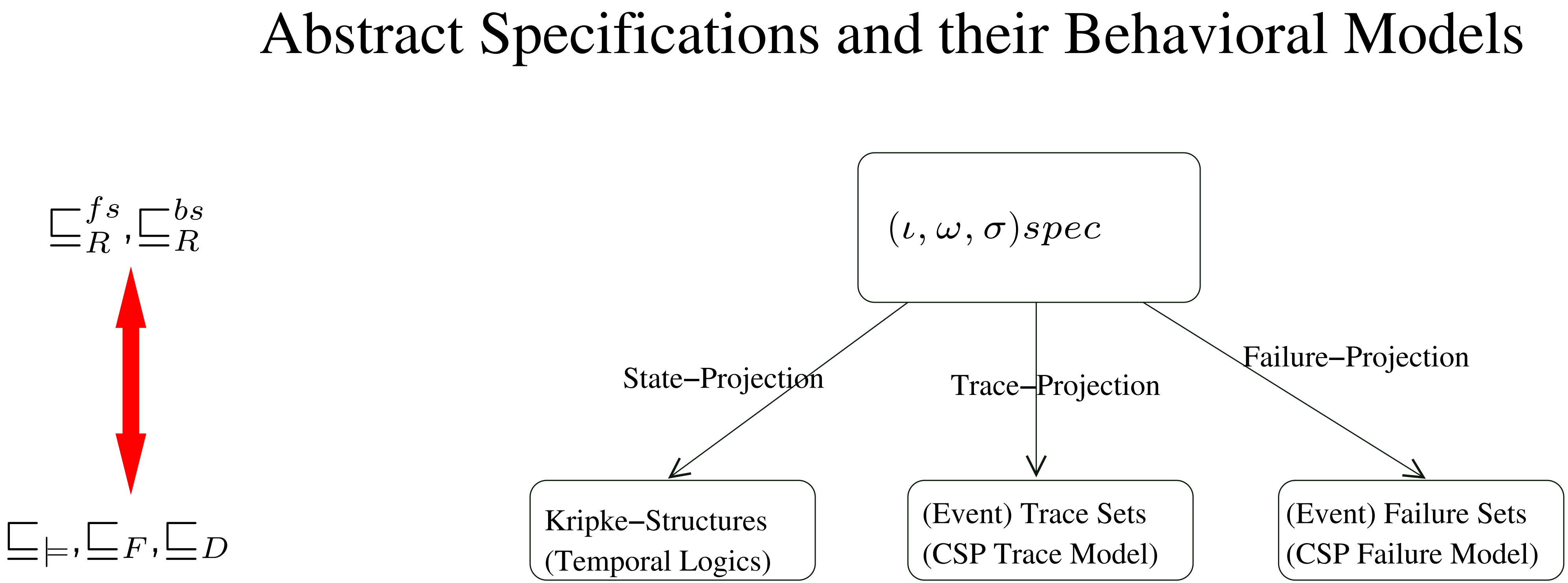
```
  is_gen_trace     ::[('i,'o,'s)spec,(('i×'s)×('o×'s))trace] ⇒ bool
" is_gen_trace  A t ≡(snd(fst(t 0)) ∈spec.init  A ∧
                      (∀i. t i ∈spec.opn A) ∧
                      (∀i. snd(snd(t i))= snd(fst(t (Suc i)))))"
 gen_traces     ::  ('i,'o,'s) spec ⇒ (('i×'s)×('o×'s))trace set
"gen_traces A ≡ { t.  is_gen_trace  A t }"

  event_traces_projection  ::  ('i,'o,'s) spec ⇒ ('i×'o)trace set
" event_traces_projection  A  ≡(λf n.(λ(x,y). (fst x, fst y))(f n))
                                                  ` (gen_traces A)"
```

In order to accomodate ('i,'o,'s)spec for CSP-processes, for example, states 's must be instantiated with process terms, and opn by transition of operational CSP semantics [A.W97] ...

Open problem: Under which conditions allows fs-refinement simplifying process-refinement, i.e. to inclusion of trace or failure sets?

Connection to Behavioral Refinement(3)

Overview:



Summary

- Refinement can be represented in a generalized framework (such as IOS-specifications)
- Approach can be used for data refinement and behavioral refinement as well
- Approach can be used for proving meta-theoretic properties of refinements (reflexive?, transitive?, composable?) too
- Approach can be used for automated proof support.

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[A.W97] A.W.Roscoe. *The Theory and Pactice of Concurrency*. Prentice Hall, 1997.

[BRW03] Achim D. Brucker, Frank Rittinger, and Burkhart Wolff. Hol-z 2.0: A proof environment for z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, February 2003.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[WD96] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science. Prentice Hall, 1996.