SPARTACAS – Automating Component Adaptation for Reuse

Brandon Morel

Master's Thesis Defense
University of Kansas
September 12, 2003

Committee:
Dr. Perry Alexander (Chair)
Dr. Susan Gauch
Dr. Costas Tsatsoulis

Introduction

- Reuse is a sound/practical design technique
- Software engineering slow to embrace reuse
- Benefits
  - Reduce errors
  - Increase productivity of engineers
  - Increase reliability/quality of software
- Costs
  - Effort to create/maintain library of components
  - Effort to search for components
  - Effort to adapt partial matches

Problem and Solution

- Problems
  - How to adapt software?
  - Can adaptation be automated?
  - Will the framework be effective?
- Solution: SPARTACAS
- Outline
  - Specification-level representation
  - Adaptation framework

Outline

– Adaptation architectures
– Adaptation method
– Evaluation results
– Future work and limitations
– Related work
– Concluding remarks

Formal Specifications

- Prior success at the specification-level
- Specification formally describe the functionality without implementation details
- DRIO specification models
  - Domain – typed input parameters
  - Range – typed output parameters
  - Input condition – pre-conditions defining legal inputs
  - Output condition – post-conditions defining valid outputs for legal inputs
  - $I(d) \Rightarrow O(d, r)$

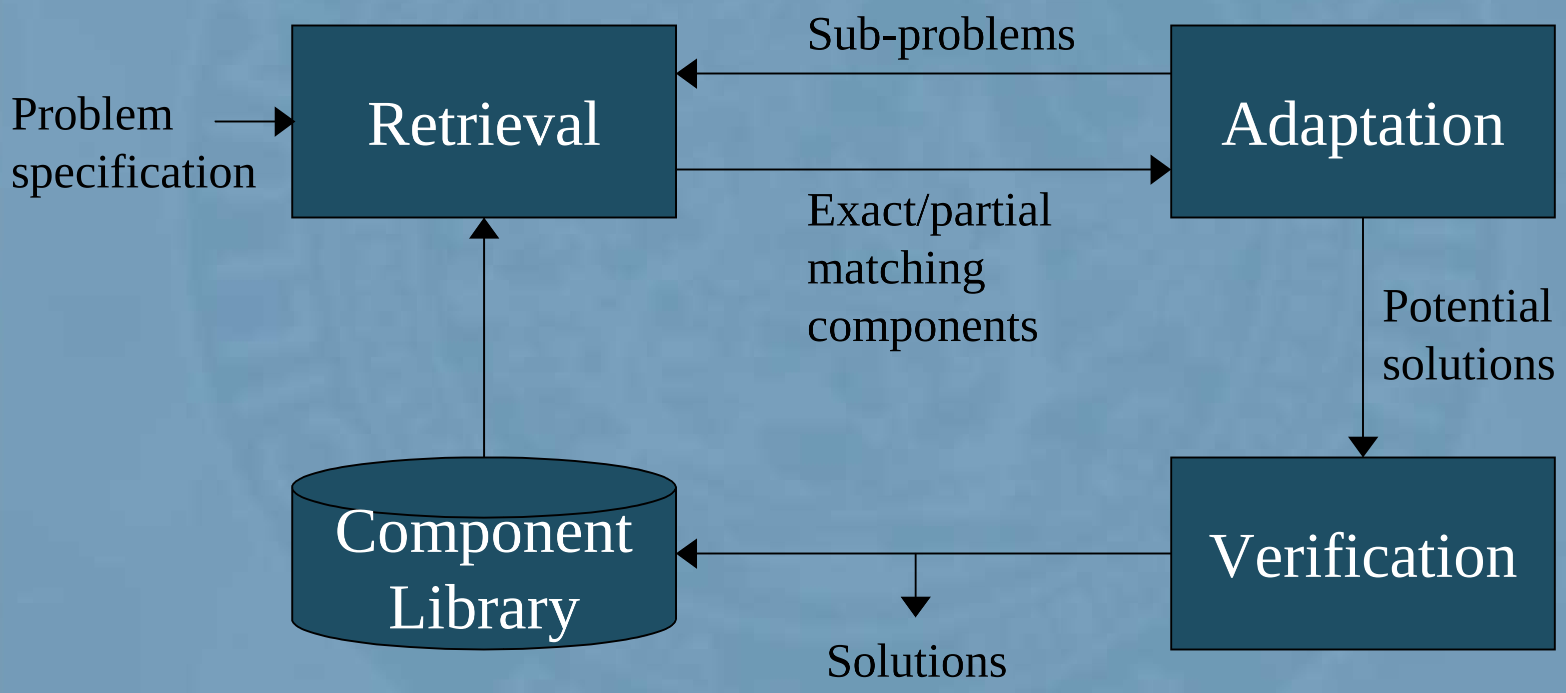
Background-Retrieval Methods

- Feature-based Retrieval
  - Component/problem assigned domain-specific features
  - Matching is based on a similarity threshold
  - Necessary condition
- Signature-based Retrieval
  - Syntactic matching of input and output ports
  - Involves currying, type coercion
- Specification-based Retrieval
  - Prove logical relationship between components
  - Match lattice used to determine *degree of satisfaction*

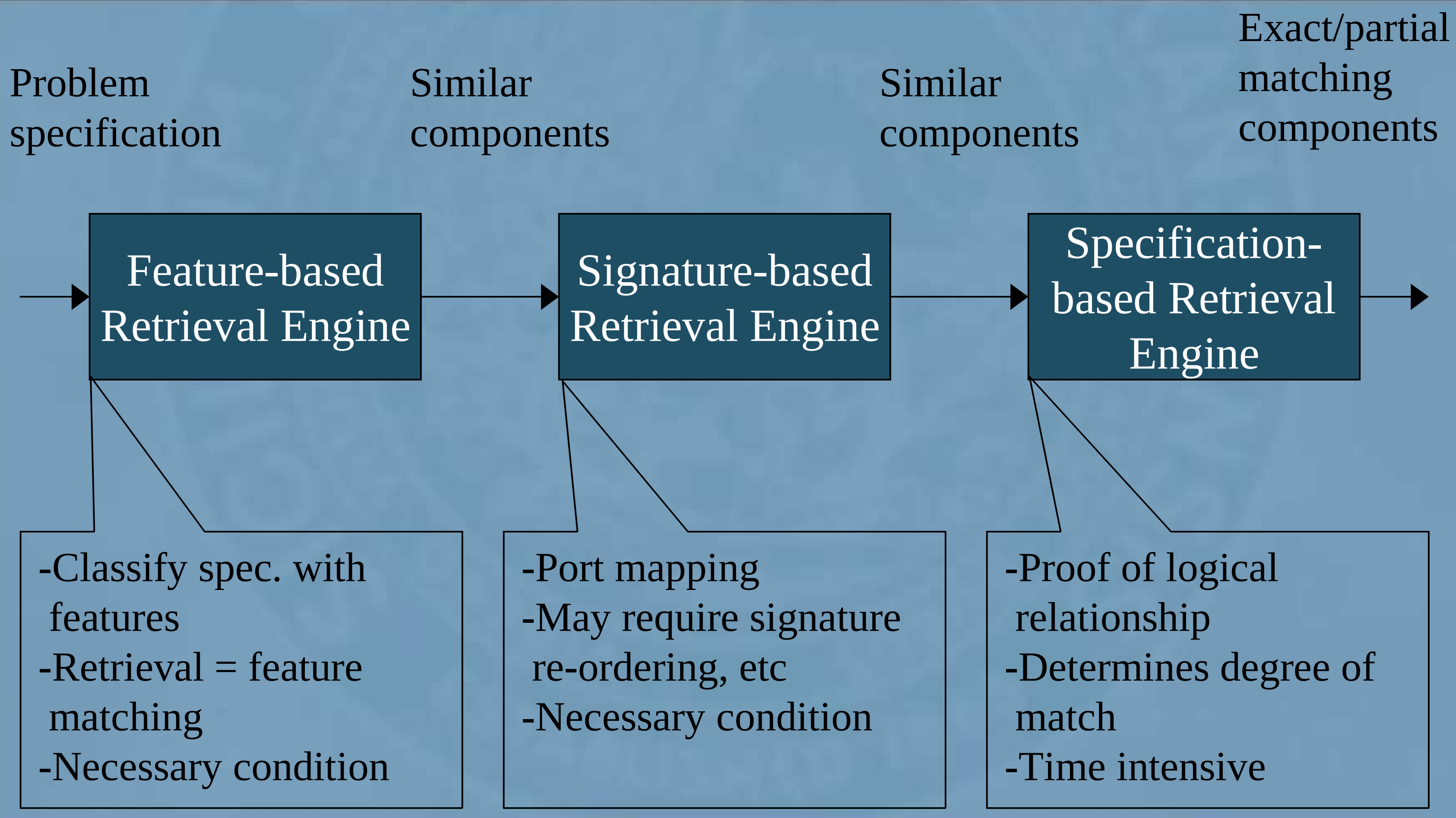
Background-Component Architectures

- Architecture is a collection of interconnected components
- Architecture theory
  - Parameterized specification
  - Specifies the configuration of sub-components in the composition of a system
  - Specifies the relationship between functionality of the system, sub-components
- To solve a problem, instantiate the theory with the problem as the system, components (other architectures) as the sub-components

SPARTACAS Framework



Retrieval Framework



Specification Match Lattice

Stronger → Weaker

Plug-in
$(I_P \Rightarrow I_C) \wedge (O_C \Rightarrow O_P)$

Weak Plug-in
$(I_P \Rightarrow I_C) \wedge (I_C \wedge O_C \Rightarrow O_P)$

Plug-in Post
$O_C \Rightarrow O_P$

Satisfies
$(I_P \Rightarrow I_C) \wedge (I_P \wedge O_C \Rightarrow O_P)$

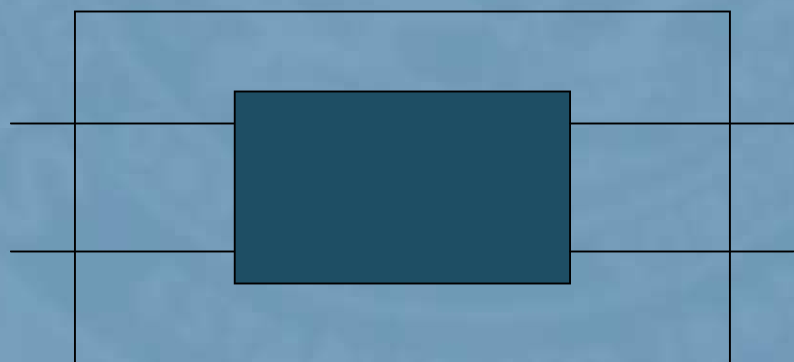
Weak Post
$I_C \wedge O_C \Rightarrow O_P$

Plug-in Pre
$I_P \Rightarrow I_C$

“Specification Matching of Software Components”, Zaremski and Wing, 1995

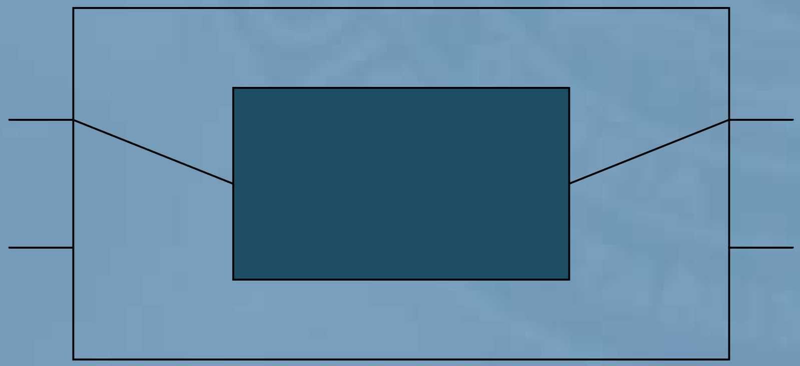
Port Connection Methods

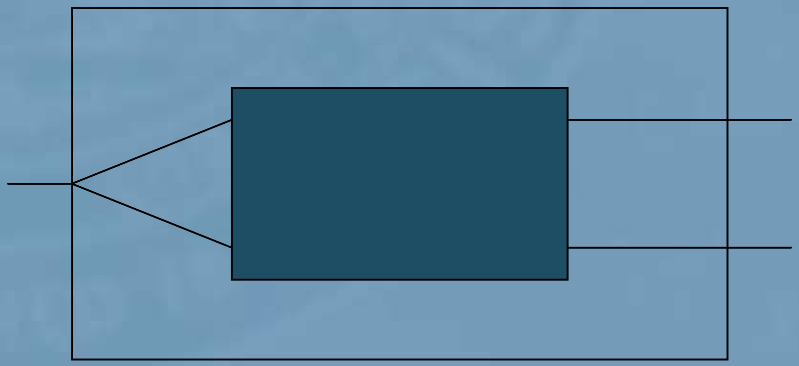
- Bijective Port Connection
  - One-to-one and onto mapping
  - Component must have equal number of ports
  - Factorial number of port combinations



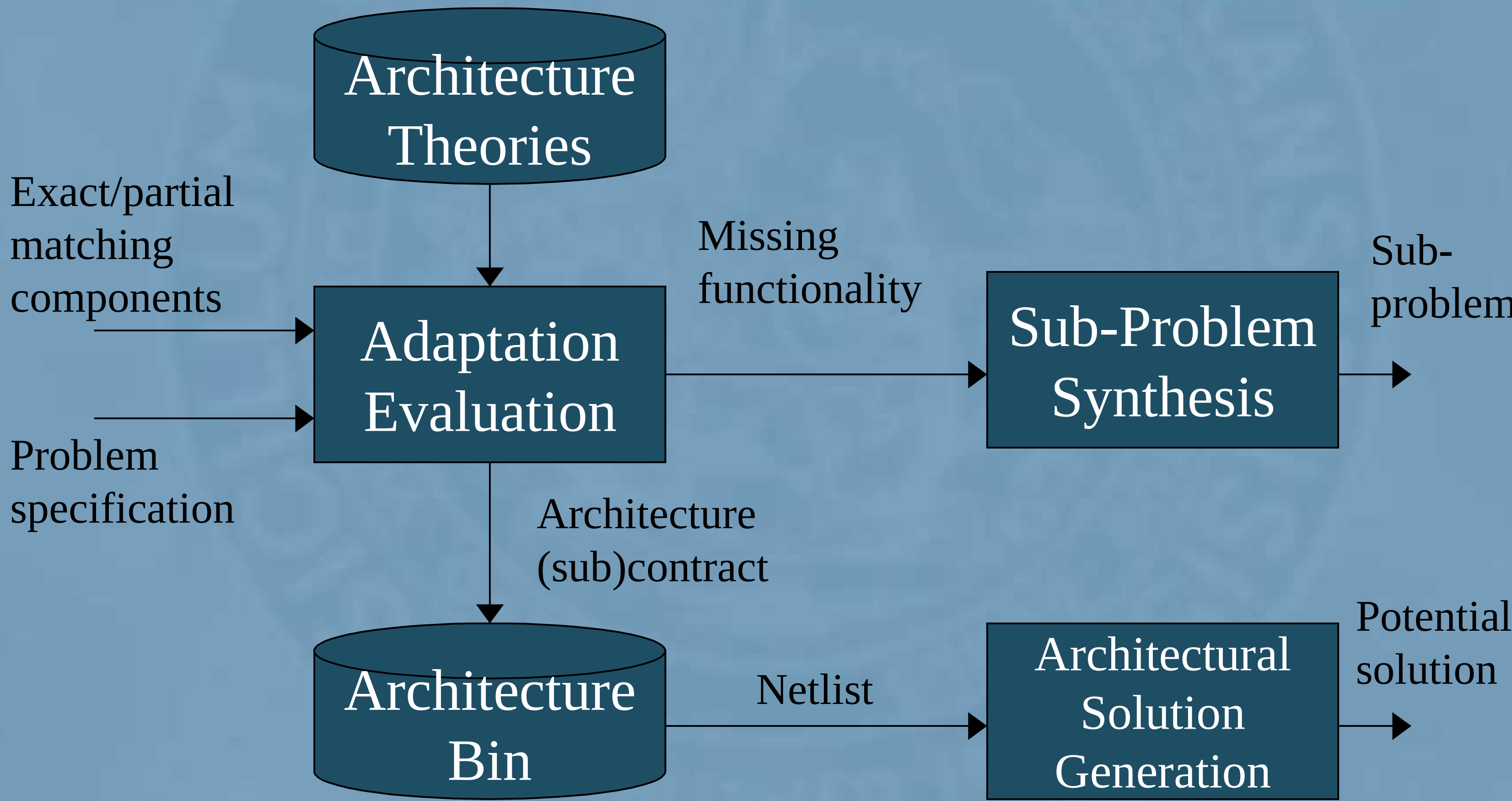
Less Restrictive PCMs

- One-to-one Port Connection
  - Component can have fewer ports than the problem
  - Binomial number of port combinations
- Onto Port Connection
  - Component can have more ports than the problem
  - Exponential number of port combinations

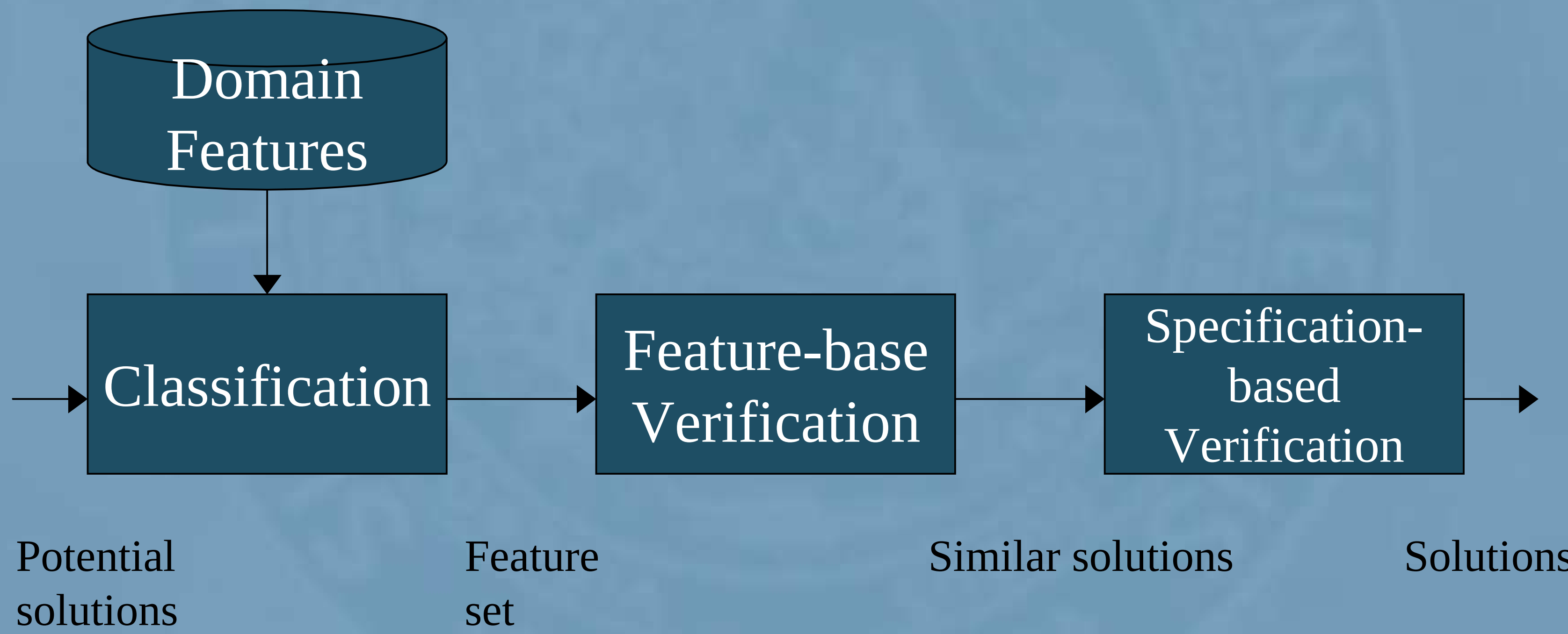




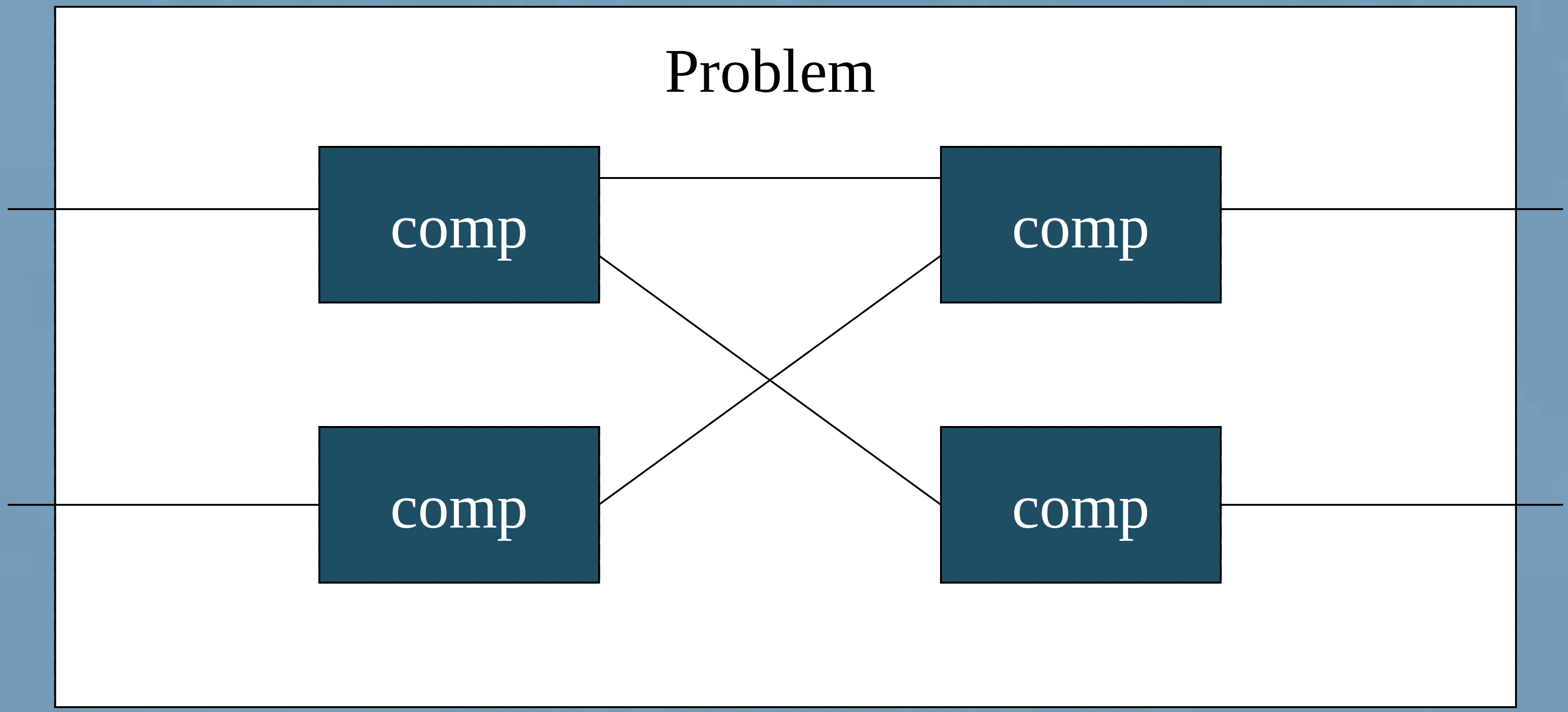
Adaptation Framework



Verification Framework

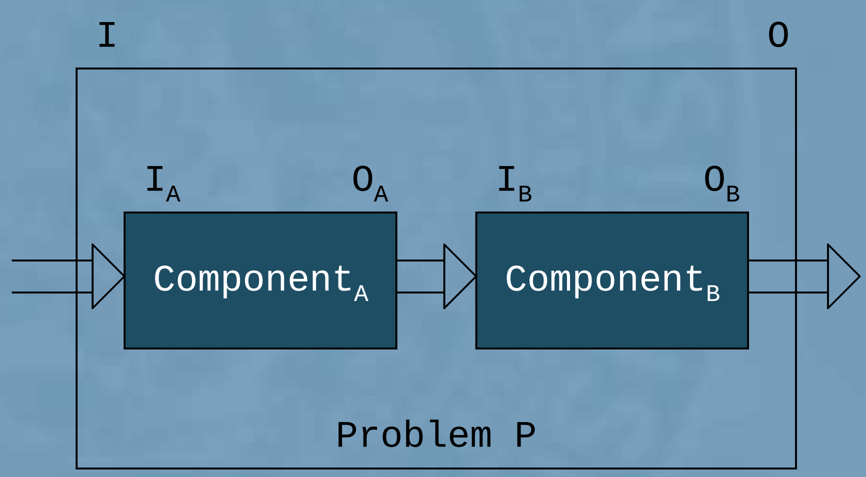


Adaptation Architectures



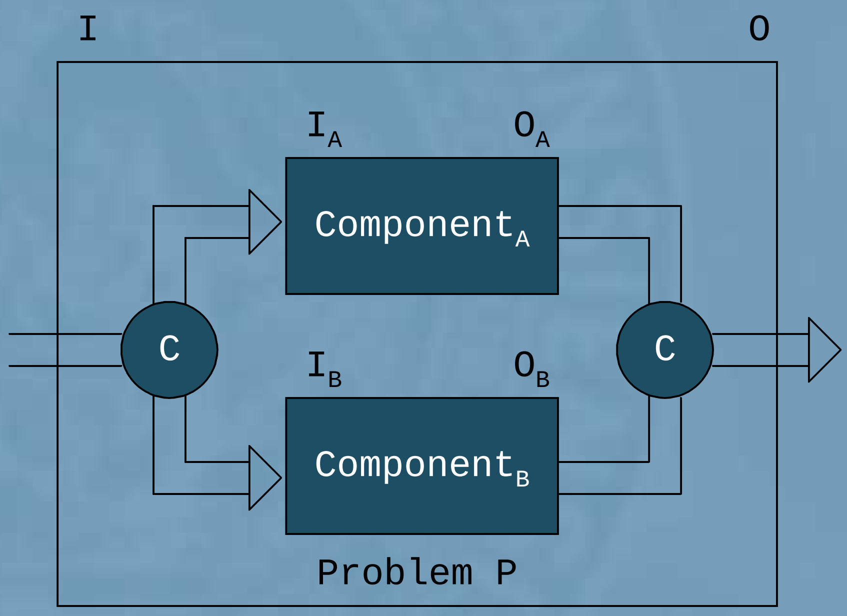
Sequential Architecture Theory

```
Sequential Architecture Theory
BEGIN
 // Problem and components
 Problem(D,R,I,O)
 Component_A(D_A,R_A,I_A,O_A)
 Component_B(D_B,R_B,I_B,O_B)

 // Port constraints
 drConstraint1: D ⊆ D_A
 drConstraint2: R_A ⊆ D_B
 drConstraint3: R_B ⊆ R

 // Behavioral constraints
 behConstraint1: ∀d:D|I(d)⇒I_A(d)
 behConstraint2: ∀d:D, x:D_B|
   I(d)∧O_A(d,x)⇒I_B(x)
 behConstraint3: ∀d:D, y:R_A, r:R|
   I(d)∧O_A(d,y)∧O_B(y,r)⇒O(d,r)

END Sequential Architecture Theory
```



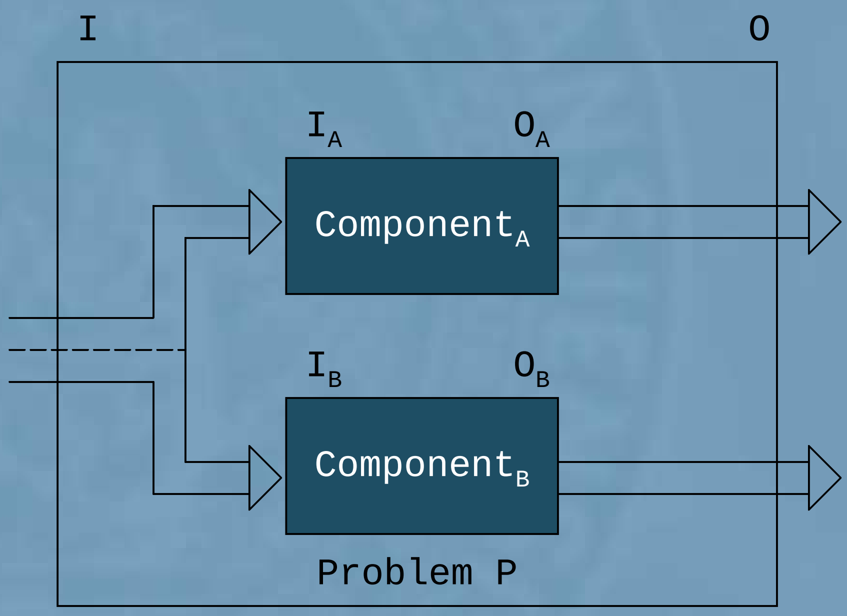
Alternative Architecture Theory

```
Alternative Architecture Theory
BEGIN
 // Problem and components
 Problem(D,R,I,O)
 Component_A(D_A,R_A,I_A,O_A)
 Component_B(D_B,R_B,I_B,O_B)

 // Port constraints
 drConstraint1: D ⊆ D_A
 drConstraint2: D ⊆ D_B
 drConstraint3: R_A ⊆ R
 drConstraint4: R_B ⊆ R

 // Behavioral constraints
 behConstraint1: ∀d:D|
   (I(d)⇒I_A(d))∨(I(d)⇒I_B(d))
 behConstraint2: ∀d:D, r:R|
   (I_A(d)∧O_A(d,r)⇒O(d,r))∨(I_B(d)∧O_B(d,r)⇒O(d,r))

END Alternative Architecture Theory
```

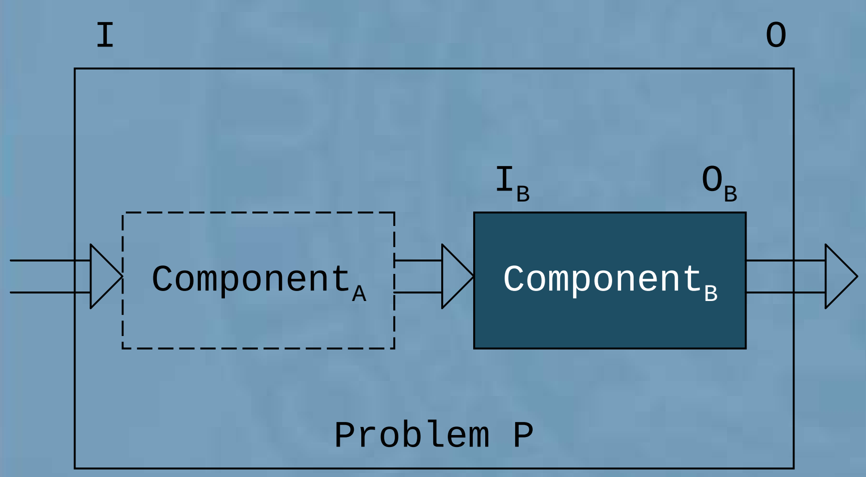


Parallel Architecture Theory

```
Parallel Architecture Theory
BEGIN

 // Problem and components
 Problem(D,R,I,O)
 Component_A(D_A,R_A,I_A,O_A)
 Component_B(D_B,R_B,I_B,O_B)

 // Port constraints
 drConstraint1: D ⊆ D_A ∪ D_B
 drConstraint2: R_A || R_B ⊆ R

 // Behavioral constraints
 behConstraint1: ∀d_1∪d_2:D|
   I(d_1∪d_2)⇒I_A(d_1)∧I_B(d_2)
 behConstraint2: ∀d_1∪d_2:D, r_1||r_2:R|
   I(d_1∪d_2)∧O_A(d_1,r_1)∧O_B(d_2,r_2)⇒O(d_1∪d_2,r_1||r_2)

END Parallel Architecture Theory
```

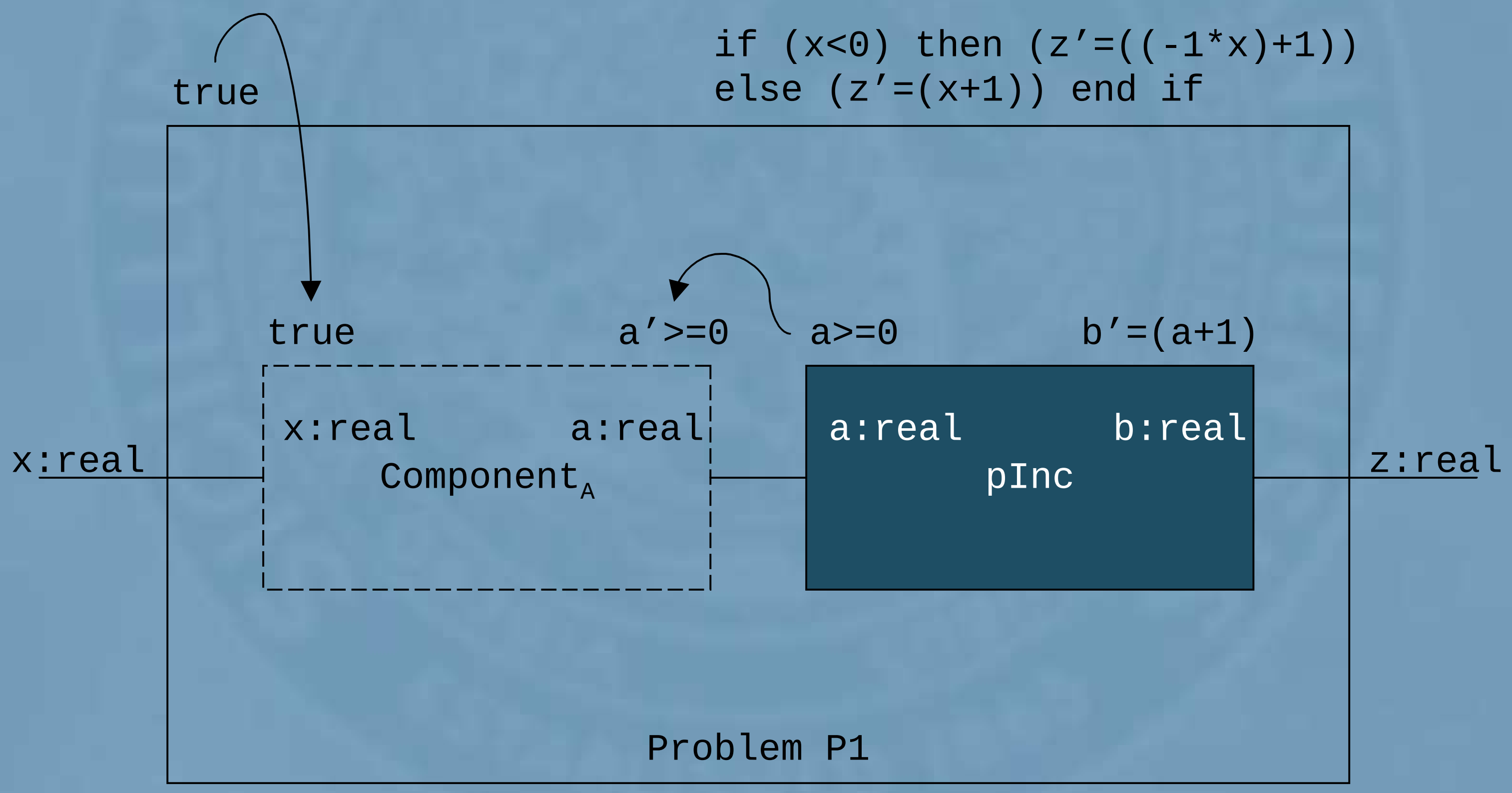


Post-match Sequential Adaptation



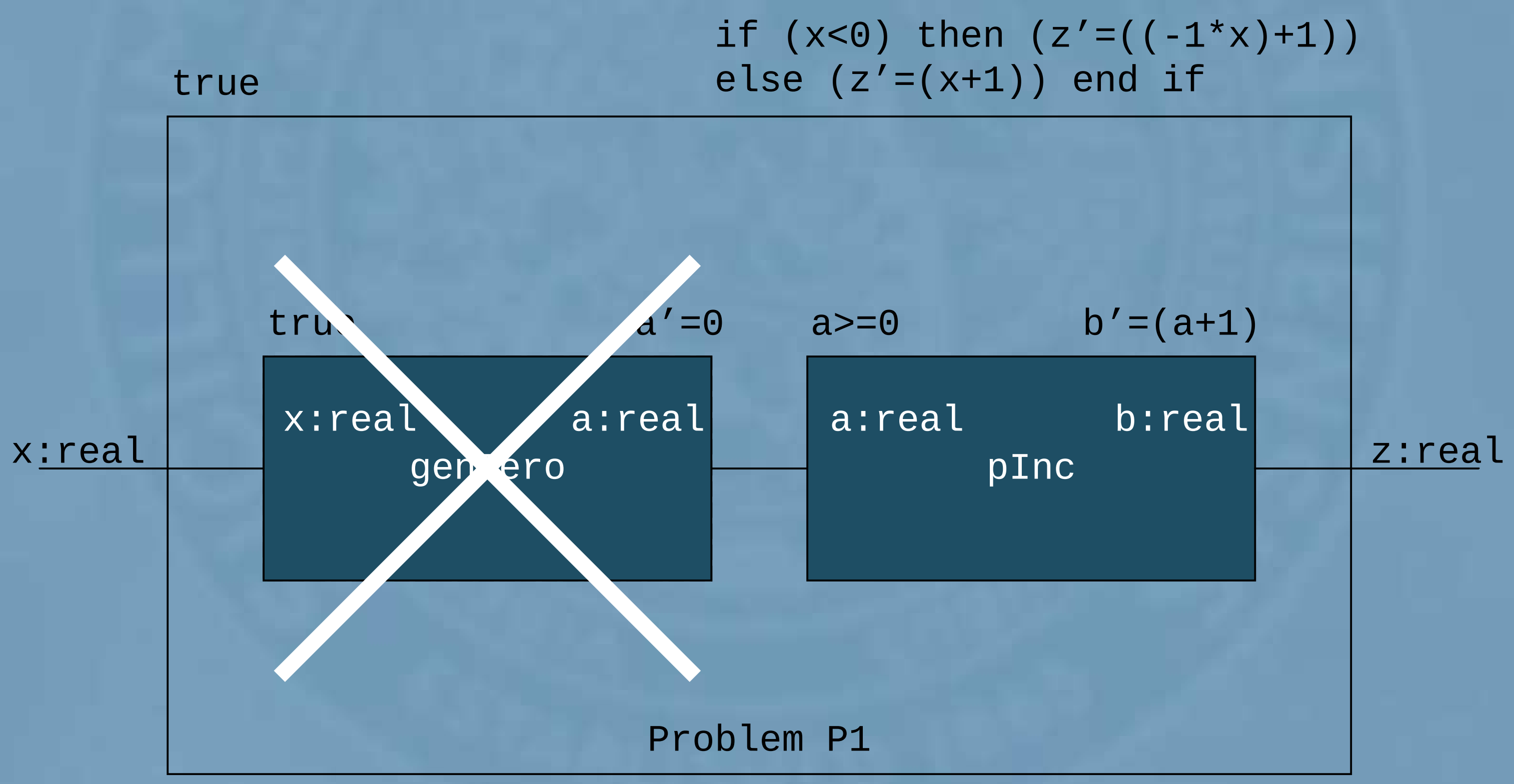
- Component$_B$ produces the required results for some set of inputs
- Tactic: find Component$_A$ that modifies all inputs to allow Component$_B$ to execute for all legal inputs

Example #1



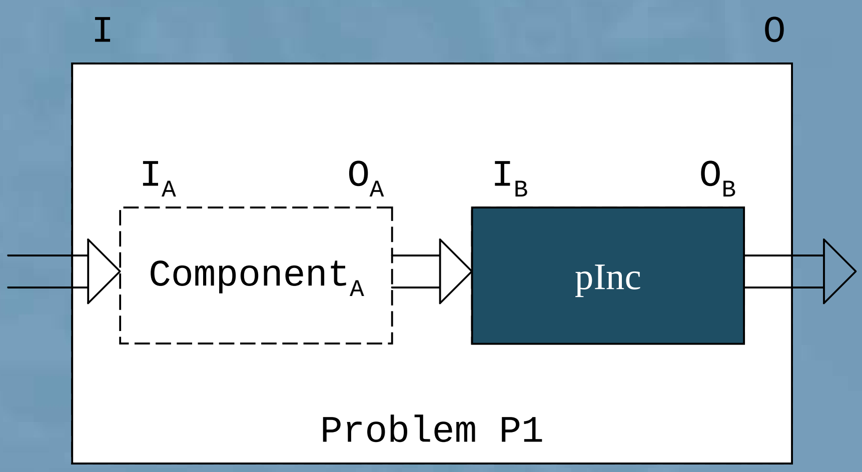
if (x<0) then (z'=((-1*x)+1))
else (z'=(x+1)) end if
true
true
a'>=0
a>=0
b'=(a+1)
x:real
x:real
a:real
Component_A
a:real
b:real
pInc
z:real
Problem P1

Example #1


if (x<0) then (z′=((-1*x)+1))
else (z′=(x+1)) end if
true
x:real
a>=0
b′=(a+1)
a:real
b:real
pInc
z:real
x:real
a:real
Problem P1

Synthesis Method

```
Sequential Architecture Theory
BEGIN
 // Problem and components
 Problem(D,R,I,O)
 ComponentA(DA,RA,IA,OA)
 ComponentB(DB,RB,IB,OB)

 // Port constraints
 drConstraint1: D ⊆ DA
 drConstraint2: RA ⊆ DB
 drConstraint3: RB ⊆ R

 // Behavioral constraints
 behConstraint1: ∀d:D|I(d)⇒IA(d)
 behConstraint2: ∀d:D, x:DB|
   I(d)∧OA(d,x)⇒IB(x)
 behConstraint3: ∀d:D, y:RA, r:R|
   I(d)∧OA(d,y)∧OB(y,r)⇒O(d,r)

END Sequential Architecture Theory
```



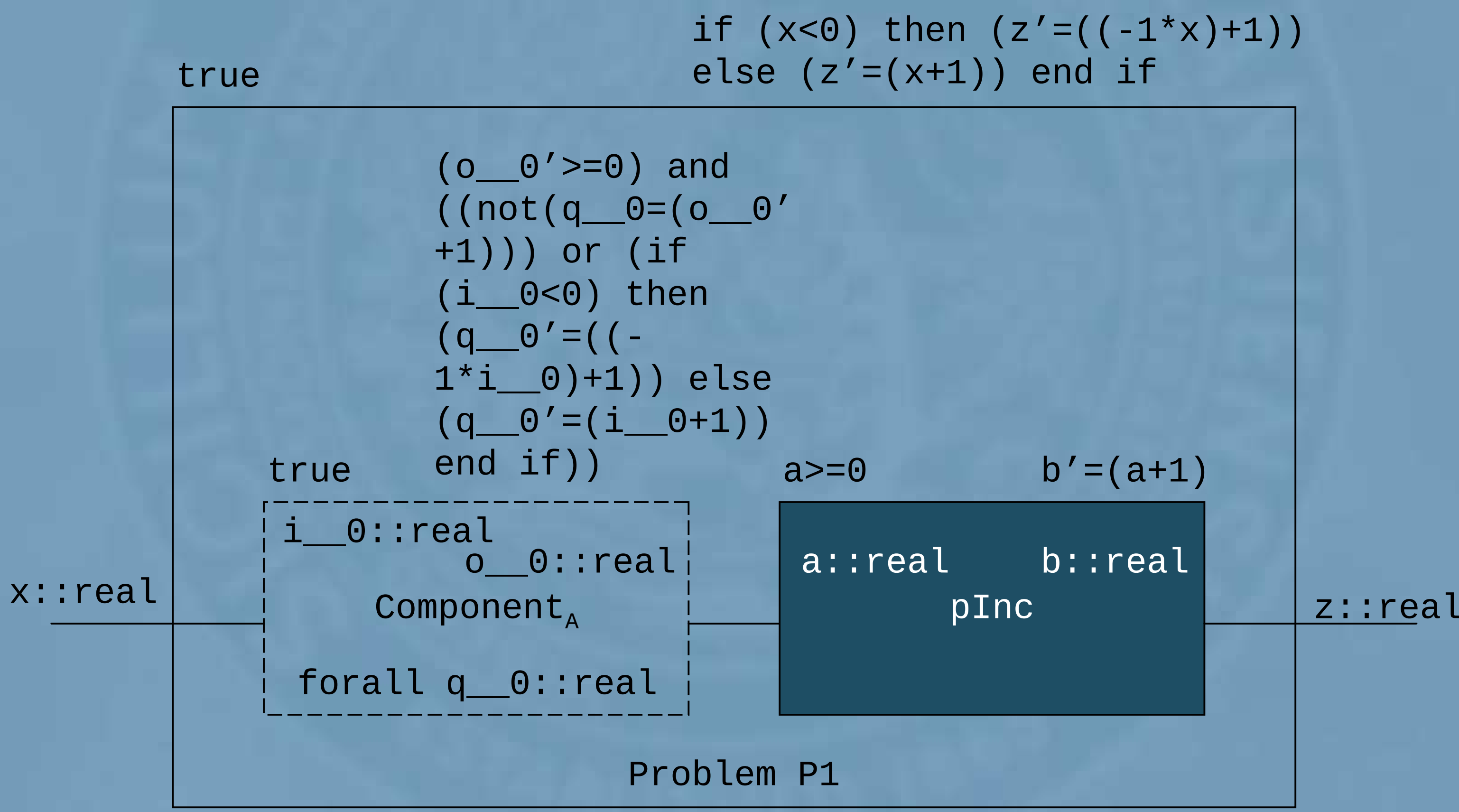
Post-match Sequential Synthesis

- $D_A = D$
- $D_B = R_A \cup \{d \in D \mid \neg\exists x \in D_A \mid \rho(x) \rightarrow d\}$
- $I_A = \forall d{:}D \mid I(d)$
- $O_A = \forall d{:}D, x{:}D_B, y{:}\{r \in R \mid \exists x \in R_B \mid \rho(x) \rightarrow r\}, r{:}R \mid I_B(x) \wedge (\neg O_B(x,y) \vee O(d,r))$

Any output ports of the problem not instantiated

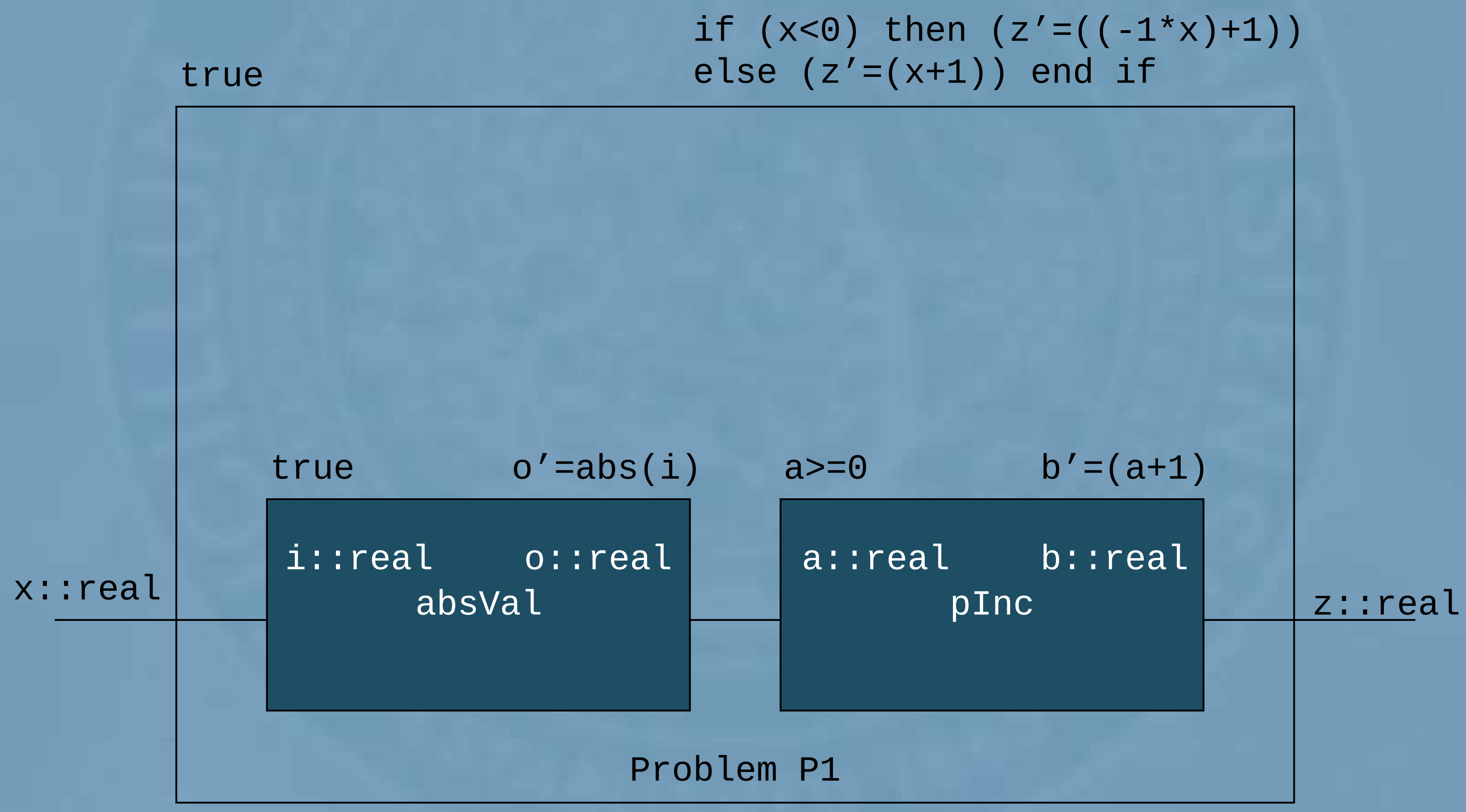
I_B is true and O_B still satisfies O

Example #1



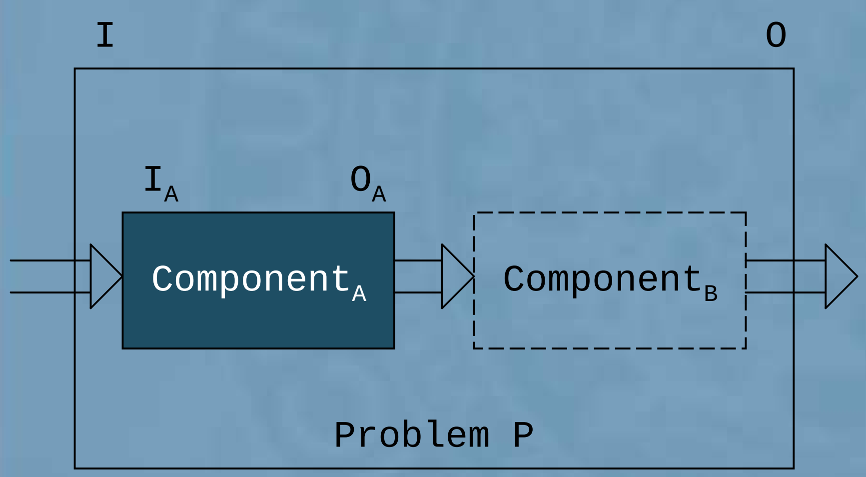
if (x<0) then (z’=((-1*x)+1))
else (z’=(x+1)) end if
true
(o__0’>=0) and
((not(q__0=(o__0’
+1))) or (if
(i__0<0) then
(q__0’=((-
1*i__0)+1)) else
(q__0’=(i__0+1))
end if))
true
a>=0
b’=(a+1)
i__0::real
o__0::real
Component_A
forall q__0::real
x::real
a::real
b::real
pInc
z::real
Problem P1

Example #1



if (x<0) then (z’=((-1*x)+1))
else (z’=(x+1)) end if
true
true
o’=abs(i)
a>=0
b’=(a+1)
i::real
o::real
absVal
a::real
b::real
pInc
x::real
z::real
Problem P1

Pre-match Sequential Adaptation

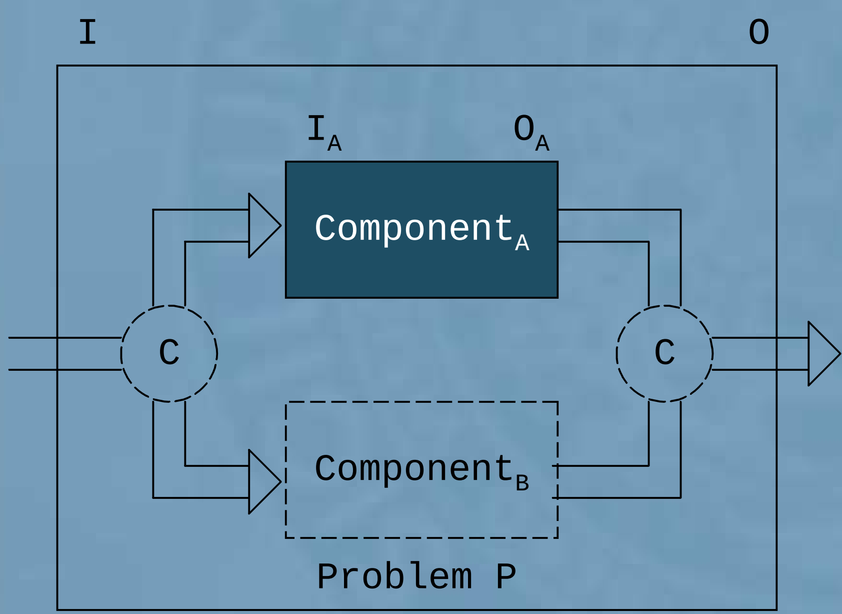


- Component$_A$ accepts the legal inputs, but does not produce valid outputs
- Tactic: find Component$_B$ that modifies all outputs such that they are valid outputs

Pre-match Sequential Synthesis

- $D_B = R_A \cup \{d \in D \mid \neg\exists x \in D_A \mid \rho(x) \rightarrow d\}$
- $R_B = R$
- $I_B = \forall d : \{x \in D \mid \exists y \in D_A \mid \rho(y) \rightarrow x\}, z : R_A \mid I(d) \wedge O_A(d, z)$
- $O_B = \forall d : D, r : R \mid O(d, r)$

Post-match Alternative Adaptation

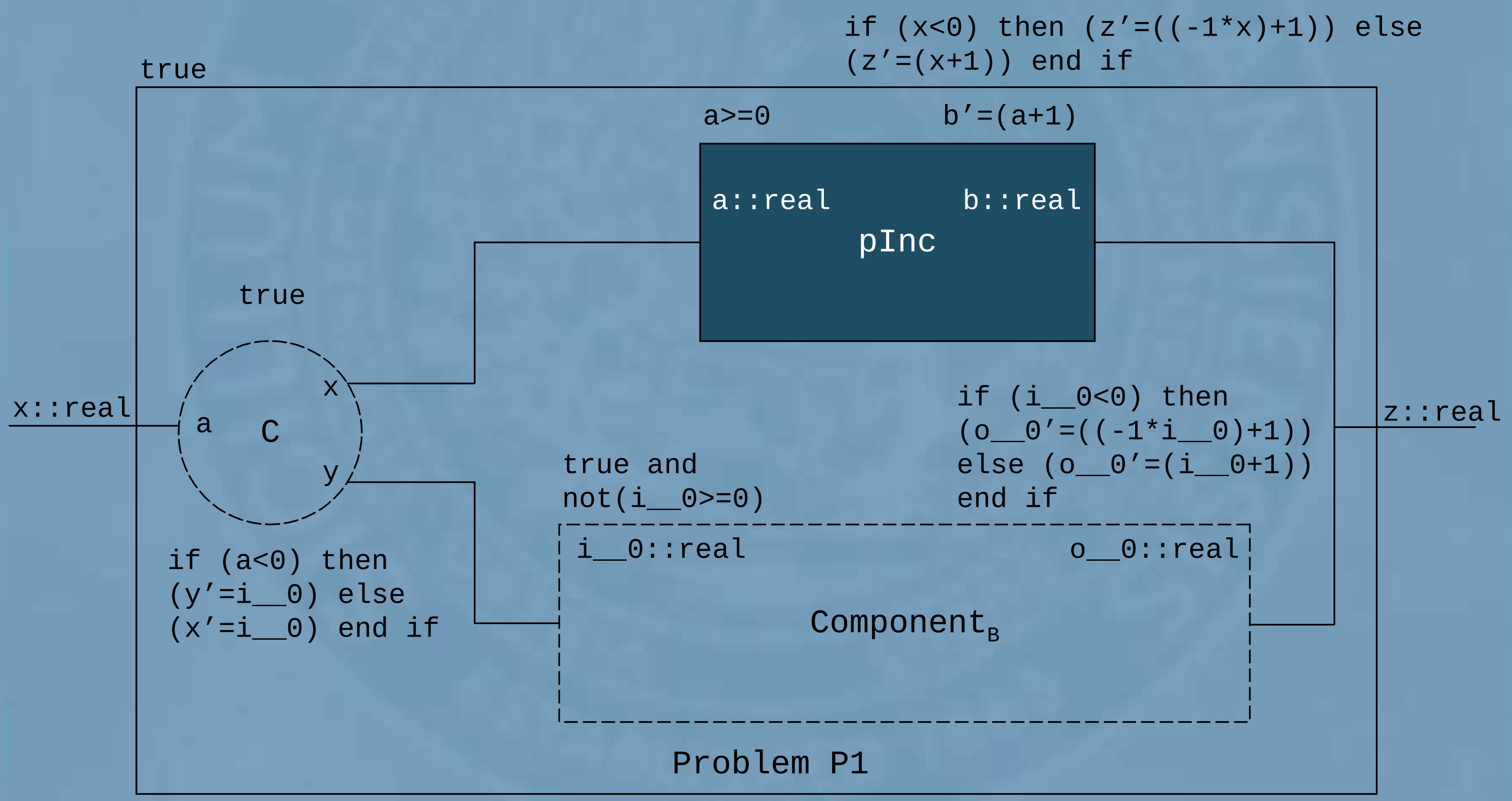


- Component$_A$ computes valid outputs for some set of inputs
- Tactic: find Component$_B$ that computes valid outputs for the rest of the inputs

Post-match Alternative Synthesis

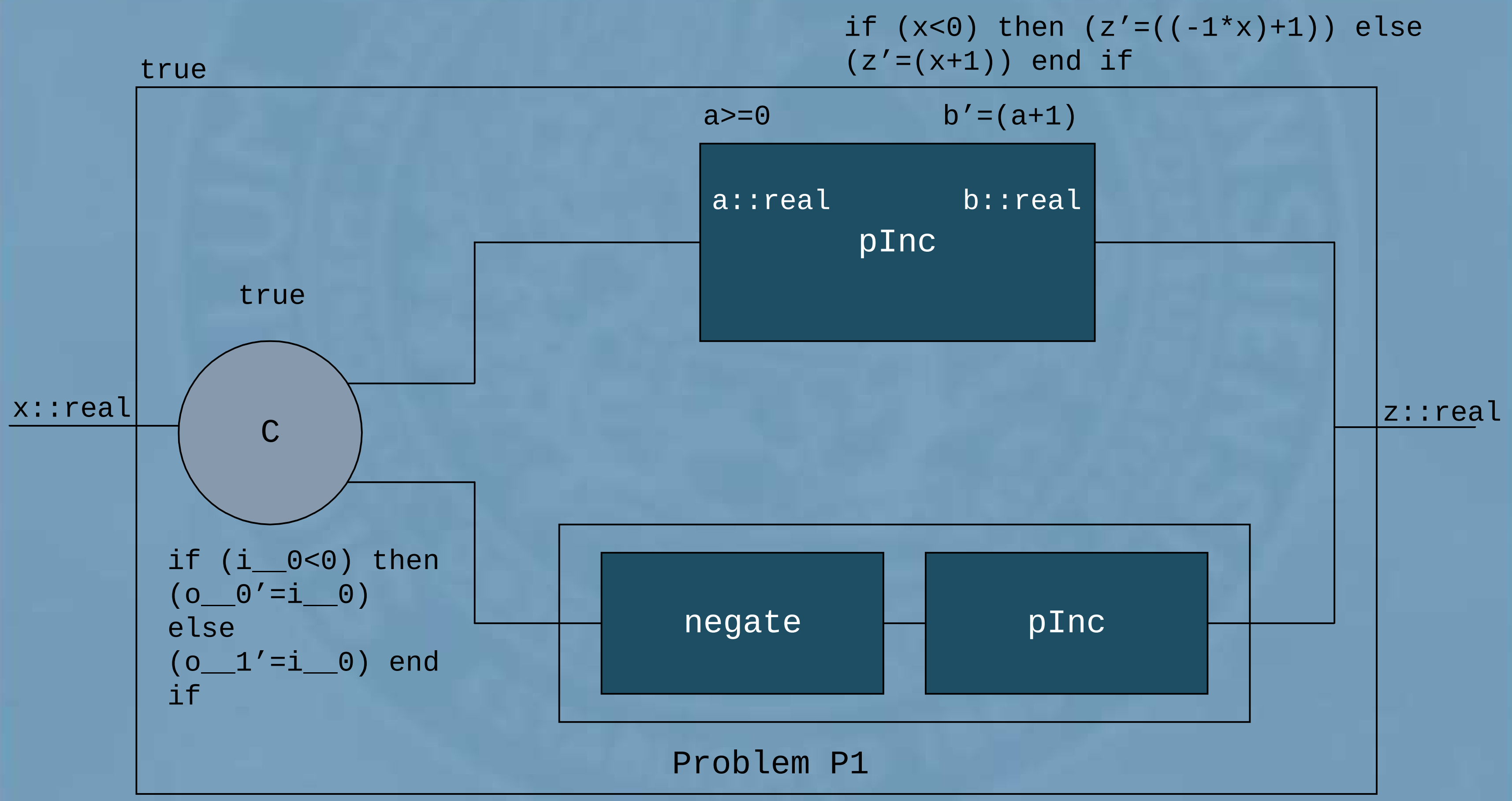
- $D_B = D$
- $R_B = R$
- $I_B = \forall d{:}D \mid I(d) \land \neg I_A(d)$
- $O_B = \forall d{:}D, r{:}R \mid O(d,r)$

Example #1



true
if (x<0) then (z'=((-1*x)+1)) else (z'=(x+1)) end if
a>=0
b'=(a+1)
a::real
b::real
pInc
true
x::real
a
C
x
y
z::real
true and not(i__0>=0)
if (i__0<0) then (o__0'=((-1*i__0)+1)) else (o__0'=(i__0+1)) end if
i__0::real
o__0::real
if (a<0) then (y'=i__0) else (x'=i__0) end if
Component_B
Problem P1

Example #1



if (x<0) then (z'=((-1*x)+1)) else
(z'=(x+1)) end if
true
a>=0
b'=(a+1)
a::real
b::real
pInc
true
x::real
C
z::real
if (i__0<0) then
(o__0'=i__0)
else
(o__1'=i__0) end
if
negate
pInc
Problem P1

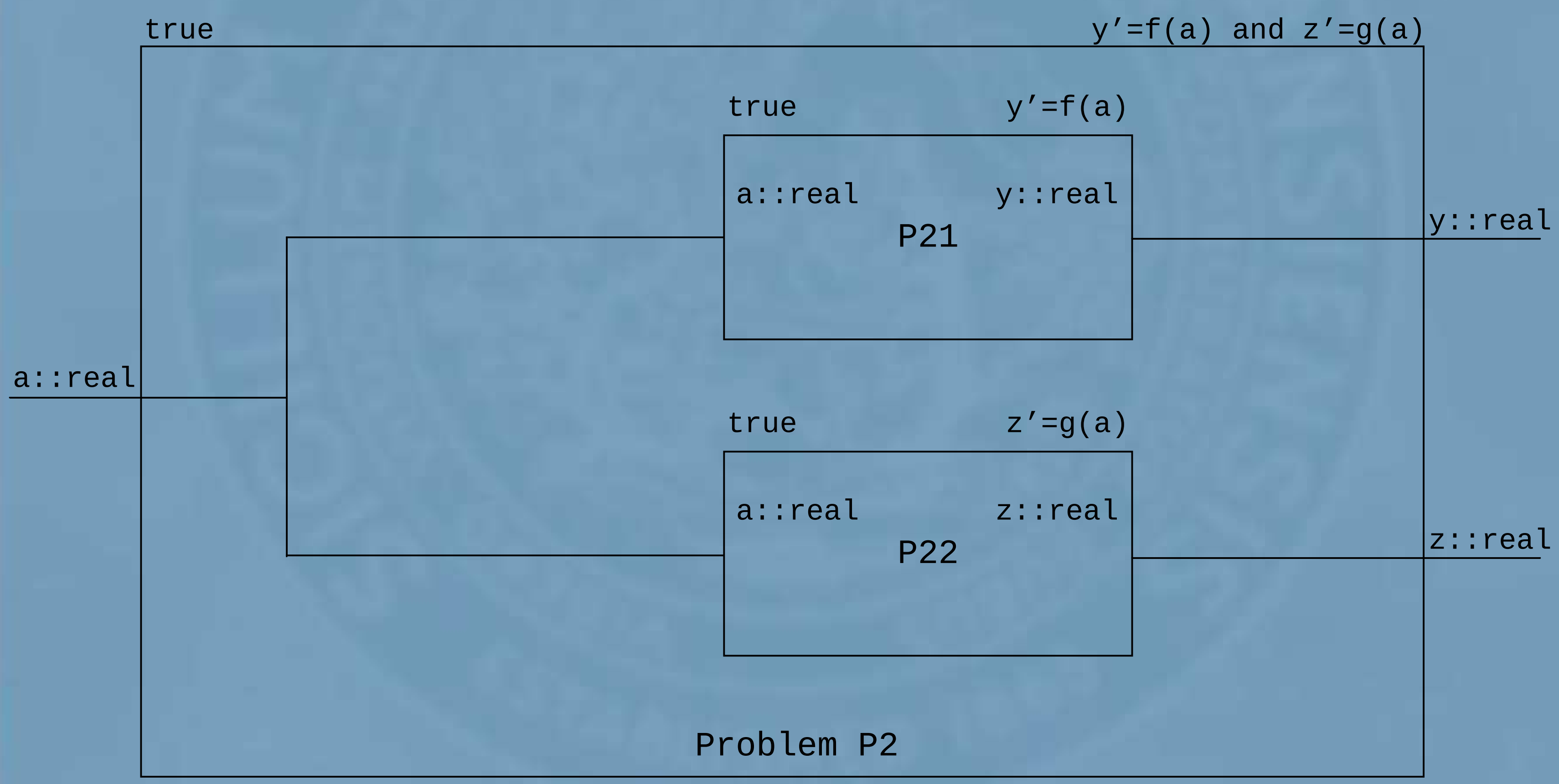
Parallel Adaptation

- Bottom-up behavioral adaptation
  - Find one component, build dynamic adaptation architecture
- Top-down behavioral adaptation
  - Decompose problem into architecture, find components
- Parallel adaptation use slicing for the top-down approach

Slicing Pseudo-Algorithm

1. Pick a range variable as the criterion
2. Select all post-conditions that affect/affected by the criterion
3. Select all pre-conditions that control the execution of the post-conditions
4. Add all range/domain variables constrained

Example #2


true
y’=f(a) and z’=g(a)
true
y’=f(a)
a::real
y::real
P21
y::real
a::real
true
z’=g(a)
a::real
z::real
P22
z::real
Problem P2

Find Example Preliminaries

- Classic adaptation example
- Goal is to find a record in a list of records given a unique key
- Library contains no constructors, only observers (e.g. firstRecord, sort, treeSearch)
- Bijective port connection fails to find solution
- Benefit of less restrictive port connection becomes apparent

Evaluation Metrics/Variables

- Evaluation metrics
  - Precision
  - Recall
  - Time-to-solution (TTS)
- Execution variables
  - Search depth (number of components)
  - Port connection methods

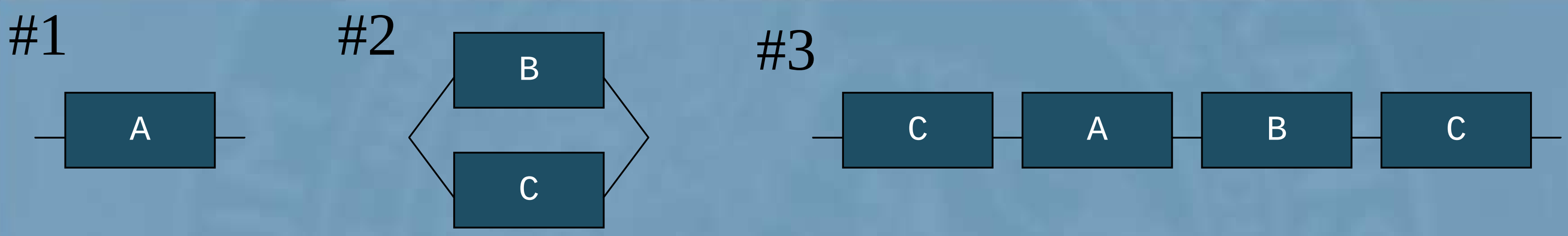
Precision and Recall Metrics

- Precision
  - Relates the purity of the retrieval set
  - # solutions retrieved/# results retrieved
- Recall
  - Relates the coverage of the solutions
  - # solutions retrieved/# solutions that exist
- Infinite solutions may exist
  - Example: f_N applied to f^{-1}_{N-1}
  - Either never stop searching or always stop with 0% recall

Recall Definitions

- $Recall_1$
  - # groups retrieved/# groups that exist
  - Group is defined as the containment of some combination (without replacement) of components such that a solution exists
  - Reduces influence of multiple/redundant configurations
- $Recall_2$
  - # groups retrieved/# groups that exist
  - Group is defined as the containment of the smallest combination (without replacement) of components such that a solution exists
  - Reduces influence of architecture expansion
- $Recall_3$
  - # solutions retrieved/# solutions that exist
  - A solution has N components or less

Recall Illustration

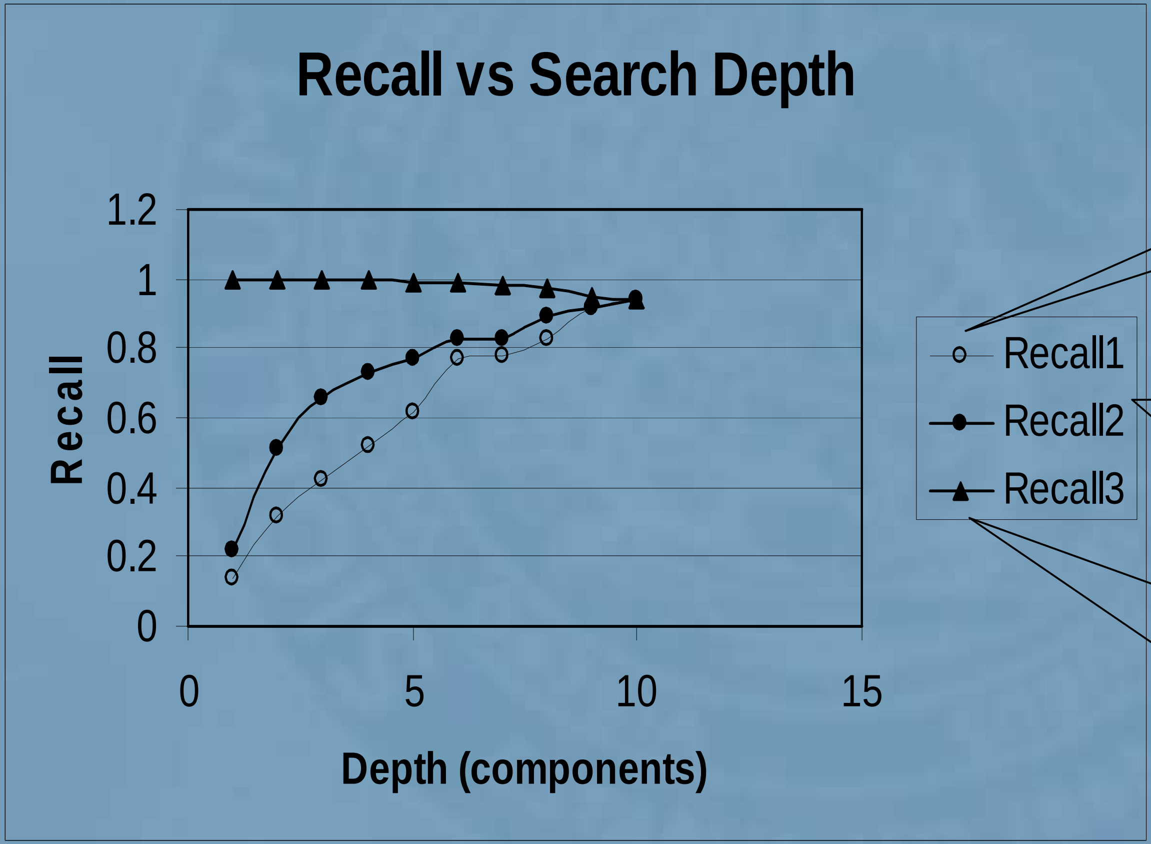


Equation	Solution Groups	No. Soln.
$Recall_1$	Group {a}: #1 Group {b,c}: #2 Group {a, b, c}: #3	3
$Recall_2$	Group {a}: #1, #3 Group {b,c}: #2, #3	2
$Recall_3$	N = 2: #1, #2	2

Evaluation Library and Queries

- Four libraries
  - 46 mathematical components
  - 106 list manipulation components
  - 33 record manipulation components
  - 42 DSP components
- 103 queries, solved by:
  - Single component architectures
  - 1:N component architectures
  - N>1 component architectures
  - Infinite number of solutions
  - Multiple sub-architectures
  - Components from multiple libraries

Recall vs. Search Depth (1-1)

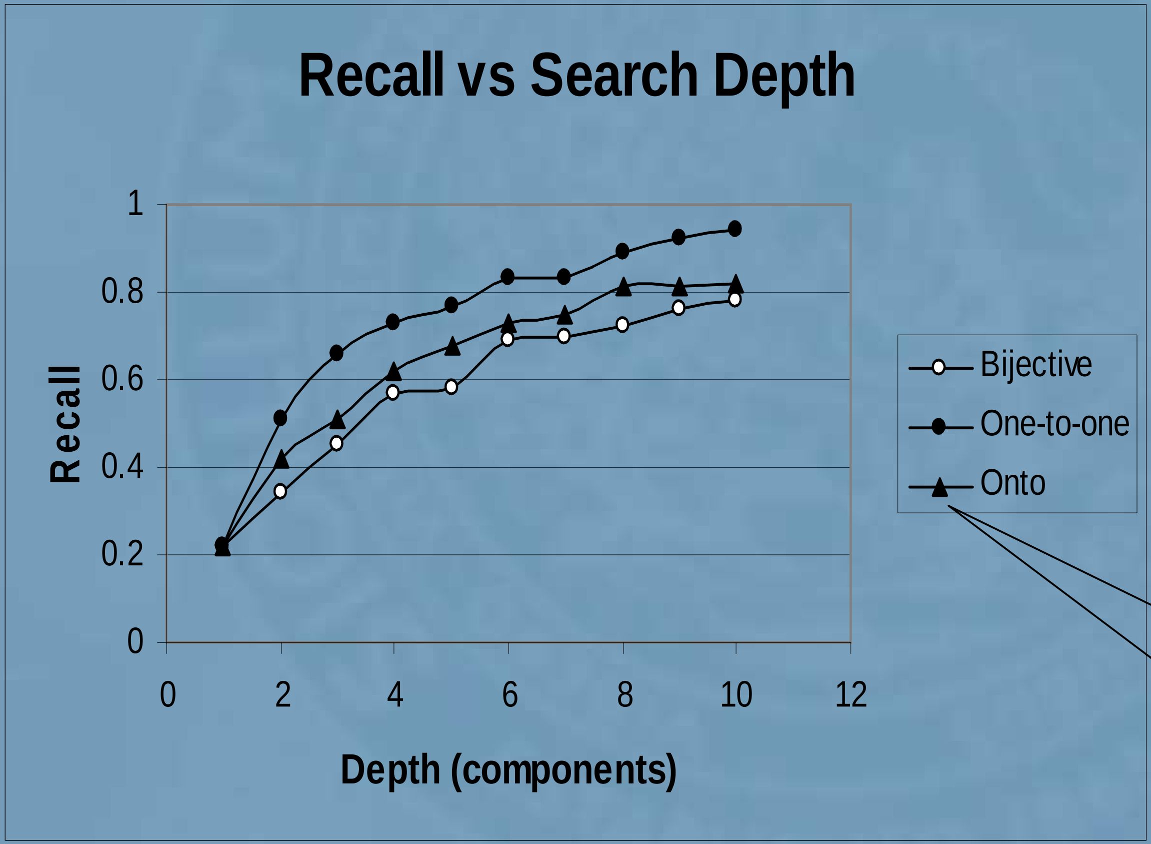


Well distributed components/arch (1-6)

Most groups found early, diminishing returns

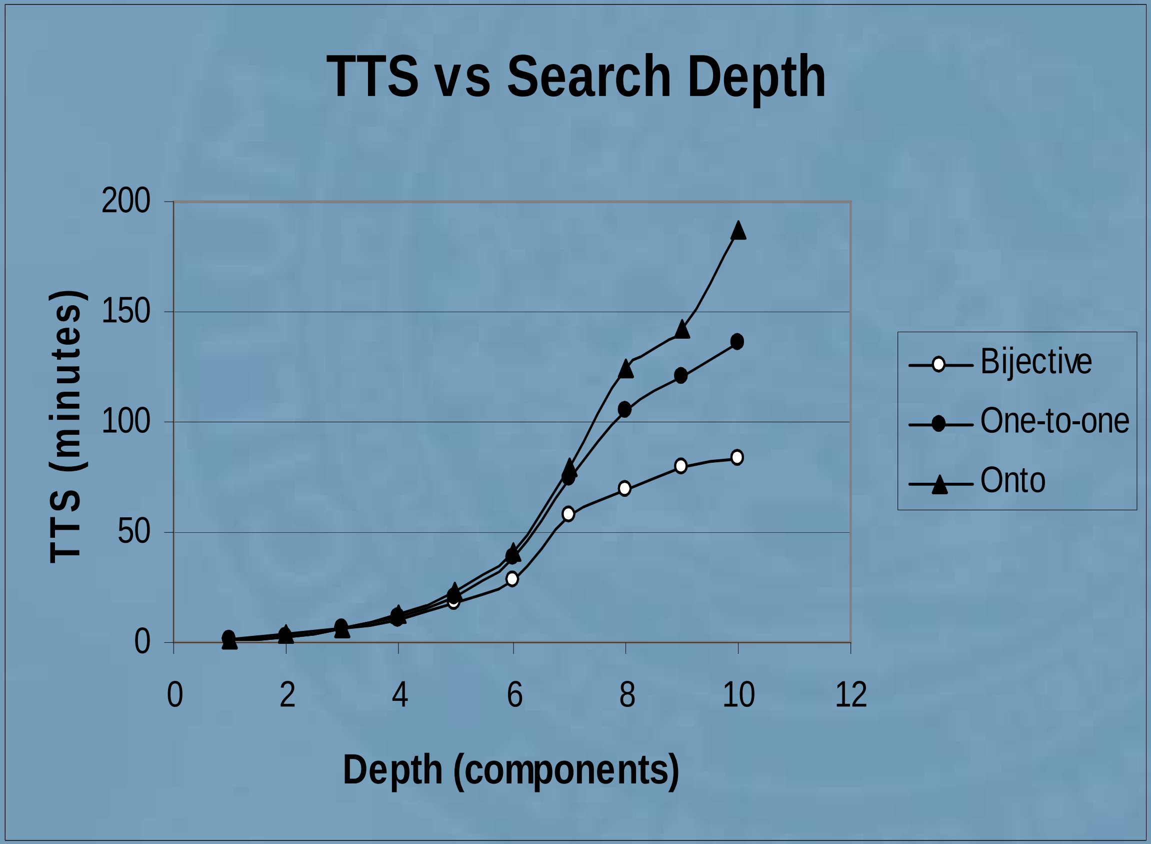
Fade due to failed proof strategies on complex sub-problems

Recall(2) vs. Search Depth



Small gains, useful in math library (e.g. power functions)

TTS vs. Search Depth



Other Results

- Precision
  - Between 98-100%
  - No tradeoff with recall
    - Formal methods for adaptation/retrieval
    - Theorem-prover precision
- Time
  - 92% spent on retrieval
  - Most of that spent on “dead-ends”
  - Hardware engineers will wait, will software engineers?

Future Work & Limitations

- Assumes shared-variable communication
  - Include communication protocols as search criteria
  - Include connector specifications in the library
- Only synthesizes three architectures
- Limited by theorem-prover, search depth
- Reduce TTS (retrieval limitation)
- Ranking of partial solutions

Related Work

- Specification-based Retrieval
  - Zaremski and Wing – developed match lattice, retrieval engine for Larch/ML specifications
  - Penix/Patil – developed REBOUND/SOCCER retrieval engine, used feature-based classification
  - Fischer – designed NORA/HAMMR retrieval engine, used a layered architecture, included model checker

Related Work

- Component Adaptation
  - Penix – Suggested using architectures for behavioral adaptation
  - Purtilo and Atlee – created NIMBLE, automated module interface adaptation
  - Jeng and Cheng – identified necessary modifications to reuse general components to specific problems

Related Work

- Synthesis, Slicing, Architecting for Reuse
  - Chen and Cheng – developed ARBIE, an architecture-based reuse framework
  - Zhao – applied slicing to ADL for reuse-of-the-large
  - Bhansali – created a hybrid approach to reuse of geometrics, uses code-level reuse, architectures, and semi-synthesis of code fragments

Conclusions

- Presented framework for specification-based component retrieval and adaptation
- Behavioral adaptation was automated using architectures
- Sequential, alternative, and parallel adaptation implemented to adapt partial matches
- Provided sound definitions to synthesize sub-problems to satisfy component adaptation
- ~94% recall, ~100% precision (tradeoffs for TTS)
- Questions