Lecture 21
CSP - Applications

David Garlan

Carnegie Mellon University

Models of Software Systems

15-671

Fall 1995

CSP Concurrency: Four Lectures

- CSP Processes
  - > basic notions of processes
  - > prefix, recursion, choice, parallel combination
  - > deadlock
- Reasoning with CSP
  - > skip, input/output
  - > semantics: traces, specifications
  - > laws of inference
- Nondeterminism
  - > external choice, interleaving, hiding
  - > the need for a richer semantic model
- Applications
  - > refinement
  - > specification of software architectures

Today's Topics

- Quick review of concepts
- Tidy up from last lecture:
  - > More on internal choice
  - > Interleaving and Hiding
- Refinement
- Specification of Software Architectures

Lecture 1

α **P: alphabet of P**

the events that P can engage in

$e \rightarrow P$: e *then* P

the process that does event e and then acts like P

$a \rightarrow P \mid b \rightarrow Q$: a then P *choose* b then Q

do a then act like P or do b then act like Q

$(x{:}B \rightarrow P(x))$: x *from* B then P of x

do one of the events from B

$P \parallel Q$: P parallel Q

run P and Q in parallel, synchronizing on shared events

Special processes: STOP, RUN

Lecture 2

- **Skip to represent successful termination**
  - SKIP = √ $\rightarrow$ STOP
- **Input/output**
  - c?in, c!out
- **Semantics of CSP**
  - meaning of a process is its set of event traces
  - specifications and "satisfies"
  - induction over traces sets
- **Axioms**
  - permit reasoning
  - allow transformation of process expressions
  - can eliminate parallel construct

Lecture 3

- Recursion is well defined when the recursive expression is guarded
- Method of proof by induction over traces
  > show that STOP satisfies SPEC
  > show that if X satisfies SPEC then F(X) satisfies SPEC
- Nondeterminism in sequencial systems
  > state transitions, relations, Z operations
- Non-determinism in concurrent systems
  > generalized (external) choice [] : a $\rightarrow$ P [] a $\rightarrow$ Q
  > internal choice $\sqcap$: a $\rightarrow$ P $\sqcap$ b $\rightarrow$ Q
- Refusals

More on Internal Choice

- The key property to keep in mind:
  $(a \rightarrow P \sqcap b \rightarrow Q) \parallel (a \rightarrow R)$
  can deadlock on the first action
  if the first process internally chooses to engage in b.
- In constrast:
  $(a \rightarrow P \,[]\, b \rightarrow Q) \parallel (a \rightarrow R)$
  must engage in the a.

Refinement

- **The basic idea of refinement is process substitutability**
  - > **Process <u>Q refines P</u> if we can replace P by Q in any context and not know the difference in terms of its behavior.**
  - > **We write P   Q for "Q refines P"**
  - > **This is a partial order**
  - > **So if P   Q and Q   P then P = Q**

Refinement continued

- What does it take for a process Q to be substitutable for process P?
- Q and P must have the same alphabet for this to make sense.
- Necessary condition: Q has a subset of the traces of P.
  - > note: if traces not the same, we can find a process that deadlocks with Q but not with P.
- However, in addition we can allow Q to resolve some of the non-determinism in P.

  Example:

  Let $\quad Chg = qtr \rightarrow (DimeFirst \sqcap NicklesFirst)$

  and $\quad Chg1 = qtr \rightarrow DimeFirst$

  Then Chg1 refines Chg

Refinement continued

- This works because we can never tell from the external behavior whether Q is really P and P simply “chooses” not to take one of its branches.
- Similarly a M&M-choosing customer refines one who sometimes chooses M&Ms, but other times chooses Snickers.

Success/Renaming/Enrichment

- CSP defines a special success event: $\surd$
  - We can also define the success process: ✓
  - ✓ = $\surd \rightarrow$ STOP
- It is sometimes useful to rename the events in a process. If f is a function from event names to event names:
  - f(p) is the process that renames all of P's events.
  - Special case where f(e) = foo.e is indicated by foo:P for f(P).
- We can enrich the alphabet of a process P using
  - P+s = the process like P, but with alphabet α P $\cup$ s
  - Formally: P+s = P || STOPs

The Challenge for Architectural Description

- Software Architecture frees us from the shackles of programming language abstractions
  - new kinds of components
    - not just ADTs and objects
  - new kinds of glue
    - not just procedure call, data sharing
  - new patterns and idioms
    - not just algorithms and data structure

- **How can we establish intellectual control over this new world?**

Wright { (1) express arch descriptions precisely

  » both syntax and semantics

(2) develop soundness criteria & tools to check

  » what type checkers and linkers do for current systems

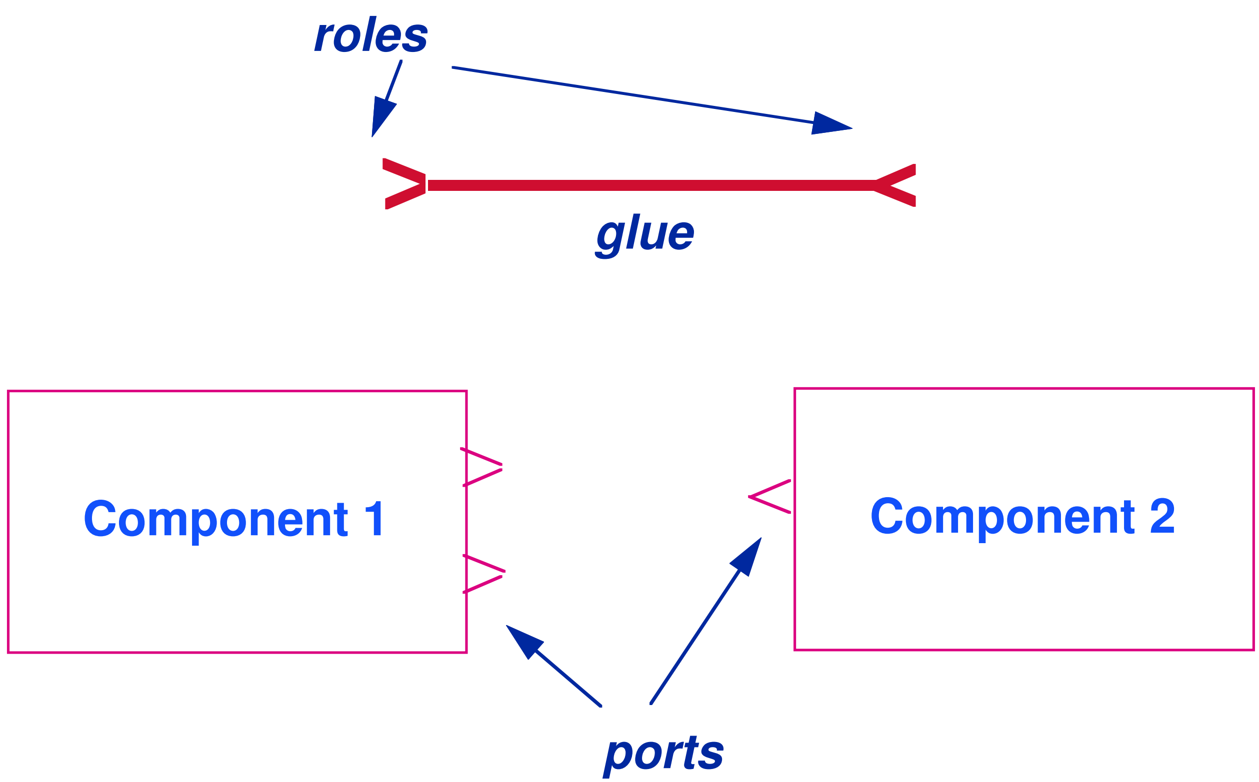
Aesop { (3) exploit patterns and styles

  » without unnecessary proliferation and isolation of new design languages and tools

Wright Specification: Key features

- **Ontological basis:**
  components, connectors, configurations
- **Components have multiple points of interaction: ports**
- **Both components <u>and</u> <u>connectors</u> are first class abstractions**
  > **can define new *types* of components & connectors**
  > **these define interfaces & semantics**
  > **they can (both) be hierarchically decomposed**

Basic Elements



Wright

System **SimpleClientServer**
  Component **Server**
    Port **provide:** ***<provide protocol>***
  Component **Client**
    Port **request:** ***<request protocol>***
  Connector **C-S-connector**
    Role **client:** ***<request protocol>***
    Role **server:** ***<server protocol>***
    Glue ***<glue protocol>***
  Instances
    s: Server ; c: Client ; cs: C-S-connector
  Attachments
    s.provide as cs.server
    c.request as cs.client

Component types

Connector types

Instances

Configurations

Wright Connectors

- **Point of View:**
  > Architectural connectors are first-class abstractions.
- **Approach:**
  1. Specification of connector types as protocols.
  2. Formal criteria for determining when a connector type is well formed.
  3. Sound rules for checking when an instance of a connector can be used in a given context.
  4. Practical tools for carrying out these checks automatically.

Key Issues

- What does it mean for a connector type to be well-formed?

  ans: they are deadlock- free
- What does it mean for a composition to be well-formed?

  ans: components ports refine connector roles
- What does it mean for a composition to be complete?

  ans: ports can be “capped” with “do-nothing” connector
- How can we check these properties?

  ans: use model-checking technology

A Formal Basis for Architectural Connection

Notation (based on CSP):

> **Events:** e, f, push, pop, request, √
>
> **Processes:** P, Q, Stack, Client, Server, ✓
>
> **Prefix:** e ` P, push ` Stack, request ` Server
>
> **Internal Choice:** P ¸ Q
>
> **External Choice:** P [] Q

Example: What is a Pipe?

Connector Pipe

Roles:

Writer = (write!x ` Writer) ¸ (close ` 3)

Reader = Read ¸ Exit

where Read = (read?x ` Reader) [] (read-eof ` Exit)

Exit = close ` 3

Glue = Writer.write?x ` Glue []

Reader.read!y ` Glue []

Writer.close ` ReadOnly []

Reader.close ` WriteOnly

where …

Semantics

Informally:

Roles are independent, constrained only by the glue.

Formally:

Use the parallel interaction operator (¨).

<u>Connector</u> C

<u>Roles</u>: <u>R1</u> = R1; ...; <u>Rn</u> = Rn

<u>Glue</u> = ***Glue***

is the process:

Glue ¨ (<u>R1:</u> R1 ¨ ... ¨ <u>Rn</u>: Rn)

where

n:P labels all of the events in process P with name n

Glue **alphabet is sufficiently large**

So What?

- Can now use the theory to analyze our descriptions:
- Example: a connector doesn't get 'stuck'.

  A process P is deadlock-free if whenever it reaches a state in which it cannot make progress, the last event to have been executed is √.
- Can check this property for finite (state) processes using commercial model checkers.
- Discover: Some of our initial descriptions were not deadlock free!
- Other properties: conservatism

Connector Instantiation

- **Attaching a port as a role:**
  - > ***Informally:*** **port stands in for the role.**
  - > ***Formally:*** **instantiating roles P1 .. Pn gives**

    Glue ¨ (R1 : P1 ¨ ... ¨ Rn: Pn)

- **Ports need not be identical to the roles**
  - > **pipe can be connected to a file**
  - > **client can use a subset of a server's facilities**
- **When is it ok to attach a port to a role?**

Compatibility
(of a Port with a Role)

- **<u>Yes</u>:**

  Port P = (push ` P) ¸ ✓

  Role R = (push ` R) ¸ (pop ` R) ¸ ✓

- **<u>No</u>:**

  Port P = (push ` P) ¸ ✓

  Role R = init ` R’

  <u>where</u> R’ = (push ` R’) ¸ (pop ` R’) ¸ ✓

Port-Role Compatibility

- **Informally:**
  > **A port P is compatible with a role R if the behavior of P does not violate the promises of R**
- **Formally we can use process refinement:**
  "R P "
- **Details**
  > **Alphabets may not be the same**
  Solved by augmenting alphabets of both processes.
  > **Only care about the behavior of the port in the context of the connection that it is participating in.**

So What?

- This theory allows us to "compatibility check" architectural descriptions
  > Analogous to (but subsumes) type checking
  > Can be automated for this notation using model checking
- Compatibility guarantees important properties

  Theorem: (soundness)
  If a connector is deadlock free and conservative then any compatible instantiation will also be deadlock free.