A Process Algebraic Approach to Software Architecture Design

Marco Bernardo

University of Urbino – Italy

Table of Contents

Starting Point:
On the Usability of Process Algebra

Process Algebra

- Conceived for reasoning about the semantics of concurrent programming around 1980 (basic concurrent languages like CCS, CSP, ACP, π-calculus, ...).
- Means for understanding communicating concurrent systems together with their various aspects (nondeterminism, priority, probability, time, ...).
- Scientific impact witnessed by a rich literature and the development of formal description techniques and automated software tools for system modeling and analysis (ISO language LOTOS; tools CADP, CWB, FDR, μCRL, ...).
- Process calculi and behavioral equivalences yielding process algebra used for teaching foundations of concurrent programming as well as model-driven design of concurrent, distributed, and mobile systems.

- Communicating concurrent systems are often required to possess a high degree of reactivity to external stimuli and are usually nonterminating.
- They are typically structured into a set of autonomous components that can evolve independently of each other and from time to time can communicate or simply synchronize.
- *Nondeterminism* in the final result or in the computation can arise due to the different speeds of the components, the interaction scheme among the components, and the scheduling policies that are adopted.
- Input-output transformation approach for sequential programs/systems no longer applicable in the case of communicating concurrent systems.

- Process algebra is the linguistic counterpart of computational models developed for communicating concurrent systems:
  - *Trace* model.
  - *Synchronization tree* model.
  - *Petri net* model.
  - *Event structure* model.
- Compositional modeling by means of behavioral operators expressing concepts like the sequential composition, the alternative composition, and the parallel composition of processes (represent the behavior of systems).
- Abstraction from certain parts of system behavior by distinguishing between visible and invisible actions (represent system activities).

Running Example: Producer-Consumer System

- General description:
  - Three components: producer, finite-capacity buffer, consumer.
  - The producer deposits items into the buffer as long as the buffer capacity is not exceeded.
  - Stored items are then withdrawn by the consumer according to some predefined discipline (like FIFO or LIFO).
- Specific scenario:
  - The buffer has only two positions.
  - Items are identical, hence the discipline is not important.

Syntax for PA

- Specification language for communicating concurrent systems.
- Support for compositionality (building complex models by combining simpler models).
- Capability of abstraction (neglecting certain details of a model).
- Based on actions and behavioral operators.
- $Name_{\mathrm{v}}$: set of visible action names.
- $Name = Name_{\mathrm{v}} \cup \{\tau\}$: set of all action names.
- $Relab = \{\varphi : Name \to Name \mid \varphi^{-1}(\tau) = \{\tau\}\}$: set of visibility-preserv. relabeling functions.
- Var: set of process variables ($Const$: set of process constants).

- Process term syntax for process language $\mathcal{PL}$:

P	$::=$	$\underline{0}$	inactive process	
	$\mid$	$a\,.\,P$	action prefix	$(a \in Name)$
	$\mid$	$P + P$	alternative composition	
	$\mid$	$P \parallel_S P$	parallel composition	$(S \subseteq Name_{\mathbf{V}})$
	$\mid$	$P \,/\, H$	hiding	$(H \subseteq Name_{\mathbf{V}})$
	$\mid$	$P \setminus L$	restriction	$(L \subseteq Name_{\mathbf{V}})$
	$\mid$	$P[\varphi]$	relabeling	$(\varphi \in Relab)$
	$\mid$	X	process variable	$(X \in Var)$
	$\mid$	$\mathrm{rec}\, X : P$	recursion	$(X \in Var)$

(process constants are defined by means of equations of the form $B \triangleq P$).

- $P_1 + P_2$ behaves as P_1 or P_2 depending on which executes first.
- The choice among several enabled actions is solved nondeterministically.
- The choice is internal if the enabled actions are all invisible, otherwise the choice can be influenced by the external environment.
- $P_1 \|_S P_2$ behaves as P_1 in parallel with P_2 under synchronization set S.
- Actions whose name does not belong to S are executed autonomously by P_1 and by P_2.
- Synchronization is forced between any action enabled by P_1 and any action enabled by P_2 that have the same name belonging to S, in which case the resulting action has the same name as the two original actions

($S = \emptyset$ implies P_1 and P_2 fully independent, $S = Name_{\mathrm{V}}$ implies P_1 and P_2 fully synchronized).

- $\underline{0}$ is a terminated process and hence cannot execute any action.
- $a \,.\, P$ can perform a and then behaves as P (action-based sequential composition).
- $P \,/\, H$ behaves as P but every action belonging to H is turned into τ (abstraction mechanism; can be used for preventing a process from communicating).
- $P \setminus L$ behaves as P but every action belonging to L is forbidden (same effect as $P \parallel_L \underline{0}$).
- $P[\varphi]$ behaves as P but every action is renamed according to function φ (redundance avoidance; encoding of the previous two operators if φ is non-visib.-pres./partial).
- Operator precedence: unary operators $>$ $+$ $>$ $\parallel$.
- Operator associativity: $+$ and $\parallel$ are left associative.

- $\mathrm{rec}\, X : P$ behaves as P with every free occurrence of process variable X being replaced by $\mathrm{rec}\, X : P$.
- A process variable is said to occur *free* in a process term if it is not in the scope of a rec binder for that variable, otherwise it is said to be *bound* in that process term.
- A process term is said to be *closed* if all of its occurrences of process variables are bound, otherwise it is said to be *open*.
- A process term is said to be *guarded* iff all of its occurrences of process variables are in the scope of action prefix operators.
- $\mathbb{P}$: set of closed and guarded process terms (fully defined, finitely branching).

- **Running example** (process syntax):
  - ⊙ Conventions: action names are verbs composed of lower-case letters, process constant names are nouns starting with an upper-case letter.
  - ⊙ The only observable activities are deposits and withdrawals.
  - ⊙ Visible actions: *deposit* and *withdraw*.
  - ⊙ Structure-independent process algebraic description:

$$ProdCons_{0/2} \stackrel{\Delta}{=} deposit \,.\, ProdCons_{1/2}$$

$$ProdCons_{1/2} \stackrel{\Delta}{=} deposit \,.\, ProdCons_{2/2} + withdraw \,.\, ProdCons_{0/2}$$

$$ProdCons_{2/2} \stackrel{\Delta}{=} withdraw \,.\, ProdCons_{1/2}$$

  - ⊙ Specification to which every correct implementation should conform.

Semantics for PA

- Mathematical model in the form of a state transition graph representing all computations and branching points (synchronization tree if unwound).
- Every process term $P \in \mathbb{P}$ is mapped to a labeled transition system $[\![P]\!]$:
  - Each state corresponds to a process term into which P can evolve.
  - The initial state corresponds to P.
  - Each transition from a source state to a target state is labeled with the action that determines the corresponding state change.
- The transition relation $\longrightarrow_P$ of $[\![P]\!]$ is contained in the smallest set of elements of $\mathbb{P} \times \mathit{Name} \times \mathbb{P}$ that satisfy some operational semantic rules defined by induction on the syntactical structure of process terms.

- Derivation of one single transition at a time by applying the operational semantic rules to the source state of the transition.
- Basic rule for action prefix, inductive rules for all the other operators.
- Different formats: dynamic operators ($.\ +$), static operators ($\| \ / \ \backslash \ [\,]$).
- No rule for $\underline{0}$: $[\![\underline{0}]\!]$ has a single state and no transitions.
- Operational semantic rule for action prefix:

$$a\,.\,P \xrightarrow{\ a\ } P$$

- Operational semantic rule for recursion:

$$\frac{P\{\mathrm{rec}\,X : P \hookrightarrow X\} \xrightarrow{\ a\ } P'}{\mathrm{rec}\,X : P \xrightarrow{\ a\ } P'} \qquad \left(\frac{B \triangleq P \qquad P \xrightarrow{\ a\ } P'}{B \xrightarrow{\ a\ } P'} \right)$$

- Operational semantic rules for alternative composition:

$$\frac{P_1 \xrightarrow{a} P_1'}{P_1 + P_2 \xrightarrow{a} P_1'} \qquad \frac{P_2 \xrightarrow{a} P_2'}{P_1 + P_2 \xrightarrow{a} P_2'}$$

- If several actions are initially enabled, the choice among them is solved nondeterministically due to the absence of quantitative information associated with them (priority, probability, time).
- The choice is internal if the initially enabled actions are all invisible.
- Otherwise the choice can be influenced by the external environment.

- Operational semantic rules for parallel execution:

$$\frac{P_1 \xrightarrow{a} P_1' \qquad a \notin S}{P_1 \parallel_S P_2 \xrightarrow{a} P_1' \parallel_S P_2} \qquad \frac{P_2 \xrightarrow{a} P_2' \qquad a \notin S}{P_1 \parallel_S P_2 \xrightarrow{a} P_1 \parallel_S P_2'}$$

- Operational semantic rule for synchronization:

$$\frac{P_1 \xrightarrow{a} P_1' \qquad P_2 \xrightarrow{a} P_2' \qquad a \in S}{P_1 \parallel_S P_2 \xrightarrow{a} P_1' \parallel_S P_2'}$$

- Parallel composition is thus given an interleaving semantics.

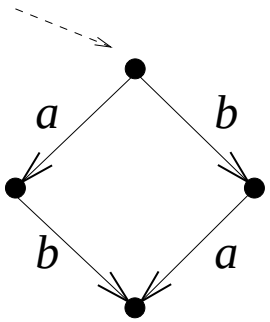
- The following process terms represent structurally different systems:

$$a\,.\,\underline{0} \parallel_{\emptyset} b\,.\,\underline{0}$$

$$a\,.\,b\,.\,\underline{0} + b\,.\,a\,.\,\underline{0}$$

  but they are indistinguishable by an external observer.

- Black-box semantics given by the same labeled transition system:



- Truly concurrent semantics are possible (via Petri net or event structure models) in which causally independent activities can also occur simultaneously.

- Operational semantic rules for hiding, restriction, relabeling:

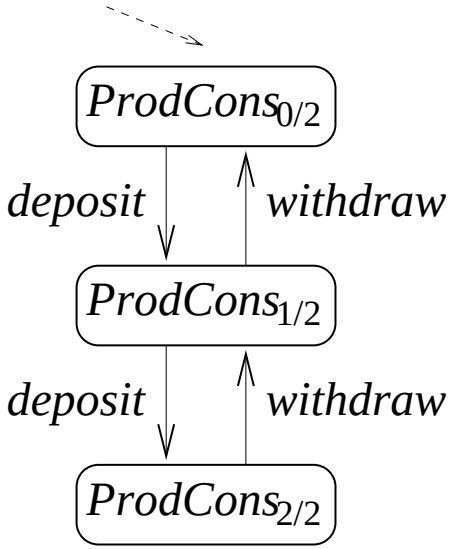
$$\frac{P \xrightarrow{a} P' \qquad a \in H}{P \,/\, H \xrightarrow{\tau} P' \,/\, H} \qquad \frac{P \xrightarrow{a} P' \qquad a \notin H}{P \,/\, H \xrightarrow{a} P' \,/\, H}$$

$$\frac{P \xrightarrow{a} P' \qquad a \notin L}{P \setminus L \xrightarrow{a} P' \setminus L}$$

$$\frac{P \xrightarrow{a} P'}{P[\varphi] \xrightarrow{\varphi(a)} P'[\varphi]}$$

- $[\![P]\!]$ is finite state if no recursive definition in P contains static operators.

- **Running example** (process semantics):
  - Labeled transition system $[\![ProdCons_{0/2}]\!]$ with explicit states:



  - Obtained by mechanically applying the operational semantic rules for process constant, alternative composition, and action prefix.

Behavioral Equivalences for PA

- Establishing whether two process terms are equivalent amounts to establishing whether the systems they represent behave the same.
- Preservation of compositionality and abstraction should be achieved.
- Useful for theoretical and applicative purposes:
  - ⊙ Comparing syntactically different process terms on the basis of the behavior they exhibit.
  - ⊙ Relating process algebraic descriptions of the same system at different abstraction levels (top-down modeling).
  - ⊙ Manipulating process algebraic descriptions in a way that preserves certain properties (state space reduction before analysis).

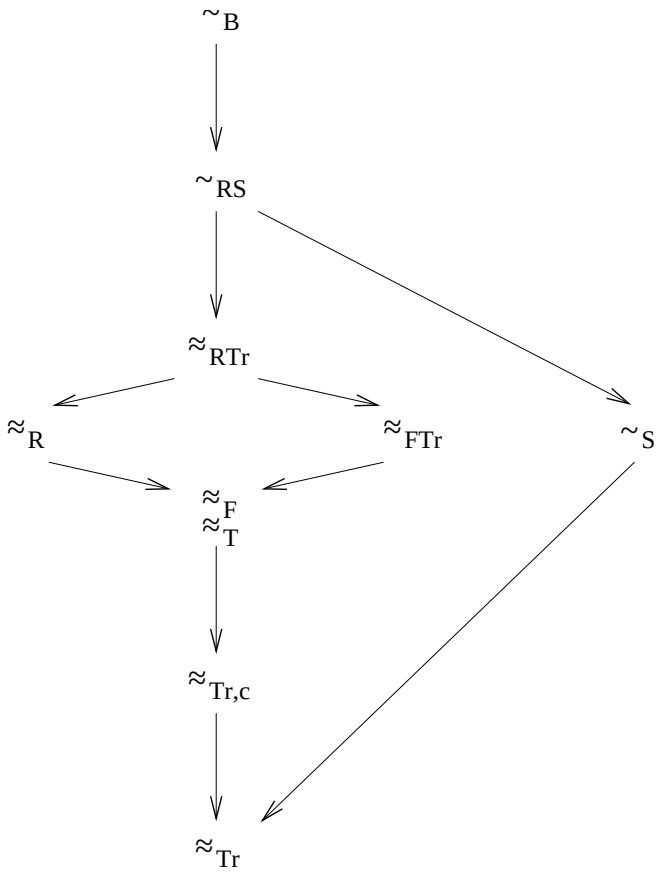
- Features of a good behavioral equivalence:
  - Being a congruence with respect to the behavioral operators, so as to support compositional reasoning.
  - Having a sound and complete axiomatization, which elucidates the fundamental equational laws of the equivalence with respect to the behavioral operators (rewriting rules for syntactical manipulation).
  - Having a modal logic characterization, which shows the behavioral properties preserved by the equivalence (diagnostic information).
  - Being equipped with an efficient verification algorithm.
  - Being able to abstract from invisible actions.

- Most studied approaches to the definition of behavioral equivalences:
  - Trace approach: two process terms are equivalent if they are able to execute the same sequences of actions ($\approx_{\mathrm{Tr}}$).
  - Abstraction from branching points leads to *deadlock insensitivity*.
  - Testing approach: two process terms are equivalent if no difference can be discovered when interacting with them by means of tests and comparing their reactions ($\approx_{\mathrm{T}}$).
  - Checking whether *every test* may/must be passed.
  - Bisimulation approach: two process terms are equivalent if they are able to mimic each other's behavior after each action execution ($\sim_{\mathrm{B}}$).
  - Faithful account of branching points leads to *overdiscrimination*.

- Variants of bisimulation equivalence:
  - Simulation equivalence: it is the intersection of two preorders, each of which considers the capability of stepwise behavior mimicking in one single direction ($\sim_{\mathrm{S}}$).
  - Ready-simulation equivalence: same as simulation equivalence, with in addition the fact that each of the two preorders checks for the equality of the sets of actions that are stepwise enabled ($\sim_{\mathrm{RS}}$).
- Less discriminating than bisimulation equivalence.
- The distinctions they make can be traced provided that reasonable operators are included in the process language.

- Deadlock-sensitive variants of trace equivalence:
  - ⊙ Completed-trace equivalence: it compares process terms also with respect to traces that lead to deadlock ($\approx_{\mathrm{Tr,c}}$).
  - ⊙ Failure equivalence: it takes into account the set of visible actions that can be refused after executing a trace ($\approx_{\mathrm{F}}$).
  - ⊙ Ready equivalence: it takes into account the set of visible actions that can be performed after executing a trace ($\approx_{\mathrm{R}}$).
  - ⊙ Failure-trace equivalence: it takes into account the sets of visible actions that can be refused at each step of a trace ($\approx_{\mathrm{FTr}}$).
  - ⊙ Ready-trace equivalence: it takes into account the sets of visible actions that can be performed at each step of a trace ($\approx_{\mathrm{RTr}}$).

- Linear-time/branching-time spectrum (for processes without invisible actions):



Bisimulation Equivalence

- Whenever a process term can perform a certain action, then any process term equivalent to the given one has to be able to perform that action.
- The derivative process terms into which all the previous process terms evolve after executing that action must still be equivalent to each other, so that this game can go on endlessly.
- A binary relation $\mathcal{B}$ over $\mathbb{P}$ is a bisimulation iff, whenever $(P_1, P_2) \in \mathcal{B}$, then for all actions $a \in \mathit{Name}$:
  - Whenever $P_1 \xrightarrow{a} P_1'$, then $P_2 \xrightarrow{a} P_2'$ with $(P_1', P_2') \in \mathcal{B}$.
  - Whenever $P_2 \xrightarrow{a} P_2'$, then $P_1 \xrightarrow{a} P_1'$ with $(P_1', P_2') \in \mathcal{B}$.
- Bisimulation equivalence $\sim_{\mathrm{B}}$ is the union of all the bisimulations.

- In order for $P_1 \sim_{\mathrm{B}} P_2$, it is necessary that for all $a \in \mathit{Name}$ there exist $P_1', P_2' \in \mathbb{P}$ such that:

$$\boxed{P_1 \xrightarrow{a} P_1' \iff P_2 \xrightarrow{a} P_2'}$$

- A binary relation $\mathcal{B}$ over $\mathbb{P}$ is a bisimulation up to $\sim_{\mathrm{B}}$ iff, whenever $(P_1, P_2) \in \mathcal{B}$, then for all actions $a \in \mathit{Name}$:
  - ⊙ Whenever $P_1 \xrightarrow{a} P_1'$, then $P_2 \xrightarrow{a} P_2'$ with $P_1' \sim_{\mathrm{B}} Q_1 \mathcal{B} Q_2 \sim_{\mathrm{B}} P_2'$.
  - ⊙ Whenever $P_2 \xrightarrow{a} P_2'$, then $P_1 \xrightarrow{a} P_1'$ with $P_1' \sim_{\mathrm{B}} Q_1 \mathcal{B} Q_2 \sim_{\mathrm{B}} P_2'$.
- Focus on important pairs of process terms that form a bisimulation.
- In order for $P_1 \sim_{\mathrm{B}} P_2$, it is sufficient to find a bisimulation up to $\sim_{\mathrm{B}}$ that contains (P_1, P_2).

- $\sim_{\mathrm{B}}$ is a congruence with respect to all the dynamic and static operators as well as recursion.
- Substituting equals for equals does not alter the overall meaning in any process context.
- Let $P_1, P_2 \in \mathbb{P}$. Whenever $P_1 \sim_{\mathrm{B}} P_2$, then:

$$a \,.\, P_1 \;\sim_{\mathrm{B}}\; a \,.\, P_2$$

$$P_1 + P \;\sim_{\mathrm{B}}\; P_2 + P \qquad\qquad P + P_1 \;\sim_{\mathrm{B}}\; P + P_2$$

$$P_1 \parallel_S P \;\sim_{\mathrm{B}}\; P_2 \parallel_S P \qquad\qquad P \parallel_S P_1 \;\sim_{\mathrm{B}}\; P \parallel_S P_2$$

$$P_1 \,/\, H \;\sim_{\mathrm{B}}\; P_2 \,/\, H$$

$$P_1 \setminus L \;\sim_{\mathrm{B}}\; P_2 \setminus L$$

$$P_1[\varphi] \;\sim_{\mathrm{B}}\; P_2[\varphi]$$

- Recursion: extend $\sim_{\mathrm{B}}$ to open process terms by replacing all variables freely occurring outside rec binders with every closed process term.
- Let $P_1, P_2 \in \mathcal{PL}$ be guarded process terms containing free occurrences of $k \in \mathbb{N}$ process variables $X_1, \ldots, X_k \in \mathit{Var}$ at most.
- We define $P_1 \sim_{\mathrm{B}} P_2$ iff:

$$P_1\{Q_i \hookrightarrow X_i \mid 1 \leq i \leq k\} \sim_{\mathrm{B}} P_2\{Q_i \hookrightarrow X_i \mid 1 \leq i \leq k\}$$

  for all $Q_1, \ldots, Q_k \in \mathbb{P}$.
- Whenever $P_1 \sim_{\mathrm{B}} P_2$, then:

$$\mathrm{rec}\, X : P_1 \sim_{\mathrm{B}} \mathrm{rec}\, X : P_2$$

- $\sim_{\mathrm{B}}$ has a sound and complete axiomatization over the set $\mathbb{P}_{\mathrm{nrec}}$ of nonrecursive process terms of $\mathbb{P}$.
- Basic laws (commutativity, associativity, and neutral element of $+$):

$(\mathcal{X}_{\mathrm{B},1})$	$P_1 + P_2$	$=$	$P_2 + P_1$
$(\mathcal{X}_{\mathrm{B},2})$	$(P_1 + P_2) + P_3$	$=$	$P_1 + (P_2 + P_3)$
$(\mathcal{X}_{\mathrm{B},3})$	$P + \underline{0}$	$=$	P

- Characterizing laws (idempotency of $+$, not valid for synchronization tree isomorphism):

$(\mathcal{X}_{\mathrm{B},4})$	$P + P$	$=$	P

- Expansion law (interleaving view of concurrency; I and J nonempty and finite):

$$(\mathcal{X}_{\mathrm{B},5}) \quad \sum_{i\in I} a_i \,.\, P_i \parallel_S \sum_{j\in J} b_j \,.\, Q_j = \sum_{k\in I, a_k\notin S} a_k \,.\, \left(P_k \parallel_S \sum_{j\in J} b_j \,.\, Q_j\right) + \sum_{h\in J, b_h\notin S} b_h \,.\, \left(\sum_{i\in I} a_i \,.\, P_i \parallel_S Q_h\right) + \sum_{k\in I, a_k\in S} \sum_{h\in J, b_h = a_k} a_k \,.\, (P_k \parallel_S Q_h)$$

$$(\mathcal{X}_{\mathrm{B},6}) \quad \sum_{i\in I} a_i \,.\, P_i \parallel_S \underline{0} = \sum_{k\in I, a_k\notin S} a_k \,.\, P_k$$

$$(\mathcal{X}_{\mathrm{B},7}) \quad \underline{0} \parallel_S \sum_{j\in J} b_j \,.\, Q_j = \sum_{h\in J, b_h\notin S} b_h \,.\, Q_h$$

$$(\mathcal{X}_{\mathrm{B},8}) \quad \underline{0} \parallel_S \underline{0} = \underline{0}$$

- Distribution laws (for unary static operators):

$$
\begin{array}{lrcll}
(\mathcal{X}_{\mathrm{B},9}) & \underline{0} \,/\, H & = & \underline{0} & \\
(\mathcal{X}_{\mathrm{B},10}) & (a \,.\, P) \,/\, H & = & \tau \,.\, (P \,/\, H) & \text{if } a \in H \\
(\mathcal{X}_{\mathrm{B},11}) & (a \,.\, P) \,/\, H & = & a \,.\, (P \,/\, H) & \text{if } a \notin H \\
(\mathcal{X}_{\mathrm{B},12}) & (P_1 + P_2) \,/\, H & = & P_1 \,/\, H + P_2 \,/\, H & \\
 & & & & \\
(\mathcal{X}_{\mathrm{B},13}) & \underline{0} \setminus L & = & \underline{0} & \\
(\mathcal{X}_{\mathrm{B},14}) & (a \,.\, P) \setminus L & = & \underline{0} & \text{if } a \in L \\
(\mathcal{X}_{\mathrm{B},15}) & (a \,.\, P) \setminus L & = & a \,.\, (P \setminus L) & \text{if } a \notin L \\
(\mathcal{X}_{\mathrm{B},16}) & (P_1 + P_2) \setminus L & = & P_1 \setminus L + P_2 \setminus L & \\
 & & & & \\
(\mathcal{X}_{\mathrm{B},17}) & \underline{0}[\varphi] & = & \underline{0} & \\
(\mathcal{X}_{\mathrm{B},18}) & (a \,.\, P)[\varphi] & = & \varphi(a) \,.\, (P[\varphi]) & \\
(\mathcal{X}_{\mathrm{B},19}) & (P_1 + P_2)[\varphi] & = & P_1[\varphi] + P_2[\varphi] &
\end{array}
$$

- $DED(\mathcal{X}_{\mathrm{B}})$: deduction system based on all the previous axioms plus:
  - Reflexivity: $\mathcal{X}_{\mathrm{B}} \vdash P = P$.
  - Symmetry: $\mathcal{X}_{\mathrm{B}} \vdash P_1 = P_2 \implies \mathcal{X}_{\mathrm{B}} \vdash P_2 = P_1$.
  - Transitivity: $\mathcal{X}_{\mathrm{B}} \vdash P_1 = P_2 \wedge \mathcal{X}_{\mathrm{B}} \vdash P_2 = P_3 \implies \mathcal{X}_{\mathrm{B}} \vdash P_1 = P_3$.
  - Substitutivity: $\mathcal{X}_{\mathrm{B}} \vdash P_1 = P_2 \implies \mathcal{X}_{\mathrm{B}} \vdash a\,.\,P_1 = a\,.\,P_2 \wedge \ldots$
- The deduction system $DED(\mathcal{X}_{\mathrm{B}})$ is sound and complete for $\sim_{\mathrm{B}}$ over $\mathbb{P}_{\mathrm{nrec}}$; i.e., for all $P_1, P_2 \in \mathbb{P}_{\mathrm{nrec}}$:

$$\boxed{\mathcal{X}_{\mathrm{B}} \vdash P_1 = P_2 \iff P_1 \sim_{\mathrm{B}} P_2}$$

- $\sim_{\mathrm{B}}$ has a modal logic characterization based on Hennessy-Milner logic.
- Basic truth values and propositional connectives, plus modal operators expressing how to behave after executing certain actions.
- Syntax of the modal language $\mathcal{ML}_{\mathrm{B}}$ $(a \in \mathit{Name})$:

ϕ	$::=$	true	basic truth value
	$\mid$	$\neg\phi$	negation
	$\mid$	$\phi \wedge \phi$	conjunction
	$\mid$	$\langle a \rangle \phi$	possibility

plus derived logical operators:

false	$\equiv$	$\neg$true	basic truth value
$\phi_1 \vee \phi_2$	$\equiv$	$\neg(\neg\phi_1 \wedge \neg\phi_2)$	disjunction
$[a]\phi$	$\equiv$	$\neg\langle a \rangle \neg\phi$	necessity

- Interpretation of $\mathcal{ML}_{\mathrm{B}}$ over $\mathbb{P}$:

P	$\models_{\mathrm{B}}$	true	
P	$\models_{\mathrm{B}}$	$\neg\phi$	if $P \not\models_{\mathrm{B}} \phi$
P	$\models_{\mathrm{B}}$	$\phi_1 \wedge \phi_2$	if $P \models_{\mathrm{B}} \phi_1$ and $P \models_{\mathrm{B}} \phi_2$
P	$\models_{\mathrm{B}}$	$\langle a \rangle \phi$	if there exists $P' \in \mathbb{P}$ such that $P \xrightarrow{a} P'$ and $P' \models_{\mathrm{B}} \phi$

plus derived logical operators:

P	$\not\models_{\mathrm{B}}$	false	
P	$\models_{\mathrm{B}}$	$\phi_1 \vee \phi_2$	if $P \models_{\mathrm{B}} \phi_1$ or $P \models_{\mathrm{B}} \phi_2$
P	$\models_{\mathrm{B}}$	$[a]\phi$	if for all $P' \in \mathbb{P}$, whenever $P \xrightarrow{a} P'$, then $P' \models_{\mathrm{B}} \phi$

- For all $P_1, P_2 \in \mathbb{P}$:

$$P_1 \sim_{\mathrm{B}} P_2 \iff (\forall \phi \in \mathcal{ML}_{\mathrm{B}}.\, P_1 \models_{\mathrm{B}} \phi \iff P_2 \models_{\mathrm{B}} \phi)$$

- $\sim_{\mathrm{B}}$ is decidable in polynomial time over the set $\mathbb{P}_{\mathrm{fin}}$ of finite-state process terms of $\mathbb{P}$ with Paige-Tarjan partition refinement algorithm.
- Based on the fact that $\sim_{\mathrm{B}}$ can be characterized as the limit of a sequence of successively finer equivalence relations:

$$\boxed{\sim_{\mathrm{B}} = \bigcap_{i \in \mathbb{N}} \sim_{\mathrm{B},i}}$$

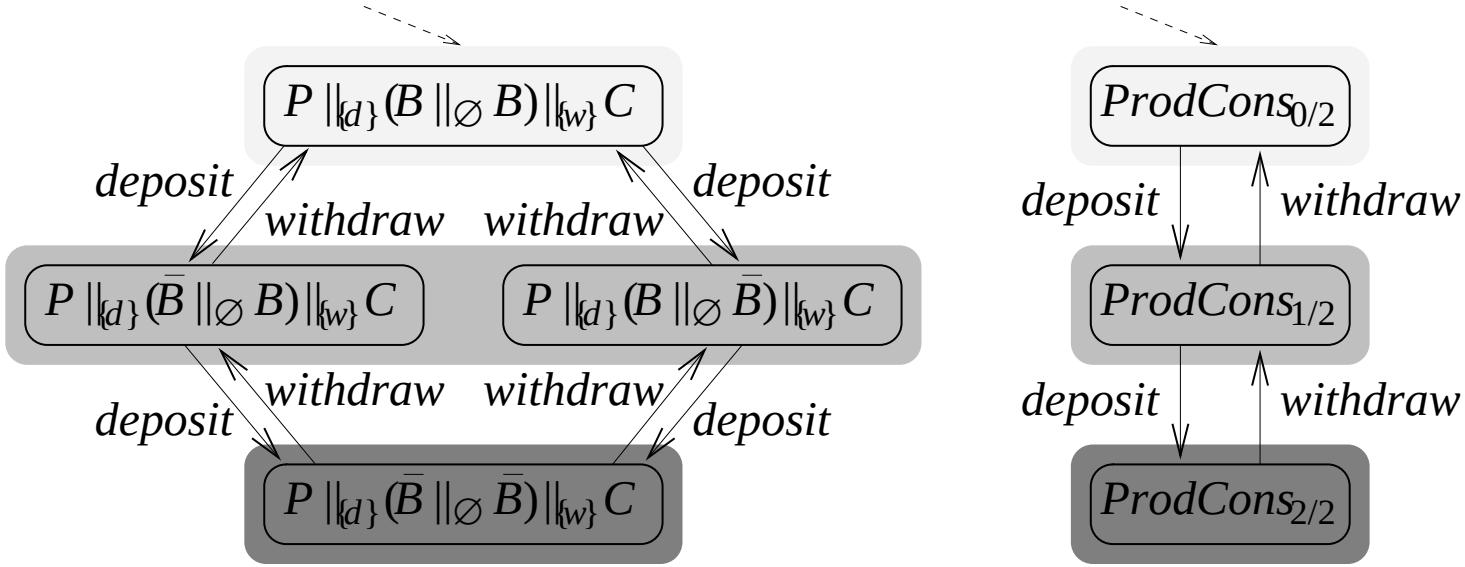
- $\sim_{\mathrm{B},0} = \mathbb{P} \times \mathbb{P}$ hence it induces the trivial partition $\{\mathbb{P}\}$.
- Whenever $P_1 \sim_{\mathrm{B},i} P_2$, $i \in \mathbb{N}_{\geq 1}$, then for all actions $a \in \mathit{Name}$:
  - ⊙ Whenever $P_1 \xrightarrow{a} P_1'$, then $P_2 \xrightarrow{a} P_2'$ with $P_1' \sim_{\mathrm{B},i-1} P_2'$.
  - ⊙ Whenever $P_2 \xrightarrow{a} P_2'$, then $P_1 \xrightarrow{a} P_1'$ with $P_1' \sim_{\mathrm{B},i-1} P_2'$.
- $\sim_{\mathrm{B},1}$ refines $\{\mathbb{P}\}$ by creating an equivalence class for each set of process terms that satisfy the necessary condition for $\sim_{\mathrm{B}}$.

- Steps of the algorithm for checking whether $P_1 \sim_{\mathrm{B}} P_2$:
  1. Build an initial partition with a single class including all the states of $[\![P_1]\!]$ and all the states of $[\![P_2]\!]$.
  2. Initialize a list of splitters with the above class as its only element.
  3. While the list of splitters is not empty, select a splitter and remove it from the list after refining the current partition for all $a \in Name_{P_1,P_2}$:
     a. Split each class of the current partition by comparing its states when executing actions of name a that lead to the selected splitter.
     b. For each class that has been split, insert its smallest subclass into the list of splitters.
  4. Return yes/no depending on whether the initial states of $[\![P_1]\!]$ and $[\![P_2]\!]$ belong to the same class of the final partition or not.
- The time complexity is $O(m \cdot \log n)$, where n is the number of states and m is the number of transitions of $[\![P_1]\!]$ and $[\![P_2]\!]$ (also for minimization).

- **Running example** (bisimulation equivalence):
  - ⊙ Concurrent implementation (with two independent one-position buffers):

$$
\begin{aligned}
PC_{\mathrm{conc},2} &\triangleq Prod \parallel_{\{deposit\}} (Buff \parallel_{\emptyset} Buff) \parallel_{\{withdraw\}} Cons \\
Prod &\triangleq deposit \,.\, Prod \\
Buff &\triangleq deposit \,.\, withdraw \,.\, Buff \\
Cons &\triangleq withdraw \,.\, Cons
\end{aligned}
$$

  - ⊙ Bisimulation proving $PC_{\mathrm{conc},2} \sim_{\mathrm{B}} ProdCons_{0/2}$:



Weak Bisimulation Equivalence

- $\sim_{\mathrm{B}}$ does not abstract from invisible actions (e.g., $a \,.\, b \,.\, \underline{0} \not\sim_{\mathrm{B}} a \,.\, \tau \,.\, b \,.\, \underline{0}$).
- Two process terms should be considered equivalent in the bisimulation approach if they are able to mimic each other's *visible* behavior stepwise.
- Extending the transition relation $\longrightarrow$ to action sequences.
- $P \stackrel{a_1 \ldots a_n}{\Longrightarrow} P'$ iff:
  - either $n = 0$ and $P \equiv P'$, meaning that P stays idle;
  - or $n \in \mathbb{N}_{\geq 1}$ and there exist $P_0, P_1, \ldots, P_n \in \mathbb{P}$ such that:
    - $P \equiv P_0$;
    - $P_{i-1} \stackrel{a_i}{\longrightarrow} P_i$ for all $1 \leq i \leq n$;
    - $P_n \equiv P'$.

- A binary relation $\mathcal{B}$ over $\mathbb{P}$ is a weak bisimulation iff, whenever $(P_1, P_2) \in \mathcal{B}$, then:
  - ⊙ Whenever $P_1 \xrightarrow{\tau} P_1'$, then $P_2 \stackrel{\tau^*}{\Longrightarrow} P_2'$ with $(P_1', P_2') \in \mathcal{B}$.
  - ⊙ Whenever $P_2 \xrightarrow{\tau} P_2'$, then $P_1 \stackrel{\tau^*}{\Longrightarrow} P_1'$ with $(P_1', P_2') \in \mathcal{B}$.

  and for all visible actions $a \in Name_{\mathrm{v}}$:
  - ⊙ Whenever $P_1 \xrightarrow{a} P_1'$, then $P_2 \stackrel{\tau^* a \tau^*}{\Longrightarrow} P_2'$ with $(P_1', P_2') \in \mathcal{B}$.
  - ⊙ Whenever $P_2 \xrightarrow{a} P_2'$, then $P_1 \stackrel{\tau^* a \tau^*}{\Longrightarrow} P_1'$ with $(P_1', P_2') \in \mathcal{B}$.
- Weak bisimulation equivalence $\approx_{\mathrm{B}}$ is the union of all the weak bisimulations.

- $\approx_{\mathrm{B}}$ is a congruence with respect to all the behavioral operators except for alternative composition (not a problem in practice).
- Additional τ-laws highlighting abstraction capabilities:

$$\begin{aligned} \tau \,.\, P &= P \\ a \,.\, \tau \,.\, P &= a \,.\, P \\ P + \tau \,.\, P &= \tau \,.\, P \\ a \,.\, (P_1 + \tau \,.\, P_2) + a \,.\, P_2 &= a \,.\, (P_1 + \tau \,.\, P_2) \end{aligned}$$

- Weak modal operators replacing those of HML ($a \in Name_{\mathrm{v}}$):

$$\begin{array}{lll} P \models_{\mathrm{B}} \langle\!\langle \tau \rangle\!\rangle \phi & \text{if there exists } P' \in \mathbb{P} \text{ such that } P \stackrel{\tau^*}{\Longrightarrow} P' \text{ and } P' \models_{\mathrm{B}} \phi \\ P \models_{\mathrm{B}} \langle\!\langle a \rangle\!\rangle \phi & \text{if there exists } P' \in \mathbb{P} \text{ such that } P \stackrel{\tau^* a \tau^*}{\Longrightarrow} P' \text{ and } P' \models_{\mathrm{B}} \phi \\ P \models_{\mathrm{B}} [[\tau]] \phi & \text{if for all } P' \in \mathbb{P}, \text{ whenever } P \stackrel{\tau^*}{\Longrightarrow} P', \text{ then } P' \models_{\mathrm{B}} \phi \\ P \models_{\mathrm{B}} [[a]] \phi & \text{if for all } P' \in \mathbb{P}, \text{ whenever } P \stackrel{\tau^* a \tau^*}{\Longrightarrow} P', \text{ then } P' \models_{\mathrm{B}} \phi \end{array}$$

- $P_1 \approx_{\mathrm{B}} P_2$ can be decided in $O(n^2 \cdot m \cdot \log n)$ time with the verification algorithm for $\sim_{\mathrm{B}}$ if preceded by the following preprocessing step:
  0. Build the reflexive and transitive closure of $\xrightarrow{\tau}$ in $[\![P_i]\!]$ for $i = 1, 2$:
     a. Add a looping τ-transition to each state.
     b. Add a τ-transition between the initial state and the final state of any sequence of at least two τ-transitions, if the two states are distinct and all the transitions in the sequence are distinct and nonlooping.
     c. Add an a-transition, $a \in \mathit{Name}_{\mathrm{v}}$, between the initial state and the final state of any sequence of at least two transitions in which one is labeled with a, if all the other transitions in the sequence are labeled with τ, distinct, and nonlooping.

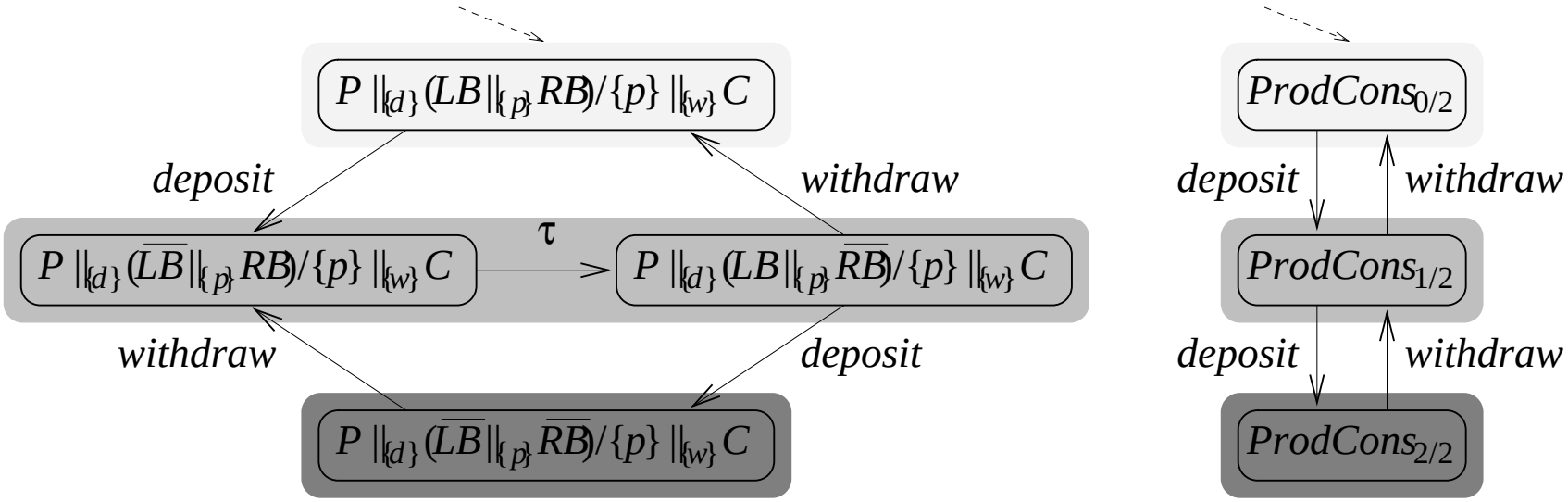
- The fact that $\approx_{\mathrm{B}}$ is not a congruence with respect to the alternative composition operator stems from $\tau \,.\, P = P$ (abstraction from initial τ-actions).
- This τ-law cannot be freely used in all contexts when P is stable (e.g., $\tau \,.\, a \,.\, \underline{0} \approx_{\mathrm{B}} a \,.\, \underline{0}$ but $\tau \,.\, a \,.\, \underline{0} + b \,.\, \underline{0} \not\approx_{\mathrm{B}} a \,.\, \underline{0} + b \,.\, \underline{0}$).
- Congruence w.r.t. the alternative composition operator can be restored by enforcing a matching on initial τ-actions in the equivalence definition.
- $P_1 \in \mathbb{P}$ is weakly bisimulation congruent to $P_2 \in \mathbb{P}$, written $P_1 \approx_{\mathrm{B}}^{\mathrm{c}} P_2$, iff for all actions $a \in \mathit{Name}$:
  - Whenever $P_1 \xrightarrow{a} P_1'$, then $P_2 \stackrel{\tau^* a \tau^*}{\Longrightarrow} P_2'$ with $P_1' \approx_{\mathrm{B}} P_2'$.
  - Whenever $P_2 \xrightarrow{a} P_2'$, then $P_1 \stackrel{\tau^* a \tau^*}{\Longrightarrow} P_1'$ with $P_1' \approx_{\mathrm{B}} P_2'$.
- $\approx_{\mathrm{B}}^{\mathrm{c}}$ is the largest congruence contained in $\approx_{\mathrm{B}}$.

- $\approx_{\mathrm{B}}$ does not fully retain the property possessed by $\sim_{\mathrm{B}}$ of respecting the branching structure of process terms.
- Given $P_1 \approx_{\mathrm{B}} P_2$, when $P_1 \stackrel{a}{\longrightarrow} P_1'$ then $P_2 \stackrel{\tau^*}{\Longrightarrow} Q \stackrel{a}{\longrightarrow} Q' \stackrel{\tau^*}{\Longrightarrow} P_2'$ with $P_1' \approx_{\mathrm{B}} P_2'$ ...
- ... but we do not know whether any relation exists between P_1 and Q and between P_1' and Q'.
- The property can be restored by requiring that P_1 be equivalent to Q and that P_1' be equivalent to Q'.
- The resulting equivalence is branching bisimulation equivalence ($\approx_{\mathrm{B,b}}$).
- Characterized by a single τ-law: $a\,.\,(\tau\,.\,(P_1 + P_2) + P_1) = a\,.\,(P_1 + P_2)$.
- $\approx_{\mathrm{B,b}}$ coincides with $\approx_{\mathrm{B}}$ on any pair of process terms with at most one of them reaching unstable process terms.

- **Running example** (weak bisimulation equivalence):
  - ⊙ Pipeline implementation (with two communicating one-position buffers):

$$
\begin{aligned}
PC_{\text{pipe},2} &\triangleq Prod \parallel_{\{deposit\}} (LBuff \parallel_{\{pass\}} RBuff) / \{pass\} \parallel_{\{withdraw\}} Cons \\
Prod &\triangleq deposit \,.\, Prod \\
LBuff &\triangleq deposit \,.\, pass \,.\, LBuff \\
RBuff &\triangleq pass \,.\, withdraw \,.\, RBuff \\
Cons &\triangleq withdraw \,.\, Cons
\end{aligned}
$$

  - ⊙ Weak bisimulation proving $PC_{\text{pipe},2} \approx_{\text{B}} ProdCons_{0/2}$:



Strengths of PA

+ System modeling is compositional thanks to a restricted set of powerful operators for building larger descriptions up from smaller ones.
+ Operational formal semantics that, for each process term, exactly defines the state transition graph that the process term stands for.
+ Syntax-level and semantics-level compositional reasoning through behavioral equivalences possibly abstracting from certain details.
+ Extensions dealing with several aspects like mobility, dependability, real-time constraints, quality of service, ...

Weaknesses of PA

- PA is rarely adopted in the practice of software development despite its precise syntax and semantics as well as automated analysis techniques.
- Its technicalities obfuscate the way in which systems are modeled

  (given a process term comprising numerous occurrences of the parallel composition operator, it is hard to understand the communication scheme among the various subterms).
- It is perceived as being difficult to learn and use by practitioners as it is not close enough to the way they think of software systems

  (PA supports compositionality and abstraction but does not support object orientation and component orientation, hence it cannot compete, e.g., with UML although it is more robust).

Integrating PA inside Software Architecture Design

- How to enhance the usability of PA?
- Need to support a friendly component-oriented way of modeling systems with PA $\Longrightarrow$ *the designer can reason in terms of composable software units without having to worry about PA technicalities.*
- Need to provide an efficient component-oriented way of analyzing functional and nonfunctional properties of systems modeled with PA returning component-oriented diagnostic information when violations are detected $\Longrightarrow$ *the designer can pinpoint the sources of error.*
- Need to integrate the use of PA in the software development process. (Requirement analysis? Architectural design? Design/selection and integration of components? Deployment? Testing? Maintenance?)

- Working at the sw architecture level, which is the *right abstraction level for PA* as implicitly demonstrated in Allen & Garlan (1997).
- By analogy to building architecture, a model of software architecture can be defined as consisting of elements, form, and rationale.
- In the software context, architectural elements are:
  - Components, which are computation units or data stores.
  - Connectors, handling component communication and coordination.
- The form is expressed through properties and relationships that constrain the choice and the placement of architectural elements.
- The rationale is the basis for the architecture in terms of the system requirements and provides motivations for the various choices made in defining the architecture.

- De Remer & Kron (1976): *"... structuring a large collection of modules to form a system is an essentially different intellectual activity from that of constructing the individual modules. That is, we distinguish programming-in-the-large from programming-in-the-small ..."*.
- Perry & Wolf (1992): *"... architecture is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions that provide a framework in which to satisfy the requirements and serve as a basis for the design ..."*.
- Shaw & Garlan (1996): *"... organization of a system as a composition of components; global control structures; protocols for communication, synchronization and data access; assignment of functionality to design elements; composition of design elements; physical distribution; scaling and performance; dimensions of evolution; and selection among design alternatives. This is the software architecture level of design ..."*.

- The software architecture design phase can benefit from the use of PA.
- Focus *not* on algorithms and data structures (programming languages), but on software components and connectors (modeling notations).
- Objective: document describing system structure and behavior at high abstraction level, to be shared by all the people who contribute to the various phases of the software development process.
- A precise architectural description language (ADL) has no ambiguity and opens the way to system property analyzability in the early stages.
- Good for *time-to-market constraints* & *development/maintenance costs*.
- How to transform PA into an ADL?
- How to drive the whole development process with the resulting PADL?

Part I:
Component-Oriented Modeling

Architectural Upgrade of Process Algebra: Transformation Guidelines

1. Separation of the behavior description from the topology description.
2. Reuse of the specification of components and connectors.
3. Elicitation of the interface of components and connectors.
4. Classification of the synchronicity of communications.
5. Classification of the multiplicity of communications.
6. Combination of textual and graphical notations.
7. Revision of dynamic operators and concealment of static operators.
8. Provision of support for architectural styles.

Running Example: Client-Server System

- General description:
  - A possibly replicated server provides a set of predefined services.
  - Clients contact the server to request some of the available services.
  - The server provides the requested services to the requesting clients according to some predefined discipline.
- Specific scenario:
  - Single replica of the server.
  - Two identically behaving clients.
  - No buffer for holding incoming requests on the server side.
  - After a request, a client cannot proceed until it receives a response.

G1: Separating Behavior/Topology Descriptions

- Hard to understand which process terms communicate with each other in a system description with several process terms composed in parallel.
- Selecting the appropriate synchronization sets and the appropriate order for the various terms is not trivial at all.
- The problem arises from the fact that the parallel composition operator is not simply a behavioral operator, but encodes both part of the system behavior and the entire system topology.
- Two distinct sections are necessary in every architectural description:
  - Architectural behavior section.
  - Architectural topology section.

G2: Reusing Component/Connector Specification

- There may be several process terms composed in parallel in a system description that differ only for the names of some of their actions or process constants.
- Recognize the presence of repeated behavioral patterns in order to avoid specification redundancy and hence to reduce the modeling time.
- The architectural behavior section contains the definition of as many architectural element types (AETs) as there are types of components and types of connectors.
- The architectural topology section contains the declaration of as many architectural element instances (AEIs) of any previously defined AET as there are components and connectors of that type in the system (process terms exhibiting the same behavioral pattern).

- **Running example** (G1/G2):
  - AETs defined in the architectural behavior section:
    - One for the server (`Server_Type`).
    - One for both clients (`Client_Type`) due to their identical behavior.
  - AEIs declared in the architectural topology section:
    - One of server type (`S`).
    - Two of client type (`C_1`, `C_2`).
  - Each of the two client AEIs can communicate with the server AEI.

G3: Eliciting Component/Connector Interface

- The actions occurring in a process term do not play the same role from the communication viewpoint.
- Distinguish explicitly between:
  - internal actions forming the component/connector implementation;
  - interactions constituting the component/connector interface:
    - input vs. output within AET definition;
    - local vs. architectural within architectural topology section.
- Static checks on the declaration of attachments between interactions:
  - *Any local interaction involved at least in one attachment, whereas no internal action/architectural interaction involved in attachments.*
  - *Every attachment from a local output interaction of an AEI to a local input interaction of another AEI* (*names of attached interactions can be different*).
  - *Connected topology or groups of isolated AEIs.*

G4: Classifying Communication Synchronicity

- The interactions occurring in a process term are not necessarily involved in communications with the same synchronicity.
- Distinguish explicitly in any AET definition among:
  - ⊙ synchronous interactions (blocking);
  - ⊙ semi-synchronous interactions (exception if other not ready, so as not to block);
  - ⊙ asynchronous interactions (decoupling start from end).
- Boolean variable `success` associated with semi-synchronous interactions for catching exceptions within the AET behavior.
- A local output synchronous, semi-synch., or asynchronous interaction can be freely attached to a local input synchronous, semi-synch., or asynchronous interaction (all nine two-by-two combinations are permitted).

G5: Classifying Communication Multiplicity

- The interactions occurring in a process term are not necessarily involved in communications with the same multiplicity.
- Distinguish explicitly in any AET definition among:
  - ⊙ uni-interactions mainly involved in one-to-one communications;
  - ⊙ and-interactions guiding inclusive one-to-many communications;
  - ⊙ or-interactions guiding selective one-to-many communications (an or-dep. ensures that an output is sent to the same AEI that issued the last corr. input).
- Static checks:
  - ⊙ *Any local uni-interaction attached to only one local interaction.*
  - ⊙ *Any local and-/or-interaction attached only to local uni-interactions each belonging to a different AEI.*
  - ⊙ *No output or-interaction occurring before the input or-interaction it depends on, and both of them attached to the same AEIs if local.*

- **Running example** (G3/G4/G5):
  - The server AET has:
    - One input interaction for receiving requests from the clients (`receive_request`).
    - One internal action modeling the computation of responses (`compute_response`).
    - One output interaction for sending responses to the clients (`send_response`).
  - The client AET has:
    - One internal action modeling the processing of tasks (`process`).
    - One output interaction for sending requests to the server (`send_request`).
    - One input interaction for receiving responses from the server (`receive_response`).

- All the interactions of the server AET are or-interactions because the server AEI is attached to both client AEIs but can communicate with only one of them at a time:
  - Dependence between `send_response` and `receive_request` since the responses computed by the server AEI have to be sent back to the client AEIs that issued the corresponding requests.
  - `receive_request` is synchronous because the server AEI stays idle as long as it does not receive requests from the client AEIs.
  - `send_response` is asynchronous so that the server AEI can proceed with further requests without being blocked by the unavailability of the client AEI that should receive the response.

- All the interactions of the client AET are uni-interactions because each of the client AEIs is attached to the only server AEI:
  - `send_request` is semi-synchronous so that a client AEI wishing to send a request when the server AEI is busy can keep working instead of passively waiting for the server AEI to become available.
  - `receive_response` is synchronous because after issuing a request a client AEI cannot proceed until it receives a response from the server AEI.
- All the interactions of the three AEIs are local:
  - Both `send_request` interactions of the two client AEIs are attached to the `receive_request` interaction of the server AEI.
  - The `send_response` interaction of the server AEI is attached to both `receive_response` interactions of the two client AEIs.

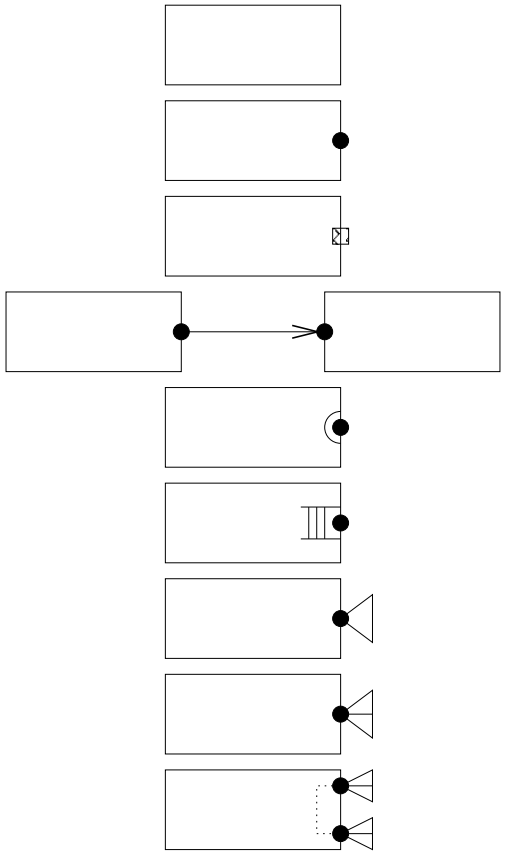
G6: Combining Textual and Graphical Notations (PADL Syntax)

- Process algebra provides just a textual notation, which may not be enough for the software designer.
- Not only a matter of introducing some architectural syntactic sugar on the basis of the previous guidelines.
- Make available also a graphical notation for providing a visual aid.
- Permit an integrated use of the two notations:
  - The graphical one for representing the system topology.
  - The textual one for describing the architectural element behavior.

- Structure of a PADL textual description (dot notation for AEI interactions):

```
ARCHI_TYPE                    ◁name and initialized formal data parameters▷
  ARCHI_BEHAVIOR
        ⋮                           ⋮
    ARCHI_ELEM_TYPE           ◁AET name and formal data parameters▷
      BEHAVIOR                ◁sequence of process algebraic equations
                                built from verbose dynamic operators only▷
      INPUT_INTERACTIONS      ◁input synchronous/semi-sync./asynchronous
                                uni/and/or-interactions▷
      OUTPUT_INTERACTIONS     ◁output synchronous/semi-sync./asynchronous
                                uni/and/or-interactions▷
        ⋮                           ⋮
  ARCHI_TOPOLOGY
    ARCHI_ELEM_INSTANCES      ◁AEI names and actual data parameters▷
    ARCHI_INTERACTIONS        ◁architecture-level AEI interactions▷
    ARCHI_ATTACHMENTS         ◁attachments between AEI local interactions▷
END
```

- Graphical notation of PADL based on enriched flow graphs:



AEI

local (synchronous uni-)interaction

architectural (synchronous uni-)interaction

attachment between local (synchronous uni-)interactions

(local) semi-synchronous (uni-)interaction

(local) asynchronous (uni-)interaction

(local synchronous) and-interaction

(local synchronous) or-interaction

dependence between (local synchronous) or-interactions

G7: Revising Dynamic Operators and Concealing Static Operators

- Static operators ($\|$ / \ [ ]) are harder to use than dynamic ones (. +).
- Conceal static operators and make available only dynamic operators for describing the behavior of AETs.
- Make dynamic operators more verbose: $\underline{0}$ becomes `stop`.
- Avoid semantic overloading of dynamic operators: + becomes `choice`.
- Enrich actions with boolean guards and the capability of exchanging data in communications (i/o actions of value passing process algebra).
- Data types: `boolean`, `integer`, `real`, `list`, `array`, `record`, `object`.

- Revised syntax for a PA defining equation (both lists can be empty):

$$B(\mathit{formal\_data\_parameter\_list};\ \mathit{data\_variable\_list}) = P$$

where (cond guard is optional):

P	::=	`stop`	inactive process
	|	$B(\mathit{actual\_data\_parameter\_list})$	process constant
	|	`cond`$(\mathit{bool\_expr})$ `->` $a\,.\,P$	action prefix
	|	`cond`$(\mathit{bool\_expr})$ `->` $a?(\mathit{var\_list})\,.\,P$	input action prefix
	|	`cond`$(\mathit{bool\_expr})$ `->` $a!(\mathit{expr\_list})\,.\,P$	output action prefix
	|	`choice`$\{P, \ldots, P\}$	alternative composition

- Formal data pars of the first defining equation have to be initialized (by possibly making use of the formal data parameters of the AET).
- Data vars are necessary for storing values received via input actions.

- Use *implicitly* static operators different from parallel composition for declaring behavioral modifications that may be useful for verifying certain properties (optional third section of a textual description; dot notation):

```
BEHAV_MODIFICATIONS
  BEHAV_HIDINGS          ◁names of actions to be hidden▷
  BEHAV_RESTRICTIONS     ◁names of actions to be restricted▷
  BEHAV_RENAMINGS        ◁names of actions to be changed▷
```

- Use *transparently* static operators for defining PADL semantics:
  - ⊙ Parallel composition for making AEIs communicate with each other.
  - ⊙ Relabeling for attached local interactions having different names.

- **Running example** (G6/G7):
  - Header of the textual description:

```
ARCHI_TYPE Client_Server(void)
```

  - Definition of the server AET:

```
ARCHI_ELEM_TYPE Server_Type(void)

  BEHAVIOR
    Server(void; void) =
      receive_request . compute_response . send_response . Server()

  INPUT_INTERACTIONS  SYNC  OR receive_request
  OUTPUT_INTERACTIONS ASYNC OR send_response   DEP receive_request
```

- Definition of the client AET:

```
ARCHI_ELEM_TYPE Client_Type(void)

  BEHAVIOR
    Client_Internal(void; void) =
      process . Client_Interacting();
    Client_Interacting(void; void) =
      send_request .
        choice
        {
          cond(send_request.success = true) ->
                                  receive_response . Client_Internal(),
          cond(send_request.success = false) ->
                                  keep_processing . Client_Interacting()
        }

  INPUT_INTERACTIONS  SYNC  UNI receive_response
  OUTPUT_INTERACTIONS SSYNC UNI send_request
```

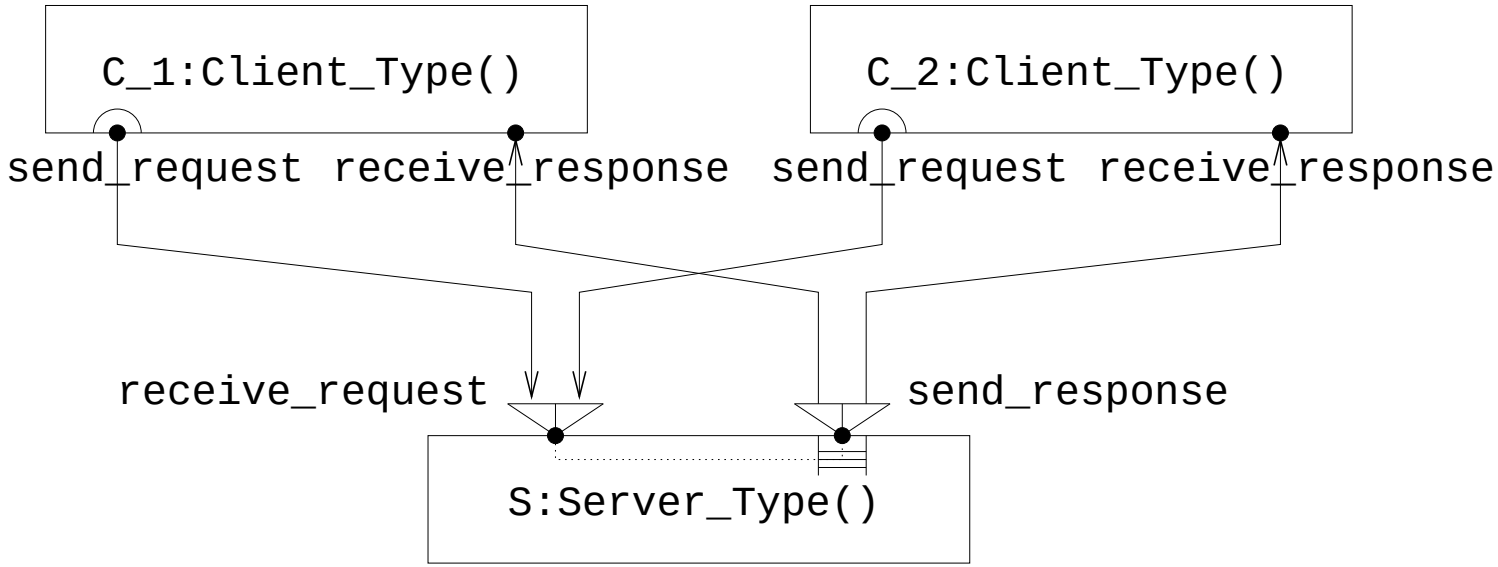
- Declaration of the topology:

```
ARCHI_ELEM_INSTANCES
  S   : Server_Type();
  C_1 : Client_Type();
  C_2 : Client_Type()

ARCHI_INTERACTIONS
  void

ARCHI_ATTACHMENTS
  FROM C_1.send_request TO S.receive_request;
  FROM C_2.send_request TO S.receive_request;
  FROM S.send_response  TO C_1.receive_response;
  FROM S.send_response  TO C_2.receive_response
```

- Enriched flow graph:



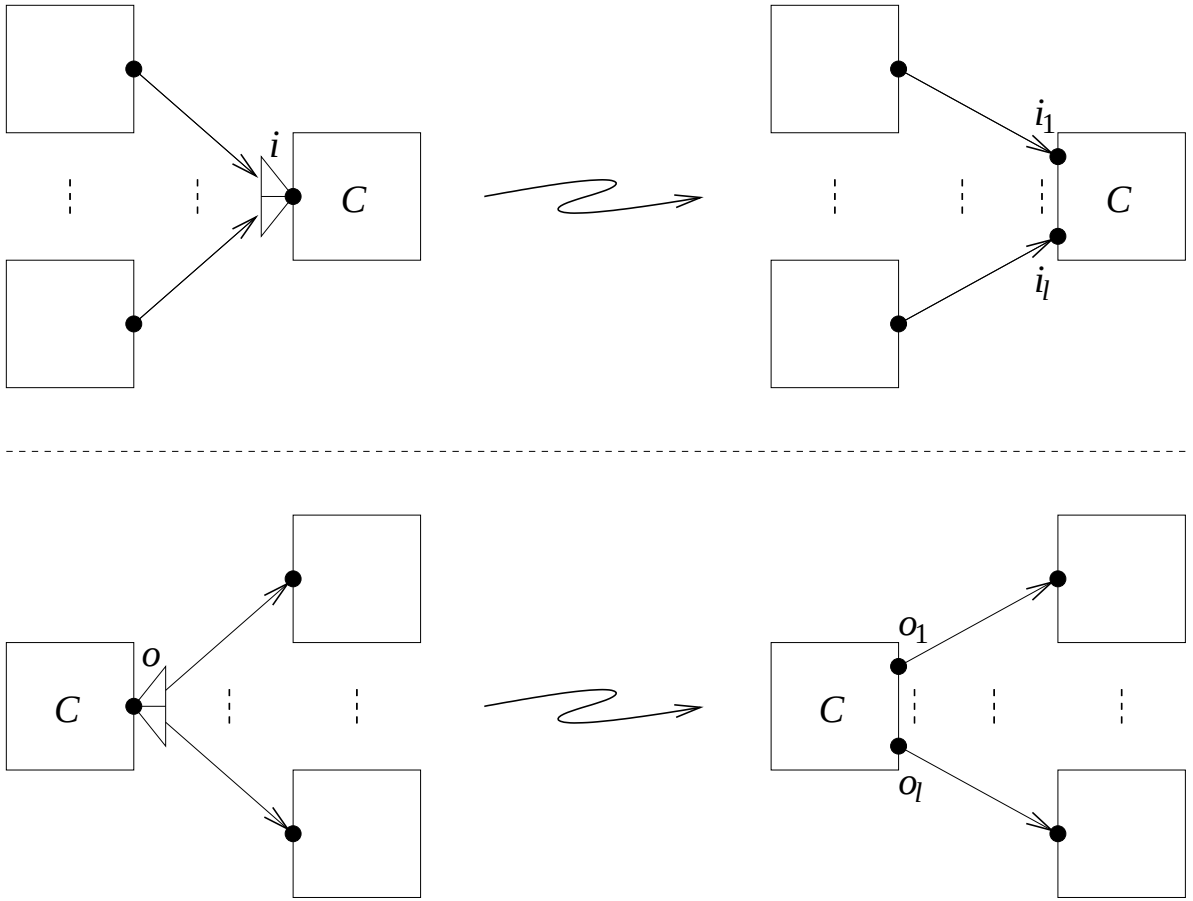
Translation Semantics for PADL

- Every PADL description is translated into a PA specification.
- *First step:*
  - The semantics of any AEI is the behavior of the corresponding AET.
  - Action occurrences preceded by the name of the AEI.
  - AET formal data parameters substituted for by AEI actual ones.
  - Local or-interactions replaced by sets of fresh local uni-interactions.
  - Additional implicit AEIs for handling local async. interactions.
- *Second step:*
  - Parallel composition of the semantics of all the AEIs.
  - Local interactions renamed on the basis of the attachments.
  - Synchronization sets determined by the new local interaction names.
  - Additional semantic rules for handling exceptions (due to local semi-sync.).
  - Hiding, restriction, and relabeling for behavioral modifications.

Semantics of Individual Elements

- Let $\mathcal{C}$ be an AET with:
  - $m \in \mathbb{N}_{\geq 0}$ formal data parameters $fp_1, \ldots, fp_m$.
  - Behavior given by a sequence $\mathcal{E}$ of PA defining equations obeying the revised syntax.
- Let C be an AEI of type $\mathcal{C}$ with:
  - $m \in \mathbb{N}_{\geq 0}$ actual data parameters $ap_1, \ldots, ap_m$ consistent by order and type with the formal data parameters.
- The kernel of the semantics of C is given by $C.\mathcal{E}$, in which every action name a becomes $C.a$.
- Then every occurrence of fp_j is substituted for by ap_j $(1 \leq j \leq m)$.
- However, every local or-interaction and local asynchronous interaction of C requires a specific treatment.

- Replace each local or-interaction of C with fresh local uni-interactions and attach them to the local uni-interactions of other AEIs to which the local or-interaction was originally attached:



- Rewrite each local or-interaction of C into its fresh local uni-interactions within the right-hand side of any PA defining equation of C.
- Only if the number $\mathit{attach\text{-}no}(\_)$ of attachments that involve the local or-interaction is greater than 1.
- Or-deps managed by keeping track of the set $\mathit{FI} \subseteq \mathit{Name}$ (initially $\emptyset$) of fresh local input uni-interactions currently in force arising from local input or-interactions on which some local output or-interaction depends.
- Function $\mathit{or\text{-}rewrite}(\_)$ defined by structural induction on the syntactical structure of the right-hand side of a set of PA defining equations.

- If a is an or-interaction such that $\mathit{attach\text{-}no}(C.a) \leq 1$ (if 0 then architectural) or a uni-/and-interaction or an internal action:

$$\boxed{\mathit{or\text{-}rewrite}_{FI}(a\,.\,P) \;=\; a\,.\,\mathit{or\text{-}rewrite}_{FI}(P)}$$

- If a is an input or-interaction on which no output or-interaction depends or an output or-interaction not depending on any input or-interaction and $\mathit{attach\text{-}no}(C.a) = l \geq 2$:

$$\boxed{\begin{array}{rcl}\mathit{or\text{-}rewrite}_{FI}(a\,.\,P) & = & \texttt{choice}\{a_1\,.\,\mathit{or\text{-}rewrite}_{FI}(P),\\ & & \vdots\\ & & a_l\,.\,\mathit{or\text{-}rewrite}_{FI}(P)\}\end{array}}$$

- If i is an input or-interaction on which an output or-interaction depends and $\mathit{attach\text{-}no}(C.i) = l \geq 2$:

$$\mathit{or\text{-}rewrite}_{FI}(i\,.\,P) \;=\; \mathtt{choice}\{i_1\,.\,\mathit{or\text{-}rewrite}_{FI\,-\,\{i_j\,|\,1\leq j\leq l\}\,\cup\,\{i_1\}}(P),\; \vdots\; i_l\,.\,\mathit{or\text{-}rewrite}_{FI\,-\,\{i_j\,|\,1\leq j\leq l\}\,\cup\,\{i_l\}}(P)\}$$

- If o is an output or-interaction depending on the input or-interaction i and $\mathit{attach\text{-}no}(C.i) = \mathit{attach\text{-}no}(C.o) \geq 2$ and $i_j \in FI$:

$$\mathit{or\text{-}rewrite}_{FI}(o\,.\,P) \;=\; o_j\,.\,\mathit{or\text{-}rewrite}_{FI}(P)$$

- Rewriting process for the remaining operators:

$$
\begin{aligned}
\mathit{or\text{-}rewrite}_{FI}(\texttt{stop}) &= \texttt{stop} \\
\mathit{or\text{-}rewrite}_{FI}(B(\mathit{actual\_data\_par\_list})) &= B_{FI}(\mathit{actual\_data\_par\_list}) \\
\mathit{or\text{-}rewrite}_{FI}(\texttt{choice}\{P_1, \ldots, P_n\}) &= \texttt{choice}\{\mathit{or\text{-}rewrite}_{FI}(P_1), \\
&\qquad \vdots \\
&\qquad \mathit{or\text{-}rewrite}_{FI}(P_n)\}
\end{aligned}
$$

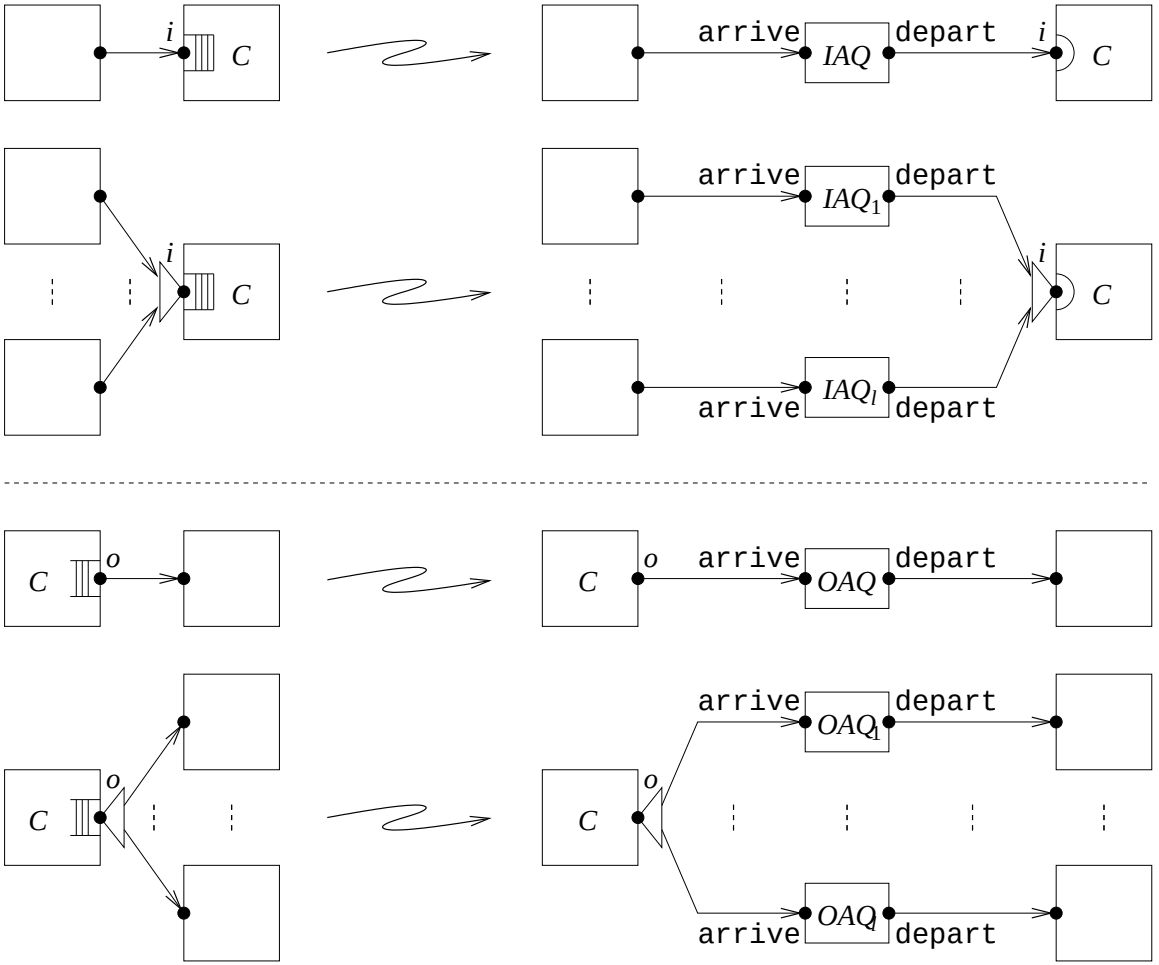
  where $B_{FI} \equiv B$ for $FI = \emptyset$.

- For $FI \neq \emptyset$:

$$
B_{FI}(\mathit{formal\_data\_par\_list};\ \mathit{data\_var\_list}) = \mathit{or\text{-}rewrite}_{FI}(P)
$$

  if $B(\mathit{formal\_data\_par\_list};\ \mathit{data\_var\_list}) = P$.

- For each local asynchronous uni-/and-interaction of C (no more or-interactions) add implicit AEIs behaving as unbounded buffers:



- AET for additional implicit input/output asynchronous queues:

```
ARCHI_ELEM_TYPE Async_Queue_Type(void)

  BEHAVIOR
    Queue(int n := 0;
          void) =
      choice
      {
        cond(true)  -> arrive . Queue(n + 1),
        cond(n > 0) -> depart . Queue(n - 1)
      }

  INPUT_INTERACTIONS  SYNC UNI arrive   /* always enabled */
  OUTPUT_INTERACTIONS SYNC UNI depart   /* only if buffer not empty */
```

- Local input asynchronous interactions become semi-synchronous.
- Local output asynchronous interactions become synchronous.

- Assume that C has:
  - $\odot$ $h \in \mathbb{N}_{\geq 0}$ local input asynchronous uni-interactions $i_1, \ldots, i_h$ handled through the related additional implicit AEIs $IAQ_1, \ldots, IAQ_h$.
  - $\odot$ $h' \in \mathbb{N}_{\geq 0}$ local input asynchronous and-interactions $i'_1, \ldots, i'_{h'}$, where each i'_j is handled through the $attach\text{-}no(C.i'_j) = il_j$ related additional implicit AEIs $IAQ_{j,1}, \ldots, IAQ_{j,il_j}$.
  - $\odot$ $k \in \mathbb{N}_{\geq 0}$ local output asynchronous uni-interactions $o_1, \ldots, o_k$ handled through the related additional implicit AEIs $OAQ_1, \ldots, OAQ_k$.
  - $\odot$ $k' \in \mathbb{N}_{\geq 0}$ local output asynchronous and-interactions $o'_1, \ldots, o'_{k'}$, where each o'_j is handled through the $attach\text{-}no(C.o'_j) = ol_j$ related additional implicit AEIs $OAQ_{j,1}, \ldots, OAQ_{j,ol_j}$.
- Isolated semantics of C:

$$[\![C]\!] = o\text{-}and^{k'}_{ol_{k'}}(\ldots o\text{-}and^{1}_{ol_1}(o\text{-}uni_k(i\text{-}and^{h'}_{il_{h'}}(\ldots i\text{-}and^{1}_{il_1}(i\text{-}uni_h(C))\ldots)))\ldots)$$

$$
\begin{array}{rcll}
\textit{i-uni}_0(C) & = & \textit{or-rewrite}_{\emptyset}(C.\mathcal{E}\{ap_1 \hookrightarrow fp_1, \ldots, ap_m \hookrightarrow fp_m\})\,[\varphi_{C,\text{async}}] & \\
\textit{i-uni}_j(C) & = & IAQ_j.\texttt{Queue(0)}\,[\varphi_{C,\text{async}}] & 1 \leq j \leq h \\
 & & \quad \|_{\{IAQ_j.\texttt{depart}\#C.i_j\}}\,(\textit{i-uni}_{j-1}(C)) & \\
\hline
\textit{i-and}_1^j(f(C)) & = & IAQ_{j,1}.\texttt{Queue(0)}\,[\varphi_{C,\text{async}}] & 1 \leq j \leq h' \\
 & & \quad \|_{\{IAQ_{j,1}.\texttt{depart}\#\ldots\#IAQ_{j,il_j}.\texttt{depart}\#C.i'_j\}}\,(f(C)) & \\
\textit{i-and}_{j'}^j(f(C)) & = & IAQ_{j,j'}.\texttt{Queue(0)}\,[\varphi_{C,\text{async}}] & 2 \leq j' \leq il_j \\
 & & \quad \|_{\{IAQ_{j,1}.\texttt{depart}\#\ldots\#IAQ_{j,il_j}.\texttt{depart}\#C.i'_j\}}\,(\textit{i-and}_{j'-1}^j(f(C))) & \\
\hline
\textit{o-uni}_0(f(C)) & = & f(C) & \\
\textit{o-uni}_j(f(C)) & = & (\textit{o-uni}_{j-1}(f(C)))\,\|_{\{C.o_j\#OAQ_j.\texttt{arrive}\}} & \\
 & & \quad OAQ_j.\texttt{Queue(0)}\,[\varphi_{C,\text{async}}] & 1 \leq j \leq k \\
\hline
\textit{o-and}_1^j(f(C)) & = & (f(C))\,\|_{\{C.o'_j\#OAQ_{j,1}.\texttt{arrive}\#\ldots\#OAQ_{j,ol_j}.\texttt{arrive}\}} & \\
 & & \quad OAQ_{j,1}.\texttt{Queue(0)}\,[\varphi_{C,\text{async}}] & 1 \leq j \leq k' \\
\textit{o-and}_{j'}^j(f(C)) & = & (\textit{o-and}_{j'-1}^j(f(C)))\,\|_{\{C.o'_j\#OAQ_{j,1}.\texttt{arrive}\#\ldots\#OAQ_{j,ol_j}.\texttt{arrive}\}} & \\
 & & \quad OAQ_{j,j'}.\texttt{Queue(0)}\,[\varphi_{C,\text{async}}] & 2 \leq j' \leq ol_j
\end{array}
$$

- **Running example** (first step of the translation):
  - ⊙ $[\![$C_1$]\!]$ (resp. $[\![$C_2$]\!]$) coincides with the sequence of PA defining equations of `Client_Type`:

```
Client_Internal(void; void) =
  process . Client_Interacting();
Client_Interacting(void; void) =
  send_request .
    choice
    {
      cond(send_request.success = true) ->
                                receive_response . Client_Internal(),
      cond(send_request.success = false) ->
                                keep_processing . Client_Interacting()
    }
```

    where action names are preceded by `C_1` (resp. `C_2`).
  - ⊙ The reason is that there are no formal data parameters, local or-interactions, and local asynchronous interactions.

- The construction of $[\![\mathtt{S}]\!]$ requires the application of *or-rewrite*:

```
Server'(void; void) =
  choice
  {
    S.receive_request_1 . S.compute_response .
                                          S.send_response_1 . Server'(),
    S.receive_request_2 . S.compute_response .
                                          S.send_response_2 . Server'()
  }
```

- Then it requires two additional implicit AEIs for the local output async. uni-interactions `S.send_response_1` and `S.send_response_2`:

```
Server'[S.send_response_1 ↦ S.send_response_1#OAQ_1.arrive,
        S.send_response_2 ↦ S.send_response_2#OAQ_2.arrive]

  ||{S.send_response_1#OAQ_1.arrive}

OAQ_1.Queue(0)[OAQ_1.arrive ↦ S.send_response_1#OAQ_1.arrive]

  ||{S.send_response_2#OAQ_2.arrive}

OAQ_2.Queue(0)[OAQ_2.arrive ↦ S.send_response_2#OAQ_2.arrive]
```

Semantics of Interacting Elements

- Let $\{C_1, \ldots, C_n\}$ be a set of AEIs and C_j be one of them.
- $\mathcal{LI}_{C_j}$: set of local interactions of C_j after rewriting local or-interactions and adding implicit AEIs for local asynchronous interactions, including:
  - Its original local nonasynchronous uni-/and-interactions.
  - Its fresh local nonasynchronous uni-interactions that replace its original local nonasynchronous or-interactions.
  - The local interactions of its additional implicit AEIs that are not attached to its originally asynchronous local interactions.
- $\mathcal{LI}_{C_j;C_1,\ldots,C_n}$: set of local interactions of C_j attached to $\{C_1, \ldots, C_n\}$.
- Rename the local interactions in each set $\mathcal{LI}_{C_j;C_1,\ldots,C_n}$ in such a way that the process terms representing the semantics of $C_1, \ldots, C_n$ can communicate even if attached interactions have different names.

- $\mathcal{S}(C_1, \ldots, C_n)$: set of fresh action names, one for each pair of attached local uni-interactions $_{\text{within } \{C_1, \ldots, C_n\}}$ and for each local and-interaction and the local uni-interactions attached to it $_{\text{within } \{C_1, \ldots, C_n\}}$.
- Original name concatenation ensures fresh name uniqueness $_{(C_j.o \# C_g.i)}$.
- $\varphi_{C_j;C_1,\ldots,C_n} : \mathcal{LI}_{C_j;C_1,\ldots,C_n} \longrightarrow \mathcal{S}(C_1, \ldots, C_n)$: element of a set of injective relabeling functions such that for $C_j.a_1$ attached to $C_g.a_2$ or to the same and-interaction as $C_g.a_2$:

$$\boxed{\varphi_{C_j;C_1,\ldots,C_n}(C_j.a_1) \;=\; \varphi_{C_g;C_1,\ldots,C_n}(C_g.a_2)}$$

- Interacting semantics of $C_j \in \{C_1, \ldots, C_n\}$ with respect to $\{C_1, \ldots, C_n\}$:

$$\boxed{[\![C_j]\!]_{C_1,\ldots,C_n} \;=\; [\![C_j]\!]\,[\varphi_{C_j;C_1,\ldots,C_n}]}$$

- $\mathcal{S}(C_j; C_1, \ldots, C_n) = \varphi_{C_j;C_1,\ldots,C_n}(\mathcal{LI}_{C_j;C_1,\ldots,C_n})$: individual sync. set of C_j with respect to $\{C_1, \ldots, C_n\}$.
- $\mathcal{S}(C_j, C_g; C_1, \ldots, C_n) = \mathcal{S}(C_j; C_1, \ldots, C_n) \cap \mathcal{S}(C_g; C_1, \ldots, C_n)$: pairwise synchronization set of C_j and C_g with respect to $\{C_1, \ldots, C_n\}$.
- Interacting semantics of $\{C'_1, \ldots, C'_{n'}\} \subseteq \{C_1, \ldots, C_n\}$ with respect to $\{C_1, \ldots, C_n\}$:

$$\begin{aligned}[\![C'_1, \ldots, C'_{n'}]\!]_{C_1,\ldots,C_n} = {} & [\![C'_1]\!]_{C_1,\ldots,C_n} \parallel_{\mathcal{S}(C'_1,C'_2;C_1,\ldots,C_n)} \\ & [\![C'_2]\!]_{C_1,\ldots,C_n} \parallel_{\mathcal{S}(C'_1,C'_3;C_1,\ldots,C_n) \cup \mathcal{S}(C'_2,C'_3;C_1,\ldots,C_n)} \\ & \cdots \parallel_{\bigcup_{j=1}^{n'-1} \mathcal{S}(C'_j,C'_{n'};C_1,\ldots,C_n)} [\![C'_{n'}]\!]_{C_1,\ldots,C_n}\end{aligned}$$

where the unions of pairwise synchronization sets are consistent with the left associativity of the parallel composition operator.

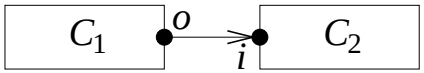
- A local semi-synchronous interaction s executed by an AEI C results in a transition labeled with $C.s$ within $[\![C]\!]$ and hence the related `success` variable is set to true.
- In an interacting context, this transition has to be relabeled as an exception if s cannot immediately participate in a communication.
- Additional semantic rules for context-sensitive relabeling to exceptions

($C_1.o$ attached to $C_2.i$ and P_j representing the current state of the interacting sem. of C_j):

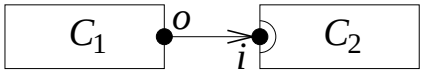
$$\frac{P_1 \xrightarrow{C_1.o\#C_2.i} \!\!\!\!\!\!\diagup \quad P_1' \qquad P_2 \xrightarrow{C_1.o\#C_2.i} P_2' \qquad C_2.i \text{ semi-synchronous}}{P_1 \parallel_S P_2 \xrightarrow{C_2.i\_exception} P_1 \parallel_S P_2' \qquad C_2.i.\texttt{success} = \texttt{false}}$$

$$\frac{P_1 \xrightarrow{C_1.o\#C_2.i} P_1' \qquad P_2 \xrightarrow{C_1.o\#C_2.i} \!\!\!\!\!\!\diagup \quad P_2' \qquad C_1.o \text{ semi-synchronous}}{P_1 \parallel_S P_2 \xrightarrow{C_1.o\_exception} P_1' \parallel_S P_2 \qquad C_1.o.\texttt{success} = \texttt{false}}$$

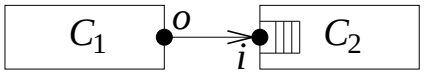
- Summary of the semantic treatment of communication synchronicity:



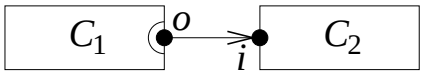
$P_1 \|_{\{C_1.o\#C_2.i\}} P_2$



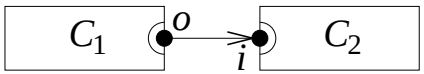
$P_1 \|_{\{C_1.o\#C_2.i\}} P_2$ — $C_2.i\_exception$



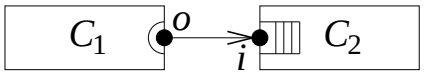
$P_1 \|_{\{C_1.o\#IAQ.\texttt{arrive}\}} (IAQ.\texttt{Queue(0)} \|_{\{IAQ.\texttt{depart}\#C_2.i\}} P_2)$ — $C_2.i\_exception$



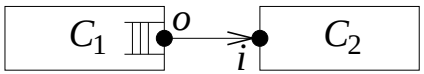
$P_1 \|_{\{C_1.o\#C_2.i\}} P_2$ — $C_1.o\_exception$



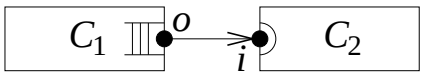
$P_1 \|_{\{C_1.o\#C_2.i\}} P_2$ — $C_1.o\_/C_2.i\_exc.$



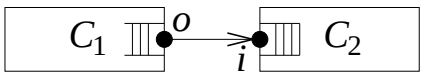
$P_1 \|_{\{C_1.o\#IAQ.\texttt{arrive}\}} (IAQ.\texttt{Queue(0)} \|_{\{IAQ.\texttt{depart}\#C_2.i\}} P_2)$ — $C_1.o\_/C_2.i\_exc.$



$(P_1 \|_{\{C_1.o\#OAQ.\texttt{arrive}\}} OAQ.\texttt{Queue(0)}) \|_{\{OAQ.\texttt{depart}\#C_2.i\}} P_2$



$(P_1 \|_{\{C_1.o\#OAQ.\texttt{arrive}\}} OAQ.\texttt{Queue(0)}) \|_{\{OAQ.\texttt{depart}\#C_2.i\}} P_2$ — $C_2.i\_exception$



$(P_1 \|_{\{C_1.o\#OAQ.\texttt{arrive}\}} OAQ.\texttt{Queue(0)}) \|_{\{OAQ.\texttt{depart}\#IAQ.\texttt{arrive}\}} (IAQ.\texttt{Queue(0)} \|_{\{IAQ.\texttt{depart}\#C_2.i\}} P_2)$ — $C_2.i\_exception$

- Let $\mathcal{A}$ be an architectural description.
- Let $\{C_1, \ldots, C_n\}$ be the set of its AEIs.
- Let $\mathcal{H}$, $\mathcal{R}$, and φ be possible behavioral modifications of $\mathcal{A}$ enforcing action hiding, action restriction, and action renaming, respectively.
- Semantics of $\mathcal{A}$ before behavioral modifications:

$$\boxed{[\![\mathcal{A}]\!]_{\text{bbm}} \;=\; [\![C_1, \ldots, C_n]\!]_{C_1,\ldots,C_n}}$$

- Semantics of $\mathcal{A}$ after behavioral modifications:

$$\boxed{[\![\mathcal{A}]\!]_{\text{abm}} \;=\; [\![\mathcal{A}]\!]_{\text{bbm}} \,/\, \mathcal{H} \setminus \mathcal{R}\,[\varphi]}$$

- **Running example** (second step of the translation):
  - ⊙ Semantics of `Client_Server`:

$$[\![\texttt{S}]\!][\texttt{S.receive\_request\_1} \mapsto \texttt{C\_1.send\_request\#S.receive\_request\_1},\ \texttt{OAQ\_1.depart} \mapsto \texttt{OAQ\_1.depart\#C\_1.receive\_response},\ \texttt{S.receive\_request\_2} \mapsto \texttt{C\_2.send\_request\#S.receive\_request\_2},\ \texttt{OAQ\_2.depart} \mapsto \texttt{OAQ\_2.depart\#C\_2.receive\_response}]$$

$$\| \{\texttt{C\_1.send\_request\#S.receive\_request\_1}, \texttt{OAQ\_1.depart\#C\_1.receive\_response}\}$$

$$[\![\texttt{C\_1}]\!][\texttt{C\_1.send\_request} \mapsto \texttt{C\_1.send\_request\#S.receive\_request\_1},\ \texttt{C\_1.receive\_response} \mapsto \texttt{OAQ\_1.depart\#C\_1.receive\_response}]$$

$$\| \{\texttt{C\_2.send\_request\#S.receive\_request\_2}, \texttt{OAQ\_2.depart\#C\_2.receive\_response}\}$$

$$[\![\texttt{C\_2}]\!][\texttt{C\_2.send\_request} \mapsto \texttt{C\_2.send\_request\#S.receive\_request\_2},\ \texttt{C\_2.receive\_response} \mapsto \texttt{OAQ\_2.depart\#C\_2.receive\_response}]$$

Summarizing Example: Pipe-Filter System

- General description:
  - Each filter reads streams of row data on its inputs, transforms them, and produces streams of processed data on its outputs (increm. process).
  - Each pipe transmits outputs of one filter to inputs of another filter.
- Specific scenario:
  - Four identical filters, each equipped with a finite buffer.
  - Single pipe forwarding any item received from the upstream filter to one of the three downstream filters, according to the availability of free positions in their buffers.
- Defining and comparing its PA specification and its PADL description: which one is better? (Our objective was the enhancement of PA usability.)

- PA specification (all buffers are initially empty and can hold up to 10 items):

$$Pipe\_Filter \triangleq Upstream\_Filter_{0/10} \parallel_{\{output\_accept\_item\}} Pipe \parallel_{\{forward\_input\_item_1\}} Downstream\_Filter^1_{0/10} \parallel_{\{forward\_input\_item_2\}} Downstream\_Filter^2_{0/10} \parallel_{\{forward\_input\_item_3\}} Downstream\_Filter^3_{0/10}$$

- Not clear which process term communicates with which process term.
- No confidence about the correctness of the synchronization sets associated with the occurrences of the parallel composition operator.
- The degree of synchronicity and multiplicity of the communications is completely obscure.

- Set of equations defining $Upstream\_Filter_{0/10}$ $(1 \leq j \leq 9)$:

$$
\begin{aligned}
Upstream\_Filter_{0/10} &\triangleq input\_item \,.\, transform\_item \,.\, Upstream\_Filter_{1/10} \\
Upstream\_Filter_{j/10} &\triangleq input\_item \,.\, transform\_item \,.\, Upstream\_Filter_{j+1/10} + \\
&\qquad output\_accept\_item \,.\, Upstream\_Filter_{j-1/10} \\
Upstream\_Filter_{10/10} &\triangleq output\_accept\_item \,.\, Upstream\_Filter_{9/10}
\end{aligned}
$$

- Not clear which actions are part of the interface of the upstream filter and which actions represent internal activities (it can only be guessed).
- Forced to use a strange name like *output_accept_item* for the action describing the communication between the upstream filter and the pipe (more natural *output_item* for the upstream filter and *accept_item* for the pipe).
- A solution to this problem would be the use of relabeling functions.

- Equation defining *Pipe*:

$$Pipe \triangleq output\_accept\_item\,.\,(forward\_input\_item_1\,.\,Pipe + forward\_input\_item_2\,.\,Pipe + forward\_input\_item_3\,.\,Pipe)$$

- Same problem as before caused by the lack of freedom in selecting synchronizing action names.
- Scalability problem: for a different number of downstream filters, the intervention of the designer is required in order to modify the part of the defining equation enclosed in parentheses.

- The set of equations defining $Downstream\_Filter^{1}_{0/10}$ $(1 \leq j \leq 9)$ ...

$$Downstream\_Filter^{1}_{0/10} \triangleq forward\_input\_item_{1} \,.\, transform\_item \,.\, Downstream\_Filter^{1}_{1/10}$$

$$Downstream\_Filter^{1}_{j/10} \triangleq forward\_input\_item_{1} \,.\, transform\_item \,.\, Downstream\_Filter^{1}_{j+1/10} + output\_item \,.\, Downstream\_Filter^{1}_{j-1/10}$$

$$Downstream\_Filter^{1}_{10/10} \triangleq output\_item \,.\, Downstream\_Filter^{1}_{9/10}$$

- ... the set of equations defining $Downstream\_Filter^{2}_{0/10}$ $(1 \leq j \leq 9)$ ...

$$Downstream\_Filter^{2}_{0/10} \triangleq forward\_input\_item_{2} \,.\, transform\_item \,.\, Downstream\_Filter^{2}_{1/10}$$

$$Downstream\_Filter^{2}_{j/10} \triangleq forward\_input\_item_{2} \,.\, transform\_item \,.\, Downstream\_Filter^{2}_{j+1/10} + output\_item \,.\, Downstream\_Filter^{2}_{j-1/10}$$

$$Downstream\_Filter^{2}_{10/10} \triangleq output\_item \,.\, Downstream\_Filter^{2}_{9/10}$$

- ... the set of equations defining $Downstream\_Filter^{3}_{0/10}$ $(1 \leq j \leq 9)$ ...

$$Downstream\_Filter^{3}_{0/10} \triangleq forward\_input\_item_3 \,.\, transform\_item \,.\, Downstream\_Filter^{3}_{1/10}$$

$$Downstream\_Filter^{3}_{j/10} \triangleq forward\_input\_item_3 \,.\, transform\_item \,.\, Downstream\_Filter^{3}_{j+1/10} + output\_item \,.\, Downstream\_Filter^{3}_{j-1/10}$$

$$Downstream\_Filter^{3}_{10/10} \triangleq output\_item \,.\, Downstream\_Filter^{3}_{9/10}$$

- ... differ only for the names of the process constants and of some actions.
- Relabeling functions should be used to avoid redundancy.
- Scalability problem: for a different number of downstream filters, the intervention of the designer is required in order to add the necessary defining equations or relabeling functions.

- PADL description header:

```
ARCHI_TYPE Pipe_Filter(const integer pf_buffer_size := 10)
```

- Explicit support for data parameterization from the beginning.
- The initialization values are the default actual values for the formal data parameters of the whole description, which can then be exploited when passing actual data parameters in the declaration of AEIs.
- The declaration of each formal data parameter of the entire description or of a single AET is preceded by `const` to remind that its value is constant (not the case with formal data parameters of PA equations inside AETs).

- Definition of the filter AET (one AET, one equation, explicit interface, free names):

```
ARCHI_ELEM_TYPE Filter_Type(const integer buffer_size)

  BEHAVIOR
    Filter(integer(0..buffer_size) item_num := 0;
           void) =
      choice
      {
        cond(item_num < buffer_size) ->
          input_item . transform_item . Filter(item_num + 1),
        cond(item_num > 0) ->
          output_item . Filter(item_num - 1)
      }

  INPUT_INTERACTIONS  SYNC UNI input_item
  OUTPUT_INTERACTIONS SYNC UNI output_item
```

- Definition of the pipe AET (scalability thanks to or-interaction):

```
ARCHI_ELEM_TYPE Pipe_Type(void)

  BEHAVIOR
    Pipe(void; void) =
      accept_item . forward_item . Pipe()

  INPUT_INTERACTIONS  SYNC UNI accept_item
  OUTPUT_INTERACTIONS SYNC OR  forward_item
```

- Declaration of the topology (clear communication scheme, can be made more concise):

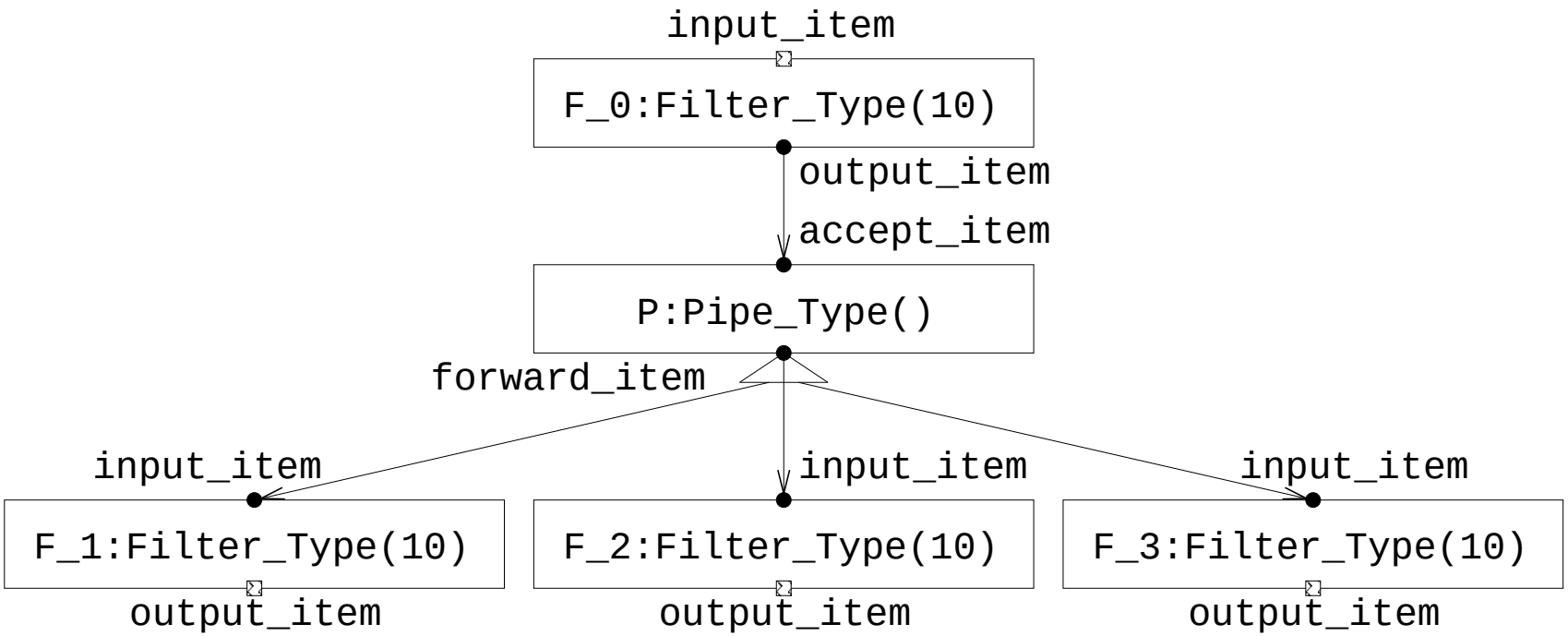
```
ARCHI_ELEM_INSTANCES
  F_0 : Filter_Type(pf_buffer_size);
  P   : Pipe_Type();
  F_1 : Filter_Type(pf_buffer_size);
  F_2 : Filter_Type(pf_buffer_size);
  F_3 : Filter_Type(pf_buffer_size)

ARCHI_INTERACTIONS
  F_0.input_item;
  F_1.output_item; F_2.output_item; F_3.output_item

ARCHI_ATTACHMENTS
  FROM F_0.output_item TO P.accept_item;
  FROM P.forward_item  TO F_1.input_item;
  FROM P.forward_item  TO F_2.input_item;
  FROM P.forward_item  TO F_3.input_item
```

- Architectural interactions allow for future structural extensions.

- Enriched flow graph:



- Isolated semantics of the various AEIs:

$$[\![\texttt{F\_0}]\!] = \texttt{F\_0.Filter}\{10 \hookrightarrow \texttt{buffer\_size}\}$$

$$[\![\texttt{F\_1}]\!] = \texttt{F\_1.Filter}\{10 \hookrightarrow \texttt{buffer\_size}\}$$

$$[\![\texttt{F\_2}]\!] = \texttt{F\_2.Filter}\{10 \hookrightarrow \texttt{buffer\_size}\}$$

$$[\![\texttt{F\_3}]\!] = \texttt{F\_3.Filter}\{10 \hookrightarrow \texttt{buffer\_size}\}$$

$$[\![\texttt{P}]\!] = \textit{or-rewrite}_{\emptyset}(\texttt{P.Pipe})$$

where $\textit{or-rewrite}_{\emptyset}(\texttt{P.Pipe})$ is given by:

```
Pipe'(void; void) =
  P.accept_item . choice
                  {
                    P.forward_item_1 . Pipe'(),
                    P.forward_item_2 . Pipe'(),
                    P.forward_item_3 . Pipe'()
                  }
```

- Semantics of `Pipe_Filter(10)`:

$$[\![\texttt{F\_0}]\!][\texttt{F\_0.output\_item} \mapsto \texttt{F\_0.output\_item\#P.accept\_item}]$$

$$\|_{\{\texttt{F\_0.output\_item\#P.accept\_item}\}}$$

$$[\![\texttt{P}]\!][\texttt{P.accept\_item} \mapsto \texttt{F\_0.output\_item\#P.accept\_item},\ \texttt{P.forward\_item\_1} \mapsto \texttt{P.forward\_item\_1\#F\_1.input\_item},\ \texttt{P.forward\_item\_2} \mapsto \texttt{P.forward\_item\_2\#F\_2.input\_item},\ \texttt{P.forward\_item\_3} \mapsto \texttt{P.forward\_item\_3\#F\_3.input\_item}]$$

$$\|_{\{\texttt{P.forward\_item\_1\#F\_1.input\_item}\}}$$

$$[\![\texttt{F\_1}]\!][\texttt{F\_1.input\_item} \mapsto \texttt{P.forward\_item\_1\#F\_1.input\_item}]$$

$$\|_{\{\texttt{P.forward\_item\_2\#F\_2.input\_item}\}}$$

$$[\![\texttt{F\_2}]\!][\texttt{F\_2.input\_item} \mapsto \texttt{P.forward\_item\_2\#F\_2.input\_item}]$$

$$\|_{\{\texttt{P.forward\_item\_3\#F\_3.input\_item}\}}$$

$$[\![\texttt{F\_3}]\!][\texttt{F\_3.input\_item} \mapsto \texttt{P.forward\_item\_3\#F\_3.input\_item}]$$

- Isomorphic to *Pipe_Filter* up to action names.

- Items are assumed to be all identical.
- However items may be different from each other (identifiers).
- We have to keep track of the order in which items arrive at buffers.
- Exploiting the value passing capability of PADL together with the data types it supports.
- New PADL description header:

```
ARCHI_TYPE Pipe_Filter_Id(const integer pf_buffer_size := 10)
```

- Only the name of the description has changed.

- Redefinition of the filter AET (list-typed parameter, input/output actions, local var):

```
ARCHI_ELEM_TYPE Filter_Type(const integer buffer_size)

  BEHAVIOR
    Filter(list(integer) item_list := list_cons();
           local integer id) =
      choice
      {
        cond(length(item_list) < buffer_size) ->
          input_item?(id) . transform_item . Filter(concat(item_list,
                                                           list_cons(id))),
        cond(length(item_list) > 0) ->
          output_item!(first(item_list)) . Filter(tail(item_list))
      }

  INPUT_INTERACTIONS  SYNC UNI input_item
  OUTPUT_INTERACTIONS SYNC UNI output_item
```

- Redefinition of the pipe AET (identity preservation throughout the system):

```
ARCHI_ELEM_TYPE Pipe_Type(void)

  BEHAVIOR
    Pipe(void;
         local integer id) =
      accept_item?(id) . forward_item!(id) . Pipe()

  INPUT_INTERACTIONS  SYNC UNI accept_item
  OUTPUT_INTERACTIONS SYNC OR  forward_item
```

- Same architectural topology section.
- Same enriched flow graph.
- Same structure of the process term formalizing the semantics.

G8: Supporting Architectural Styles

- An architectural style defines a vocabulary of components/connectors and a set of constraints on how they should behave and be combined.
- Family of software systems sharing specific organizational principles:
  - Call-return systems (main-subroutines, object oriented, layered, client-server).
  - Dataflow systems (pipe-filter, compilers).
  - Repositories (databases, hypertexts).
  - Virtual machines (interpreters).
  - Event-based systems (publish-subscribe).
- Shaw & Garlan (1996): *"... enable the designer to capitalize on codified principles and experience to specify, analyze, plan, and monitor system construction with high levels of efficiency and confidence ..."*.

Architectural Types

- The concept of architectural style is hard to formalize.
- At least two degrees of freedom:
  - Variability of the architectural element behavior.
  - Variability of the architectural topology.
- Variabilities interpretable in different ways, with their interpretation possibly changing from style to style.
- Approximate an architectural style with an architectural type (AT).
- The behavior of the architectural elements and the overall topology are allowed to vary in a controlled way from AT instance to AT instance:
  - Only the internal behavior of AETs can change.
  - Only some topological variations are admitted.

- All the instances of an AT are generated via architectural invocations of the definition of the AT, which consists of a PADL description.
- Explicit support for passing data pars and architectural parameters:
  - Actual values for formal data parameters (replace default values).
  - Actual AETs *preserving the observable behavior of formal AETs*.
  - Actual topology *complying with the formal topology up to admitted topological variations* (four groups of parameters).
  - Actual behavioral modifications (three groups of parameters).
  - Actual names for architectural interactions (hierarchical modeling).
- Symbol @ is used as separator of groups of actual parameters (if omitted, a group of actual architectural parameters coincides with the respective formal group).
- The semantics of an AT instance is built the same way as the semantics of the definition of the AT by using actual parameters instead of formal ones and by relabeling architectural interactions to their actual names.

Hierarchical Modeling

- Simplest form of AT invocation (no behavioral or topological variation):
  - Passing actual data parameters (if any).
  - Reusing formal AETs, topology, and behavioral modifications.
- Still useful for hierachical modeling purposes if we extend PADL so that an AET can be defined as an instance of a previously defined AT.
- The AET behavior is given by the semantics of the AT instance.
- AET interactions become actual names for AT architectural interactions (whose synchronicity and multiplicity qualifiers are overridden).
- Related AEIs are graphically represented as boxes with double border.
- Unification between AEI interactions and AT architectural interactions is graphically represented through dashed edges.

- **Example**: hierarchical variant of the client-server system.
- The server has the same structure and behavior as the pipe-filter system.
- Header of the textual description:

```
ARCHI_TYPE H_Client_Server(const integer hcs_buffer_size := 10)
```

- Formal data parameters are now necessary due to the presence of buffers.
- The or-dependence mechanism can no longer be exploited.
- When an architectural element results from the combination of other elements, their interleaving may alter the relationships between local output or-interactions and local input or-interactions.
- Each request needs to carry the client identifier in order to ensure that responses are sent to the clients that issued the corresponding requests.

- Redefinition of the client AET (inclusion of the client identifier):

```
ARCHI_ELEM_TYPE Client_Type(const integer id)

  BEHAVIOR
    Client_Internal(void; void) =
      process . Client_Interacting();
    Client_Interacting(void; void) =
      send_request!(id) .
        choice
        {
          cond(send_request.success = true) ->
                                  receive_response . Client_Internal(),
          cond(send_request.success = false) ->
                                  keep_processing . Client_Interacting()
        }

  INPUT_INTERACTIONS  SYNC  UNI receive_response
  OUTPUT_INTERACTIONS SSYNC UNI send_request
```

- Redefinition of the server AET (concise unification based on output values):

```
ARCHI_ELEM_TYPE Server_Type(const integer buffer_size)

  BEHAVIOR
    Server(void; void) =
      Pipe_Filter_Id(buffer_size @
                     @         /* reuse formal AETs */
                     @ @ @ @   /* reuse formal topology */
                     @ @ @     /* no behavioral modifications */
                     UNIFY F_0.input_item WITH receive_request;
                     FOR_ALL 1 <= j <= 2
                       UNIFY F_1.output_item!(j),
                             F_2.output_item!(j),
                             F_3.output_item!(j) WITH send_response[j])

  INPUT_INTERACTIONS  SYNC  OR  receive_request
  OUTPUT_INTERACTIONS ASYNC UNI send_response[1]; send_response[2]
```

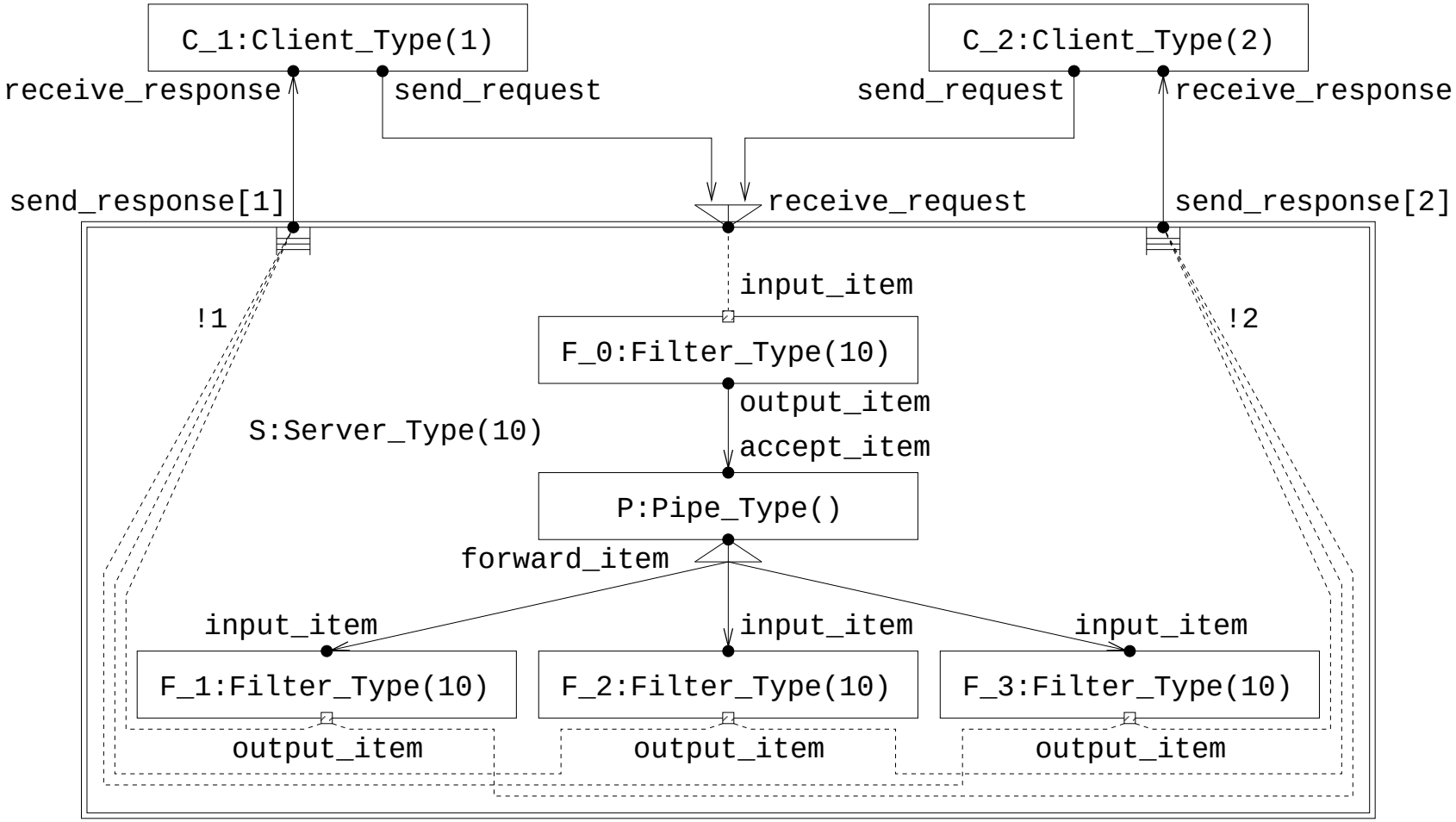
- Redeclaration of the topology:

```
ARCHI_ELEM_INSTANCES
  S   : Server_Type(hcs_buffer_size);
  C_1 : Client_Type(1);
  C_2 : Client_Type(2)

ARCHI_INTERACTIONS
  void

ARCHI_ATTACHMENTS
  FROM C_1.send_request   TO S.receive_request;
  FROM C_2.send_request   TO S.receive_request;
  FROM S.send_response[1] TO C_1.receive_response;
  FROM S.send_response[2] TO C_2.receive_response
```

- Enriched flow graph:



Behavioral Conformity

- AT invocation with actual AETs different from formal AETs and the actual topology introducing no variations w.r.t. the formal topology.
- An AT instance $\mathcal{A}_1$ behaviorally conforms to another AT instance $\mathcal{A}_2$ if both exhibit the same observable behavior.
- Formalized through weak bisimulation equivalence $\approx_{\mathrm{B}}$:
  - Abstraction from internal actions (they play no role in behav. conf.).
  - Congruence w.r.t. static operators (AT-level b.c. from AET-level b.c.).
  - Not too coarse (preserving many properties of interest).
- The complexity of the AT behavioral conformity check is thus linear in the number of AETs (instead of being exponential in the number of AEIs).
- Relabeling for interactions, hiding for internal actions.

- Let $\mathcal{C}_1, \mathcal{C}_2$ be two AETs with:
  - $\mathcal{D}_1, \mathcal{D}_2$ being the sets of their formal data parameters.
  - $\mathcal{E}_1, \mathcal{E}_2$ being the sequences of their PA defining equations.
  - $\mathcal{N}_1, \mathcal{N}_2$ being the sets of their internal actions.
  - $\mathcal{I}_1, \mathcal{I}_2$ being the sets of their interactions.
- Assume parameters in $\mathcal{D}_1, \mathcal{D}_2$ consistent by number, order, and type and interactions in $\mathcal{I}_1, \mathcal{I}_2$ consistent by number, order, and qualifiers.
- $\mathcal{C}_1$ behaviorally conforms to $\mathcal{C}_2$ iff there exist two injective relabeling functions φ_1, φ_2 for $\mathcal{I}_1, \mathcal{I}_2$, which have the same codomain and are qualifier-consistent, such that for all substitutions σ of $\mathcal{D}_1, \mathcal{D}_2$:

$$(\mathcal{E}_1\,\sigma)\,/\,\mathcal{N}_1\,[\varphi_1] \approx_{\mathrm{B}} (\mathcal{E}_2\,\sigma)\,/\,\mathcal{N}_2\,[\varphi_2]$$

- An AT instance $\mathcal{A}_1$ (strictly) behaviorally conforms to another AT instance $\mathcal{A}_2$ iff:
  - ⊙ Their actual data parameters are consistent by number, order, and type (and value).
  - ⊙ *Their AETs are consistent by number, order, and behavioral conformity.*
  - ⊙ Their AEIs are consistent by number, order, and type and have actual data parameters consistent by number, order, and type (and value).
  - ⊙ Their architectural interactions are consistent by number, order, qualifiers, and AEI membership.
  - ⊙ Their attachments are consistent by number, order, and qualifiers and AEI membership of the involved local interactions.

- Let $\mathcal{A}_1, \mathcal{A}_2$ be two AT instances with:
  - $\mathcal{N}_1, \mathcal{N}_2$ being the sets of internal actions of their AEIs.
  - $\mathcal{I}_1, \mathcal{I}_2$ being the sets of interactions of their AEIs.

  Whenever $\mathcal{A}_1$ strictly behaviorally conforms to $\mathcal{A}_2$, then there exist two relabeling functions φ_1, φ_2 for $\mathcal{I}_1, \mathcal{I}_2$, which are injective at least on local interactions, have the same codomain, and are qualifier-consistent, such that:

$$\boxed{[\![\mathcal{A}_1]\!]_{\mathrm{bbm}} \,/\, \mathcal{N}_1\,[\varphi_1] \;\approx_{\mathrm{B}}\; [\![\mathcal{A}_2]\!]_{\mathrm{bbm}} \,/\, \mathcal{N}_2\,[\varphi_2]}$$

- **Example**: instance of the pipe-filter AT with faulty filters.
- Obtained via architectural invocation (new AET `Faulty_Filter_Type` as actual param):

```
Pipe_Filter(@       /* reuse default values of formal data parameters */
            Faulty_Filter_Type;
            Pipe_Type @
            F_0 : Faulty_Filter_Type(pf_buffer_size);
            P   : Pipe_Type();
            F_1 : Faulty_Filter_Type(pf_buffer_size);
            F_2 : Faulty_Filter_Type(pf_buffer_size);
            F_3 : Faulty_Filter_Type(pf_buffer_size) @
            @ @ @   /* reuse rest of formal topology */
            @ @ @   /* no behavioral modifications */
            )       /* reuse names of actual architectural interactions */
```

- Definition of the faulty filter AET (subject to failures and subsequent repairs):

```
ARCHI_ELEM_TYPE Faulty_Filter_Type(const integer buffer_size)

  BEHAVIOR
    Faulty_Filter(integer(0..buffer_size) item_num := 0;
                  void) =
      choice
      {
        cond(item_num < buffer_size) ->
          input_item . transform_item . Faulty_Filter(item_num + 1),
        cond(item_num > 0) ->
          output_item . Faulty_Filter(item_num - 1),
        fail . repair . Faulty_Filter(item_num)
      }

  INPUT_INTERACTIONS  SYNC UNI input_item
  OUTPUT_INTERACTIONS SYNC UNI output_item
```

- Is the AT instance associated with the AT invocation a legal instance of the invoked AT?
- The `Pipe_Filter` instance originated by the AT invocation strictly behaviorally conforms to the `Pipe_Filter` instance originated by the AT definition because:
  - ⊙ They have the same data parameters.
  - ⊙ They have consistent topologies (no topological variations).
  - ⊙ They have the same pipe AET.
  - ⊙ `Faulty_Filter_Type` and `Filter_Type` have the same data params and the same interactions (identical relabeling function).
  - ⊙ $\approx_B$ equates `Faulty_Filter(0)/{transform_item,fail,repair}` and `Filter(0)/{transform_item}` for all values of `buffer_size`.

Exogenous Variations

- AT invocation with the actual topology introducing variations with respect to the formal topology.
- Addition of AEIs obtained by attaching some of them to the topological frontier, which is the set of architectural interactions.
- An exogenous variation is expressed within the actual topology by means of keyword `EXO` followed by four parameters:
  - Set of additional AEIs, which must be instances of the actual AETs.
  - Set of replacements of some of the actual architectural interactions with new architectural interactions belonging to the additional AEIs.
  - Set of additional attachments involving all additional AEIs and all replaced actual architectural interactions, which thus become local.
  - Possible nested exogenous variations.

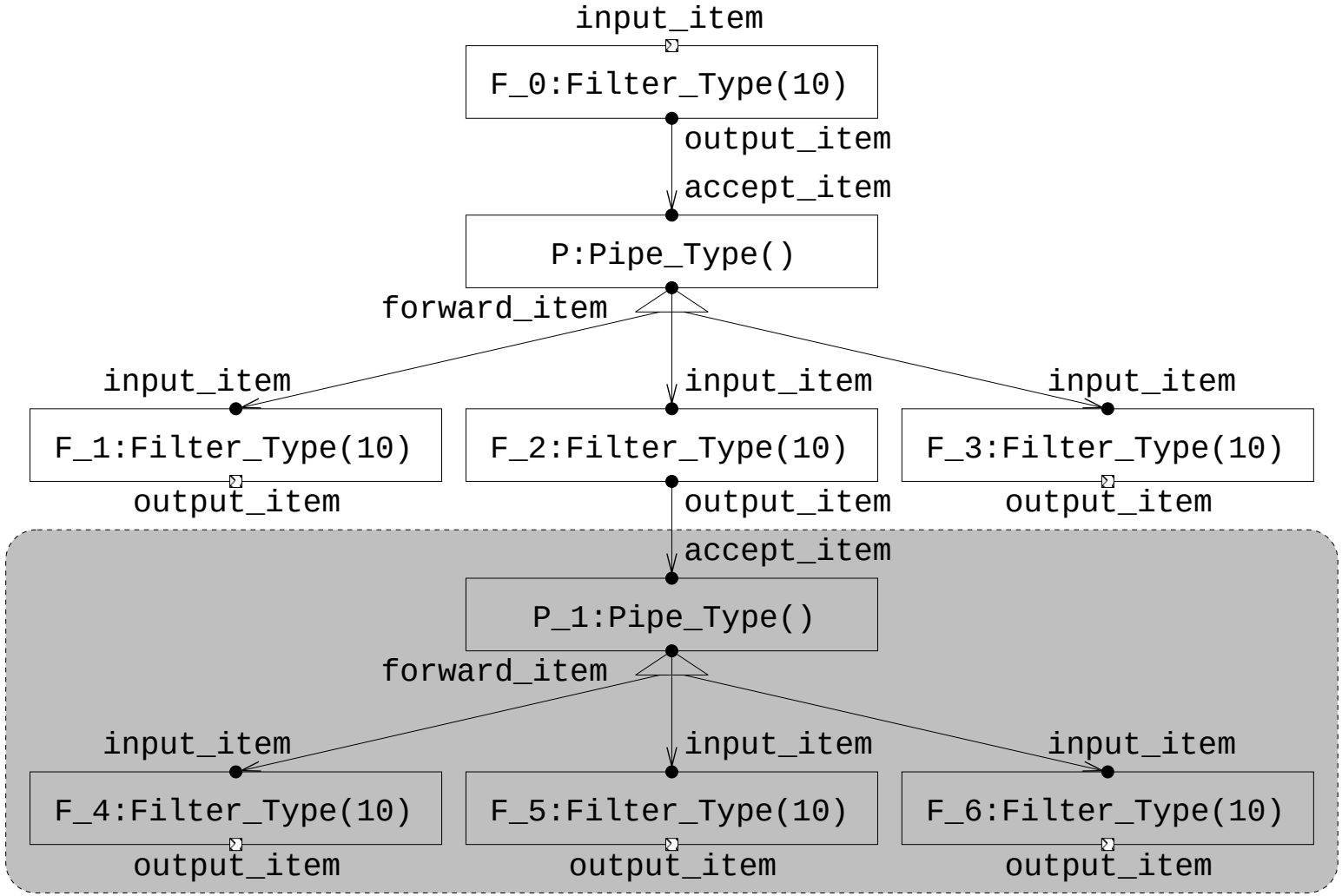
- With respect to a portion of the formal topology of the invoked AT, the addendum must be complete and contain no new kinds of attachment.
- An exogenous variation (strictly) topologically conforms to the formal topology iff there exists an injective function *corr* defined from the set of additional AEIs to the set of actual AEIs such that:
  - C and $corr(C)$ have the same type (and actual data param values).
  - For all interactions a of an arbitrary additional AEI C:
    - $C.a$ is local/architectural iff $corr(C).a$ is local/architectural.
    - There is an additional AEI C' with an additional attachment from $C.a$ to $C'.a'$ iff there is an attachment from $corr(C).a$ to $corr(C').a'$ (same in the opposite direction).
    - There is an additional attachment from $C.a$ to the replaced architectural interaction $K.b$ iff there is an actual AEI K' of the same type as K with an attachment from $corr(C).a$ to $K'.b$ (same in the opposite direction).

- **Example**: exogenous variation of the pipe-filter AT.
- The topological frontier is composed of:
  - `F_0.input_item`.
  - `F_1.output_item`, `F_2.output_item`, `F_3.output_item`.
- An exogenous variation at `F_0.input_item` must replicate the defined topology by viewing `F_0` as a downstream filter.
- An exogenous variation at `F_1.output_item`, `F_2.output_item`, or `F_3.output_item` must replicate the defined topology by viewing `F_1`, `F_2`, or `F_3`, respectively, as an upstream filter.

- Architectural invocation for an exogenous variation at `F_2.output_item`:

```
Pipe_Filter(@          /* reuse default values of formal data parameters */
            @          /* reuse formal AETs */
            @ @ @      /* reuse formal AEIs, arch. inters, and attachs */
            EXO(P_1 : Pipe_Type();
                F_4 : Filter_Type(pf_buffer_size);
                F_5 : Filter_Type(pf_buffer_size);
                F_6 : Filter_Type(pf_buffer_size) @
                REPLACE F_2.output_item WITH F_4.output_item,
                                             F_5.output_item,
                                             F_6.output_item @
                FROM F_2.output_item  TO P_1.accept_item;
                FROM P_1.forward_item TO F_4.input_item;
                FROM P_1.forward_item TO F_5.input_item;
                FROM P_1.forward_item TO F_6.input_item @
                ) @    /* no nested exogenous variations */
            @ @ @      /* no behavioral modifications */
            )          /* reuse names of actual architectural interactions */
```

- Enriched flow graph:



Endogenous Variations

- A different kind of topological variation is the one that takes place inside the topological frontier.
- Changing the number of AEIs of certain types in certain positions of the topology, without altering the number of attachments in which local and-/or-interactions are involved.
- The variable number of AEIs of each of the involved types is defined as a data parameter of the AT.
- Indexing mechanism via `FOR_ALL` construct for the concise declaration in the topology section of arbitrarily many AEIs of the same type, of their architectural interactions, and of the attachments involving them.
- All these variations conform to the formal topology by construction.
- New kinds of attachment may be created when varying the number of AEIs of some of the involved types from one to more than one.

- **Example**: endogenous variation of a ring of pipe-filter systems.
- Every system in the ring waits for items from the previous system in the ring, processes the received items, and then sends the processed items to the next system in the ring.
- Presence of an initial component in the ring, which accepts items from the outside and returns items to the outside.
- Any incoming item has to perform an entire traversal of the ring before leaving it.
- Allowing for a variable number of pipe-filter systems.
- Header of the textual description:

```
ARCHI_TYPE Pipe_Filter_R(const integer pfr_system_num  :=  3,
                         const integer pfr_buffer_size := 10)
```

- Definition of the initial component AET (aware that the ring may saturate):

```
ARCHI_ELEM_TYPE Init_Comp_Type(void)

  BEHAVIOR
    Init_Comp_In(void; void) =
      choice
      {
        accept_item . Init_Comp_New(),
        receive_item . return_item . Init_Comp_In()
      };
    Init_Comp_New(void; void) =
      choice
      {
        send_item . Init_Comp_In(),
        receive_item . return_item . Init_Comp_New()
      }

  INPUT_INTERACTIONS  SYNC UNI accept_item; receive_item
  OUTPUT_INTERACTIONS SYNC UNI return_item; send_item
```

- Definition of the pipe-filter system AET:

```
ARCHI_ELEM_TYPE PF_System_Type(const integer buffer_size)

  BEHAVIOR
    PF_System(void; void) =
      Pipe_Filter(buffer_size @
                  @          /* reuse formal AETs */
                  @ @ @ @    /* reuse formal topology */
                  @ @ @      /* no behavioral modifications */
                  UNIFY F_0.input_item  WITH receive_item;
                  UNIFY F_1.output_item,
                        F_2.output_item,
                        F_3.output_item WITH send_item)

  INPUT_INTERACTIONS  SYNC UNI receive_item
  OUTPUT_INTERACTIONS SYNC UNI send_item
```

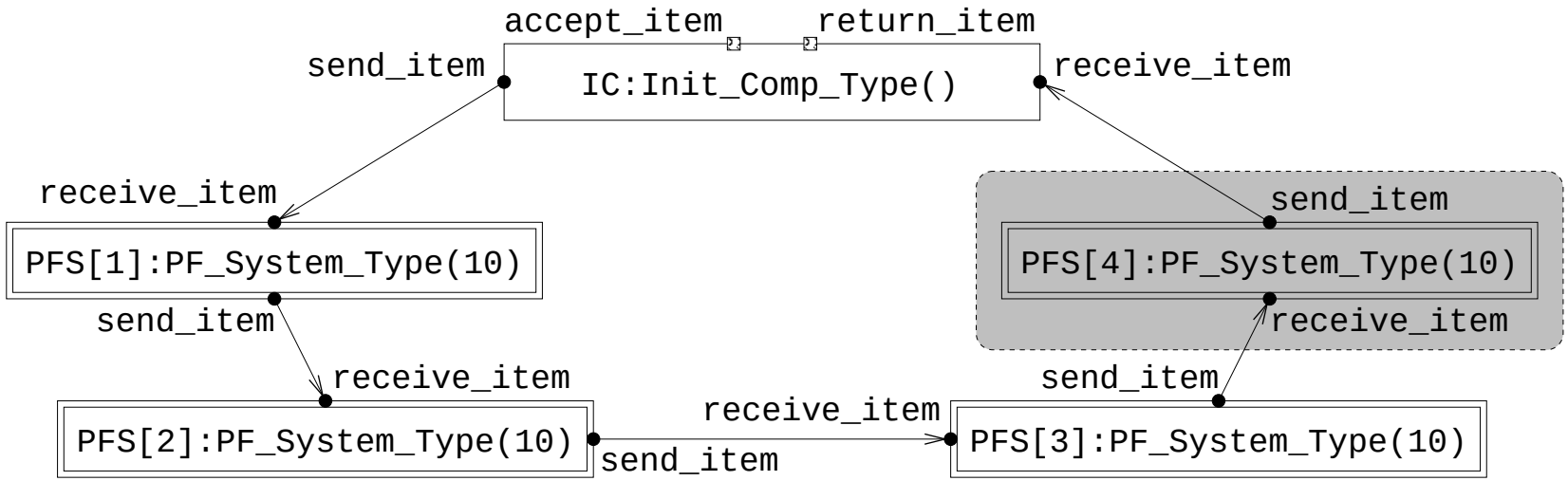
- Declaration of the topology:

```
ARCHI_ELEM_INSTANCES
  IC : Init_Comp_Type();
  FOR_ALL 1 <= j <= pfr_system_num
    PFS[j] : PF_System_Type(pfr_buffer_size)

ARCHI_INTERACTIONS
  IC.accept_item;
  IC.return_item

ARCHI_ATTACHMENTS
  FROM IC.send_item TO PFS[1].receive_item;
  FOR_ALL 1 <= j <= pfr_system_num - 1
    FROM PFS[j].send_item TO PFS[j + 1].receive_item;
  FROM PFS[pfr_system_num].send_item TO IC.receive_item
```

- Enriched flow graph of `Pipe_Filter_R(4, 10 @ @ @ @ @ @ @ @ @)`:



Multiplicity Variations

- Another kind of topological variation taking place inside the topological frontier is possible.
- Changing the number of AEIs of certain types that are attached to certain local and-/or-interactions.
- Variable numbers of involved AEIs defined as AT data parameters.
- Indexing mechanism via `FOR_ALL` construct for the topology section.
- No new kinds of attachment can be created when varying the number of considered AEIs.
- These variations are admissible iff the involved local and-/or-interactions support variability.
- A local and-/or-interaction of an AEI supports variability if the AEI is not attached with uni-interactions to any of the AEIs attached to the considered local and-/or-interaction.

- **Example**: multiplicity variation of the pipe-filter AT.
- Allow for a variable number of downstream filters.
- Achieved by introducing topological variability at the or-interaction `forward_item` of the pipe.
- Header of the textual description:

```
ARCHI_TYPE OV_Pipe_Filter(const integer ovpf_downstr_num :=  3,
                          const integer ovpf_buffer_size := 10)
```

- The definitions of the filter AET and of the pipe AET do not change.

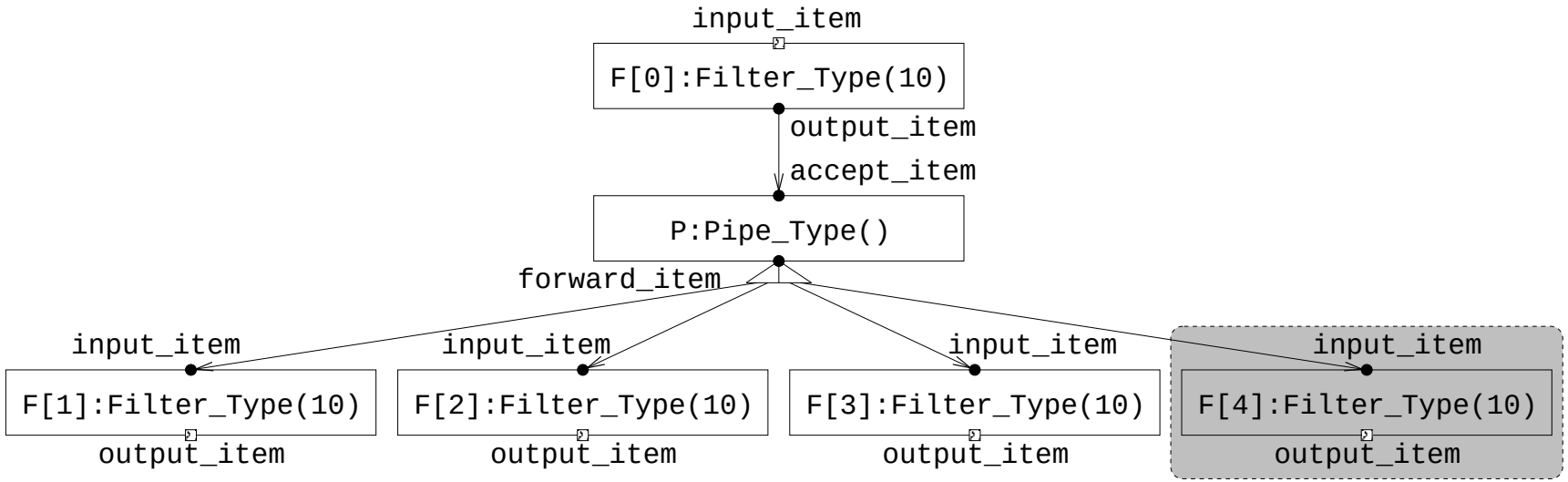
- Declaration of the topology:

```
ARCHI_ELEM_INSTANCES
  F[0] : Filter_Type(ovpf_buffer_size);
  P    : Pipe_Type();
  FOR_ALL 1 <= j <= ovpf_downstr_num
    F[j] : Filter_Type(ovpf_buffer_size)

ARCHI_INTERACTIONS
  F[0].input_item;
  FOR_ALL 1 <= j <= ovpf_downstr_num
    F[j].output_item

ARCHI_ATTACHMENTS
  FROM F[0].output_item TO P.accept_item;
  FOR_ALL 1 <= j <= ovpf_downstr_num
    FROM P.forward_item TO F[j].input_item
```

- Enriched flow graph of `OV_Pipe_Filter(4, 10 @ @ @ @ @ @ @ @ @)`:



Comparison with PA

- PADL has been specifically structured on the basis of a set of guidelines aimed at improving usability with respect to process algebra.
- Behavior description sharply separated from topology description (no longer intertwinedly encoded through the parallel composition operator).
- Intended use of every action made clear via a set of explicit qualifiers (no longer to be inferred from the occurrences of the parallel composition operator).
- Communicating interactions explicitly related by means of attachments (no longer required to have matching names; clear communication scheme).
- Error-prone situations easily detected via static checks (conseq. of prev. two).
- Only simpler-to-use behavioral operators available to the designer.
- Textual notation accompanied by a richer graphical notation.
- Higher degree of specification reuse at the component level (AET) and at the system level (AT).

Comparison with Parallel Composition Operators

- The communication mechanism of PADL is more adequate for modeling purposes than typical parallel composition operators (simple, expressive).
- CCS:
  - Two-way synchronizations only, complementary name constraint.
  - Restriction operator necessary to enforce synchronization.
- CSP:
  - Multiway synchronizations, identical name constraint.
  - Explicit synchronization sets whose contents depend on the order.
- ACP:
  - Definition of a communication function over the set of action names.
  - Restriction operator necessary to enforce synchronization.
- Similar to flexible n-ary operators with explicit declaration of process interface able to handle m-among-n synchronizations ($2 \leq m \leq n$).

Comparison with Other Process Algebraic ADLs

- Only static architectures to balance expressiveness and analyzability (dynamic/mobile architectures representable in Dynamic Wright, Darwin/FSP, LEDA, π-ADL).
- Textual notation and translation semantics inspired by Wright, but:
  - Components and connectors are handled uniformly (no trivial connectors).
  - Interfaces are simply expressed through actions (no ports/roles).
- Elicitation of interaction synchronicity, multiplicity, and dependences.
- Behavioral modifications for conducting ad-hoc analyses.
- Architectural types for approximating architectural styles (analyzability):
  - Hierarchical modeling via architectural invocations.
  - Internal behavioral variations respecting behavioral conformity.
  - Restricted number of topological variations.
- Other ADLs: Aesop, UniCon, Rapide, MetaH, SADL, C2, Acme, AADL plus approaches based on Z, CHAM, graph grammars, UML, Java.

Part II:
Component-Oriented Functional Verification

Architecture-Level Mismatch Detection

- PADL is much easier to use than PA for modeling purposes.
- △ *What about the verification of functional properties?*
- All the analysis techniques applicable to PA like equivalence checking and model checking are inherited by PADL.
- △ *Are those techniques adequate at the architectural level?*
- What we need to check for is the absence of architectural mismatches.
- Errors stemming from the inappropriate assembly of several components each of which is correct when considered in isolation.
- If not detected in the early design stages, then a lack of coordination among components will emerge at run time.

- Inferring the absence of architectural mismatches in a software system from the properties of its individual components.
- Producing diagnostic information in order to enable the identification of the components responsible for well-formedness violations.
- Information helpful for modifying the design of in-house components or synthesizing models of wrappers/adaptors for off-the-shelf components.
- Topological reduction process MismDet uses behavioral equivalences:
  - In case of success, repeatedly replaces basic topological portions with single architectural elements having the same observable behavior.
  - In case of failure, returns distinguishing modal logic formulas.
- Relying on process algebraic machinery only: congruence properties and modal logic characterizations of behavioral equivalences.
- Scaling to arbitrary topologies and architectural types.

Class of Properties

- Architectural mismatches can derive only from the wrong interplay of local interactions (internal actions and arch. interactions not involved in communications).
- MismDet focuses on properties expressing the possibility/necessity of executing certain local interactions in a certain order.
- Formulas of an action-based modal or temporal logic in which negation does not occur (simplification of property derivation from topologically reduced variant).
- Class Ψ of those properties $\mathcal{P}$ for each of which there exists $\approx_{\mathcal{P}}$ that is:
  - $\mathcal{P}$-preserving (fundamental for enabling the topological reduction process).
  - Congruent w.r.t. static operators (application to single topological portions).
  - Weak (abstraction from unimportant activities).
  - Characterizable via modal logic (production of diagnostic information).
- **Example**: deadlock freedom or any negation-free formula of weak HML is included in Ψ and preserved by weak bisimulation equivalence $\approx_{\mathrm{B}}$.

Detection Strategy

- MismDet considers a variant of enriched flow graph where:
  - Vertices correspond to AEIs.
  - Two vertices are linked by an edge iff attachments have been declared among the interactions of their corresponding AEIs.
- This graph, which abstracts from direction and number of attachments and is an arbitrary combination of possibly intersecting stars and cycles (basic topological formats), is assumed to be strongly connected.
- MismDet applies architectural checks locally to stars/cycles of AEIs:
  - Objective: verify whether the whole topology can be reduced to a single $\approx_{\mathcal{P}}$-equivalent AEI, which is then checked against $\mathcal{P} \in \Psi$.
  - Strategy: replace any set of AEIs forming a star or a cycle with a single $\approx_{\mathcal{P}}$-equivalent AEI in the set.
- The congruence property of $\approx_{\mathcal{P}}$ w.r.t. static operators avoids verifying the preservation of the possible validity of $\mathcal{P}$ after each reduction step.

- Let $\{C_1, \ldots, C_n\}$ be a set of AEIs forming a star or a cycle.
- Before applying the check, for each C_j in the set we have to hide all of its internal actions and architectural interactions as well as all of its local interactions that are not attached to $\{C_1, \ldots, C_n\}$.
- These actions cannot result in mismatches within $\{C_1, \ldots, C_n\}$, but may hamper the topological reduction process if left visible.
- For each AEI C_j in the set we also have to exclude all of its additional implicit AEIs that are not attached to $\{C_1, \ldots, C_n\}$ (they are not necessary).
- The only actions that remain observable are those in $\mathcal{LI}_{C_j;C_1,\ldots,C_n}$ and those in $\mathcal{OALI}_{C_j}$.
- $\mathcal{OALI}_{C_j}$: set of originally asynchronous local interactions of C_j:
  - Local interactions of the related additional implicit AEIs to which they have been re-attached (synchronization sets of isolated semantics).
  - Exceptions that may be raised by the local input semi-synchronous interactions corresponding to local input asynchronous interactions.

- Partially closed variant and totally closed variant of the interacting semantics set the visibility of each action as needed by MismDet.
- Both parameterized with respect to a set of AEIs $\{C''_1, \ldots, C''_{n''}\}$ determining the additional implicit AEIs that have to be included.
- Interacting semantics of $C_j \in \{C_1, \ldots, C_n\}$ with respect to $\{C_1, \ldots, C_n\}$ without buffers for the originally asynchronous local interactions of C_j:

$$[\![C_j]\!]^{\text{wob}}_{C_1,\ldots,C_n} = \mathit{or\text{-}rewrite}_{\emptyset}(C_j.\mathcal{E}\{ap_i \hookrightarrow fp_i \mid 1 \leq i \leq m\})\,[\varphi_{C_j,\text{async}}]\,[\varphi_{C_j;C_1,\ldots,C_n}]$$

- $[\![C_j]\!]^{\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n}$: variant of $[\![C_j]\!]^{\text{wob}}_{C_1,\ldots,C_n}$ including the buffers for the originally asynchronous local interactions of C_j attached to $\{C''_1,\ldots,C''_{n''}\}$.

- Partially closed interacting semantics of $C_j \in \{C_1, \ldots, C_n\}$ with respect to $\{C_1, \ldots, C_n\}$ including its buffers attached to $\{C''_1, \ldots, C''_{n''}\}$:

$$[\![C_j]\!]^{\mathrm{pc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} = [\![C_j]\!]^{\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} / (Name - \mathcal{V}_{C_j;C_1,\ldots,C_n})$$

  where:

$$\mathcal{V}_{C_j;C_1,\ldots,C_n} = \varphi_{C_j;C_1,\ldots,C_n}(\mathcal{LI}_{C_j;C_1,\ldots,C_n}) \cup \varphi_{C_j,\mathrm{async}}(\mathcal{OALI}_{C_j})$$

- $[\![C_j]\!]^{\mathrm{pc;wob}}_{C_1,\ldots,C_n}$: shorthand for the case $n'' = 0$.

- Partially closed interacting semantics of $\{C'_1, \ldots, C'_{n'}\} \subseteq \{C_1, \ldots, C_n\}$ w.r.t. $\{C_1, \ldots, C_n\}$ including their buffers attached to $\{C''_1, \ldots, C''_{n''}\}$:

$$\begin{aligned}
&[\![C'_1, \ldots, C'_{n'}]\!]^{\mathrm{pc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} = \\
&\qquad [\![C'_1]\!]^{\mathrm{pc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} \parallel_{\mathcal{S}(C'_1,C'_2;C_1,\ldots,C_n)} \\
&\qquad\quad [\![C'_2]\!]^{\mathrm{pc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} \parallel_{\mathcal{S}(C'_1,C'_3;C_1,\ldots,C_n) \cup \mathcal{S}(C'_2,C'_3;C_1,\ldots,C_n)} \cdots \\
&\qquad\quad \cdots \parallel_{\bigcup_{i=1}^{n'-1} \mathcal{S}(C'_i,C'_{n'};C_1,\ldots,C_n)} [\![C'_{n'}]\!]^{\mathrm{pc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n}
\end{aligned}$$

- Totally closed interacting semantics of $C_j \in \{C_1, \ldots, C_n\}$ with respect to $\{C_1, \ldots, C_n\}$ including its buffers attached to $\{C''_1, \ldots, C''_{n''}\}$:

$$[\![C_j]\!]^{\mathrm{tc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} = [\![C_j]\!]^{\mathrm{pc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} / \varphi_{C_j,\mathrm{async}}(\mathcal{OALI}_{C_j})$$

- $[\![C_j]\!]^{\mathrm{tc;wob}}_{C_1,\ldots,C_n}$: shorthand for the case $n'' = 0$.

- Totally closed interacting semantics of $\{C'_1, \ldots, C'_{n'}\} \subseteq \{C_1, \ldots, C_n\}$ w.r.t. $\{C_1, \ldots, C_n\}$ including their buffers attached to $\{C''_1, \ldots, C''_{n''}\}$:

$$\begin{aligned}
[\![C'_1, \ldots, C'_{n'}]\!]^{\mathrm{tc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} = {} & \\
& [\![C'_1]\!]^{\mathrm{tc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} \|_{\mathcal{S}(C'_1,C'_2;C_1,\ldots,C_n)} \\
& [\![C'_2]\!]^{\mathrm{tc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} \|_{\mathcal{S}(C'_1,C'_3;C_1,\ldots,C_n)\cup\mathcal{S}(C'_2,C'_3;C_1,\ldots,C_n)} \cdots \\
& \cdots \|_{\bigcup_{i=1}^{n'-1} \mathcal{S}(C'_i,C'_{n'};C_1,\ldots,C_n)} [\![C'_{n'}]\!]^{\mathrm{tc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n}
\end{aligned}$$

- $[\![C'_1, \ldots, C'_{n'}]\!]^{\mathrm{tc};\#C''_1,\ldots,C''_{n''};C'''_1,\ldots,C'''_{n'''}}_{C_1,\ldots,C_n}$: variant totally closed up to $\{C'''_1, \ldots, C'''_{n'''}\} \subset \{C'_1, \ldots, C'_{n'}\}$.

Architectural Compatibility of Star-Shaped Topologies

- A star is a portion of the abstract enriched flow graph of an architectural description $\mathcal{A}$, which is not part of a cyclic subgraph and is formed by:
  - The central AEI K.
  - The border $\mathcal{B}_K = \{C_1, \ldots, C_n\}$ of all the AEIs attached to K.
- MismDet investigates the validity of a property $\mathcal{P} \in \Psi$ by analyzing the interplay between the central AEI K and each of the AEIs in the border, as there cannot be attachments among AEIs in the border.
- Coordination is ensured if the actual observable behavior of every $C_j \in \mathcal{B}_K$ coincides with the observable behavior expected by K.
- K is compatible with C_j if the observable behavior of K is not altered by the insertion of C_j into the border of the star.

- The center K of the star is $\mathcal{P}$-compatible with $C_j \in \mathcal{B}_K$ iff:

$$\boxed{(\llbracket K \rrbracket_{\mathcal{A}}^{\mathrm{pc};\#C_j} \parallel_{\mathcal{S}(K,C_j;\mathcal{A})} \llbracket C_j \rrbracket_{K,\mathcal{B}_K}^{\mathrm{tc};\#K}) \,/\, (H_{K,C_j} \cup E_{K,C_j}) \approx_{\mathcal{P}} \llbracket K \rrbracket_{\mathcal{A}}^{\mathrm{pc;wob}}}$$

- H_{K,C_j}: set of interactions of additional implicit AEIs of K that are attached to interactions of C_j.
- E_{K,C_j}: set of exceptions that may be raised by semi-sync. interactions involved in attachments between K and C_j.
- The totally closed interacting semantics of C_j with respect to the AEIs inside the star hides:
  - ⊙ All possible originally asynchronous local interactions of C_j.
  - ⊙ All interactions of C_j possibly attached to AEIs outside the star.
- $H_{K,C_j} \cup E_{K,C_j} = \emptyset$ whenever there are no nonsynchronous interactions involved in attachments inside the star, in which case all interacting semantics between $\llbracket K \rrbracket_{\mathcal{A}}^{\mathrm{pc};\#\mathcal{A}}$ and $\llbracket K \rrbracket_{\mathcal{A}}^{\mathrm{pc;wob}}$ coincide with $\llbracket K \rrbracket_{\mathcal{A}}^{\mathrm{tc;wob}}$.

- Whenever K is $\mathcal{P}$-compatible with every $C_j \in \mathcal{B}_K$, then:

$$\boxed{[\![K, \mathcal{B}_K]\!]^{\mathrm{tc};\#K,\mathcal{B}_K;K}_{K,\mathcal{B}_K} \,/\, \bigcup_{j=1}^{n} (H_{K,C_j} \cup E_{K,C_j}) \approx_{\mathcal{P}} [\![K]\!]^{\mathrm{pc;wob}}_{\mathcal{A}}}$$

  thus $[\![K, \mathcal{B}_K]\!]^{\mathrm{tc};\#K,\mathcal{B}_K;K}_{K,\mathcal{B}_K} / \bigcup_{j=1}^{n} (H_{K,C_j} \cup E_{K,C_j})$ sat. $\mathcal{P}$ iff so does $[\![K]\!]^{\mathrm{pc;wob}}_{\mathcal{A}}$.

- If the $\mathcal{P}$-compatibility check is passed with respect to every $C_j \in \mathcal{B}_K$, then the star can be reduced to its center K (and this is verified against $\mathcal{P}$).

- If the $\mathcal{P}$-compatibility check is not passed with respect to some $C_g \in \mathcal{B}_K$, then this reveals a potential lack of coordination between K and C_g.

- The cost of the $\mathcal{P}$-compatibility-check-based verification grows linearly with the size of $\mathcal{B}_K$, while the cost of directly verifying the whole star against $\mathcal{P}$ grows exponentially with the size of $\{K\} \cup \mathcal{B}_K$.

Case Study: Compressing Proxy System

- Improving the performance of a Unix-based web browser over a slow network by causing the HTTP server to compress data with the gzip program before sending them across the network.
- The HTTP server is a series of filters strung together that communicate through a function-call-based stream interface that allows an upstream filter to push data into a downstream filter.
- The gzip program is instead a Unix filter communicating through pipes.
- Assembling the compressing proxy system from the existing HTTP server and the gzip program as they are, i.e., without modifications.
- Objective: design the system in such a way to ensure deadlock freedom.

- The gzip program explicitly chooses when to get data, while the HTTP filters are forced to read when data are pushed to them.
- The gzip program may attempt to output a portion of the compressed data before finishing getting all the input data (internal buffer full).
- Need for a software adaptor between two consecutive filters of the HTTP server because of the different communication mechanisms of the two software units.
- The adaptor has to redirect to the gzip program data coming from the upstream filter and to deliver to the downstream filter compressed data produced by the gzip program.
- Textual description header:

```
ARCHI_TYPE Compressing_Proxy(void)
```

- Definition of the upstream filter AET:

```
ARCHI_ELEM_TYPE U_Filter_Type(void)

  BEHAVIOR
    U_Filter(void; void) =
      write_data . U_Filter()

  INPUT_INTERACTIONS  void
  OUTPUT_INTERACTIONS SYNC UNI write_data
```

- Definition of the downstream filter AET:

```
ARCHI_ELEM_TYPE D_Filter_Type(void)

  BEHAVIOR
    D_Filter(void; void) =
      read_data . D_Filter()

  INPUT_INTERACTIONS  SYNC UNI read_data
  OUTPUT_INTERACTIONS void
```

- Definition of the adaptor AET:

```
ARCHI_ELEM_TYPE Adaptor_Type(void)

  BEHAVIOR
    Adaptor_From_Filter(void; void) =
      receive_from_filter . put_to_gzip . Adaptor_To_Gzip();
    Adaptor_To_Gzip(void; void) =
      choice
      {
        put_to_gzip . Adaptor_To_Gzip(),
        put_eoi_gzip . get_from_gzip . Adaptor_From_Gzip()
      };
    Adaptor_From_Gzip(void; void) =
      choice
      {
        get_from_gzip . Adaptor_From_Gzip(),
        get_eoo_gzip . Adaptor_To_Filter()
      };
    Adaptor_To_Filter(void; void) =
      send_to_filter . Adaptor_From_Filter()

  INPUT_INTERACTIONS  SYNC UNI receive_from_filter; get_from_gzip; get_eoo_gzip
  OUTPUT_INTERACTIONS SYNC UNI send_to_filter; put_to_gzip; put_eoi_gzip
```

- Definition of the gzip AET:

```
ARCHI_ELEM_TYPE Gzip_Type(void)

  BEHAVIOR
    Gzip(void; void) =
      get_data . Gzip_In();
    Gzip_In(void; void) =
      choice
      {
        get_data . Gzip_In(),
        get_eoi . compress . put_data . Gzip_Out(),
        saturate_buffer . compress . put_data . Gzip_Out()
      };
    Gzip_Out(void; void) =
      choice
      {
        put_data . Gzip_Out(),
        put_eoo . Gzip()
      }

  INPUT_INTERACTIONS  SYNC UNI get_data; get_eoi
  OUTPUT_INTERACTIONS SYNC UNI put_data; put_eoo
```

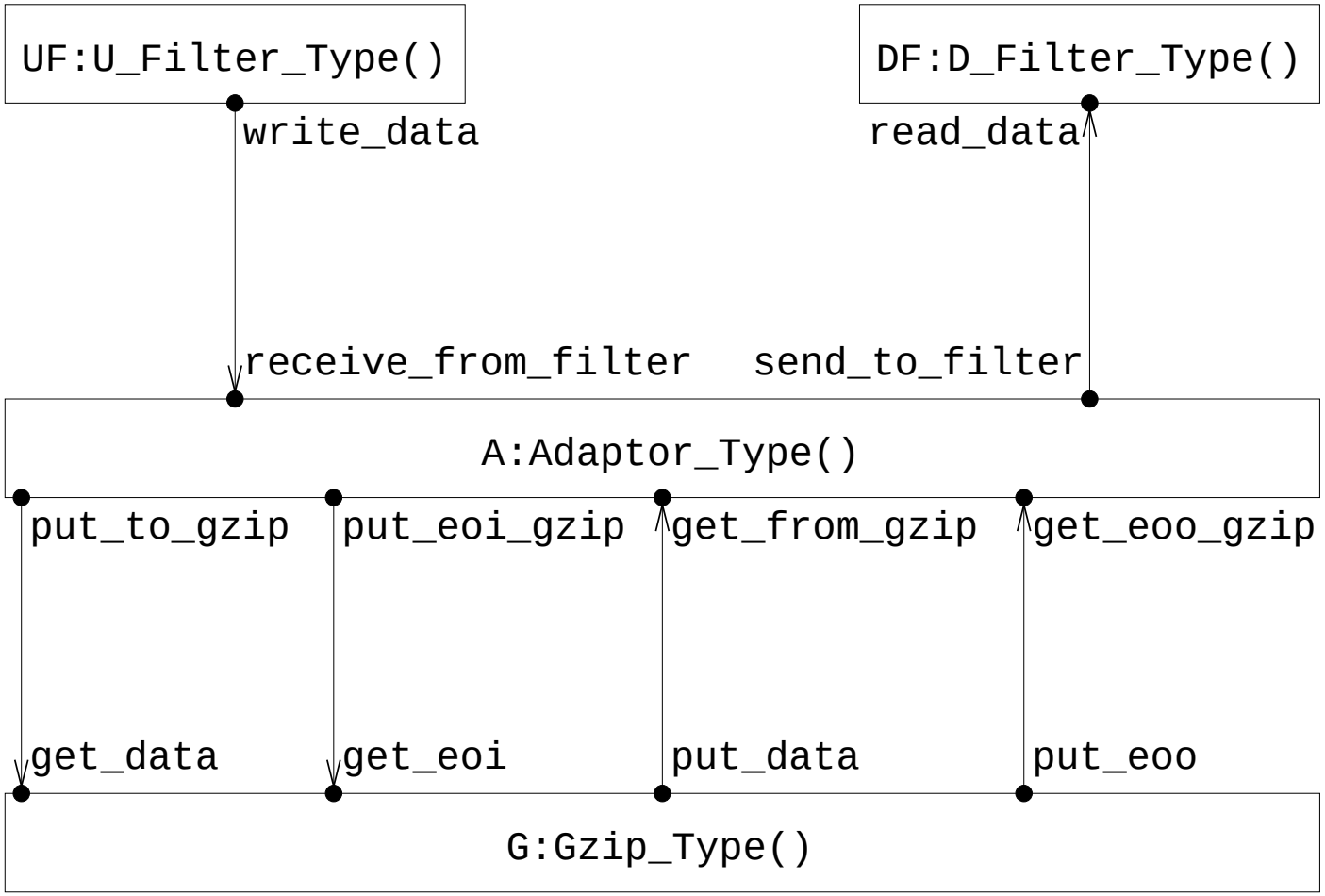
- Declaration of the topology:

```
ARCHI_ELEM_INSTANCES
  UF : U_Filter_Type();
  DF : D_Filter_Type();
  A  : Adaptor_Type();
  G  : Gzip_Type()

ARCHI_INTERACTIONS
  void

ARCHI_ATTACHMENTS
  FROM UF.write_data    TO A.receive_from_filter;
  FROM A.put_to_gzip    TO G.get_data;
  FROM A.put_eoi_gzip   TO G.get_eoi;
  FROM G.put_data       TO A.get_from_gzip;
  FROM G.put_eoo        TO A.get_eoo_gzip;
  FROM A.send_to_filter TO DF.read_data
```

- Enriched flow graph:



- The abstract enriched flow graph is a star whose central AEI is `A`.
- The architectural compatibility check can be applied by taking the totally closed interacting semantics without buffers of the various AEIs as there are no local nonsynchronous interactions in the star.
- `A` is not deadlock-freedom-compatible with `G` because:

$$[\![\mathtt{A}]\!]^{\mathrm{tc;wob}}_{\mathtt{A,UF,DF,G}} \parallel_{\mathcal{S}(\mathtt{A,G;A,UF,DF,G})} [\![\mathtt{G}]\!]^{\mathrm{tc;wob}}_{\mathtt{A,UF,DF,G}} \not\approx_{\mathrm{B}} [\![\mathtt{A}]\!]^{\mathrm{tc;wob}}_{\mathtt{A,UF,DF,G}}$$

- Distinguishing weak HML formula:

$$\langle\!\langle \mathtt{UF.write\_data\#A.receive\_from\_filter} \rangle\!\rangle \langle\!\langle \mathtt{A.put\_to\_gzip\#G.get\_data} \rangle\!\rangle \neg \langle\!\langle \mathtt{A.put\_eoi\_gzip\#G.get\_eoi} \rangle\!\rangle \ \mathrm{true}$$

- Actual deadlock when `G` autonomously decides to start sending compressed data back to `A` because of buffer saturation.
- Circular waiting because `G` can send compressed data to `A` if and only if `A` has signaled eoi to `G`.
- Solution:
  - `A` must be redesigned in order to account for the possibility of receiving compressed data from `G` before signaling end of input.
  - `G` must inform `A` about its intention to start sending compressed data in advance.

- Redefinition of the adaptor AET:

```
ARCHI_ELEM_TYPE Adaptor_Type(void)

  BEHAVIOR
    Adaptor_From_Filter(void; void) =
      receive_from_filter . put_to_gzip . Adaptor_To_Gzip();
    Adaptor_To_Gzip(void; void) =
      choice
      {
        put_to_gzip . Adaptor_To_Gzip(),
        put_eoi_gzip . get_from_gzip . Adaptor_From_Gzip(),
        notified_buffer_full . get_from_gzip . Adaptor_Suspended()
      };
    Adaptor_Suspended(void; void) =
      choice
      {
        get_from_gzip . Adaptor_Suspended(),
        get_eoo_gzip . put_to_gzip . Adaptor_To_Gzip()
      };
```

```
  Adaptor_From_Gzip(void; void) =
    choice
    {
      get_from_gzip . Adaptor_From_Gzip(),
      get_eoo_gzip . Adaptor_To_Filter()
    };
  Adaptor_To_Filter(void; void) =
    send_to_filter . Adaptor_From_Filter()

INPUT_INTERACTIONS  SYNC UNI receive_from_filter;
                             get_from_gzip; get_eoo_gzip;
                             notified_buffer_full
OUTPUT_INTERACTIONS SYNC UNI send_to_filter;
                             put_to_gzip; put_eoi_gzip
```

- Redefinition of the gzip AET:

```
ARCHI_ELEM_TYPE Gzip_Type(void)

  BEHAVIOR
    Gzip(void; void) =
      get_data . Gzip_In();
    Gzip_In(void; void) =
      choice
      {
        get_data . Gzip_In(),
        get_eoi . compress . put_data . Gzip_Out(),
        notify_buffer_full . compress . put_data . Gzip_Out()
      };
    Gzip_Out(void; void) =
      choice
      {
        put_data . Gzip_Out(),
        put_eoo . Gzip()
      }

  INPUT_INTERACTIONS  SYNC UNI get_data; get_eoi
  OUTPUT_INTERACTIONS SYNC UNI put_data; put_eoo;
                               notify_buffer_full
```

- Redeclaration of the attachments:

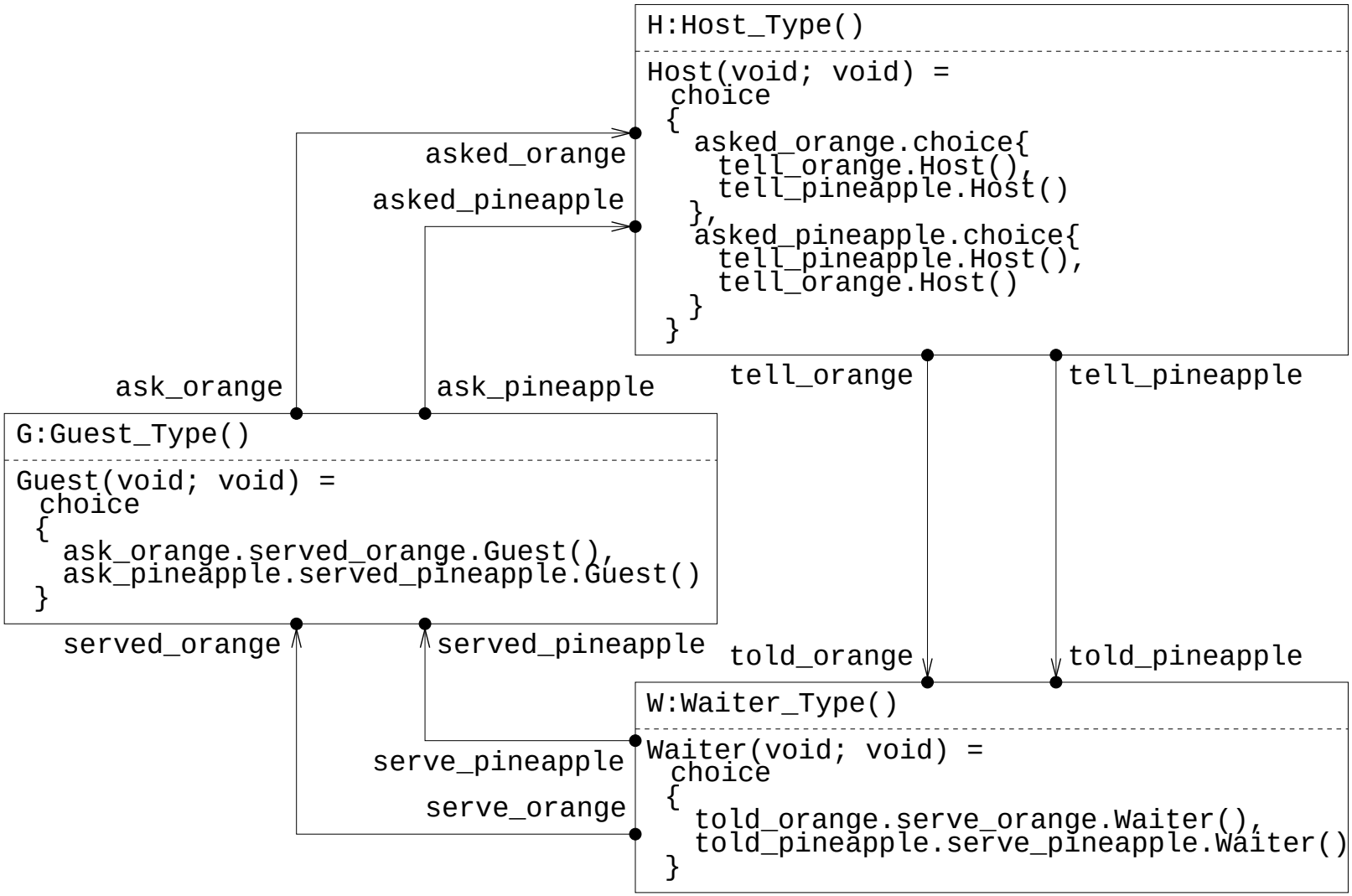
```
ARCHI_ATTACHMENTS
  FROM UF.write_data          TO A.receive_from_filter;
  FROM A.put_to_gzip          TO G.get_data;
  FROM A.put_eoi_gzip         TO G.get_eoi;
  FROM G.notify_buffer_full   TO A.notified_buffer_full;
  FROM G.put_data             TO A.get_from_gzip;
  FROM G.put_eoo              TO A.get_eoo_gzip;
  FROM A.send_to_filter       TO DF.read_data
```

- Now `A` is deadlock-freedom-compatible with `G`.
- `A` is also deadlock-freedom-compatible with `UF` and `DF`.
- Then `Compressing_Proxy` is deadlock free because so is `A`.

Architectural Interoperability of Cycle-Shaped Topologies

- The compatibility check is not enough when dealing with cycles.
- **Example**: guest analogy.
- A party involves three kinds of actor: guest, host, waiter.
- The guest can ask the host for an orange juice or a pineapple juice.
- The host is expected to tell the waiter to bring the requested drink.
- What if the host is absentminded or malicious, and hence tells the waiter to bring a drink different from the one requested by the guest?

- Enriched flow graph integrated with behavioral description:



- Can the party deadlock?
- Deadlock corresponds to the embarrassing situation in which the guest receives a drink different from the requested one, as the guest may then be led to refrain from asking for further drinks.
- `H` is deadlock-freedom-compatible both with `W` and with `G`.
- `H` is deadlock free.
- But the overall system is not deadlock free!
- The abstract enriched flow graph is a cycle, not a star!
- This example demonstrates that the architectural compatibility check does not suffice in the presence of cycles.

- A cycle is a closed simple path in the abstract enriched flow graph of an architectural description $\mathcal{A}$, which traverses a set $\mathcal{Y} = \{C_1, \ldots, C_n\}$ of $n \geq 3$ AEIs.
- Each AEI in the cycle may arbitrarily interfere with any of the other AEIs in the cycle.
- When MismDet investigates a property $\mathcal{P} \in \Psi$, the AEIs traversed by the cycle cannot be considered two-by-two as in the compatibility check.
- Coordination is ensured if the actual observable behavior of any AEI C_j in the cycle coincides with the observable behavior expected by the rest of the cycle.
- C_j interoperates with the rest of the cycle if the observable behavior of C_j is not altered by the insertion of C_j itself into the cycle.

- C_j $\mathcal{P}$-interoperates with the other AEIs in the cycle iff:

$$[\![\mathcal{Y}]\!]_{\mathcal{A}}^{\mathrm{tc};\#\mathcal{Y};C_j} \,/\, (Name - \mathcal{V}_{C_j;\mathcal{A}}) \,/\, (H_{C_j,\mathcal{Y}} \cup E_{C_j,\mathcal{Y}}) \approx_{\mathcal{P}} [\![C_j]\!]_{\mathcal{A}}^{\mathrm{pc};\mathrm{wob}}$$

- $H_{C_j,\mathcal{Y}}$: set of interactions of additional implicit AEIs of C_j that are attached to $\mathcal{Y}$.
- $E_{C_j,\mathcal{Y}}$: set of exceptions that may be raised by semi-sync. interactions involved in attachments between C_j and $\mathcal{Y}$.
- The totally closed interacting semantics of the other AEIs in the cycle and visibility restricted to $\mathcal{V}_{C_j;\mathcal{A}}$ hide:
  - ⊙ All possible originally asynchronous local interactions of those AEIs.
  - ⊙ All interactions of those AEIs that are not attached to C_j.
- Whenever C_j has no local nonsynchronous interactions and is not attached to semi-synchronous interactions of other AEIs in the cycle, then $H_{C_j,\mathcal{Y}} \cup E_{C_j,\mathcal{Y}} = \emptyset$ and both $[\![C_j]\!]_{\mathcal{A}}^{\mathrm{pc};\#\mathcal{Y}}$ and $[\![C_j]\!]_{\mathcal{A}}^{\mathrm{pc};\mathrm{wob}}$ coincide with $[\![C_j]\!]_{\mathcal{A}}^{\mathrm{tc};\mathrm{wob}}$.

- Interoperability is an adaptation of compatibility to cycles:

$$
\begin{array}{rcl}
(\llbracket C_1 \rrbracket_{\mathcal{A}}^{\mathrm{pc};\#\mathcal{Y}} \parallel_{\overline{S}_{1,n}} \llbracket C_2, \ldots, C_n \rrbracket_{\mathcal{A}}^{\mathrm{tc};\#\mathcal{Y}}) \,/\, NV_1 & \approx_{\mathcal{P}} & \llbracket C_1 \rrbracket_{\mathcal{A}}^{\mathrm{pc;wob}} \\
\vdots & \vdots & \vdots \\
(\llbracket \ldots, C_{j-1} \rrbracket_{\mathcal{A}}^{\mathrm{tc};\#\mathcal{Y}} \parallel_{S_{j-1}} \llbracket C_j \rrbracket_{\mathcal{A}}^{\mathrm{pc};\#\mathcal{Y}} \parallel_{\overline{S}_{j,n}} \llbracket C_{j+1}, \ldots \rrbracket_{\mathcal{A}}^{\mathrm{tc};\#\mathcal{Y}}) \,/\, NV_j & \approx_{\mathcal{P}} & \llbracket C_j \rrbracket_{\mathcal{A}}^{\mathrm{pc};wob} \\
\vdots & \vdots & \vdots \\
(\llbracket C_1, \ldots, C_{n-1} \rrbracket_{\mathcal{A}}^{\mathrm{tc};\#\mathcal{Y}} \parallel_{S_{n-1}} \llbracket C_n \rrbracket_{\mathcal{A}}^{\mathrm{pc};\#\mathcal{Y}}) \,/\, NV_n & \approx_{\mathcal{P}} & \llbracket C_n \rrbracket_{\mathcal{A}}^{\mathrm{pc;wob}}
\end{array}
$$

where:

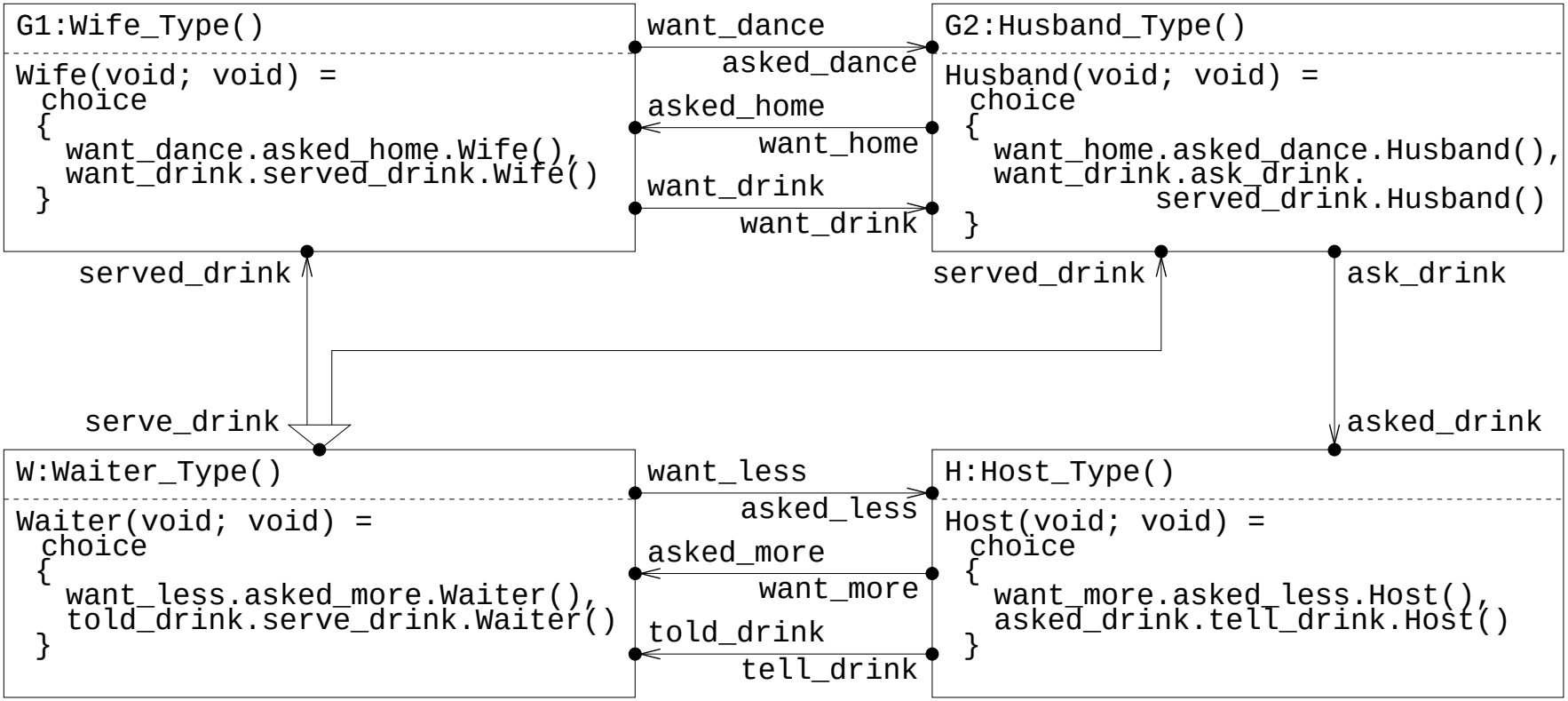
- $\overline{S}_{j,n} = \bigcup_{f=1}^{j} \bigcup_{g=j+1}^{n} \mathcal{S}(C_f, C_g; \mathcal{A})$.
- $S_j = \bigcup_{g=1}^{j} \mathcal{S}(C_g, C_{j+1}; \mathcal{A})$.
- $NV_j = (Name - \mathcal{V}_{C_j;\mathcal{A}}) \cup (H_{C_j,\mathcal{Y}} \cup E_{C_j,\mathcal{Y}})$.

- Whenever there exists $C_j \in \mathcal{Y}$ that $\mathcal{P}$-interoperates with the other AEIs in the cycle, then $[\![\mathcal{Y}]\!]_{\mathcal{A}}^{\mathrm{tc};\#\mathcal{Y};C_j} \,/\, (Name - \mathcal{V}_{C_j;\mathcal{A}}) \,/\, (H_{C_j,\mathcal{Y}} \cup E_{C_j,\mathcal{Y}})$ satisfies $\mathcal{P}$ iff so does $[\![C_j]\!]_{\mathcal{A}}^{\mathrm{pc;wob}}$.
- If the $\mathcal{P}$-interoperability check is passed by some $C_j \in \mathcal{Y}$, then the cycle can be reduced to C_j (and this is verified against $\mathcal{P}$).
- If the $\mathcal{P}$-interoperability check is not passed by any $C_g \in \mathcal{Y}$, then this reveals a potential lack of coordination among the AEIs in the cycle.
- The cost of the $\mathcal{P}$-interoperability-check-based verification grows exponentially with the size of $\mathcal{Y}$.
- Can be mitigated if the state space of the cycle is built compositionally and minimized at each step with respect to $\approx_{\mathcal{P}}$ by taking into account the interfaces of the various AEIs in the cycle (numerous actions to be hidden).

- Not immediately clear which AEIs are responsible for possible well-formedness violations whenever no AEI in a cycle passes the architectural interoperability check.
- Cycle shrinking algorithm:
  - Consider an AEI $C_g \in \mathcal{Y}$ that does not $\mathcal{P}$-interoperate with the other AEIs in the cycle.
  - Use the modal-logic diagnostic information coming from the failure of the $\mathcal{P}$-interoperability check for C_g in order to determine where the source of a potential well-formedness violation is.
  - If within C_g, modify C_g and repeat the $\mathcal{P}$-interoperability check.
  - Otherwise shrink the cycle by replacing C_{g-1}, C_g, and C_{g+1} with a new AEI whose behavior is the parallel composition of the semantics of the three original AEIs, then repeat the $\mathcal{P}$-interoperability check.

- Considering sets of adjacent AEIs instead of single AEIs can also be exploited for improving the effectiveness of the interoperability check.
- **Example**: guest analogy revisited.
- Party with four actors: couple of guests, host, and waiter.
- As before, whenever a guest wants something to drink, the guest has to ask the host.
- Differently from before, the drink mentioned to the waiter by the host always coincides with the drink requested by the guest.
- First conflict: the wife wants to dance with her husband, whereas the husband wants to go home (solved by means of an agreement on having another drink).
- Second conflict: the host asks the waiter to work one more hour, whereas the waiter asks the host to quit working earlier (service takes precedence).

- Enriched flow graph integrated with behavioral description (only conflicts):



- Can the party deadlock?
- None of the AEIs deadlock-freedom-interoperates with the other AEIs in the cycle because of the wife-husband and host-waiter conflicts.
- However the AEI resulting from the parallel composition of `G1` and `G2` deadlock-freedom-interoperates with `H` and `W`!
- Therefore the party does not deadlock!
- This example demonstrates that the outcome of the architectural interoperability check depends on the number of adjacent AEIs that are considered during the application of the check.

- Given a cycle $\mathcal{Y} = \{C_1, \ldots, C_n\}$, take a subset of l adjacent AEIs $\mathcal{J} = \{C'_1, \ldots, C'_l\}$ in the cycle with $1 \leq l \leq n/2$ (limit to $n/2$ due to symmetry).
- Let $\mathcal{T} = \mathcal{Y} - \mathcal{J}$ be the set of the other AEIs in the cycle.
- The l adjacent AEIs interoperate with the rest of the cycle if the observable behavior of those AEIs is not altered by the insertion of the AEIs themselves into the cycle.
- $\mathcal{J} = \{C'_1, \ldots, C'_l\}$ $\mathcal{P}$-interoperates with the other AEIs in the cycle iff:

$$[\![\mathcal{Y}]\!]_{\mathcal{A}}^{\mathrm{tc};\#\mathcal{Y};\mathcal{J}} \,/\, (Name - \bigcup_{j=1}^{l} \mathcal{V}_{C'_j;\mathcal{A}}) \,/\, \bigcup_{j=1}^{l} (H_{C'_j,\mathcal{T}} \cup E_{C'_j,\mathcal{T}}) \approx_{\mathcal{P}} [\![\mathcal{J}]\!]_{\mathcal{A}}^{\mathrm{pc};\#\mathcal{J}}$$

- Whenever there exists $\mathcal{J} = \{C'_1, \ldots, C'_l\} \subseteq \mathcal{Y}$ that $\mathcal{P}$-interoperates with the other AEIs in the cycle, then $[\![\mathcal{Y}]\!]_{\mathcal{A}}^{\mathrm{tc};\#\mathcal{Y};\mathcal{J}} \,/\, (Name - \bigcup_{j=1}^{l} \mathcal{V}_{C'_j;\mathcal{A}}) \,/\, \bigcup_{j=1}^{l} (H_{C'_j,\mathcal{T}} \cup E_{C'_j,\mathcal{T}})$ satisfies $\mathcal{P}$ iff so does $[\![\mathcal{J}]\!]_{\mathcal{A}}^{\mathrm{pc};\#\mathcal{J}}$.

Case Study: Cruise Control System

- Standard automobile equipped with two pedals: accelerator, brake.
- Cruise control system governed through three buttons: on, off, resume.
- When on is pressed, the cruise control system records the current speed and then maintains the automobile at that speed.
- When the accelerator, the brake, or off is pressed, the cruise control system disengages but retains the speed setting.
- If resume is pressed later on, then the cruise control system accelerates or decelerates the automobile to the previously recorded speed.
- Objective: design the system in such a way to ensure deadlock freedom.

- Software units: sensor, speed detector, speed controller, speed actuator.
- The sensor forwards the driver commands to the speed controller.
- The speed detector periodically measures the number of wheel revolutions per time unit and communicates it to the speed actuator.
- The speed controller triggers the speed actuator on the basis of the driver commands that are received from the sensor (inactive, active, cruising, suspended).
- The speed actuator adjusts the throttle on the basis of the triggers received from the controller and of the speed measured by the detector (disabled, enabled).
- Textual description header:

```
ARCHI_TYPE Cruise_Control(void)
```

- Definition of the sensor AET:

```
ARCHI_ELEM_TYPE Sensor_Type(void)

  BEHAVIOR
    Sensor_Off(void; void) =
      detected_engine_on . turn_engine_on . Sensor_On();
    Sensor_On(void; void) =
      choice
      {
        detected_accelerator . press_accelerator . Sensor_On(),
        detected_brake . press_brake . Sensor_On(),
        detected_on . press_on . Sensor_On(),
        detected_off . press_off . Sensor_On(),
        detected_resume . press_resume . Sensor_On(),
        detected_engine_off . turn_engine_off . Sensor_Off()
      }

  INPUT_INTERACTIONS  SYNC UNI detected_engine_on; detected_engine_off;
                               detected_accelerator; detected_brake;
                               detected_on; detected_off; detected_resume
  OUTPUT_INTERACTIONS SYNC UNI press_accelerator; press_brake;
                               press_on; press_off; press_resume
                          AND turn_engine_on; turn_engine_off
```

- Definition of the speed detector AET:

```
ARCHI_ELEM_TYPE Detector_Type(void)

  BEHAVIOR
    Detector_Off(void; void) =
      turned_engine_on . Detector_On();
    Detector_On(void; void) =
      choice
      {
        measure_speed . signal_speed . Detector_On(),
        turned_engine_off . Detector_Off()
      }

  INPUT_INTERACTIONS  SYNC UNI turned_engine_on; turned_engine_off
  OUTPUT_INTERACTIONS SYNC UNI signal_speed
```

- Definition of the speed controller AET:

```
ARCHI_ELEM_TYPE Controller_Type(void)

  BEHAVIOR
    Inactive(void; void) =
      turned_engine_on . Active();
    Active(void; void) =
      choice
      {
        pressed_accelerator . Active(),
        pressed_brake . Active(),
        pressed_on . trigger_record . Cruising(),
        pressed_off . Active(),
        pressed_resume . Active(),
        turned_engine_off . Inactive()
      };
```

```
Cruising(void; void) =
  choice
  {
    pressed_accelerator . trigger_disable . Suspended(),
    pressed_brake . trigger_disable . Suspended(),
    pressed_on . Cruising(),
    pressed_off . trigger_disable . Suspended(),
    pressed_resume . Cruising(),
    turned_engine_off . Inactive()
  };
```

```
  Suspended(void; void) =
    choice
    {
      pressed_accelerator . Suspended(),
      pressed_brake . Suspended(),
      pressed_on . trigger_record . Cruising(),
      pressed_off . Suspended(),
      pressed_resume . trigger_resume . Cruising(),
      turned_engine_off . Inactive()
    }

INPUT_INTERACTIONS  SYNC UNI turned_engine_on; turned_engine_off;
                             pressed_accelerator; pressed_brake;
                             pressed_on; pressed_off; pressed_resume
OUTPUT_INTERACTIONS SYNC UNI trigger_record; trigger_resume;
                             trigger_disable
```

- Definition of the speed actuator AET:

```
ARCHI_ELEM_TYPE Actuator_Type(void)

  BEHAVIOR
    Disabled(void; void) =
      choice
      {
        signaled_speed . Disabled(),
        triggered_record . record_speed . Enabled(),
        triggered_resume . resume_speed . Enabled()
      };
    Enabled(void; void) =
      choice
      {
        signaled_speed . adjust_throttle . Enabled(),
        triggered_disable . disable_control . Disabled()
      }

  INPUT_INTERACTIONS  SYNC UNI triggered_record; triggered_resume;
                               triggered_disable;
                               signaled_speed
  OUTPUT_INTERACTIONS void
```

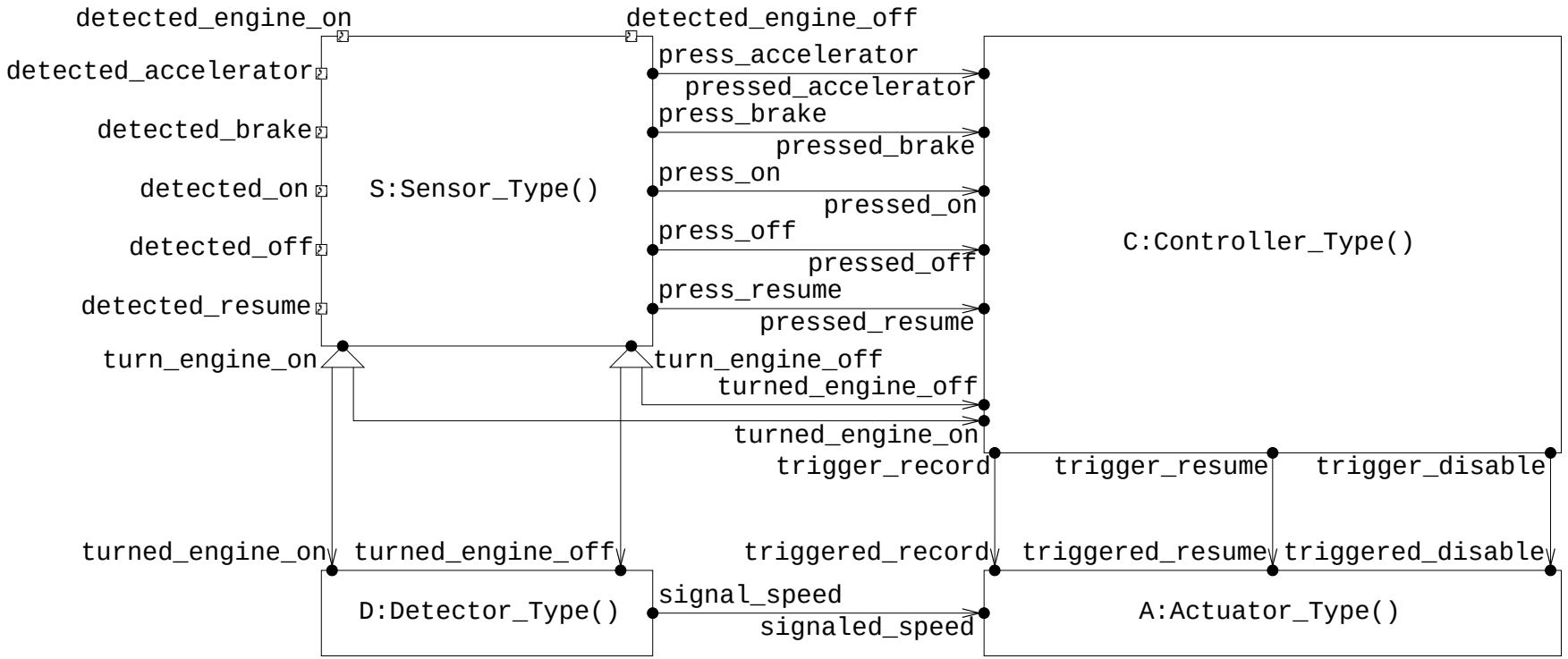
- Declaration of the topology:

```
ARCHI_ELEM_INSTANCES
  S : Sensor_Type();
  D : Detector_Type();
  C : Controller_Type();
  A : Actuator_Type()

ARCHI_INTERACTIONS
  S.detected_engine_on; S.detected_engine_off;
  S.detected_accelerator; S.detected_brake;
  S.detected_on; S.detected_off; S.detected_resume
```

```
ARCHI_ATTACHMENTS
  FROM S.turn_engine_on    TO D.turned_engine_on;
  FROM S.turn_engine_on    TO C.turned_engine_on;
  FROM S.turn_engine_off   TO D.turned_engine_off;
  FROM S.turn_engine_off   TO C.turned_engine_off;
  FROM S.press_accelerator TO C.pressed_accelerator;
  FROM S.press_brake       TO C.pressed_brake;
  FROM S.press_on          TO C.pressed_on;
  FROM S.press_off         TO C.pressed_off;
  FROM S.press_resume      TO C.pressed_resume;
  FROM D.signal_speed      TO A.signaled_speed;
  FROM C.trigger_record    TO A.triggered_record;
  FROM C.trigger_resume    TO A.triggered_resume;
  FROM C.trigger_disable   TO A.triggered_disable
```

- Enriched flow graph:



- The abstract enriched flow graph is a cycle formed by `S`, `D`, `C`, `A`.
- The architectural interoperability check can be applied by taking the totally closed interacting semantics without buffers of the various AEIs as there are no local nonsynchronous interactions in the cycle.
- All the AEIs in the cycle are deadlock free but none of them deadlock-freedom-interoperates with the others.
- We can suspect that some mismatch exists.
- `Cruise_Control` is deadlock free because speed signaling activities take place endlessly between `D` and `A` as long as the engine is running.
- If we hide those activities, a deadlock shows up.
- Those activities are hidden in the interoperability checks for `S` and `C`, so we can try to derive useful diagnostic information from such checks.

- S does not deadlock-freedom-interoperate with D, C, A because:

$$[\![\mathtt{S},\mathtt{D},\mathtt{C},\mathtt{A}]\!]^{\mathrm{tc;wob}}_{\mathtt{S},\mathtt{D},\mathtt{C},\mathtt{A}} \,/\, (\mathit{Name} - \mathcal{V}_{\mathtt{S};\mathtt{S},\mathtt{D},\mathtt{C},\mathtt{A}}) \;\not\approx_{\mathrm{B}}\; [\![\mathtt{S}]\!]^{\mathrm{tc;wob}}_{\mathtt{S},\mathtt{D},\mathtt{C},\mathtt{A}}$$

- Distinguishing weak HML formula:

$$\langle\!\langle \mathtt{S.turn\_engine\_on\#D.turned\_engine\_on\#C.turned\_engine\_on} \rangle\!\rangle$$
$$\langle\!\langle \mathtt{S.press\_on\#C.pressed\_on} \rangle\!\rangle$$
$$\langle\!\langle \mathtt{S.turn\_engine\_off\#D.turned\_engine\_off\#C.turned\_engine\_off} \rangle\!\rangle$$
$$\langle\!\langle \mathtt{S.turn\_engine\_on\#D.turned\_engine\_on\#C.turned\_engine\_on} \rangle\!\rangle$$
$$\langle\!\langle \mathtt{S.press\_on\#C.pressed\_on} \rangle\!\rangle$$
$$\neg \langle\!\langle \mathtt{S.press\_brake\#C.pressed\_brake} \rangle\!\rangle \; \mathrm{true}$$

- If we turn the engine on and off and in the meanwhile we press on, then the speed actuator remains enabled instead of being disengaged.
- Danger: cruise control system no longer sensitive to brake pressure!

- Solution: modify the cruising behavior of the speed controller AET so that a disabling trigger is sent also when the engine is turned off:

```
Cruising(void; void) =
  choice
  {
    pressed_accelerator . trigger_disable . Suspended(),
    pressed_brake . trigger_disable . Suspended(),
    pressed_on . Cruising(),
    pressed_off . trigger_disable . Suspended(),
    pressed_resume . Cruising(),
    turned_engine_off . trigger_disable . Inactive()
  }
```

- Now `S` deadlock-freedom-interoperates with the other AEIs.
- Then `Cruise_Control` is deadlock free even if we hide the speed signaling activities.

Generalization to Arbitrary Topologies

- The abstract enriched flow graph underlying an architectural description may contain arbitrarily many stars and cycles.
- MismDet applies the compatibility and interoperability checks several times in a way that hopefully converges towards the reduction of the entire topology to a single architectural element.
- Prominent role played by AEIs that belong to the intersection of cycles with acyclic portions of the topology or other cycles, as they can be exploited for reduction purposes.
- The reduction to any AEI in a cycle that does not interact with AEIs outside the cycle leads the process to a dead end.
- Frontier of $\{C_1, \ldots, C_n\}$:

$$\mathcal{F}_{C_1,\ldots,C_n} = \{C_j \in \{C_1, \ldots, C_n\} \mid \mathcal{LI}_{C_j;C_1,\ldots,C_n} \neq \mathcal{LI}_{C_j}\}$$

- The topological reduction process can take place in several different ways depending on the order in which the various stars and cycles of the topology are considered.
- All cycles have to be taken into account as the reduction process tends to transform an arbitrary topology into an acyclic topology.
- A cycle covering algorithm for $\mathcal{A}$ is defined as follows:
  1. All the AEIs of $\mathcal{A}$ are initially unmarked.
  2. While there are unmarked AEIs in the cycles of the abstract enriched flow graph of $\mathcal{A}$:
     a. Pick out one such AEI, say C.
     b. Mark all the AEIs in $\mathcal{CU}_C$.
- The cyclic union of C, denoted $\mathcal{CU}_C$, is the union of the sets of AEIs traversed by a cycle that traverses also C.

- The result of a cycle covering algorithm κ is a set $\mathcal{CU}(\kappa)$ of cyclic unions that include every AEI belonging to a cycle in the abstract enriched flow graph of $\mathcal{A}$.
- Any two cyclic unions in $\mathcal{CU}(\kappa)$ are connected at most through a single shared AEI or through the attachments between a single AEI of one cyclic union and a single AEI of the other cyclic union.
- The resulting set of cyclic unions $\mathcal{CU}(\kappa)$ is total iff the topology becomes acyclic after replacing every cyclic union $\mathcal{Y} = \{C_1, \ldots, C_n\} \in \mathcal{CU}(\kappa)$ with an AEI whose behavior is given by:

$$[\![\mathcal{Y}]\!]_{\mathcal{A}}^{\mathbf{tc};\#\mathcal{Y};\mathcal{F}_{C_1,\ldots,C_n}} \,/\, (Name - \bigcup_{C_j \in \mathcal{F}_{C_1,\ldots,C_n}} \mathcal{V}_{C_j;\mathcal{A}}) \,/ \bigcup_{C_j \in \mathcal{F}_{C_1,\ldots,C_n}} (H_{C_j,\mathcal{Y}} \cup E_{C_j,\mathcal{Y}})$$

- Let $\mathcal{A}$ be an architectural description for which the following conditions hold:
  1. For each $K \in \mathcal{A}$ belonging to an acyclic portion or to the intersection of some cycle with acyclic portions of the abstract enriched flow graph of $\mathcal{A}$, K is $\mathcal{P}$-compatible with every $C \in \mathcal{B}_K - \mathcal{CU}_K$.
  2. If $\mathcal{A}$ is cyclic, then there exists a cycle covering algorithm κ with total result such that for each cyclic union $\{C_1, \ldots, C_n\} \in \mathcal{CU}(\kappa)$:
     a. If $\mathcal{F}_{C_1,\ldots,C_n} = \emptyset$, there exists $C_j \in \{C_1, \ldots, C_n\}$ that $\mathcal{P}$-interoperates with the other AEIs in the cyclic union.
     b. If $\mathcal{F}_{C_1,\ldots,C_n} \neq \emptyset$, every $C_j \in \mathcal{F}_{C_1,\ldots,C_n}$ $\mathcal{P}$-interoperates with the other AEIs in the cyclic union.
     c. If no $C_j \in \mathcal{F}_{C_1,\ldots,C_n}$ is such that $[\![C_j]\!]_{\mathcal{A}}^{\mathrm{pc;wob}}$ satisfies $\mathcal{P}$ and there exists $C_g \in \{C_1, \ldots, C_n\} - \mathcal{F}_{C_1,\ldots,C_n}$ such that $[\![C_g]\!]_{\mathcal{A}}^{\mathrm{pc;wob}}$ satisfies $\mathcal{P}$, at least one such C_g $\mathcal{P}$-interoperates with the other AEIs in the cyclic union.

  Then $[\![\mathcal{A}]\!]_{\mathrm{bbm}}^{\mathrm{pc};\#\mathcal{A}}$ satisfies $\mathcal{P}$ iff so does $[\![C]\!]_{\mathcal{A}}^{\mathrm{pc;wob}}$ for some $C \in \mathcal{A}$.

Case Study: Simulator for Cruise Control System

- The applet has a panel with seven software buttons and a text area.
- The seven software buttons correspond to turning the engine on/off, pressing the accelerator/brake, and pressing on/off/resume.
- The text area shows the sequence of buttons successfully pressed so far.
- Each of the seven software buttons can be pressed at any time.
- If pressing one of them succeeds, the panel can interact with the sensor and the text area is updated accordingly.
- If it fails, the panel cannot interact with the sensor and emits a beep (e.g., pressing the accelerator button when the engine is off).
- Objective: design the system in such a way to ensure deadlock freedom.

- The description of the architectural type `Cruise_Control` can be reused thanks to the architectural interactions provided by the sensor.
- The applet is started/stopped by the user.
- Its panel sends the user commands to the sensor, which then propagates them inside the cruise control system.
- The simulator must not block whenever the pressure of a software button fails.
- The interactions of the panel (attached to the architectural interactions of the sensor) must then be semi-synchronous.
- Textual description header:

```
ARCHI_TYPE Cruise_Control_Simulator(void)
```

- Definition of the cruise control system AET (architectural invocation):

```
ARCHI_ELEM_TYPE Cruise_Control_System_Type(void)

  BEHAVIOR
    Cruise_Control_System(void; void) =
      Cruise_Control(@          /* no data parameters */
                     @          /* reuse formal AETs */
                     @ @ @ @    /* reuse formal topology */
                     @ @ @      /* no behavioral modifications */
                     UNIFY S.detected_engine_on   WITH engine_on;
                     UNIFY S.detected_engine_off  WITH engine_off;
                     UNIFY S.detected_accelerator WITH accelerator;
                     UNIFY S.detected_brake       WITH brake;
                     UNIFY S.detected_on          WITH on;
                     UNIFY S.detected_off         WITH off;
                     UNIFY S.detected_resume      WITH resume)

  INPUT_INTERACTIONS  SYNC UNI engine_on; engine_off;
                               accelerator; brake; on; off; resume
  OUTPUT_INTERACTIONS void
```

- Definition of the panel AET:

```
ARCHI_ELEM_TYPE Panel_Type(void)

  BEHAVIOR
    Unallocated(void; void) =
      init_applet . start_applet . Active();
    Active(void; void) =
      choice
      {
        signal_engine_on . Checking(signal_engine_on.success),
        signal_accelerator . Checking(signal_accelerator.success),
        signal_brake . Checking(signal_brake.success),
        signal_on . Checking(signal_on.success),
        signal_off . Checking(signal_off.success),
        signal_resume . Checking(signal_resume.success),
        signal_engine_off . Checking(signal_engine_off.success),
        stop_applet . Inactive()
      };
```

```
  Checking(boolean success; void) =
    choice
    {
      cond(success = true)  -> update . Active(),
      cond(success = false) -> beep . Active()
    };
  Inactive(void; void) =
    choice
    {
      start_applet . Active(),
      destroy_applet . Unallocated()
    }

INPUT_INTERACTIONS  SYNC  UNI init_applet; start_applet;
                              stop_applet; destroy_applet
OUTPUT_INTERACTIONS SSYNC UNI signal_engine_on; signal_engine_off;
                              signal_accelerator; signal_brake;
                              signal_on; signal_off; signal_resume
```

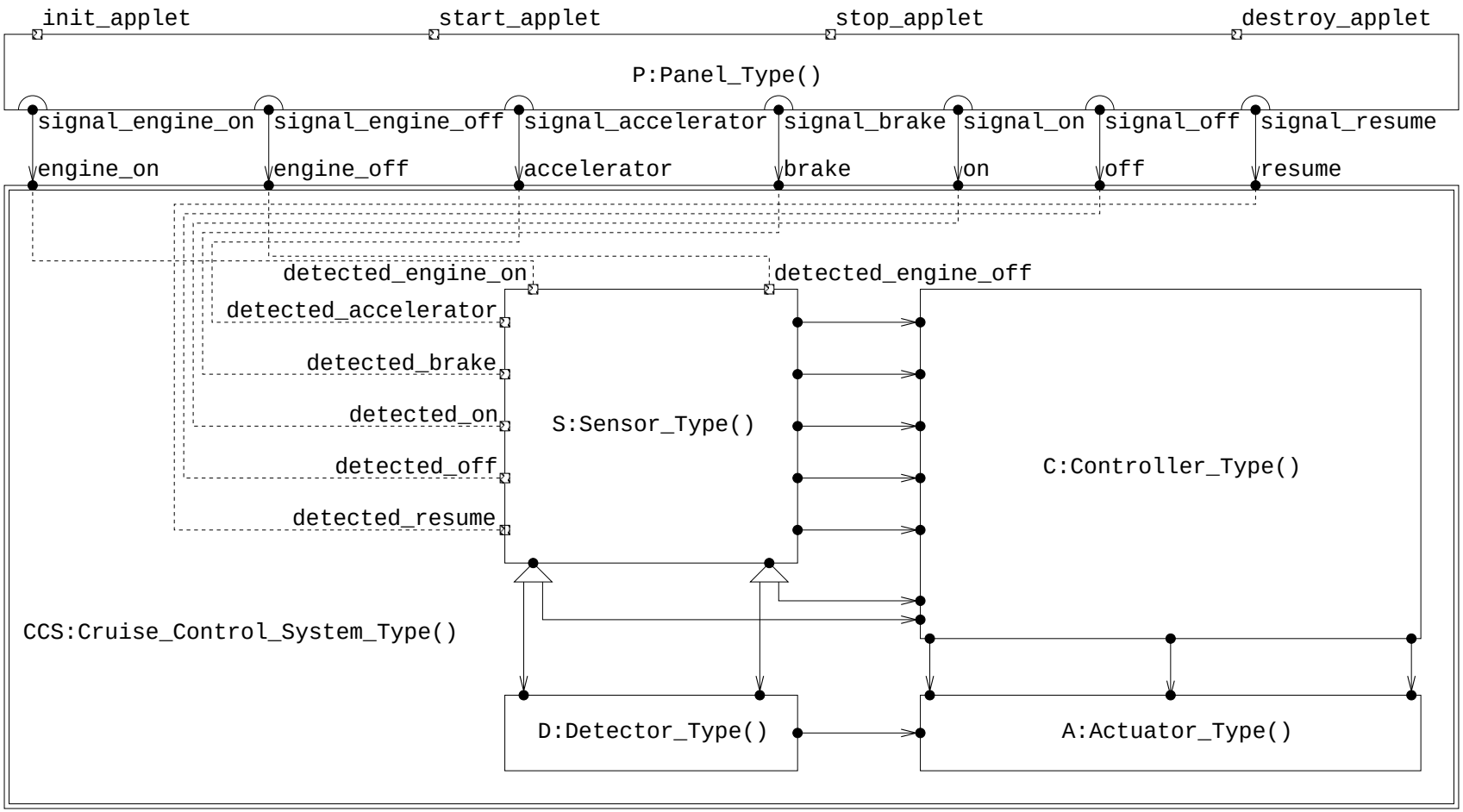
- Declaration of the topology:

```
ARCHI_ELEM_INSTANCES
  P   : Panel_Type();
  CCS : Cruise_Control_System_Type()

ARCHI_INTERACTIONS
  P.init_applet; P.start_applet;
  P.stop_applet; P.destroy_applet

ARCHI_ATTACHMENTS
  FROM P.signal_engine_on   TO CCS.engine_on;
  FROM P.signal_engine_off  TO CCS.engine_off;
  FROM P.signal_accelerator TO CCS.accelerator;
  FROM P.signal_brake       TO CCS.brake;
  FROM P.signal_on          TO CCS.on;
  FROM P.signal_off         TO CCS.off;
  FROM P.signal_resume      TO CCS.resume
```

- Enriched flow graph:



- `P`, `S`, `C`, `D`, `A` are all deadlock free.
- Consider the flattened topology where the architectural interactions provided by `S` have become local.
- Intersection of a cycle and a star.
- Cycle traversing `S`, `D`, `C`, `A` within `CCS`.
- `S` deadlock-freedom-interoperates with `D`, `C`, `A`, hence the cycle can be reduced to $[\![\texttt{S}]\!]^{\text{tc;wob}}_{\texttt{P,S,D,C,A}}$.
- `S`, the only AEI in the cycle frontier, is deadlock-freedom-compatible with `P`, the only AEI in the only acyclic portion of the topology (all exceptions that `P` may raise are hidden when applying the check).
- Then `Cruise_Control_Simulator` is deadlock free because so is `S`.

Generalization to Architectural Types

- The process algebraic techniques for architectural mismatch detection are not only useful for deriving information about an architectural description from star-shaped and cycle-shaped portions of its topology.
- These techniques can also be exploited for deducing information about an entire architectural type from the absence of architectural mismatches investigated by means of MismDet in one of its instances.
- This avoids the cost resulting from the application of the architectural compatibility and interoperability checks to every single instance of the considered architectural type.
- Several cases related to behavioral variations and topological variations.

- First case: *internal behavioral variations*.
- Two AT instances behaviorally conform to each other if they exhibit the same observable behavior according to $\approx_{\mathrm{B}}$.
- When introducing internal behavioral variations, the absence of architectural mismatches is then preserved for each property $\mathcal{P} \in \Psi$ such that $\approx_{\mathcal{P}}$ is coarser than $\approx_{\mathrm{B}}$.
- Let $\mathcal{A}$ be an AT instance for which conditions 1 and 2 hold. Whenever $\approx_{\mathrm{B}} \subseteq \approx_{\mathcal{P}}$, then for each AT instance $\mathcal{A}'$ that strictly behaviorally conforms to $\mathcal{A}$ it turns out that $[\![\mathcal{A}']\!]_{\mathrm{bbm}}^{\mathrm{pc};\#\mathcal{A}'}$ satisfies $\mathcal{P}$ iff so does $[\![C]\!]_{\mathcal{A}}^{\mathrm{pc};\mathrm{wob}}$ for some $C \in \mathcal{A}$.

- **Example**: instance of the pipe-filter AT with faulty filters.
- The original architectural type `Pipe_Filter` is deadlock free because:
  - From the topological viewpoint it is a star.
  - Its central AEI `P` is deadlock free.
  - `P` is deadlock-freedom-compatible with `F_0`, `F_1`, `F_2`, and `F_3`.
- Then its strictly behaviorally conformant instance in which the AET `Filter_Type` is replaced by the new AET `Faulty_Filter_Type` is deadlock free too.

- Second case: *exogenous variations*.
- An exogenous variation of an AT instance consists of adding AEIs outside the topological frontier, with some of these AEIs being attached to architectural interactions.
- All the architectural interactions at which an exogenous variation takes place become local interactions, hence they must be left visible when applying the architectural compatibility and interoperability checks.
- An exogenous variation may alter the set of original cyclic unions, in which case nothing can be deduced from the absence of mismatches in the original AT instance whenever new kinds of cycle are generated.
- Think of an exogenous variation of a pipe-filter system whose only pipe has not only several downstream filters, but also several upstream filters.

- $\mathcal{AI}_{C_j}$: set of architectural interactions of C_j.
- Partially semi-closed interacting semantics of $C_j \in \{C_1, \ldots, C_n\}$ with respect to $\{C_1, \ldots, C_n\}$ including its buffers attached to $\{C''_1, \ldots, C''_{n''}\}$:

$$[\![C_j]\!]^{\mathrm{psc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} = [\![C_j]\!]^{\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} / (Name - \mathcal{V}_{C_j;C_1,\ldots,C_n} - \mathcal{AI}_{C_j})$$

- $[\![C_j]\!]^{\mathrm{psc;wob}}_{C_1,\ldots,C_n}$: shorthand for the case $n'' = 0$.
- Partially semi-closed interacting sem. of $\{C'_1, \ldots, C'_{n'}\} \subseteq \{C_1, \ldots, C_n\}$ w.r.t. $\{C_1, \ldots, C_n\}$ including their buffers attached to $\{C''_1, \ldots, C''_{n''}\}$:

$$\begin{aligned}
&[\![C'_1, \ldots, C'_{n'}]\!]^{\mathrm{psc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} = \\
&\qquad [\![C'_1]\!]^{\mathrm{psc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} \parallel_{\mathcal{S}(C'_1,C'_2;C_1,\ldots,C_n)} \\
&\qquad\quad [\![C'_2]\!]^{\mathrm{psc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} \parallel_{\mathcal{S}(C'_1,C'_3;C_1,\ldots,C_n) \cup \mathcal{S}(C'_2,C'_3;C_1,\ldots,C_n)} \cdots \\
&\qquad\quad \cdots \parallel_{\bigcup_{i=1}^{n'-1} \mathcal{S}(C'_i,C'_{n'};C_1,\ldots,C_n)} [\![C'_{n'}]\!]^{\mathrm{psc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n}
\end{aligned}$$

- Totally semi-closed interacting semantics of $C_j \in \{C_1, \ldots, C_n\}$ with respect to $\{C_1, \ldots, C_n\}$ including its buffers attached to $\{C''_1, \ldots, C''_{n''}\}$:

$$[\![C_j]\!]^{\text{tsc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} = [\![C_j]\!]^{\text{psc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} / \varphi_{C_j,\text{async}}(\mathcal{OALI}_{C_j})$$

- $[\![C_j]\!]^{\text{tsc;wob}}_{C_1,\ldots,C_n}$: shorthand for the case $n'' = 0$.

- Totally semi-closed interacting sem. of $\{C'_1, \ldots, C'_{n'}\} \subseteq \{C_1, \ldots, C_n\}$ w.r.t. $\{C_1, \ldots, C_n\}$ including their buffers attached to $\{C''_1, \ldots, C''_{n''}\}$:

$$\begin{aligned}
[\![C'_1, \ldots, C'_{n'}]\!]^{\text{tsc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} = \\
[\![C'_1]\!]^{\text{tsc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} \|_{\mathcal{S}(C'_1,C'_2;C_1,\ldots,C_n)} \\
[\![C'_2]\!]^{\text{tsc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n} \|_{\mathcal{S}(C'_1,C'_3;C_1,\ldots,C_n) \cup \mathcal{S}(C'_2,C'_3;C_1,\ldots,C_n)} \cdots \\
\cdots \|_{\bigcup_{i=1}^{n'-1} \mathcal{S}(C'_i,C'_{n'};C_1,\ldots,C_n)} [\![C'_{n'}]\!]^{\text{tsc};\#C''_1,\ldots,C''_{n''}}_{C_1,\ldots,C_n}
\end{aligned}$$

- $[\![C'_1, \ldots, C'_{n'}]\!]^{\text{tsc};\#C''_1,\ldots,C''_{n''};C'''_1,\ldots,C'''_{n'''}}_{C_1,\ldots,C_n}$: variant totally semi-closed up to $\{C'''_1, \ldots, C'''_{n'''}\} \subset \{C'_1, \ldots, C'_{n'}\}$.

- The definitions of these variants of the partially and totally closed interacting semantics coincide with the respective original definitions in the absence of architectural interactions.
- The center K of a star is $\mathcal{P}$-semi-compatible with $C_j \in \mathcal{B}_K$ iff:

$$(\llbracket K \rrbracket_{\mathcal{A}}^{\text{psc};\#C_j} \parallel_{\mathcal{S}(K,C_j;\mathcal{A})} \llbracket C_j \rrbracket_{K,\mathcal{B}_K}^{\text{tc};\#K}) \,/\, (H_{K,C_j} \cup E_{K,C_j}) \approx_{\mathcal{P}} \llbracket K \rrbracket_{\mathcal{A}}^{\text{psc};\text{wob}}$$

- An AEI C_j in a cycle $\mathcal{P}$-semi-interoperates with the other AEIs in the cycle iff:

$$\llbracket \mathcal{Y} \rrbracket_{\mathcal{A}}^{\text{tsc};\#\mathcal{Y};C_j} \,/\, (Name - \mathcal{V}_{C_j;\mathcal{A}} - \mathcal{AI}_{C_j}) \,/\, (H_{C_j,\mathcal{Y}} \cup E_{C_j,\mathcal{Y}}) \approx_{\mathcal{P}} \llbracket C_j \rrbracket_{\mathcal{A}}^{\text{psc};\text{wob}}$$

- These revised architectural compatibility and interoperability notions imply the respective original ones if hiding architectural interactions thanks to the congruence property of $\approx_{\mathcal{P}}$ w.r.t. static operators.

- Semi-frontier of $\{C_1, \ldots, C_n\}$:

  $$\mathcal{SF}_{C_1,\ldots,C_n} = \{C_j \in \{C_1, \ldots, C_n\} \mid \mathcal{LI}_{C_j;C_1,\ldots,C_n} \neq \mathcal{LI}_{C_j} \vee \mathcal{AI}_{C_j} \neq \emptyset\}$$

  which coincides with $\mathcal{F}_{C_1,\ldots,C_n}$ plus other AEIs having architectural interactions.

- A cyclic union $\mathcal{CU}_1$ is (strictly) topologically equivalent to another cyclic union $\mathcal{CU}_2$ iff there exists a bijection between them that preserves for corresponding AEIs:
  - ⊙ Their type (and their actual data parameter values).
  - ⊙ Their attachments within the cyclic union.
  - ⊙ Their membership to the frontier of the cyclic union.

- Let κ be a cycle covering alg. with total result for an AT instance $\mathcal{A}$ and $\mathcal{A}'$ be an AT instance resulting from a strictly topologically conformant exogenous variation of $\mathcal{A}$.
- The exogenous variation of κ for $\mathcal{A}'$ is defined as follows:
  1. All the AEIs of $\mathcal{A}'$ are initially unmarked.
  2. For each cyclic union $\mathcal{CU}_C^{\mathcal{A}} \in \mathcal{CU}^{\mathcal{A}}(\kappa)$:
     a. Pick out C.
     b. Mark all the AEIs in $\mathcal{CU}_C^{\mathcal{A}'}$.
  3. While there is an unmarked additional AEI C in the cycles of the abstract enriched flow graph of $\mathcal{A}'$ such that there is $\mathcal{CU}_{C'}^{\mathcal{A}} \in \mathcal{CU}^{\mathcal{A}}(\kappa)$ with $C' = corr(C)$ and $\mathcal{CU}_C^{\mathcal{A}'}$ strictly topologically equivalent to $\mathcal{CU}_{C'}^{\mathcal{A}}$:
     a. Pick out C.
     b. Mark all the AEIs in $\mathcal{CU}_C^{\mathcal{A}'}$.
- $\mathcal{A}'$ is exo-coverable by κ iff all the AEIs in the cycles of $\mathcal{A}'$ are marked, the exogenous variation of κ has total result, and for each $C \in \mathcal{A}$ such that $\mathcal{CU}_C^{\mathcal{A}} \in \mathcal{CU}^{\mathcal{A}}(\kappa)$ it holds $\mathcal{CU}_C^{\mathcal{A}'} = \mathcal{CU}_C^{\mathcal{A}}$ (no new kind of cycle can be present).

- Let $\mathcal{A}$ be an AT instance for which conditions 1 and 2 hold and $\mathcal{A}'$ be an AT instance resulting from a strictly topologically conformant exogenous var. of $\mathcal{A}$ for which the following additional conditions hold:

  $3^{\text{ex}}_{\cdot}$ For each $K \in \mathcal{A}$ belonging to an acyclic portion or to the intersection of some cycle with acyclic portions of the abstract enriched flow graph of $\mathcal{A}$, if K is of the same type as an AEI having architectural interactions at which the exogenous variation takes place, then K is $\mathcal{P}$-semi-compatible with every $C \in \mathcal{B}_K - \mathcal{CU}^{\mathcal{A}}_K$.

  $4^{\text{ex}}_{\cdot}$ If $\mathcal{A}'$ is cyclic, then $\mathcal{A}'$ is exo-coverable by κ (cycle covering alg. of cond. 2) and for each $C_j \in \mathcal{SF}_{C_1,\ldots,C_n}$ with $\{C_1, \ldots, C_n\} \in \mathcal{CU}^{\mathcal{A}}(\kappa)$, if C_j is of the same type as an AEI having architectural interactions at which the exogenous variation takes place, then C_j $\mathcal{P}$-semi-interoperates with the other AEIs in $\{C_1, \ldots, C_n\}$.

  Then $[\![\mathcal{A}']\!]^{\text{pc};\#\mathcal{A}'}_{\text{bbm}}$ satisfies $\mathcal{P}$ iff so does $[\![C]\!]^{\text{pc;wob}}_{\mathcal{A}}$ for some $C \in \mathcal{A}$.

- **Example**: exogenous variation of the pipe-filter AT.
- The original architectural type `Pipe_Filter` is deadlock free because:
  - From the topological viewpoint it is a star.
  - Its central AEI `P` is deadlock free.
  - `P` is deadlock-freedom-compatible with `F_0`, `F_1`, `F_2`, and `F_3`.
- Then its exogenous variation taking place at `F_2.output_item` through an additional pipe `P_1` having the already existing AEI `F_2` as upstream filter and the new AEIs `F_4`, `F_5`, and `F_6` as downstream filters is deadlock free too.

- Third case: *endogenous variations*.
- An endogenous variation of an AT instance consists of changing the number of AEIs of certain types in certain positions of the original topology – without altering the number of attachments involving local and-/or-interactions – by adding or removing AEIs of those types.
- An endogenous variation may create new kinds of attachment not present in the original topology, which may compromise compatibility within some altered acyclic portion.
- An endogenous variation may cancel existing attachments – like those involving removed AEIs or those broken to allow for added AEIs – that are important for the validity of $\mathcal{P} \in \Psi$.
- An endogenous variation may alter the set of original cyclic unions, in which case nothing can be deduced from the absence of mismatches in the original AT instance whenever new kinds of cycle are generated or interoperability is compromised along some altered cycle.

- Let κ be a cycle covering alg. with total result for an AT instance $\mathcal{A}$ and $\mathcal{A}'$ be an AT instance resulting from an endogenous variation of $\mathcal{A}$.
- The endogenous variation of κ for $\mathcal{A}'$ is defined as follows:
  1. All the AEIs of $\mathcal{A}'$ are initially unmarked.
  2. For each cyclic union $\mathcal{CU}_C^{\mathcal{A}} \in \mathcal{CU}^{\mathcal{A}}(\kappa)$:
     a. Pick out C.
     b. Mark all the AEIs in $\mathcal{CU}_C^{\mathcal{A}'}$.
  3. While there is an unmarked additional AEI C in the cycles of the abstract enriched flow graph of $\mathcal{A}'$ such that there is $\mathcal{CU}_{C'}^{\mathcal{A}} \in \mathcal{CU}^{\mathcal{A}}(\kappa)$ with C' of the same type as C and $\mathcal{CU}_C^{\mathcal{A}'}$ strictly top. eq. to $\mathcal{CU}_{C'}^{\mathcal{A}}$:
     a. Pick out C.
     b. Mark all the AEIs in $\mathcal{CU}_C^{\mathcal{A}'}$.
- $\mathcal{A}'$ is endo-coverable by κ iff all the AEIs in the cycles of $\mathcal{A}'$ are marked, the endogenous variation of κ has total result, and for each $C \in \mathcal{A}$ such that $\mathcal{CU}_C^{\mathcal{A}} \in \mathcal{CU}^{\mathcal{A}}(\kappa)$ it holds $\mathcal{CU}_C^{\mathcal{A}'} = \mathcal{CU}_C^{\mathcal{A}}$ up to added/removed AEIs.

- Let $\mathcal{A}$ be an AT instance for which conditions 1 and 2 hold and $\mathcal{A}'$ be an AT instance resulting from an endogenous variation of $\mathcal{A}$ for which the following additional conditions hold:

  $3^{\text{en}}_{\cdot}$ For each attachment in $\mathcal{A}'$ from interaction o of an AEI C'_1, which belongs to an acyclic portion or to the intersection of some cycle with acyclic portions of the abstract enriched flow graph of $\mathcal{A}'$, to interaction i of an AEI $C'_2 \in \mathcal{B}_{C'_1} - \mathcal{CU}^{\mathcal{A}'}_{C'_1}$, there exists an attachment in $\mathcal{A}$ from interaction o of an AEI C_1 of the same type as C'_1, with C_1 belonging to an acyclic portion or to the intersection of some cycle with acyclic portions of the abstract enriched flow graph of $\mathcal{A}$, to interaction i of an AEI $C_2 \in \mathcal{B}_{C_1} - \mathcal{CU}^{\mathcal{A}}_{C_1}$ of the same type as C'_2.

  $4^{\text{en}}_{\cdot}$ No local interaction occurring in $\mathcal{P}$ is involved in canceled attachments.

  $5^{\text{en}}_{\cdot}$ If $\mathcal{A}$ or $\mathcal{A}'$ is cyclic, then $\mathcal{A}'$ is endo-coverable by κ (cycle covering alg. of cond. 2) and for each cyclic union $\mathcal{CU}^{\mathcal{A}'}_{C}$ generated by the endogenous variation of κ:

  a. No local interaction of the AEIs of $\mathcal{CU}^{\mathcal{A}}_{C}$ that $\mathcal{P}$-interoperate with the other AEIs in $\mathcal{CU}^{\mathcal{A}}_{C}$ by virtue of condition 2 is involved in canceled attachments.

  b. No possibly added AEI in $\mathcal{CU}^{\mathcal{A}'}_{C}$ belongs to the frontier of $\mathcal{CU}^{\mathcal{A}'}_{C}$.

  c. If $C \in \mathcal{A}$, then $[\![\mathcal{CU}^{\mathcal{A}'}_{C}]\!]^{\text{pc};\#\mathcal{CU}^{\mathcal{A}'}_{C}}_{\mathcal{CU}^{\mathcal{A}'}_{C}} / H \approx_{\mathcal{P}} [\![\mathcal{CU}^{\mathcal{A}}_{C}]\!]^{\text{pc};\#\mathcal{CU}^{\mathcal{A}}_{C}}_{\mathcal{CU}^{\mathcal{A}}_{C}} / H$ where H has all local interactions of the added/removed AEIs and those attached to them.

  Then $[\![\mathcal{A}']\!]^{\text{pc};\#\mathcal{A}'}_{\text{bbm}}$ satisfies $\mathcal{P}$ iff so does $[\![C]\!]^{\text{pc;wob}}_{\mathcal{A}}$ for some $C \in \mathcal{A}$.

- **Example**: endogenous variation of a ring of pipe-filter systems.
- The original architectural type `Pipe_Filter_R` is deadlock free because:
  - From the topological viewpoint it is a cyclic union.
  - Each of its instances of AET `Pipe_Type` is deadlock free.
  - Each such AEI deadlock-freedom-interoperates with the other AEIs in the cyclic union.
- Then its endogenous variation in which the number of pipe-filter systems changes from 3 to 4 is deadlock free too.

- Fourth case: *multiplicity variations*.
- A multiplicity variation of an AT instance consists of changing the number of AEIs of certain types that are attached to certain local and-/or-interactions, by adding or removing AEIs of those types.
- A multiplicity variation may cancel existing attachments that are important for the validity of $\mathcal{P} \in \Psi$.
- Scalability not supported by or-interactions attached to uni-interactions that may raise exceptions or enabled a number of times that does not allow all the uni-interactions attached to them to be executed.
- A multiplicity variation may alter the set of original cyclic unions.
- In each of the cases above, nothing can be deduced from the absence of architectural mismatches in the original AT instance.

- Let $\mathcal{A}$ be an AT instance for which conditions 1 and 2 hold and $\mathcal{A}'$ be an AT instance resulting from a multiplicity variation of $\mathcal{A}$ for which the following additional conditions hold:

  $3^{\mathrm{mu}}_{\cdot}$ No local interaction occurring in $\mathcal{P}$ is involved in attachments canceled by the multiplicity variation.

  $4^{\mathrm{mu}}_{\cdot}$ No local or-interaction involved in the multiplicity variation is attached to a semi-synchronous uni-interaction or to an input asynchronous uni-interaction.

  $5^{\mathrm{mu}}_{\cdot}$ Each local or-interaction involved in the multiplicity variation is enabled infinitely often.

  $6^{\mathrm{mu}}_{\cdot}$ If $\mathcal{A}$ or $\mathcal{A}'$ is cyclic, then $\mathcal{CU}^{\mathcal{A}}(\kappa) = \mathcal{CU}^{\mathcal{A}'}(\kappa)$ (cycle covering alg. of cond. 2).

  Then $[\![\mathcal{A}']\!]^{\mathrm{pc};\#\mathcal{A}'}_{\mathrm{bbm}}$ satisfies $\mathcal{P}$ iff so does $[\![C]\!]^{\mathrm{pc};\mathrm{wob}}_{\mathcal{A}}$ for some $C \in \mathcal{A}$.

- **Example**: multiplicity variation of the pipe-filter AT.
- The original architectural type `OV_Pipe_Filter` is deadlock free because:
  - From the topological viewpoint it is a star.
  - Its central AEI `P` is deadlock free.
  - `P` is deadlock-freedom-compatible with `F[0]`, `F[1]`, `F[2]`, and `F[3]`.
- Then its multiplicity variation in which the number of downstream filters changes from 3 to 4 is deadlock free too.

Comparison with Related Techniques

- MismDet focuses on an entire class of properties rather than on individual properties in order to gain generality.
- MismDet uses property-specific behavioral equivalences instead of a single general behavioral equivalence in order to gain flexibility.
- MismDet exploits the hiding operator for highlighting only the parts of the system behavior that are important for well-formedness verification.
- Stars and cycles generalize topological formats considered in the past:
  - Two software units from the point of view of a single communication involving both of them.
  - Two software units from the point of view of all the communications involving both of them.
- Scalability from sets of software units each forming a star or a cycle to arbitrary topologies and to architectural types (under certain constraints).
- Provision of useful information for code generation and test generation.

Part III:
Component-Oriented Performance Evaluation

Performance-Driven Architectural Selection

- Integrated view of functional and nonfunctional aspects in the software development process.
  - △ *How to choose among a number of alternative architectural designs that are functionally correct?*
  - △ *How to choose among a number of alternative off-the-shelf components that provide a given set of functionalities?*
- Those choices are typically driven by the objective of optimizing the performance of the final system.
  - △ *How to improve the performance of a specific architectural design?*
- Component-oriented diagnostic information for guiding performance-improving modifications.

- Quickly predicting/improving/comparing the performance of different architectural designs.
- Procedure PERFSEL combines different performance-aware notations:
  - ⊙ Stochastic process algebraic architectural description languages for performance-aware component-oriented modeling (ÆMILIA).
  - ⊙ Product-form queueing networks for component-oriented evaluation and diagnosis of typical average performance indicators (PFQN).
  - ⊙ Reward structures and logics for component-oriented specification of arbitrary performance measures (MSL).
- Behavioral equivalences are not used because they are qualitative means for establishing whether distinct elements possess the same properties, while we need to quantify the extent to which properties are satisfied.
- Definition of a quantitative semantics transforming ÆMILIA descriptions into QN models (whenever certain constraints are satisfied).

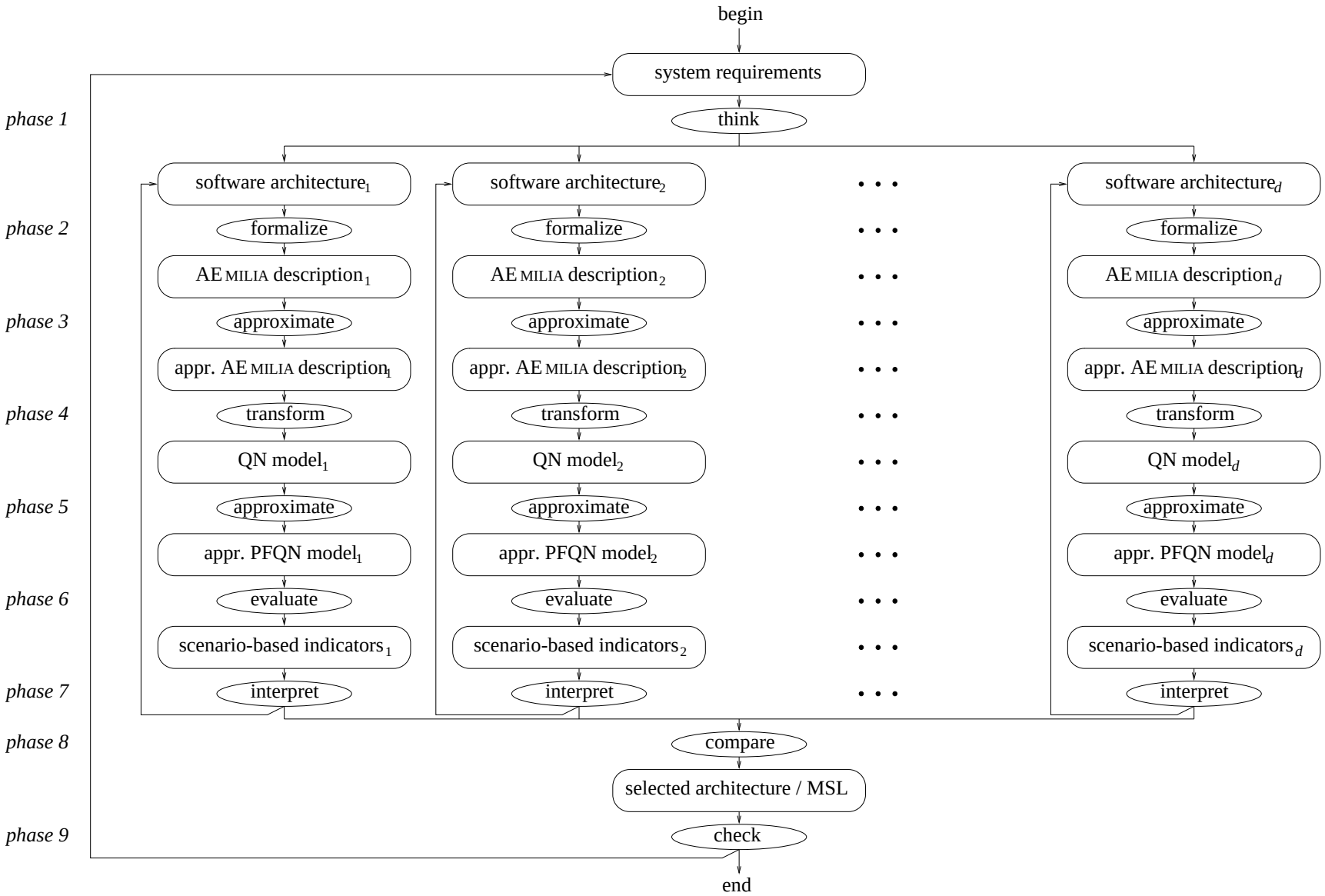
Class of Measures

- PerfSel focuses on typical average indicators that refer to both system level and component level and give insights into the overall performance.
- Throughput: mean number of tasks executed by a system/component per unit of time.
- Measure of the productivity of a system/component.
- Useful for singling out components that are bottlenecks and hence must be redesigned.
- Utilization: mean fraction of time during which a system/component is running.
- Measure of the relative usage of the computational resources by a system/component.
- Useful at deployment time for a balanced distribution of the workload among the computational resources.

- Mean queue length: mean number of items in a system/component handling data.
- Measure of the space complexity of data repositories.
- Useful for setting their dimension in such a way to avoid component execution blocking due to under-sized buffers as well as waste of memory due to over-sized buffers.
- Mean response time: mean time needed by a system/component to complete or repeat its execution.
- Measure of the time complexity of a system/component.
- Useful for predicting the quality of service that will be perceived by the software user on average.

Selection Strategy

- PerfSel is a multiphase procedure working on an arbitrary number of alternative architectural designs.
- Focus on general average indicators that can be efficiently computed and give insights into the overall performance (prediction).
- Indicators calculated at the system level and at the component level in order to provide diagnostic information (improvement).
- Scenario-based assessment of the indicators for the various alternative designs (comparison).
- Final check of the selected architecture against the specific performance requirements due to the possible introduction of approximations and the usage of general indicators.


begin
system requirements
phase 1
think
software architecture1
software architecture2
• • •
software architectured
phase 2
formalize
formalize
formalize
AE MILIA description1
AE MILIA description2
AE MILIA descriptiond
phase 3
approximate
approximate
approximate
appr. AE MILIA description1
appr. AE MILIA description2
appr. AE MILIA descriptiond
phase 4
transform
transform
transform
QN model1
QN model2
QN modeld
phase 5
approximate
approximate
approximate
appr. PFQN model1
appr. PFQN model2
appr. PFQN modeld
phase 6
evaluate
evaluate
evaluate
scenario-based indicators1
scenario-based indicators2
scenario-based indicatorsd
phase 7
interpret
interpret
interpret
phase 8
compare
selected architecture / MSL
phase 9
check
end

Performance Modeling: ÆMILIA

- PERFSEL models nonfunctional aspects by means of ÆMILIA.
- ÆMILIA is a performance-aware variant of PADL inheriting the structure of the descriptions, the construction of the translation semantics, and the functional verification techniques based on behavioral equivalences.
- Each of its actions is composed of a name and a rate.
- Actions are divided into exponentially timed, immediate, and passive.
- Semantics by translation into extended Markovian process algebra with generative-reactive synchronizations (EMPA_{gr}).
- Labeled *multi*transition systems in order to take into account transition multiplicity ($P + P = P$ no longer holds).
- Performance analysis of any performance-closed ÆMILIA description on a continuous-time Markov chain model (CTMC).

- A Markov chain is a discrete-state stochastic process $\{RV(t) \mid t \in \mathbb{R}_{\geq 0}\}$ such that for all $n \in \mathbb{N}$, time instants $t_0 < t_1 < \ldots < t_n < t_{n+1}$, and states $s_0, s_1, \ldots, s_n, s_{n+1} \in S$:

$$\begin{aligned} &\Pr\{RV(t_{n+1}) = s_{n+1} \mid RV(t_0) = s_0 \wedge RV(t_1) = s_1 \wedge \ldots \wedge RV(t_n) = s_n\} \\ &\quad = \Pr\{RV(t_{n+1}) = s_{n+1} \mid RV(t_n) = s_n\} \end{aligned}$$

- The past history is completely summarized by the current state.
- Equivalently, the stochastic process has no memory of the past.
- Time homogeneity: probabilities independent of state change times.
- The solution of a Markov chain is its state probability distribution $\boldsymbol{\pi}()$ at an arbitrary time instant.

- In the continuous-time case:
  - ⊙ State transitions are described by a rate matrix $\boldsymbol{Q}$.
  - ⊙ The sojourn time in any state is exponentially distributed.
  - ⊙ Given $\boldsymbol{\pi}(0)$, the transient solution $\boldsymbol{\pi}(t)$ is obtained by solving:

$$\boldsymbol{\pi}(t) \cdot \boldsymbol{Q} = \frac{\mathrm{d}\boldsymbol{\pi}(t)}{\mathrm{d}t}$$

  - ⊙ The stationary solution $\boldsymbol{\pi} = \lim_{t \to \infty} \boldsymbol{\pi}(t)$ is obtained (if any) by solving:

$$\begin{aligned} \boldsymbol{\pi} \cdot \boldsymbol{Q} &= \mathbf{0} \\ \sum_{s \in S} \pi[s] &= 1 \end{aligned}$$

- Exponentially distributed random variables are the only continuous random variables satisfying the memoryless property:

$$\Pr\{RV \leq v + v' \mid RV > v'\} = \Pr\{RV \leq v\}$$

- Exponentially distributed durations for timed actions not so restrictive.
- The memoryless property of the exponential distribution results in a simpler mathematical treatment without sacrificing expressiveness:
  - Compliance with the interleaving semantics of parallel composition.
  - Easy calculation of state sojourn times and transition probabilities.
  - Adequate for modeling the timing of many real-life phenomena like arrival processes, failure events, and chemical reactions.
  - Most appropriate stochastic approximation in the case in which only the average duration of an activity is known.
  - Proper combinations (phase-type distributions) approximate most of general distributions arbitrarily closely.

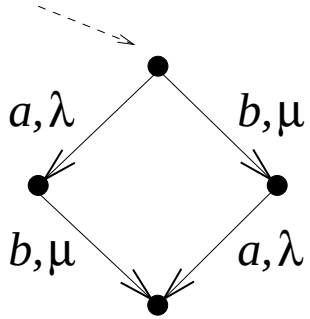
- An exponentially timed action $<a, \lambda>$ takes place at rate $\lambda \in \mathbb{R}_{>0}$.
- The duration of $<a, \lambda>$ is described by the exponentially distributed random variable Exp_λ.
- The probability distribution function for the duration of $<a, \lambda>$ is $\Pr\{Exp_\lambda \leq t\} = 1 - \mathrm{e}^{-\lambda \cdot t}$ for all $t \in \mathbb{R}_{\geq 0}$.
- The average duration of $<a, \lambda>$ is $\mathrm{E}\{Exp_\lambda\} = 1 / \lambda$.
- The stochastic process underlying an $\mathrm{EMPA}_{\mathrm{gr}}$ description comprising only exponentially timed actions turns out to be a pure CTMC.

- Apply the race policy if several exponentially timed actions with rates $\lambda_1, \ldots, \lambda_h$ are enabled in a process term.
- The random variable quantifying the sojourn time associated with that term is thus the minimum of the random variables quantifying the durations of the actions enabled in that term.
- The sojourn time is exponentially distributed because:

$$\boxed{\min(Exp_{\lambda_1}, \ldots, Exp_{\lambda_h}) \;=\; Exp_{\lambda_1 + \ldots + \lambda_h}}$$

- The average sojourn time is therefore given by $1 \,/\, (\lambda_1 + \ldots + \lambda_h)$.
- The execution probability of exponentially timed action with rate λ_i is $\lambda_i \,/\, (\lambda_1 + \ldots + \lambda_h)$.

- Interleaving semantics valid for concurrent exponentially timed actions:
  - ⊙ Due to the memoryless property of the exponential distribution, the execution of an exponentially timed action can be thought of as being started in the last state in which the action is enabled.
  - ⊙ Due to the infinite support of the exponential distribution, the probability of simultaneous termination of two concurrent exponentially timed actions is zero.
- Labeled (multi)transition system of $<a, \lambda>.\underline{0} \parallel_{\emptyset} <b, \mu>.\underline{0}$:



- Interleave concurrent exponentially timed actions without the need of adjusting their rates inside transition labels.

- An immediate action $<a, \infty_{l,w}>$ takes place at an unbounded rate.
- The duration of $<a, \infty_{l,w}>$ is zero.
- Performance abstraction mechanism:
  - Activities that are several orders of magnitude faster than those important for evaluating certain performance measures.
  - No timing can be associated with selections among logical events (e.g., the reception of a message vs. its loss).
- Capability of expressing prioritized/probabilistic choices.
- The stochastic process underlying an EMPA_{gr} description comprising also immediate actions turns out to be a CTMC with vanishing states (which can be eliminated by connecting their incoming transitions to their derivative states).

- Immediate actions take precedence over exponentially timed ones.
- They all have the same zero duration.
- Each immediate action has an associated priority level $l \in \mathbb{N}_{>0}$ and an associated weight $w \in \mathbb{R}_{>0}$ to solve choices.
- Apply the generative preselection policy if several immediate actions are enabled in a process term.
- Consider only those having the highest priority level.
- Assume that their weights are $w_1, \ldots, w_h$.
- The execution probability of immediate action with weight w_i is $w_i \,/\, (w_1 + \ldots + w_h)$.

- A passive action $<a, *_{l,w}>$ takes place at an unspecified rate.
- The duration of $<a, *_{l,w}>$ becomes specified only when that action synchronizes with an exponentially timed or immediate action.
- Mechanism for representing passivity with respect to communications.
- Useful to overcome the fact that the maximum of two exponentially distributed random variables is not exponentially distributed.
- Taking the maximum would be the natural choice for the duration of a synchronization between two concurrent exponentially timed actions, but leads outside the field of memoryless distributions.
- The stochastic process underlying an EMPA_{gr} description comprising also passive actions is defined only if there are no passive transitions (performance closure).

- Each passive action has an associated priority constraint $l' \in \mathbb{N}$ and an associated weight $w \in \mathbb{R}_{>0}$ to solve choices.
- Apply the reactive preselection policy if several passive actions are enabled in a process term.
- The choice among passive actions with different names is solved nondeterministically.
- The choice among passive actions with the same name is restricted to those having the highest priority constraint.
- Assume that their weights are $w_1, \ldots, w_h$.
- The execution probability of passive action with weight w_i is $w_i \,/\, (w_1 + \ldots + w_h)$.

- Generative-reactive synchronizations: in EMPA_{gr} exponentially timed and immediate actions can synchronize only with passive actions.
- Passive actions with priority constraint 0 can synchronize only with exponentially timed actions.
- Passive actions with reactive priority constraint $l' > 0$ can synchronize only with immediate actions having priority level $l = l'$.
- Among all the enabled nonpassive actions whose names belong to the synchronization set, the proposal of an action name is *generated* based on priorities and rates/weights of those actions.
- The enabled passive actions having the same name as the proposed one *react* by performing a selection based on their priorities and weights.
- The nonpassive action winning the generative selection synchronize with the passive action winning the reactive selection.

- Rules for generative-reactive and reactive-reactive synchronizations:

$$\frac{P_1 \xrightarrow{a,\lambda}_{\mathrm{M}} P_1' \qquad P_2 \xrightarrow{a,*0,w}_{\mathrm{M}} P_2' \qquad a \in S}{P_1 \parallel_S P_2 \xrightarrow{a,\lambda \cdot \frac{w}{weight(P_2,a,0)}}_{\mathrm{M}} P_1' \parallel_S P_2'} \quad \text{(plus symmetrical rule)}$$

$$\frac{P_1 \xrightarrow{a,\infty_{l,v}}_{\mathrm{M}} P_1' \qquad P_2 \xrightarrow{a,*l,w}_{\mathrm{M}} P_2' \qquad a \in S}{P_1 \parallel_S P_2 \xrightarrow{a,\infty_{l,v} \cdot \frac{w}{weight(P_2,a,l)}}_{\mathrm{M}} P_1' \parallel_S P_2'} \quad \text{(plus symmetrical rule)}$$

$$\frac{P_1 \xrightarrow{a,*l,w_1}_{\mathrm{M}} P_1' \qquad P_2 \xrightarrow{a,*l,w_2}_{\mathrm{M}} P_2' \qquad a \in S}{P_1 \parallel_S P_2 \xrightarrow{a,*l,norm(w_1,w_2,a,l,P_1,P_2)}_{\mathrm{M}} P_1' \parallel_S P_2'}$$

$$weight(P,a,l) = \sum \{\!|\, w \in \mathbb{R}_{>0} \mid \exists P'.\, P \xrightarrow{a,*l,w}_{\mathrm{M}} P' \,|\!\}$$

$$norm(w_1,w_2,a,l,P_1,P_2) = \frac{w_1}{weight(P_1,a,l)} \cdot \frac{w_2}{weight(P_2,a,l)} \cdot (weight(P_1,a,l) + weight(P_2,a,l))$$

- In an ÆMILIA description:
  - The rate of an exponentially timed action is denoted by `exp( )`.
  - The rate of an immediate action is denoted by `inf( , )`.
  - The rate of a passive action is denoted by `_( , )`.
  - The default value for priorities/weights is `1` (use of `inf` and `_` alone permitted).
  - Headers can contain rate-/priority-/weight-typed parameters.
  - The occurrences of an action name within the behavior of an AET must be all exponentially timed, all immediate, or all passive.
  - Every set of local interactions attached to each other can contain at most one nonpassive local interaction.

- **Example**: performance-aware variant of the pipe-filter AT.
- There is only one system activity that introduces nonnegligible delays: item transformation.
- For a correct performance modeling, item transformation must be separated from item buffering by means of two distinct AETs.
- Different transformation rates for the various filters.
- The pipe is most likely to forward items to faster downstream filters with free positions.
- Different forward probabilities towards downstream filters.

- Textual description header:

```
ARCHI_TYPE PA_Pipe_Filter(const integer papf_downstr_num :=  3,
                          const integer papf_buffer_size := 10,
                          const rate    papf_tran_rate_0 := 60,
                          const weight  papf_forw_prob_0 :=  1,
                          const array(papf_downstr_num, rate)
                            papf_tran_rate := array_cons(70,
                                                         45,
                                                         30),
                          const array(papf_downstr_num, weight)
                            papf_forw_prob := array_cons(0.5,
                                                         0.3,
                                                         0.2))
```

- Definition of the filter buffer AET (all actions are passive):

```
ARCHI_ELEM_TYPE Filter_Buffer_Type(const integer buffer_size,
                                   const weight  forward_prob)

  BEHAVIOR
    Filter_Buffer(integer(0..buffer_size) item_num := 0;
                  void) =
      choice
      {
        cond(item_num < buffer_size) ->
          <input_item, _(1, forward_prob)> . Filter(item_num + 1),
        cond(item_num > 0) ->
          <pass_item, _> . Filter(item_num - 1)
      }

  INPUT_INTERACTIONS  SYNC UNI input_item
  OUTPUT_INTERACTIONS SYNC UNI pass_item
```

- Definition of the filter core AET (every action is exponentially timed or immediate):

```
ARCHI_ELEM_TYPE Filter_Core_Type(const rate transf_rate)

  BEHAVIOR
    Filter_Core(void; void) =
      <select_item, inf> . <transform_item, exp(transf_rate)> .
                                        <output_item, inf> . Filter_Core()

  INPUT_INTERACTIONS  SYNC UNI select_item
  OUTPUT_INTERACTIONS SYNC UNI output_item
```

- Definition of the pipe AET (a single output or-interaction is still enough because forward probabilities have been encoded inside the filters):

```
ARCHI_ELEM_TYPE Pipe_Type(void)

  BEHAVIOR
    Pipe(void; void) =
      <accept_item, _> . <forward_item, inf> . Pipe()

  INPUT_INTERACTIONS  SYNC UNI accept_item
  OUTPUT_INTERACTIONS SYNC OR  forward_item
```

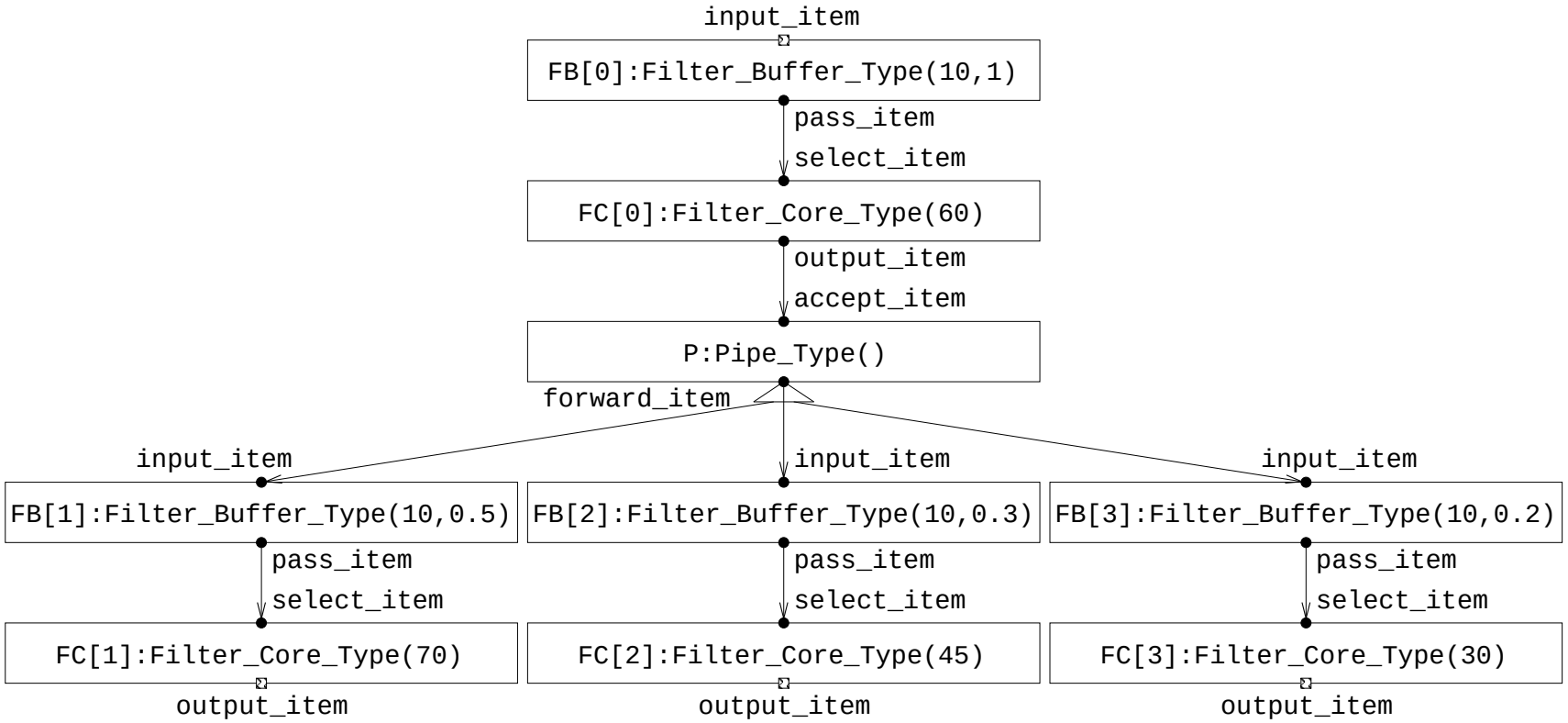
- Declaration of the topology:

```
ARCHI_ELEM_INSTANCES
  FB[0] : Filter_Buffer_Type(papf_buffer_size, papf_forw_prob_0);
  FC[0] : Filter_Core_Type(papf_tran_rate_0);
  P     : Pipe_Type();
  FOR_ALL 1 <= j <= papf_downstr_num
    FB[j] : Filter_Buffer_Type(papf_buffer_size, papf_forw_prob[j]);
  FOR_ALL 1 <= j <= papf_downstr_num
    FC[j] : Filter_Core_Type(papf_tran_rate[j])

ARCHI_INTERACTIONS
  FB[0].input_item;
  FOR_ALL 1 <= j <= papf_downstr_num
    FC[j].output_item

ARCHI_ATTACHMENTS
  FROM FB[0].pass_item   TO FC[0].select_item;
  FROM FC[0].output_item TO P.accept_item;
  FOR_ALL 1 <= j <= papf_downstr_num
    FROM P.forward_item TO FB[j].input_item;
  FOR_ALL 1 <= j <= papf_downstr_num
    FROM FB[j].pass_item TO FC[j].select_item
```

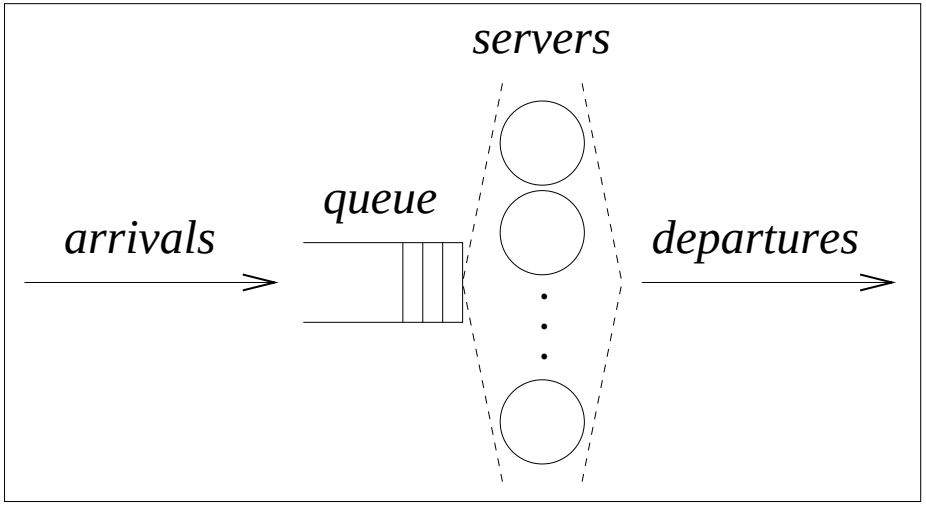
- Enriched flow graph:



Performance Analysis: PFQN

- CTMC models derivable from ÆMILIA descriptions are state based, hence they are not suited for the architectural design level.
- PERFSEL evaluates nonfunctional aspects by means of QN models.
- A queueing network is a set of interacting service centers that represent resources shared by classes of customers, where customer competition for resources corresponds to queueing into the service centers.
- QNs are structured performance models because they elucidate system components and their connectivity.
- Advantages with respect to CTMCs:
  - Average performance indicators at system/component level.
  - Fast solution algorithms for specific families of QNs.
  - Symbolic analysis possible in some cases.

- The simplest QN is composed of a single service center and it is called a queueing system.
- A QS includes a source of arrivals, a queue, and a set of servers:



- Every customer needing a certain service arrives at the QS, waits in the queue for a while, is served by one of the servers, and finally departs.

- Single-class/multiclass depending on the number of classes of customers (different arrival processes and different service demands).
- A QS $A/B/m/c/p$ is characterized by the following six parameters:
  - ⊙ Customer interarrival time probability distribution A (M, D, PH, G).
  - ⊙ Customer service time probability distribution B (M, D, PH, G).
  - ⊙ Number m of independent servers.
  - ⊙ Queue capacity c (default value ∞).
  - ⊙ Customer population size p (default value ∞).
  - ⊙ Queueing discipline.

- There are several queueing disciplines, each establishing a different order in which the customers in the queue have to be served.
- First come first served (FCFS): customers are served in the order of their arrival. This is the default queueing discipline.
- Last come first served (LCFS): customers are served in the reverse order of their arrival.
- Last come first served with preemptive resume (LCFS-PR): same as LCFS, but each arriving customer interrupts the current service, if any, and begins to be served; the interrupted service of a customer is resumed when all customers arrived later than that customer have departed.

- Service in random order (SIRO): the next customer to be served is chosen probabilistically, with equal probabilities assigned to all the waiting customers.
- Nonpreemptive priority (NP): customers are assigned fixed priorities; the waiting customer with the highest priority is served first; if several waiting customers have the same highest priority, they are served in the order of their arrival; once begun, a service cannot be interrupted by the arrival of a higher priority customer.
- Preemptive priority (PP): same as NP, but each arriving higher priority customer interrupts the current service, if any, and begins to be served; a customer whose service was interrupted resumes service when there are no higher priority customers to be served.

- Round robin (RR): each customer is given service for a maximum interval of time called a quantum; if the customer service demand is not satisfied during the quantum, the customer reenters the queue and waits to receive an additional quantum, repeating this process until the service demand is satisfied; the waiting customers are served in the order in which they last entered the queue.
- Processor sharing (PS): all the waiting customers receive service simultaneously with equal shares of the service rate. This queueing discipline is an approximation of RR.
- Infinite server (IS): no queueing takes place as each arriving customer always finds an available server. This queueing discipline needs an unbounded number of servers.

- The solution of a QS is the probability distribution of the number of customers in the system.
- The QS can be analyzed during a given time interval (transient analysis) or after it has reached a stability condition (steady-state analysis).
- The stability condition requires that the customer arrival rate be less than the service rate, so as not to saturate the QS.
- The number N_1 of customers in a QS M/M/1 with arrival rate λ and service rate μ is geometrically distributed with parameter $\rho_1 = \lambda/\mu < 1$ under FCFS, LCFS, LCFS-PR, SIRO, PS.
- The probability that there are $k \in \mathbb{N}$ customers in the system is $\Pr\{N_1 = k\} = \rho_1^k \cdot (1 - \rho_1)$.

- Throughput of M/M/1:

$$\boxed{\overline{T}_1 \;=\; \mu \cdot \Pr\{N_1 > 0\} \;=\; \mu \cdot \rho_1 \;=\; \lambda}$$

- Utilization of M/M/1:

$$\boxed{\overline{U}_1 \;=\; 1 - \Pr\{N_1 = 0\} \;=\; \rho_1}$$

- Mean number of customers in M/M/1:

$$\boxed{\overline{N}_1 \;=\; \sum_{k=0}^{\infty} k \cdot \Pr\{N_1 = k\} \;=\; \frac{\rho_1}{1-\rho_1}}$$

- Mean response time of M/M/1 (Little's law):

$$\boxed{\overline{R}_1 \;=\; \overline{N}_1/\lambda \;=\; \frac{1}{\mu \cdot (1-\rho_1)}}$$

- Stability condition for M/M/m:

$$\boxed{\rho_m \;=\; \lambda/(m\cdot\mu) \;<\; 1}$$

- Average performance indicators for M/M/m:

$$\overline{T}_m \;=\; \sum_{k=1}^{m-1} k\cdot\mu\cdot\Pr\{N_m=k\} + m\cdot\mu\cdot\Pr\{N_m\geq m\} \;=\; \lambda$$

$$\overline{U}_m \;=\; 1-\Pr\{N_m=0\} \;=\; 1-\left(\sum_{k=0}^{m-1}\frac{(m\cdot\rho_m)^k}{k!}+\frac{(m\cdot\rho_m)^m}{m!\cdot(1-\rho_m)}\right)^{-1}$$

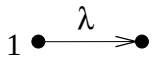
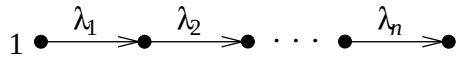
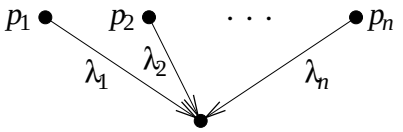
$$\overline{N}_m \;=\; \sum_{k=0}^{\infty} k\cdot\Pr\{N_m=k\} \;=\; m\cdot\rho_m+\frac{\Pr\{N_m=0\}\cdot\rho_m\cdot(m\cdot\rho_m)^m}{m!\cdot(1-\rho_m)^2}$$

$$\overline{R}_m \;=\; \overline{N}_m/\lambda \;=\; \frac{1}{\mu}\cdot\left(1+\frac{\Pr\{N_m=0\}\cdot\rho_m\cdot(m\cdot\rho_m)^{m-1}}{m!\cdot(1-\rho_m)^2}\right)$$

- A QN is a set of interconnected service centers, which are QSs when considered in isolation and hence for each of them it is necessary to specify the six parameters.
- The description of a QN is completed by specifying its topology by means of a matrix of routing probabilities governing the customer flow through the network.
- Single-class/multiclass depending on the number of classes of customers (different arrival processes, different service demands, and different routing probabilities).
- Open/closed/mixed depending on the extent to which external arrivals and departures are allowed for the various classes of customers.
- The solution of a QN is the probability distribution of the number of customers in the network, expressed as a tuple holding the numbers of customers in the various service centers.

- Product-form QN: the probability that it contains a given number of customers $(k_1, k_2, \ldots, k_n)$ is the product of the probabilities that each service center i contains k_i customers (up to a normaliz. constant for closed QNs).
- Solve each service center in isolation, then multiply their solutions.
- The average performance indicators can be obtained at the global level and at the local level for an open product-form QN composed of M/M service centers by using for each center the corresponding formulas.
- The arrival rates for the various service centers can be computed by solving the linear system of traffic equations defined by the routing probabilities.
- In the case of closed or mixed product-form QNs, the same average indicators can be derived at the global level and at the local level by applying suitable algorithms.

- BCMP theorem: an open/closed/mixed single-class/multiclass QN is product form if it has Poisson arrivals (with possibly state-dependent rates), Markovian routing, and service centers featuring:
  - FCFS with the same exponentially distributed service time for all the classes of customers.
  - LCFS-PR, PS, IS with phase-type distributed service time possibly different for the various classes of customers.
- In the second case, the values of the average performance indicators do not change if the phase-type distributed service times are replaced by exponentially distributed service times with the same expected values (time to absorption in finite-state CTMCs having exactly one absorbing state):

Exponential distribution:	*Hypoexponential distribution:*	*Hyperexponential distribution:*
		

Transforming ÆMILIA Descriptions into QN Models

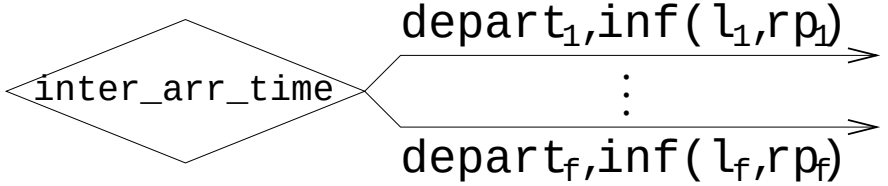
- PERFSEL needs to derive QN models from ÆMILIA descriptions.
- Several differences between the two formalisms (apart from being structured):
  - ÆMILIA is a completely formal, general-purpose architectural description language handling functional and performance aspects, whose basic ingredients are actions and behavioral operators.
  - QNs are instances of a queue-based graphical notation for performance modeling purposes only, in which some details like the queueing disciplines are usually expressed in natural language.
- Only some ÆMILIA descriptions can be transformed into QN models, depending on whether they follow a queue-like pattern or not.
- Impose general syntactical restrictions that single out a reasonably wide family of ÆMILIA descriptions from which QN models can be derived.

- The AEIs of an ÆMILIA description obeying all the general syntactical restrictions cannot always be mapped to the service centers of some QN.
- It is more reasonable to expect to be able to map groups of AEIs to service centers of the form PH/PH/m/c/p.
- Equivalently, try to map each AEI to a QN basic element:
  - Arrival process.
  - Buffer.
  - Service process.
  - Fork process.
  - Join process.
  - Routing process.
- Impose specific syntactical restrictions to single out those AEIs that can be mapped to QN basic elements.

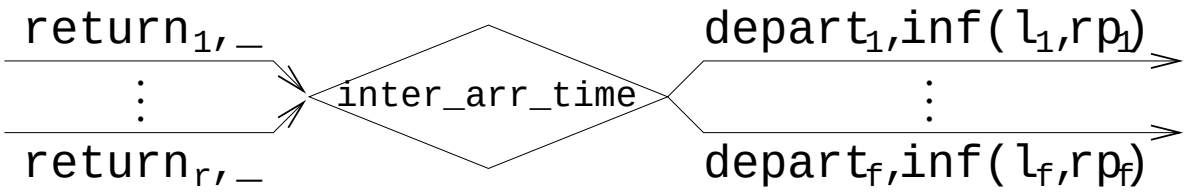
- General syntactical restrictions:
  - ⊙ Every AEI must conform to a QN basic element and be suitably connected to the other AEIs in order to yield a well-formed QN.
  - ⊙ Every interaction must be immediate or passive, thus simplifying the detection of AEIs representing arrival or service processes.
  - ⊙ No AEI can contain exponentially timed actions alternative to each other, as arrival processes and service processes are sequential.
  - ⊙ In order to simplify the detection of phase-type distributed delays, no AEI can contain:
    - ∗ Exp. timed actions alternative to immediate or passive actions.
    - ∗ Immediate actions alternative to passive actions.
    - ∗ Interactions alternative to internal actions.
- Only queueing disciplines with noninterruptable service for a fixed number of servers (FCFS, LCFS, SIRO, NP) are consequently admitted.

- Arrival process:
  - ⊙ Generator of arrivals of customers of a certain class.
  - ⊙ Interarrival times follow a phase-type distribution.
  - ⊙ Two different kinds depending on the customer population size (in the second case, distribution scaling through as many AEIs as there are customers):

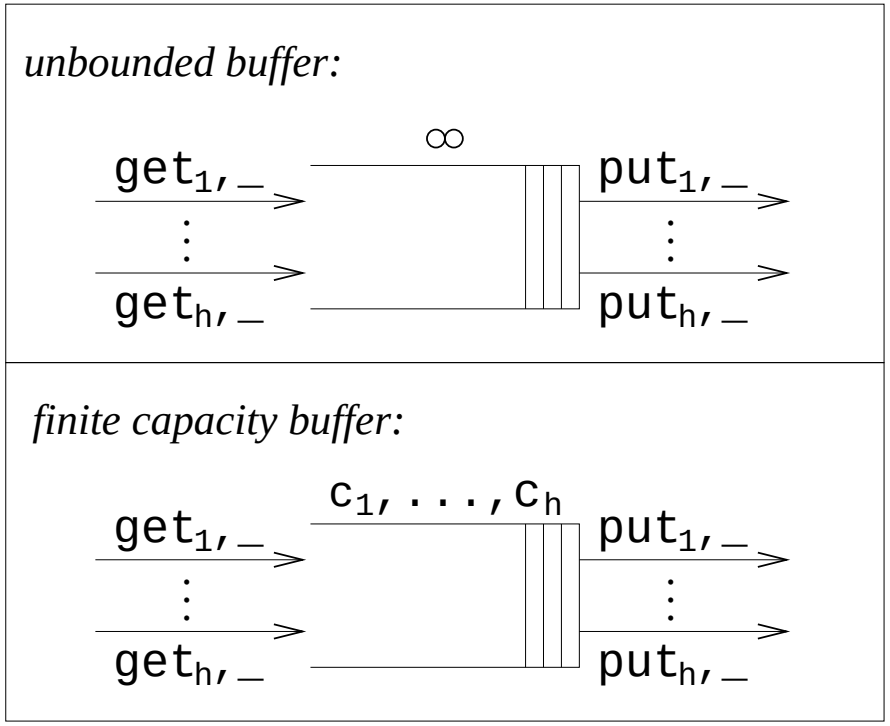
arrival process for unbounded population:



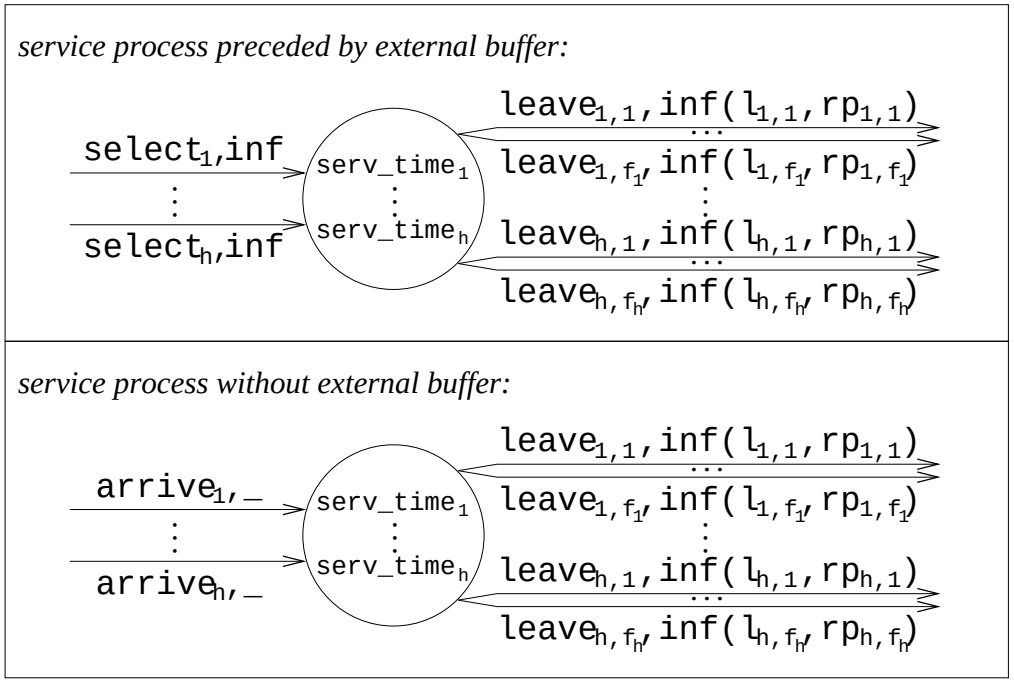
arrival process for single customer of finite population:



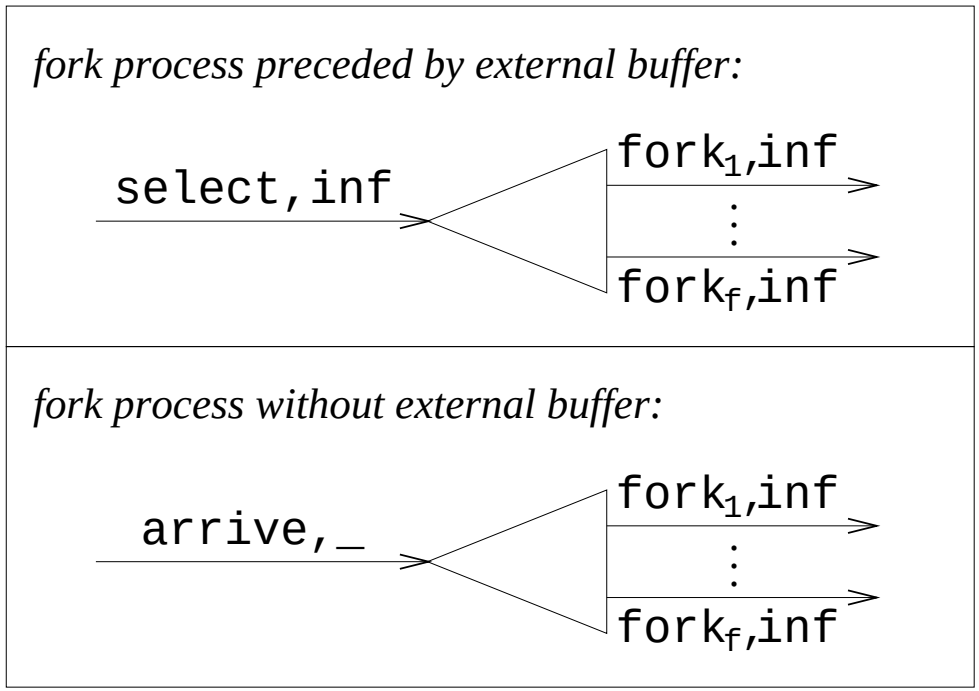
- Buffer (no exponentially timed internal actions):
  - ⊙ Repository of waiting customers of different classes.
  - ⊙ Any noninterruptable queueing discipline (FCFS, LCFS, SIRO, NP).
  - ⊙ Two different kinds depending on the buffer capacity:



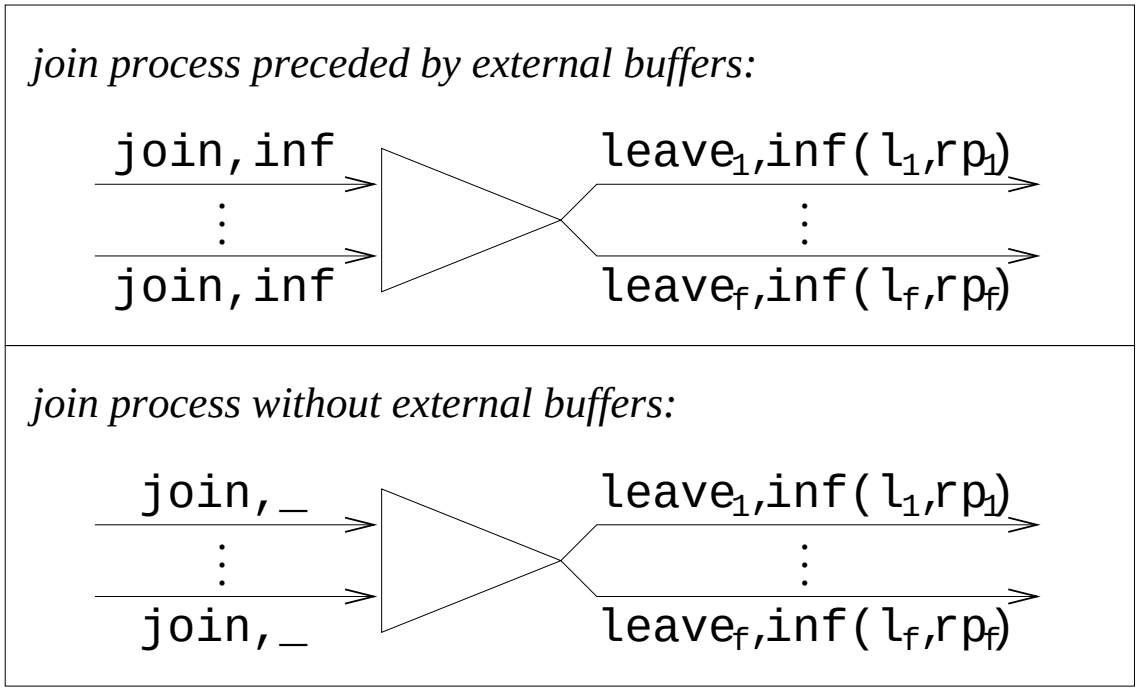
- Service process:
  - ⊙ Server for customers of different classes.
  - ⊙ Service times follow a phase-type distribution for each class.
  - ⊙ Two different kinds depending on the presence of an external buffer where customers can wait before being served (some outputs may be absent; as many AEIs as there are servers in the service center):



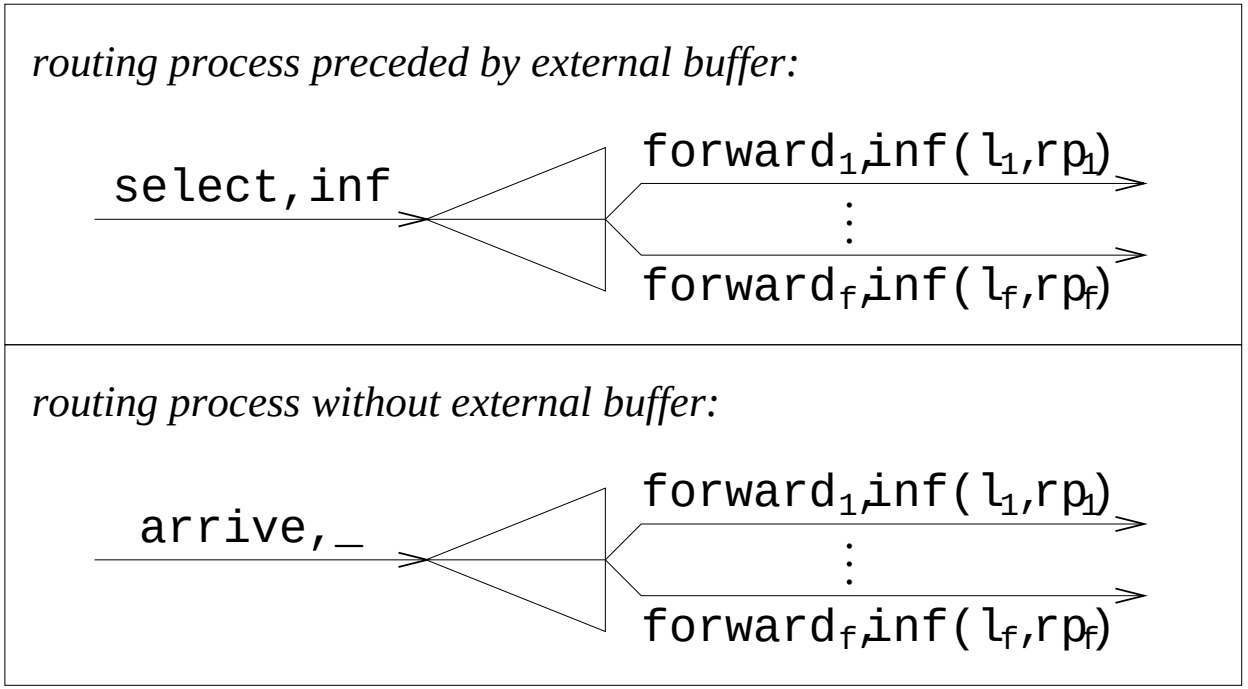
- Fork process (no exponentially timed internal actions):
  - ⊙ Splitter of requests coming from customers of a certain class into subrequests directed to different service centers (single and-interaction or several consecutive uni-interactions on the output side).
  - ⊙ Two different kinds depending on the presence of an external buffer where requests can wait before being split:



- Join process (no exponentially timed internal actions):
  - ⊙ Merger of subrequests coming from customers of a certain class after they have been served at different service centers (input and-interaction).
  - ⊙ Two different kinds depending on the presence of external buffers where subrequests can wait before merging (some outputs may be absent):



- Routing process (no exponentially timed internal actions):
  - ⊙ Router of customers of a certain class.
  - ⊙ Two different kinds depending on the presence of an external buffer where customers can wait before routing (some outputs may be absent):



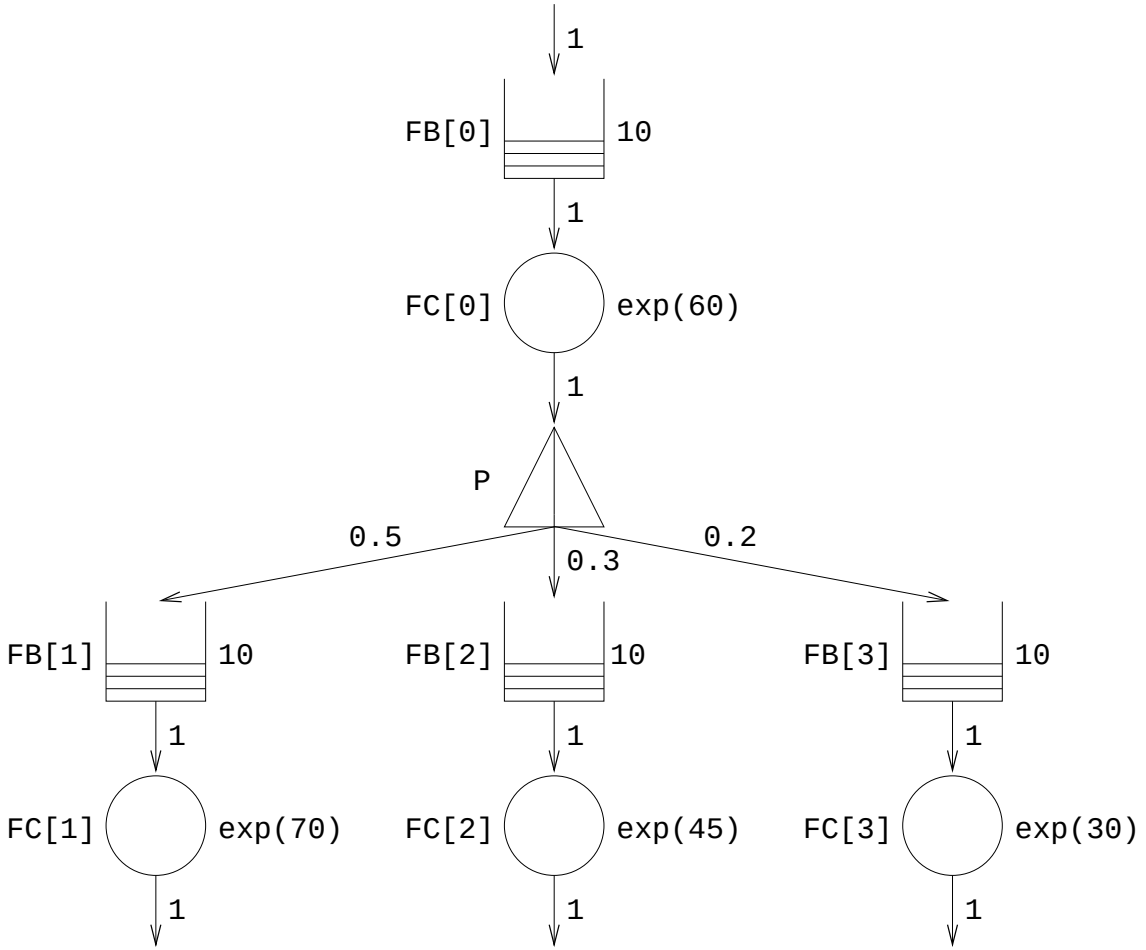
- Attachment-driven composition of the obtained QN basic elements.
- Specific restrictions ensuring that the QN basic elements derivable from the various AEIs of an ÆMILIA description result in a well-formed QN (closed/open if all arrival processes are for finite/unbounded populations, otherwise mixed):
  - An arrival process can be followed only by a service or fork process possibly preceded by a buffer.
  - A buffer can be followed only by a service, fork, join, or routing process.
  - A service process can be followed by any QN basic element.
  - A fork process can be followed only by a service process or another fork process possibly preceded by a buffer.
  - A join process can be followed by any QN basic element.
  - A routing process can be followed by any QN basic element.

- Every AEI satisfying all syntactical restrictions is transformed by means of two groups of functions that provide the attributes that label the resulting QN basic element.
- Documental functions:
  - *qnbe* specifies the kind of QN basic element into which the AEI is transformed.
  - *name* associates the name of the AEI with the corresponding QN basic element.
  - *input* associates the local input interactions of the AEI with the incoming arcs of the corresponding QN basic element.
  - *output* associates the local output interactions of the AEI with the outgoing arcs of the corresponding QN basic element.

- Characterizing functions:
  - *inter_arr_time* computes the phase-type distribution governing the interarrival times of the customers of a certain class for an AEI transformed into an arrival process.
  - *capacity* computes the capacity for an AEI transformed into a buffer.
  - *queueing_disc* establishes the queueing discipline for an AEI transformed into a buffer.
  - *serv_time* computes the phase-type distribution governing the service times related to the customers of a certain class for an AEI transformed into a service process.
  - *routing_prob* computes the routing probabilities of the customers of a certain class for an AEI transformed into an arrival, service, join, or routing process (also considers the return of customers to unbounded populations).

- **Example**: QN model for the performance-aware pipe-filter AT.
- The ÆMILIA description satisfies all the syntactical restrictions.
- `FB[0]`, `FB[1]`, `FB[2]`, `FB[3]` are buffers.
- `FC[0]`, `FC[1]`, `FC[2]`, `FC[3]` are service processes each preceded by a buffer.
- `P` is a routing process not preceded by a buffer.
- An additional arrival process would be necessary to characterize the workload, so that the model would become performance closed and hence the value of the typical average performance indicators could then be calculated.

- QN model:


1
FB[0]
10
1
FC[0]
exp(60)
1
P
0.5
0.3
0.2
FB[1]
10
FB[2]
10
FB[3]
10
1
1
1
FC[1]
exp(70)
FC[2]
exp(45)
FC[3]
exp(30)
1
1
1

Case Study: Selecting Compiler Architectures

- Six phases: lexical analysis, parsing, type checking, intermediate code generation, intermediate code optimization, and code synthesis.
- Two classes of programs: to be optimized, not to be optimized.
- Several alternative architectures:
  - Sequential.
  - Pipeline.
  - Concurrent.
- Objective: choose the best one from a performance viewpoint.
- Assumption: all the delays are exponentially distributed.

- *Sequential compiler*: only one program at a time can be compiled.
- Single buffer for incoming programs of the two classes.
- Textual description header:

```
ARCHI_TYPE Sequential_Compiler(const rate sc_lambda_1 :=  ,
                               const rate sc_lambda_2 :=  ,
                               const rate sc_mu_l     :=  ,
                               const rate sc_mu_p     :=  ,
                               const rate sc_mu_c     :=  ,
                               const rate sc_mu_g     :=  ,
                               const rate sc_mu_o     :=  ,
                               const rate sc_mu_s     :=  )
```

- λ_1, λ_2: symbolic actual values for the arrival rates of the two classes of programs.
- $1/\mu_{\mathtt{l}}, 1/\mu_{\mathtt{p}}, 1/\mu_{\mathtt{c}}, 1/\mu_{\mathtt{g}}, 1/\mu_{\mathtt{o}}, 1/\mu_{\mathtt{s}}$: symbolic actual values for the average durations of the six compilation phases.

- Definition of the program generator AET:

```
ARCHI_ELEM_TYPE Program_Generator_Type(const rate lambda)

  BEHAVIOR
    Program_Generator(void; void) =
      <generate_prog, exp(lambda)> . <deliver_prog, inf> .
                                                Program_Generator()

  INPUT_INTERACTIONS  void
  OUTPUT_INTERACTIONS SYNC UNI deliver_prog
```

- Definition of the program buffer AET for the two classes of programs:

```
ARCHI_ELEM_TYPE Program_Buffer_2C_Type(void)

  BEHAVIOR
    Program_Buffer_2C(integer n_1 := 0,
                      integer n_2 := 0;
                      void) =
      choice
      {
        <get_prog_1, _> . Program_Buffer_2C(n_1 + 1, n_2),
        <get_prog_2, _> . Program_Buffer_2C(n_1, n_2 + 1),
        cond(n_1 > 0) -> <put_prog_1, _> . Program_Buffer_2C(n_1 - 1, n_2),
        cond(n_2 > 0) -> <put_prog_2, _> . Program_Buffer_2C(n_1, n_2 - 1)
      }

  INPUT_INTERACTIONS  SYNC UNI get_prog_1; get_prog_2
  OUTPUT_INTERACTIONS SYNC UNI put_prog_1; put_prog_2
```

- Definition of the sequential compiler AET:

```
ARCHI_ELEM_TYPE Seq_Compiler_Type(const rate mu_l,
                                  const rate mu_p,
                                  const rate mu_c,
                                  const rate mu_g,
                                  const rate mu_o,
                                  const rate mu_s)

  BEHAVIOR
    Seq_Compiler(void; void) =
      choice
      {
        <select_prog_1, inf> . <recognize_tokens, exp(mu_l)> .
          <parse_phrases, exp(mu_p)> . <check_phrases, exp(mu_c)> .
          <generate_icode, exp(mu_g)> . <optimize_icode, exp(mu_o)> .
          <synthesize_code, exp(mu_s)> . Seq_Compiler(),
        <select_prog_2, inf> . <recognize_tokens, exp(mu_l)> .
          <parse_phrases, exp(mu_p)> . <check_phrases, exp(mu_c)> .
          <generate_icode, exp(mu_g)> . <synthesize_code, exp(mu_s)> . Seq_Compiler()
      }

  INPUT_INTERACTIONS  SYNC UNI select_prog_1; select_prog_2
  OUTPUT_INTERACTIONS void
```

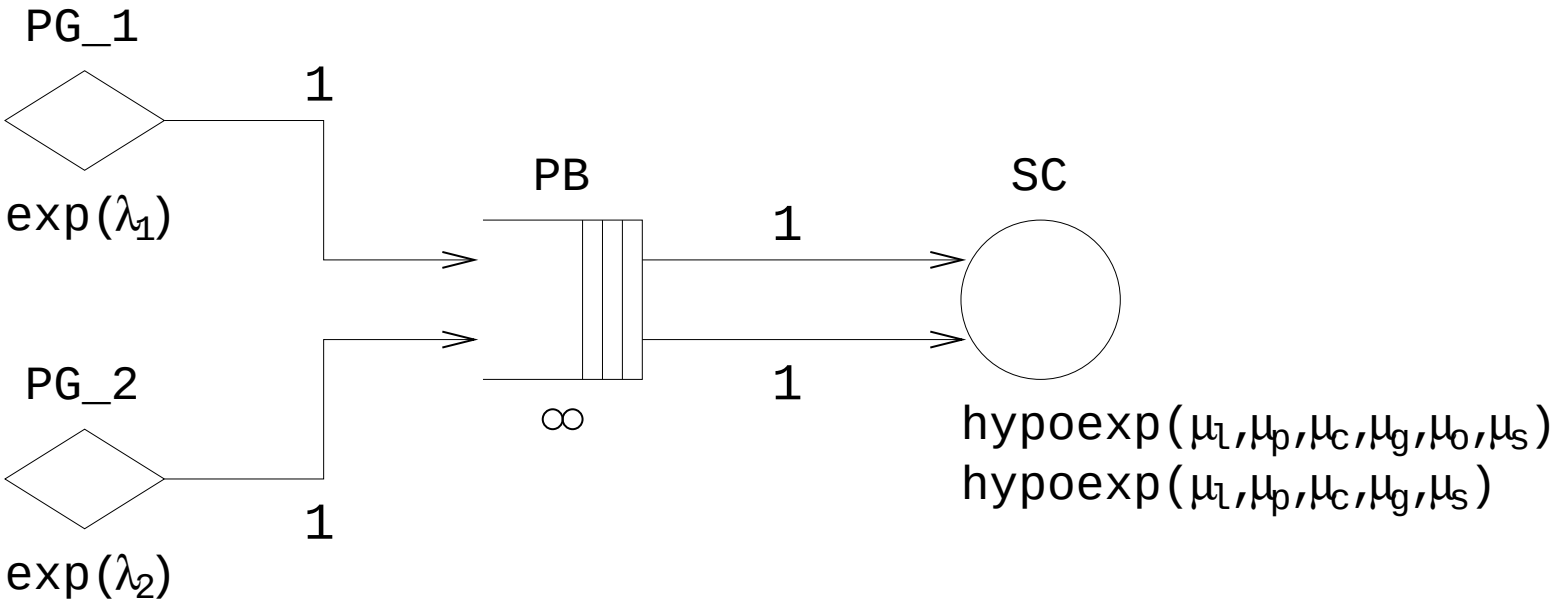
- Declaration of the topology:

```
ARCHI_ELEM_INSTANCES
  PG_1 : Program_Generator_Type(sc_lambda_1);
  PG_2 : Program_Generator_Type(sc_lambda_2);
  PB   : Program_Buffer_2C_Type();
  SC   : Seq_Compiler_Type(sc_mu_l,
                           sc_mu_p,
                           sc_mu_c,
                           sc_mu_g,
                           sc_mu_o,
                           sc_mu_s)

ARCHI_INTERACTIONS
  void

ARCHI_ATTACHMENTS
  FROM PG_1.deliver_prog TO PB.get_prog_1;
  FROM PG_2.deliver_prog TO PB.get_prog_2;
  FROM PB.put_prog_1     TO SC.select_prog_1;
  FROM PB.put_prog_2     TO SC.select_prog_2
```

- QN model (QS M/PH/1 for two classes of customers):



- In order to exploit the formulas for single-class QSs M/M/1, we have to merge the two classes into a single one and then introduce some average-preserving exponential distributions on the service side.
- Aggregated arrival rate:

$$\boxed{\lambda = \lambda_1 + \lambda_2}$$

- Convert the two hypoexponential service times into two average-preserving exponential service times:

$$\boxed{\begin{aligned} 1/\mu_1 &= 1/\mu_1 + 1/\mu_{\mathtt{p}} + 1/\mu_{\mathtt{c}} + 1/\mu_{\mathtt{g}} + 1/\mu_{\mathtt{o}} + 1/\mu_{\mathtt{s}} \\ 1/\mu_2 &= 1/\mu_1 + 1/\mu_{\mathtt{p}} + 1/\mu_{\mathtt{c}} + 1/\mu_{\mathtt{g}} + 1/\mu_{\mathtt{s}} \end{aligned}}$$

- Convert the convex combination of the two derived exponential service times into a single average-preserving exponential service time:

$$\boxed{1/\mu = \frac{\lambda_1}{\lambda} \cdot (1/\mu_1) + \frac{\lambda_2}{\lambda} \cdot (1/\mu_2)}$$

- Stability condition for the resulting QS M/M/1:

$$\boxed{\rho_{\mathtt{seq}} = \lambda/\mu < 1}$$

- Throughput of the sequential compiler system:

$$\boxed{\overline{T}_{\mathtt{seq}} = \lambda}$$

- Utilization of the sequential compiler system:

$$\boxed{\overline{U}_{\mathtt{seq}} = \rho_{\mathtt{seq}}}$$

- Mean number of programs in the sequential compiler system:

$$\boxed{\overline{N}_{\mathtt{seq}} = \frac{\rho_{\mathtt{seq}}}{1-\rho_{\mathtt{seq}}}}$$

- Mean compilation time of the sequential compiler system:

$$\boxed{\overline{R}_{\mathtt{seq}} = \frac{1}{\mu\cdot(1-\rho_{\mathtt{seq}})}}$$

- *Pipeline compiler*: simultaneous compilation of several programs at different phases.
- Phases are decoupled by means of intermediate buffers for 1-2 classes.
- Textual description header:

```
ARCHI_TYPE Pipeline_Compiler(const rate pc_lambda_1 :=  ,
                             const rate pc_lambda_2 :=  ,
                             const rate pc_mu_l     :=  ,
                             const rate pc_mu_p     :=  ,
                             const rate pc_mu_c     :=  ,
                             const rate pc_mu_g     :=  ,
                             const rate pc_mu_o     :=  ,
                             const rate pc_mu_s     :=  )
```

- `Program_Generator_Type` and `Program_Buffer_2C_Type` are unchanged.
- `Seq_Compiler_Type` needs to be replaced by six new AETs.

- Definition of the program buffer AET for one class of programs:

```
ARCHI_ELEM_TYPE Program_Buffer_1C_Type(void)

  BEHAVIOR
    Program_Buffer_1C(integer n := 0;
                      void) =
      choice
      {
        <get_prog, _> . Program_Buffer_1C(n + 1),
        cond(n > 0) -> <put_prog, _> . Program_Buffer_1C(n - 1)
      }

  INPUT_INTERACTIONS  SYNC UNI get_prog
  OUTPUT_INTERACTIONS SYNC UNI put_prog
```

- Definition of the lexical analyzer AET:

```
ARCHI_ELEM_TYPE Lexer_Type(const rate mu)

  BEHAVIOR
    Lexer(void; void) =
      choice
      {
        <select_prog_1, inf> . <recognize_tokens, exp(mu)> .
                                    <deliver_tokens_1, inf> . Lexer(),
        <select_prog_2, inf> . <recognize_tokens, exp(mu)> .
                                    <deliver_tokens_2, inf> . Lexer()
      }

  INPUT_INTERACTIONS  SYNC UNI select_prog_1; select_prog_2
  OUTPUT_INTERACTIONS SYNC UNI deliver_tokens_1; deliver_tokens_2
```

- Definition of the parser AET:

```
ARCHI_ELEM_TYPE Parser_Type(const rate mu)

  BEHAVIOR
    Parser(void; void) =
      choice
      {
        <select_tokens_1, inf> . <parse_phrases, exp(mu)> .
                                   <deliver_phrases_1, inf> . Parser(),
        <select_tokens_2, inf> . <parse_phrases, exp(mu)> .
                                   <deliver_phrases_2, inf> . Parser()
      }

  INPUT_INTERACTIONS  SYNC UNI select_tokens_1; select_tokens_2
  OUTPUT_INTERACTIONS SYNC UNI deliver_phrases_1; deliver_phrases_2
```

- Definition of the type checker AET:

```
ARCHI_ELEM_TYPE Checker_Type(const rate mu)

  BEHAVIOR
    Checker(void; void) =
      choice
      {
        <select_phrases_1, inf> . <check_phrases, exp(mu)> .
                                    <deliver_cphrases_1, inf> . Checker(),
        <select_phrases_2, inf> . <check_phrases, exp(mu)> .
                                    <deliver_cphrases_2, inf> . Checker()
      }

  INPUT_INTERACTIONS  SYNC UNI select_phrases_1; select_phrases_2
  OUTPUT_INTERACTIONS SYNC UNI deliver_cphrases_1; deliver_cphrases_2
```

- Definition of the intermediate code generator AET:

```
ARCHI_ELEM_TYPE Generator_Type(const rate mu)

  BEHAVIOR
    Generator(void; void) =
      choice
      {
        <select_cphrases_1, inf> . <generate_icode, exp(mu)> .
                                        <deliver_icode_1, inf> . Generator(),
        <select_cphrases_2, inf> . <generate_icode, exp(mu)> .
                                        <deliver_icode_2, inf> . Generator()
      }

  INPUT_INTERACTIONS  SYNC UNI select_cphrases_1; select_cphrases_2
  OUTPUT_INTERACTIONS SYNC UNI deliver_icode_1; deliver_icode_2
```

- Definition of the intermediate code optimizer AET (slightly different):

```
ARCHI_ELEM_TYPE Optimizer_Type(const rate mu)

  BEHAVIOR
    Optimizer(void; void) =
      <select_icode, inf> . <optimize_icode, exp(mu)> .
                                  <deliver_oicode, inf> . Optimizer()

  INPUT_INTERACTIONS  SYNC UNI select_icode
  OUTPUT_INTERACTIONS SYNC UNI deliver_oicode
```

- Definition of the code synthesizer AET:

```
ARCHI_ELEM_TYPE Synthesizer_Type(const rate mu)

  BEHAVIOR
    Synthesizer(void; void) =
      choice
      {
        <select_oicode_1, inf> . <synthesize_code, exp(mu)> . Synthesizer(),
        <select_icode_2, inf> . <synthesize_code, exp(mu)> . Synthesizer()
      }

  INPUT_INTERACTIONS  SYNC UNI select_oicode_1; select_icode_2
  OUTPUT_INTERACTIONS void
```

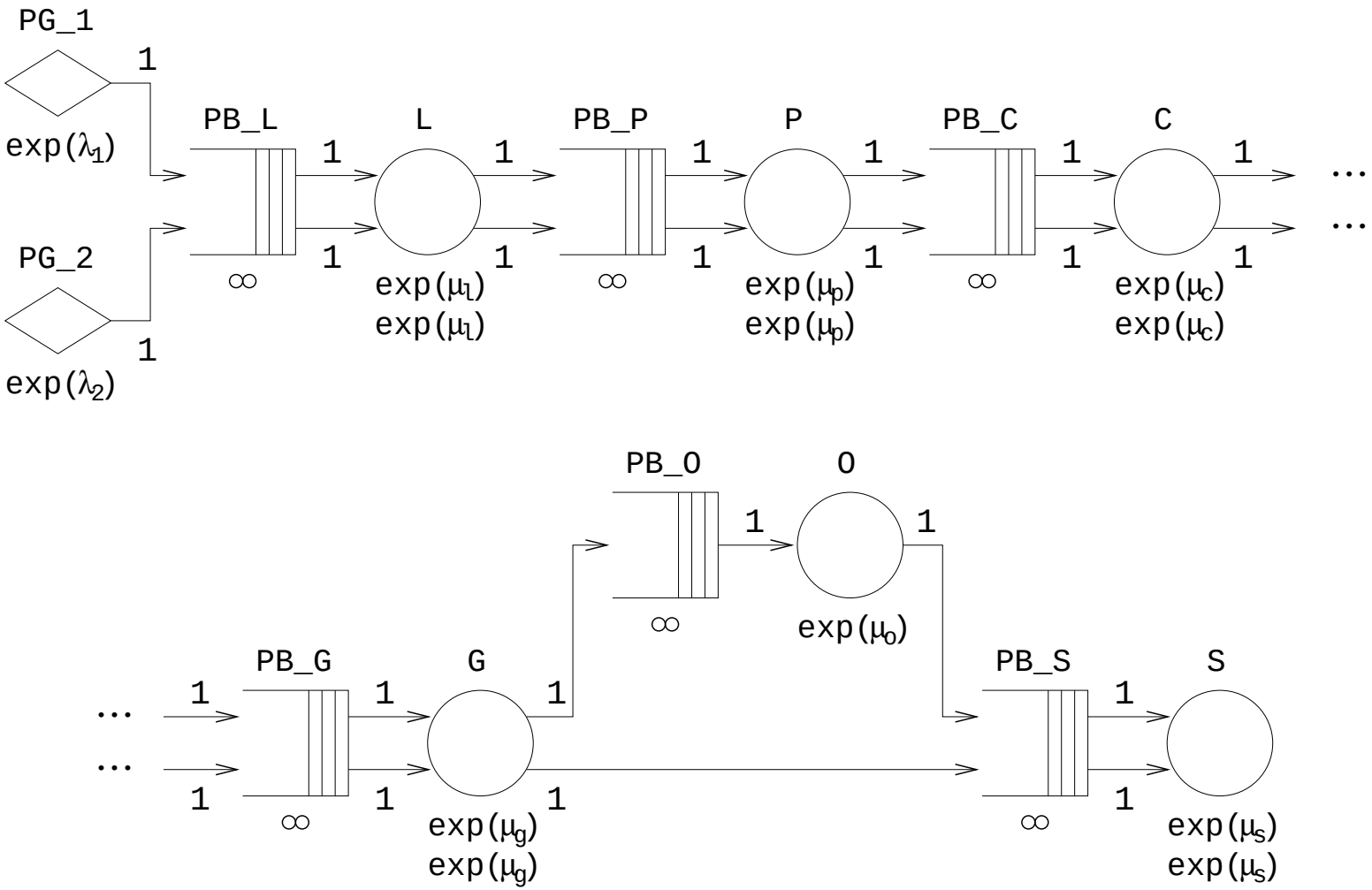
- Declaration of the topology:

```
ARCHI_ELEM_INSTANCES
  PG_1 : Program_Generator_Type(pc_lambda_1);
  PG_2 : Program_Generator_Type(pc_lambda_2);
  PB_L : Program_Buffer_2C_Type();
  L    : Lexer_Type(pc_mu_l);
  PB_P : Program_Buffer_2C_Type();
  P    : Parser_Type(pc_mu_p);
  PB_C : Program_Buffer_2C_Type();
  C    : Checker_Type(pc_mu_c);
  PB_G : Program_Buffer_2C_Type();
  G    : Generator_Type(pc_mu_g);
  PB_O : Program_Buffer_1C_Type();
  O    : Optimizer_Type(pc_mu_o);
  PB_S : Program_Buffer_2C_Type();
  S    : Synthesizer_Type(pc_mu_s)

ARCHI_INTERACTIONS
  void
```

```
ARCHI_ATTACHMENTS
  FROM PG_1.deliver_prog    TO PB_L.get_prog_1;
  FROM PG_2.deliver_prog    TO PB_L.get_prog_2;
  FROM PB_L.put_prog_1      TO L.select_prog_1;
  FROM PB_L.put_prog_2      TO L.select_prog_2;
  FROM L.deliver_tokens_1   TO PB_P.get_prog_1;
  FROM L.deliver_tokens_2   TO PB_P.get_prog_2;
  FROM PB_P.put_prog_1      TO P.select_tokens_1;
  FROM PB_P.put_prog_2      TO P.select_tokens_2;
  FROM P.deliver_phrases_1  TO PB_C.get_prog_1;
  FROM P.deliver_phrases_2  TO PB_C.get_prog_2;
  FROM PB_C.put_prog_1      TO C.select_phrases_1;
  FROM PB_C.put_prog_2      TO C.select_phrases_2;
  FROM C.deliver_cphrases_1 TO PB_G.get_prog_1;
  FROM C.deliver_cphrases_2 TO PB_G.get_prog_2;
  FROM PB_G.put_prog_1      TO G.select_cphrases_1;
  FROM PB_G.put_prog_2      TO G.select_cphrases_2;
  FROM G.deliver_icode_1    TO PB_O.get_prog;
  FROM G.deliver_icode_2    TO PB_S.get_prog_2;
  FROM PB_O.put_prog        TO O.select_icode;
  FROM O.deliver_oicode     TO PB_S.get_prog_1;
  FROM PB_S.put_prog_1      TO S.select_oicode_1;
  FROM PB_S.put_prog_2      TO S.select_icode_2
```

- QN model (open network of six QSs M/M/1):



- In order to exploit the BCMP theorem and the formulas for single-class QSs M/M/1, we have to merge the two classes into a single one.
- The service rate is uniquely defined for each phase.
- Aggregated arrival rate for all phases excluding optimization:

$$\boxed{\lambda = \lambda_1 + \lambda_2}$$

- The arrival rate for the optimization phase is λ_1.
- The probability that a program leaving the code generator enters the optimizer (resp. the synthesizer) is λ_1/λ (resp. λ_2/λ).

- Stability conditions for the various constituting QSs:

$$\begin{array}{lcll} \rho_{\mathtt{l}} & = & \lambda/\mu_{\mathtt{l}} & < 1 \\ \rho_{\mathtt{p}} & = & \lambda/\mu_{\mathtt{p}} & < 1 \\ \rho_{\mathtt{c}} & = & \lambda/\mu_{\mathtt{c}} & < 1 \\ \rho_{\mathtt{g}} & = & \lambda/\mu_{\mathtt{g}} & < 1 \\ \rho_{\mathtt{o}} & = & \lambda_1/\mu_{\mathtt{o}} & < 1 \\ \rho_{\mathtt{s}} & = & \lambda/\mu_{\mathtt{s}} & < 1 \end{array}$$

- Overall stability condition for the resulting QN:

$$\lambda < \min(\mu_{\mathtt{l}}, \mu_{\mathtt{p}}, \mu_{\mathtt{c}}, \mu_{\mathtt{g}}, \mu_{\mathtt{o}} \cdot \frac{\lambda}{\lambda_1}, \mu_{\mathtt{s}})$$

- Throughput of phase i:

$$\boxed{\begin{array}{lcll} \overline{T}_i & = & \lambda & \text{if } i \in \{\mathtt{l}, \mathtt{p}, \mathtt{c}, \mathtt{g}, \mathtt{s}\} \\ \overline{T}_\circ & = & \lambda_1 & \end{array}}$$

- Utilization of phase i:

$$\boxed{\overline{U}_i = \rho_i}$$

- Mean number of programs in phase i:

$$\boxed{\overline{N}_i = \frac{\rho_i}{1-\rho_i}}$$

- Mean processing time of phase i:

$$\boxed{\overline{R}_i = \frac{1}{\mu_i \cdot (1-\rho_i)}}$$

- Throughput of the pipeline compiler system:

$$\boxed{\overline{T}_{\texttt{pipe}} = \overline{T}_{\texttt{s}}}$$

- Utilization of the pipeline compiler system:

$$\boxed{\overline{U}_{\texttt{pipe}} = 1 - \prod_i (1 - \overline{U}_i)}$$

- Mean number of programs in the pipeline compiler system:

$$\boxed{\overline{N}_{\texttt{pipe}} = \sum_i \overline{N}_i}$$

- Mean compilation time of the pipeline compiler system:

$$\boxed{\overline{R}_{\texttt{pipe}} = \frac{\lambda_1}{\lambda} \cdot \sum_i \overline{R}_i + \frac{\lambda_2}{\lambda} \cdot \sum_{i \neq \texttt{o}} \overline{R}_i}$$

- *Concurrent compiler*: simultaneous compilation of several programs thanks to the presence of several replicas of the sequential compiler.
- All the replicas share the same buffer for incoming programs.
- Textual description header:

```
ARCHI_TYPE Concurrent_Compiler(const integer cc_seq_num  := 2,
                               const rate    cc_lambda_1 :=  ,
                               const rate    cc_lambda_2 :=  ,
                               const rate    cc_mu_l     :=  ,
                               const rate    cc_mu_p     :=  ,
                               const rate    cc_mu_c     :=  ,
                               const rate    cc_mu_g     :=  ,
                               const rate    cc_mu_o     :=  ,
                               const rate    cc_mu_s     :=  )
```

- Same AETs as `Sequential_Compiler`, with `put_prog_1` and `put_prog_2` becoming or-interactions.

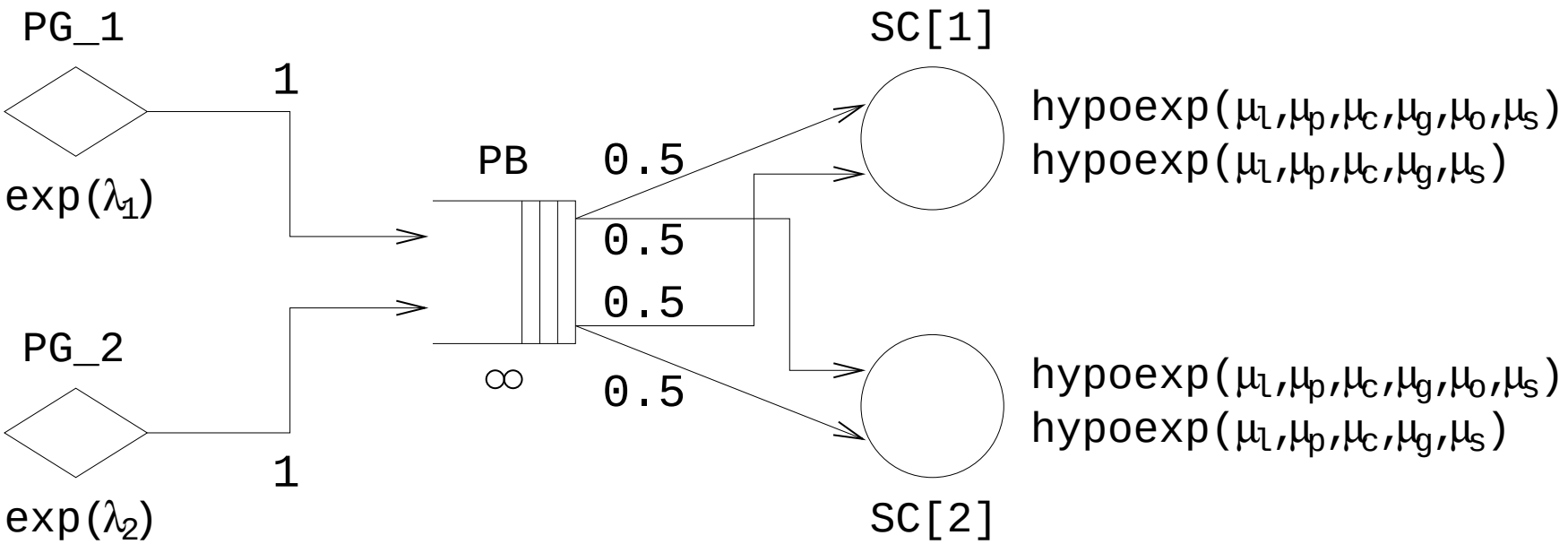
- Declaration of the topology:

```
ARCHI_ELEM_INSTANCES
  PG_1 : Program_Generator_Type(cc_lambda_1);
  PG_2 : Program_Generator_Type(cc_lambda_2);
  PB   : Program_Buffer_2C_Type();
  FOR_ALL 1 <= j <= cc_seq_num
    SC[j] : Seq_Compiler_Type(cc_mu_l, cc_mu_p, cc_mu_c, cc_mu_g, cc_mu_o, cc_mu_s)

ARCHI_INTERACTIONS
  void

ARCHI_ATTACHMENTS
  FROM PG_1.deliver_prog TO PB.get_prog_1;
  FROM PG_2.deliver_prog TO PB.get_prog_2;
  FOR_ALL 1 <= j <= cc_seq_num
    FROM PB.put_prog_1 TO SC[j].select_prog_1;
  FOR_ALL 1 <= j <= cc_seq_num
    FROM PB.put_prog_2 TO SC[j].select_prog_2
```

- QN model (QS M/PH/2 for two classes of customers):



- In order to exploit the formulas for single-class QSs M/M/2, we have to merge the two classes into a single one and then introduce some average-preserving exponential distributions on the service side.
- Manipulations similar to those for the sequential compiler:

$$\begin{aligned} \lambda &= \lambda_1 + \lambda_2 \\ 1/\mu &= \frac{\lambda_1}{\lambda} \cdot (1/\mu_{\mathtt{l}} + 1/\mu_{\mathtt{p}} + 1/\mu_{\mathtt{c}} + 1/\mu_{\mathtt{g}} + 1/\mu_{\mathtt{o}} + 1/\mu_{\mathtt{s}}) + \\ &\quad \frac{\lambda_2}{\lambda} \cdot (1/\mu_{\mathtt{l}} + 1/\mu_{\mathtt{p}} + 1/\mu_{\mathtt{c}} + 1/\mu_{\mathtt{g}} + 1/\mu_{\mathtt{s}}) \end{aligned}$$

- Stability condition for the resulting QS M/M/2:

$$\boxed{\rho_{\text{conc}} = \lambda/(2 \cdot \mu) < 1}$$

- Throughput of the concurrent compiler system:

$$\boxed{\overline{T}_{\text{conc}} = \lambda}$$

- Utilization of the concurrent compiler system:

$$\boxed{\overline{U}_{\text{conc}} = \frac{2 \cdot \rho_{\text{conc}}}{1 + \rho_{\text{conc}}}}$$

- Mean number of programs in the concurrent compiler system:

$$\boxed{\overline{N}_{\text{conc}} = \frac{2 \cdot \rho_{\text{conc}}}{1 - \rho_{\text{conc}}^2}}$$

- Mean compilation time of the concurrent compiler system:

$$\boxed{\overline{R}_{\text{conc}} = \frac{1}{\mu_{\text{conc}} \cdot (1 - \rho_{\text{conc}}^2)}}$$

- *Scenario-based comparison* of the three compiler architectures.
- The comparison must be fair (e.g., reuse symbolic actual values already introduced):
  - Identical actual values of service rates for each compilation phase across the three architectures.
  - The actual values of the arrival rates for each class of programs can be different from architecture to architecture, but must ensure that the frequencies p_1 and p_2 of the two classes of programs do not vary across the three architectures.
- Focus on throughput (mean number of programs compiled per unit of time).
- Under light load the specific architecture does not really matter, as the relations among the three throughputs directly depend on the relations among the three cumulative arrival rates.
- Assume heavy load when comparing.

- Each of the three architectures works close to its maximum throughput (cumulative arrival rates arbitrarily close to their corresponding overall service rates).
- From the stability conditions we derive that:

$$\begin{aligned} \overline{T}_{\mathtt{seq,max}} &= \mu \\ \overline{T}_{\mathtt{pipe,max}} &= \min(\mu_{\mathtt{l}}, \mu_{\mathtt{p}}, \mu_{\mathtt{c}}, \mu_{\mathtt{g}}, \mu_{\mathtt{o}}/p_{\mathtt{1}}, \mu_{\mathtt{s}}) \\ \overline{T}_{\mathtt{conc,max}} &= 2 \cdot \mu \end{aligned}$$

- Symbolic performance comparison in three scenarios:
  - All compilation phases have the same average duration $\mu_{\mathtt{avg}}^{-1}$.
  - There is a compilation phase whose average duration is several orders of magnitude greater than the average duration of the other phases.
  - The average duration of all phases ranges between $\mu_{\mathtt{max}}^{-1}$ and $\mu_{\mathtt{min}}^{-1}$, where the two endpoints may be several orders of magnitude apart.

- First scenario:

$$\begin{aligned} \overline{T}_{\texttt{seq,max}} &= (5+p_1)^{-1} \cdot \mu_{\texttt{avg}} \\ \overline{T}_{\texttt{pipe,max}} &= \mu_{\texttt{avg}} \\ \overline{T}_{\texttt{conc,max}} &= 2 \cdot (5+p_1)^{-1} \cdot \mu_{\texttt{avg}} \end{aligned}$$

- Ratios:

$$\begin{aligned} \overline{T}_{\texttt{pipe,max}} / \overline{T}_{\texttt{seq,max}} &= 5+p_1 \\ \overline{T}_{\texttt{pipe,max}} / \overline{T}_{\texttt{conc,max}} &= 2.5+0.5 \cdot p_1 \\ \overline{T}_{\texttt{conc,max}} / \overline{T}_{\texttt{seq,max}} &= 2 \end{aligned}$$

- The pipeline architecture wins in this case.

- Second scenario (suppose lexical analysis is the bottleneck):

$$\boxed{\begin{aligned} \overline{T}_{\texttt{seq,max}} &= \mu_{\texttt{l}} \\ \overline{T}_{\texttt{pipe,max}} &= \mu_{\texttt{l}} \\ \overline{T}_{\texttt{conc,max}} &= 2 \cdot \mu_{\texttt{l}} \end{aligned}}$$

- Ratios:

$$\boxed{\begin{aligned} \overline{T}_{\texttt{conc,max}} / \overline{T}_{\texttt{pipe,max}} &= 2 \\ \overline{T}_{\texttt{conc,max}} / \overline{T}_{\texttt{seq,max}} &= 2 \\ \overline{T}_{\texttt{pipe,max}} / \overline{T}_{\texttt{seq,max}} &= 1 \end{aligned}}$$

- The concurrent architecture wins in this case.

- Third scenario (generalization of the previous two):

$$\begin{array}{rcccl} (5+p_1)^{-1}\cdot\mu_{\texttt{min}} & \leq & \overline{T}_{\texttt{seq,max}} & \leq & (5+p_1)^{-1}\cdot\mu_{\texttt{max}} \\ \mu_{\texttt{min}} & \leq & \overline{T}_{\texttt{pipe,max}} & \leq & \mu_{\texttt{min}} \\ 2\cdot(5+p_1)^{-1}\cdot\mu_{\texttt{min}} & \leq & \overline{T}_{\texttt{conc,max}} & \leq & 2\cdot(5+p_1)^{-1}\cdot\mu_{\texttt{max}} \end{array}$$

- Ratios:

$$\begin{array}{rcccl} (5+p_1)\cdot\frac{\mu_{\texttt{min}}}{\mu_{\texttt{max}}} & \leq & \overline{T}_{\texttt{pipe,max}}/\overline{T}_{\texttt{seq,max}} & \leq & 5+p_1 \\ (2.5+0.5\cdot p_1)\cdot\frac{\mu_{\texttt{min}}}{\mu_{\texttt{max}}} & \leq & \overline{T}_{\texttt{pipe,max}}/\overline{T}_{\texttt{conc,max}} & \leq & 2.5+0.5\cdot p_1 \\ 2 & \leq & \overline{T}_{\texttt{conc,max}}/\overline{T}_{\texttt{seq,max}} & \leq & 2 \end{array}$$

- The concurrent one is twice more productive than the sequential one.
- The pipeline architecture can perform better/worse than the other two depending on the distance between $\mu_{\texttt{min}}$ and $\mu_{\texttt{max}}$.

Performance Measure Specification: MSL

- The last phase of PerfSel checks the chosen Æmilia description against specific performance requirements (possible approximations, general indicators).
- Notations for performance modeling are accompanied by notations for specifying performance measures.
- PerfSel formalizes performance measures by means of MSL.
- MSL combines ideas from reward structures, logic-based approaches, and state-/action-based approaches in a component-oriented flavor.
- Increasing usability up to the intrinsic limit of choosing correct numeric values for rewards.
- Core logic plus measure definition mechanism.

- Instant-of-time measures refer to a particular time instant.
- Interval-of-time/cumulative measures refer to a time interval.
- Both kinds of measures refer to stationary or transient behavior.
- Traditionally expressed through rewards to be associated with CTMC states and transitions:
  - The rate reward associated with a state expresses the rate at which a gain/loss is accumulated while sojourning in that state.
  - The instantaneous reward associated with a transition expresses the instantaneous gain/loss implied by the execution of that transition.

- Value of an instant-of-time performance measure specified through a reward structure over an action-labeled CTMC:

$$\boxed{\sum_{s \in S} R_{\mathrm{r}}(s) \cdot \pi[s] + \sum_{s \xrightarrow{a,\lambda} s'} R_{\mathrm{i}}(s, a, \lambda, s') \cdot \phi(s, a, \lambda, s')}$$

  where:

  - $R_{\mathrm{r}}(s)$ is the rate reward associated with s.
  - $\pi[s]$ is the probability of being in s at the considered instant of time.
  - $R_{\mathrm{i}}(s, a, \lambda, s')$ is the instantaneous reward associated with $s \xrightarrow{a,\lambda} s'$.
  - $\phi(s, a, \lambda, s')$ is the frequency of $s \xrightarrow{a,\lambda} s'$ at the considered instant of time: $\phi(s, a, \lambda, s') = \pi[s] \cdot \lambda$.

- The core logic of MSL associates rewards with states and transitions of an action-labeled CTMC (underlying an ÆMILIA description).
- Each global state s is viewed as a vector of local states $[z_1, \ldots, z_n]$ representing the current behavior of each AEI.
- Based on a set of first-order predicates concerned with:
  - ⊙ Local states in a set $Z \subseteq S_{\text{local}}$ relevant to the measure.
  - ⊙ Activities in a set $A \subseteq \mathit{Name}$ relevant to the measure.
- Six formula schemas resulting from the combination of:
  - ⊙ Universal quantification, existential quantification (over Z or A).
  - ⊙ Direct state rewards, indirect state rewards, transition rewards.
- Universal closure with respect to S of the six formula schemas.

- First formula schema:

$$\forall z \in Z\,(is\_local(z,s) \Rightarrow eq(lstate\_contrib(z,s), lstate\_rew(z))) \Rightarrow$$
$$eq(state\_rew(s), sum\_lstate\_contrib(s,Z))$$

- Every local state $z \in Z$ of the current state s directly provides a contribution of value $lstate\_rew(z)$ to the rate at which the reward is accumulated while staying in that state.

- Local state contribution additivity assumption: the contributions of all the local states of s belonging to Z have to be summed up.

- Second formula schema:

$$\forall a \in A\,(is\_trans(s, a, \lambda, s') \Rightarrow eq(act\_contrib(s, a, \lambda, s'), act\_rew(a, \lambda))) \Rightarrow$$
$$eq(state\_rew(s), sum\_act\_contrib(s, A))$$

- Each transition labeled with $a \in A$ that departs from the current state s indirectly provides a contribution of value $act\_rew(a, \lambda)$ to the rate at which the reward is accumulated while staying in that state.
- Activity contribution additivity assumption: the contributions of all the outgoing transitions of s labeled with $a \in A$ have to be summed up.

- Third formula schema:

$$\forall a \in A\,(is\_trans(s, a, \lambda, s') \Rightarrow \\ eq(trans\_rew(s, a, \lambda, s'), act\_rew(a, \lambda)))$$

- Each transition labeled with $a \in A$ that departs from the current state s gains an instantaneous reward of value $act\_rew(a, \lambda)$ whenever it is executed.

- Fourth formula schema:

$$\exists z \in Z\,(is\_local(z, s)) \Rightarrow$$
$$eq(state\_rew(s), choose\_lstate\_rew(s, Z, cf))$$

- The current state s gains a contribution to the rate at which the reward is accumulated while staying in that state if at least one of its local states belongs to Z.

- The value of the contribution is obtained by applying a choice function cf to the direct state rewards $lstate\_rew(z)$ associated with the local states of s belonging to Z.

- Fifth formula schema:

$$\exists a \in A\,(is\_trans(s, a, \lambda, s')) \Rightarrow$$
$$eq(state\_rew(s), choose\_act\_rew(s, A, cf))$$

- The current state s gains a contribution to the rate at which the reward is accumulated while staying in that state if at least one of its outgoing transitions is labeled with $a \in A$.

- The value of the contribution is obtained by applying a choice function cf to the indirect state rewards $act\_rew(a, \lambda)$ associated with the outgoing transitions of s labeled with $a \in A$.

- Sixth formula schema:

$$\exists a \in A\,(is\_trans(s, a, \lambda, s')) \Rightarrow$$
$$eq(trans\_rew(choose\_trans(s, A, cf)), choose\_trans\_rew(s, A, cf))$$

- Only one of the transitions labeled with $a \in A$ that depart from the current state s gains an instantaneous reward of value $act\_rew(a, \lambda)$ upon execution.

- This transition is obtained by applying a choice function cf, which takes into account the transition rewards $act\_rew(a, \lambda)$ associated with the outgoing transitions of s labeled with $a \in A$ multiplied by the frequencies of the transitions themselves.

- The measure definition mechanism of MSL enhances usability by means of a component-oriented level on top of the core logic.
- Association of mnemonic names with performance measures defined through sets of formula schemas of the core logic.
- Parameterization of performance measures with respect to component behaviors and activities.
- Invocation of performance measure identifiers allowed in arithmetical operations and mathematical functions.

- Syntax of a performance measure definition:

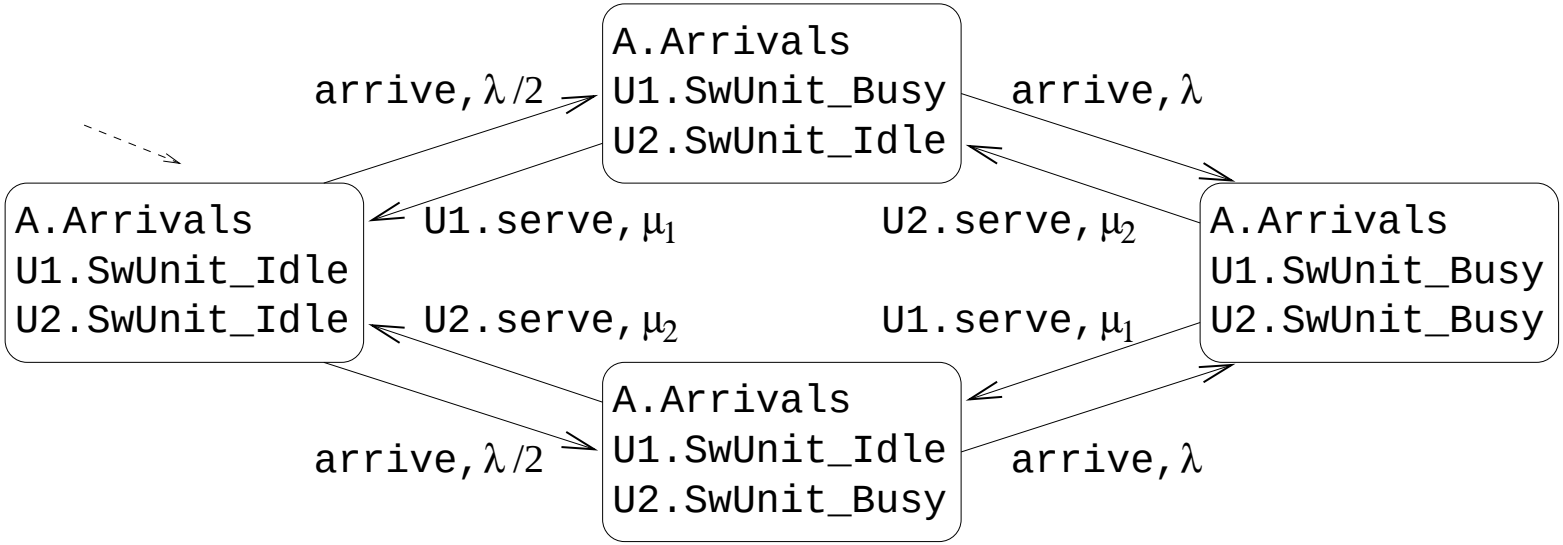
  `MEASURE` ◁*name*▷ (◁*formal component-oriented parameters*▷) `IS` ◁*body*▷

- The name is the identifier of the performance measure.
- When used inside the body of another performance measure, it denotes the value of the measure it represents (computed on a given action-labeled CTMC).
- Formal component-oriented parameters given by component behaviors together with possibly associated real numbers result in local state sets for formula schemas of the first/fourth type.
- Formal component-oriented parameters given by component activities result in activity sets for formula schemas of all the other types.

- The body of a basic measure definition is a set of formula schemas of the core logic.
- The body of a derived measure definition is an expression involving identifiers of previously defined measures, arithmetical operators, and mathematical functions.
- Libraries of basic measure definitions (provided by performance experts) and of derived measure definitions (provided by nonexperts too).
- The difficulties with performance measure specification should be hidden inside the body, especially of basic measures.
- Usability is achieved by requesting the designer to provide only suitable actual component-oriented parameters.

- **Example**: illustrating the core logic.
- Service center composed of two independent software units.
- Requests arrive at the service center at rate $\lambda \in \mathbb{R}_{>0}$.
- An incoming request has the same probability to be served by the two software units when both are idle.
- An incoming request cannot be accepted when both software units are busy as there is no buffer.
- The two software units serve requests at rate $\mu_1, \mu_2 \in \mathbb{R}_{>0}$ respectively.

- Underlying action-labeled CTMC model:



- First formula schema:
  - System throughput:

$$\begin{aligned} Z &= \{\texttt{U1.SwUnit\_Busy}, \texttt{U2.SwUnit\_Busy}\} \\ \mathit{lstate\_rew}(\texttt{U1.SwUnit\_Busy}) &= \mu_1 \\ \mathit{lstate\_rew}(\texttt{U2.SwUnit\_Busy}) &= \mu_2 \end{aligned}$$

  - Throughput of `U1`:

$$\begin{aligned} Z &= \{\texttt{U1.SwUnit\_Busy}\} \\ \mathit{lstate\_rew}(\texttt{U1.SwUnit\_Busy}) &= \mu_1 \end{aligned}$$

- Second formula schema:
  - System throughput:

$$\begin{aligned} A &= \{\texttt{U1.serve}, \texttt{U2.serve}\} \\ act\_rew(\texttt{U1.serve}, any) &= \mu_1 \\ act\_rew(\texttt{U2.serve}, any) &= \mu_2 \end{aligned}$$

  - Throughput of `U1`:

$$\begin{aligned} A &= \{\texttt{U1.serve}\} \\ act\_rew(\texttt{U1.serve}, any) &= \mu_1 \end{aligned}$$

- Third formula schema:
  - System throughput:

$$\begin{aligned} A &= \{\texttt{U1.serve}, \texttt{U2.serve}\} \\ act\_rew(\texttt{U1.serve}, any) &= 1 \\ act\_rew(\texttt{U2.serve}, any) &= 1 \end{aligned}$$

  - Throughput of `U1`:

$$\begin{aligned} A &= \{\texttt{U1.serve}\} \\ act\_rew(\texttt{U1.serve}, any) &= 1 \end{aligned}$$

- Fourth formula schema:
  - System utilization:

$$\begin{aligned} Z &= \{\texttt{U1.SwUnit\_Busy}, \texttt{U2.SwUnit\_Busy}\} \\ \mathit{lstate\_rew}(\texttt{U1.SwUnit\_Busy}) &= 1 \\ \mathit{lstate\_rew}(\texttt{U2.SwUnit\_Busy}) &= 1 \\ cf &= \min \end{aligned}$$

  - Utilization of `U1`:

$$\begin{aligned} Z &= \{\texttt{U1.SwUnit\_Busy}\} \\ \mathit{lstate\_rew}(\texttt{U1.SwUnit\_Busy}) &= 1 \\ cf &= \min \end{aligned}$$

- Fifth formula schema:
  - System utilization:

$$\begin{aligned} A &= \{\texttt{U1.serve}, \texttt{U2.serve}\} \\ act\_rew(\texttt{U1.serve}, any) &= 1 \\ act\_rew(\texttt{U2.serve}, any) &= 1 \\ cf &= \min \end{aligned}$$

  - Utilization of `U1`:

$$\begin{aligned} A &= \{\texttt{U1.serve}\} \\ act\_rew(\texttt{U1.serve}, any) &= 1 \\ cf &= \min \end{aligned}$$

- Sixth formula schema:
  - ⊙ Actual arrival rate (less than λ due to the absence of a buffer):

$$\begin{aligned} A &= \{\texttt{arrive}\} \\ act\_rew(\texttt{arrive}, any) &= 1 \\ cf &= \min \end{aligned}$$

- **Example**: illustrating the measure definition mechanism.
- Parameterized definitions of the typical average performance indicators for a component-oriented description:
  - Throughput.
  - Utilization.
  - Mean queue length.
  - Mean response time.
- When using them, the designer has only to provide suitable actual component-oriented parameters.

- Definition of throughput:

$$\begin{array}{l}\texttt{MEASURE}\ \mathit{throughput}(C_1.a_1, \ldots, C_n.a_n)\ \texttt{IS} \\ \quad \forall a \in \{C_1.a_1, \ldots, C_n.a_n\} \\ \quad\quad (\mathit{is\_trans}(s, a, \lambda, s') \Rightarrow \\ \quad\quad\quad \mathit{eq}(\mathit{act\_contrib}(s, a, \lambda, s'), \mathit{act\_rew}(a, \lambda))) \Rightarrow \\ \quad\quad\quad\quad \mathit{eq}(\mathit{state\_rew}(s), \mathit{sum\_act\_contrib}(s, \{C_1.a_1, \ldots, C_n.a_n\}))\end{array}$$

  where $\mathit{act\_rew}(a, \lambda) = \lambda$ for all $a \in \{C_1.a_1, \ldots, C_n.a_n\}$.

- Alternative definition of throughput:

$$\begin{array}{l}\texttt{MEASURE}\ \mathit{throughput}(C_1.a_1, \ldots, C_n.a_n)\ \texttt{IS} \\ \quad \forall a \in \{C_1.a_1, \ldots, C_n.a_n\} \\ \quad\quad (\mathit{is\_trans}(s, a, \lambda, s') \Rightarrow \\ \quad\quad\quad \mathit{eq}(\mathit{trans\_rew}(s, a, \lambda, s'), 1))\end{array}$$

- Definition of utilization:

$$\begin{array}{l}\texttt{MEASURE}\ \mathit{utilization}(C.a_1, \ldots, C.a_n)\ \texttt{IS} \\ \quad \exists a \in \{C.a_1, \ldots, C.a_n\} \\ \quad\quad (\mathit{is\_trans}(s, a, \lambda, s')) \Rightarrow \\ \quad\quad\quad \mathit{eq}(\mathit{state\_rew}(s), \mathit{choose\_act\_rew}(s, \{C.a_1, \ldots, C.a_n\}, \min))\end{array}$$

where $act\_rew(a, any) = 1$ for all $a \in \{C.a_1, \ldots, C.a_n\}$.

- Definition of mean queue length:

$$\texttt{MEASURE}\ mean\_queue\_length(C.B_1(k_1), \ldots, C.B_n(k_n))\ \texttt{IS}$$
$$\exists z \in \{C.B_1, \ldots, C.B_n\}$$
$$(is\_local(z, s)) \Rightarrow$$
$$eq(state\_rew(s), choose\_lstate\_rew(s, \{C.B_1, \ldots, C.B_n\}, \min))$$

where $lstate\_rew(z) = k_i$ whenever $z = C.B_i$ for some $1 \leq i \leq n$.

- Definition of mean response time:

$$\texttt{MEASURE}\ mean\_response\_time(C.B_1(k_1/\lambda), \ldots, C.B_n(k_n/\lambda))\ \texttt{IS}$$
$$\exists z \in \{C.B_1, \ldots, C.B_n\}$$
$$(is\_local(z, s)) \Rightarrow$$
$$eq(state\_rew(s), choose\_lstate\_rew(s, \{C.B_1, \ldots, C.B_n\}, \min))$$

where $lstate\_rew(z) = k_i/\lambda$ whenever $z = C.B_i$ for some $1 \leq i \leq n$.

- Definition of mean queue length over a set of components (derived measure):

$$\texttt{MEASURE}\ total\_mean\_queue\_length(C_1.B_{1,1}(k_{1,1}), \ldots, C_1.B_{1,n_1}(k_{1,n_1}),$$
$$C_2.B_{2,1}(k_{2,1}), \ldots, C_2.B_{2,n_2}(k_{2,n_2}),$$
$$\vdots$$
$$C_m.B_{m,1}(k_{m,1}), \ldots, C_m.B_{m,n_m}(k_{m,n_m}))\ \texttt{IS}$$
$$mean\_queue\_length(C_1.B_{1,1}(k_{1,1}), \ldots, C_1.B_{1,n_1}(k_{1,n_1})) +$$
$$mean\_queue\_length(C_2.B_{2,1}(k_{2,1}), \ldots, C_2.B_{2,n_2}(k_{2,n_2})) +$$
$$\vdots$$
$$mean\_queue\_length(C_m.B_{m,1}(k_{m,1}), \ldots, C_m.B_{m,n_m}(k_{m,n_m}))$$

Comparison with Related Techniques

- PerfSel generates QN models from component-oriented descriptions rather than labeled transition systems in order to gain effectiveness.
- PerfSel focuses on typical average performance indicators instead of specific performance measures in order to quicken the comparison among alternative architectural designs (precise analysis at the end).
- PerfSel resorts to the component-oriented specification of performance measures in order to strengthen usability.
- Integrated view of functional and nonfunctional aspects that:
  - Overcomes the well-known drawbacks related to the insularity of performance modeling.
  - Ensures consistency of results derived from performance evaluation with results obtained from functional verification.

References

[1] G.D. Abowd, R. Allen, and D. Garlan, *"Formalizing Style to Understand Descriptions of Software Architecture"*, in ACM Trans. on Software Engineering and Methodology 4:319–364, 1995.

[2] S. Abramsky, *"Observational Equivalence as a Testing Equivalence"*, in Theoretical Computer Science 53:225–241, 1987.

[3] A. Acquaviva, A. Aldini, M. Bernardo, A. Bogliolo, E. Bontà, and E. Lattanzi, *"A Methodology Based on Formal Methods for Predicting the Impact of Dynamic Power Management"*, in *Formal Methods for Mobile Computing*, Springer, LNCS 3465:155–189, 2005.

[4] A. Aldini and M. Bernardo, *"On the Usability of Process Algebra: An Architectural View"*, in Theoretical Computer Science 335:281–329, 2005.

[5] A. Aldini and M. Bernardo, *"Mixing Logics and Rewards for the Component-Oriented Specification of Performance Measures"*, in Theoretical Computer Science 382:3–23, 2007.

[6] A. Aldini and M. Bernardo, *"A Formal Approach to the Integrated Analysis of Security and QoS"*, in Reliability Engineering & System Safety 92:1503–1520, 2007.

[7] J. Aldrich, C. Chambers, and D. Notkin, *"ArchJava: Connecting Software Architecture to Implementation"*, in Proc. of the *24th Int. Conf. on Software Engineering (ICSE 2002)*, ACM Press, pp. 187–197, Orlando (FL), 2002.

[8] R. Allen, R. Douence, and D. Garlan, *"Specifying and Analyzing Dynamic Software Architectures"*, in Proc. of the *1st Int. Conf. on Fundamental Approaches to Software Engineering (FASE 1998)*, Springer, LNCS 1382:21–37, Lisbon (Portugal), 1998.

[9] R. Allen and D. Garlan, *"A Formal Basis for Architectural Connection"*, in ACM Trans. on Software Engineering and Methodology 6:213–249, 1997.

[10] F. Aquilani, S. Balsamo, and P. Inverardi, *"Performance Analysis at the Software Architectural Design Level"*, in Performance Evaluation 45:205–221, 2001.

[11] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop, *"Ready-Trace Semantics for Concrete Process Algebra with the Priority Operator"*, in Computer Journal 30:498–506, 1987.

[12] J.C.M. Baeten and W.P. Weijland, *"Process Algebra"*, Cambridge University Press, 1990.

[13] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen, *"Automated Performance and Dependability Evaluation Using Model Checking"*, in *Performance Evaluation of Complex Systems: Techniques and Tools*, Springer, LNCS 2459:261–289, 2002.

[14] S. Balsamo, M. Bernardo, and M. Simeoni, *"Performance Evaluation at the Software Architecture Level"*, in [24]:207–258.

[15] F. Baskett, K.M. Chandy, R.R. Muntz, and G. Palacios, *"Open, Closed, and Mixed Networks of Queues with Different Classes of Customers"*, in Journal of the ACM 22:248–260, 1975.

[16] H. Bekic, *"Towards a Mathematical Theory of Processes"*, Tech. Rep. TR 25.125, IBM Laboratory Vienna (Austria), 1971 (published in *"Programming Languages and Their Definition"*, Springer, LNCS 177:168–206, 1984).

[17] J.A. Bergstra, A. Ponse, and S.A. Smolka (eds.), *"Handbook of Process Algebra"*, Elsevier, 2001.

[18] M. Bernardo and E. Bontà, *"Generating Well-Synchronized Multithreaded Programs from Software Architecture Descriptions"*, in Proc. of the *4th Working IEEE/IFIP Conf. on Software Architecture (WICSA 2004)*, IEEE-CS Press, pp. 167–176, Oslo (Norway), 2004.

[19] M. Bernardo and E. Bontà, *"Preserving Architectural Properties in Multithreaded Code Generation"*, in Proc. of the *7th Int. Conf. on Coordination Models and Languages (COORDINATION 2005)*, Springer, LNCS 3454:188–203, Namur (Belgium), 2005.

[20] M. Bernardo, E. Bontà, and A. Aldini, *"Handling Communications in Process Algebraic Architectural Description Languages: Modeling, Verification, and Implementation"*, in Journal of Systems and Software 83:1404–1429, 2010.

[21] M. Bernardo and M. Bravetti, *"Performance Measure Sensitive Congruences for Markovian Process Algebras"*, in Theoretical Computer Science 290:117–160, 2003.

[22] M. Bernardo, P. Ciancarini, and L. Donatiello, *"Architecting Families of Software Systems with Process Algebras"*, in ACM Trans. on Software Engineering and Methodology 11:386–426, 2002.

[23] M. Bernardo, L. Donatiello, and P. Ciancarini, *"Stochastic Process Algebra: From an Algebraic Formalism to an Architectural Description Language"*, in *Performance Evaluation of Complex Systems: Techniques and Tools*, Springer, LNCS 2459:236–260, 2002.

[24] M. Bernardo and P. Inverardi (eds.), *"Formal Methods for Software Architectures"*, Springer, LNCS 2804, 2003.

[25] A. Bertolino, F. Corradini, P. Inverardi, and H. Muccini, *"Deriving Test Plans from Architectural Descriptions"*, in Proc. of the *22nd Int. Conf. on Software Engineering (ICSE 2000)*, ACM Press, pp. 220–229, Limerick (Ireland), 2000.

[26] A. Bertolino, P. Inverardi, and H. Muccini, *"An Explorative Journey from Architectural Tests Definition Downto Code Tests Execution"*, in Proc. of the *23rd Int. Conf. on Software Engineering (ICSE 2001)*, ACM Press, pp. 211–220, Toronto (Canada), 2001.

[27] B. Bloom, S. Istrail, and A.R. Meyer, *"Bisimulation Can't Be Traced"*, in Journal of the ACM 42:232–268, 1995.

[28] T. Bolognesi and E. Brinksma, *"Introduction to the ISO Specification Language LOTOS"*, in Computer Networks and ISDN Systems 14:25–59, 1987.

[29] E. Bontà, M. Bernardo, J. Magee, and J. Kramer, *"Synthesizing Concurrency Control Components from Process Algebraic Specifications"*, in Proc. of the *8th Int. Conf. on Coordination Models and Languages (COORDINATION 2006)*, Springer, LNCS 4038:28–43, Bologna (Italy), 2006.

[30] A. Bracciali, A. Brogi, and C. Canal, *"A Formal Approach to Component Adaptation"*, in Journal of Systems and Software 74:45–54, 2005.

[31] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe, *"A Theory of Communicating Sequential Processes"*, in Journal of the ACM 31:560–599, 1984.

[32] J.P. Buzen, *"Computational Algorithms for Closed Queueing Networks with Exponential Servers"*, in Comm. of the ACM 16:527–531, 1973.

[33] C. Canal, E. Pimentel, and J.M. Troya, *"Specification and Refinement of Dynamic Software Architectures"*, in Proc. of the *1st Working IFIP Conf. on Software Architecture (WICSA 1999)*, Kluwer, pp. 107–126, San Antonio (TX), 1999.

[34] C. Canal, E. Pimentel, and J.M. Troya, *"Compatibility and Inheritance in Software Architectures"*, in Science of Computer Programming 41:105–138, 2001.

[35] K.M. Chandy and C.H. Sauer, *"Computational Algorithms for Product Form Queueing Networks"*, in Comm. of the ACM 23:573–583, 1980.

[36] G. Clark, S. Gilmore, J. Hillston, and M. Ribaudo, *"Exploiting Modal Logic to Express Performance Measures"*, in Proc. of the *11th Int. Conf. on Modeling Techniques and Tools for Computer Performance Evaluation (PERFORMANCE TOOLS 2000)*, Springer, LNCS 1786:247–261, Schaumburg (IL), 2000.

[37] E.M. Clarke, O. Grumberg, and D.A. Peled, *"Model Checking"*, MIT Press, 1999.

[38] R. Cleaveland and M. Hennessy, *"Testing Equivalence as a Bisimulation Equivalence"*, in Formal Aspects of Computing 5:1–20, 1993.

[39] R. Cleaveland, J. Parrow, and B. Steffen, *"The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems"*, in ACM Trans. on Programming Languages and Systems 15:36–72, 1993.

[40] R. Cleaveland and O. Sokolsky, *"Equivalence and Preorder Checking for Finite-State Systems"*, in [17]:391–424.

[41] A.E. Conway and N.D. Georganas, *"RECAL – A New Efficient Algorithm for the Exact Analysis of Multiple-Chain Closed Queueing Networks"*, in Journal of the ACM 33:786–791, 1986.

[42] T.M. Cover and J.A. Thomas, *"Elements of Information Theory"*, John Wiley & Sons, 1991.

[43] T.R. Dean and J.R. Cordy, *"A Syntactic Theory of Software Architecture"*, in IEEE Trans. on Software Engineering 21:302–313, 1995.

[44] R. De Nicola and M. Hennessy, *"Testing Equivalences for Processes"*, in Theoretical Computer Science 34:83–133, 1984.

[45] F. De Remer and H.H. Kron, *"Programming-in-the-Large Versus Programming-in-the-Small"*, in IEEE Trans. on Software Engineering 2:80–86, 1976.

[46] P.H. Feiler, B.A. Lewis, and S. Vestal, *"The SAE Architecture Analysis & Design Language (AADL) – A Standard for Engineering Performance Critical Systems"*, in Proc. of the *14th IEEE Int. Symp. on Computer-Aided Control Systems Design (CACSD 2006)*, IEEE-CS Press, pp. 1206–1211, Munich (Germany), 2006.

[47] D. Ferrari, *"Considerations on the Insularity of Performance Evaluation"*, in IEEE Trans. on Software Engineering 12:678–683, 1986.

[48] H. Garavel, R. Mateescu, F. Lang, and W. Serwe, *"CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes"*, in Proc. of the *19th Int. Conf. on Computer Aided Verification (CAV 2007)*, Springer, LNCS 4590:158–163, Berlin (Germany), 2007.

[49] H. Garavel and M. Sighireanu, *"A Graphical Parallel Composition Operator for Process Algebras"*, in Proc. of the *IFIP Joint Int. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV 1999)*, Kluwer, pp. 185–202, Beijing (China), 1999.

[50] D. Garlan, R. Allen, and J. Ockerbloom, *"Exploiting Style in Architectural Design Environments"*, in Proc. of the *2nd ACM Int. Symp. on the Foundations of Software Engineering (FSE-2)*, ACM Press, pp. 175–188, New Orleans (LA), 1994.

[51] D. Garlan, R. Allen, and J. Ockerbloom, *"Architectural Mismatch: Why Reuse Is So Hard"*, in IEEE Software 12(6):17–26, 1995.

[52] D. Garlan, R.T. Monroe, and D. Wile, *"Acme: Architectural Description of Component-Based Systems"*, in *Foundations of Component-Based Systems*, Cambridge University Press, pp. 47–68, 2000.

[53] R.J. van Glabbeek, *"The Linear Time – Branching Time Spectrum I"*, in [17]:3–99.

[54] R.J. van Glabbeek, S.A. Smolka, and B. Steffen, *"Reactive, Generative and Stratified Models of Probabilistic Processes"*, in Information and Computation 121:59–80, 1995.

[55] R.J. van Glabbeek and W.P. Weijland, *"Branching Time and Abstraction in Bisimulation Semantics"*, in Journal of the ACM 43:555–600, 1996.

[56] G. Gössler and J. Sifakis, *"Composition for Component-Based Modeling"*, in Proc. of the *1st Int. Symp. on Formal Methods for Components and Objects (FMCO 2002)*, Springer, LNCS 2852:443–466, Leiden (The Netherlands), 2002.

[57] S. Graf, B. Steffen, and G. Lüttgen, *"Compositional Minimization of Finite State Systems Using Interface Specifications"*, in Formal Aspects of Computing 8:607–616, 1996.

[58] B.R. Haverkort and K.S. Trivedi, *"Specification Techniques for Markov Reward Models"*, in Discrete Event Dynamic Systems: Theory and Applications 3:219–247, 1993.

[59] M. Hennessy, *"Acceptance Trees"*, in Journal of the ACM 32:896–928, 1985.

[60] M. Hennessy, *"Algebraic Theory of Processes"*, MIT Press, 1988.

[61] M. Hennessy and R. Milner, *"Algebraic Laws for Nondeterminism and Concurrency"*, in Journal of the ACM 32:137–162, 1985.

[62] C.A.R. Hoare, *"Communicating Sequential Processes"*, Prentice Hall, 1985.

[63] R.A. Howard, *"Dynamic Probabilistic Systems"*, John Wiley & Sons, 1971.

[64] P. Inverardi and S. Uchitel, *"Proving Deadlock Freedom in Component-Based Programming"*, in Proc. of the *4th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2001)*, Springer, LNCS 2029:60–75, Genoa (Italy), 2001.

[65] P. Inverardi and A.L. Wolf, *"Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model"*, in IEEE Trans. on Software Engineering 21:373–386, 1995.

[66] P. Inverardi, A.L. Wolf, and D. Yankelevich, *"Static Checking of System Behaviors Using Derived Component Assumptions"*, in ACM Trans. on Software Engineering and Methodology 9:239–272, 2000.

[67] P.C. Kanellakis and S.A. Smolka, *"CCS Expressions, Finite State Processes, and Three Problems of Equivalence"*, in Information and Computation 86:43–68, 1990.

[68] L. Kleinrock, *"Queueing Systems"*, John Wiley & Sons, 1975.

[69] J. Kramer and J. Magee, *"Exposing the Skeleton in the Coordination Closet"*, in Proc. of the *2nd Int. Conf. on Coordination Models and Languages (COORDINATION 1997)*, Springer, LNCS 1282:18–31, Berlin (Germany), 1997.

[70] K.G. Larsen and A. Skou, *"Bisimulation Through Probabilistic Testing"*, in Information and Computation 94:1–28, 1991.

[71] E.D. Lazowska, J. Zahorjan, G. Scott Graham, and K.C. Sevcik, *"Quantitative System Performance: Computer System Analysis Using Queueing Network Models"*, Prentice Hall, 1984.

[72] D. Le Metayer, *"Describing Software Architecture Styles Using Graph Grammars"*, in IEEE Trans. on Software Engineering 24:521–533, 1998.

[73] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann, *"Specification and Analysis of System Architecture Using Rapide"*, in IEEE Trans. on Software Engineering 21:336–355, 1995.

[74] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, *"Specifying Distributed Software Architectures"*, in Proc. of the *5th European Software Engineering Conf. (ESEC 1995)*, Springer, LNCS 989:137–153, Barcelona (Spain), 1995.

[75] J. Magee and J. Kramer, *"Concurrency: State Models & Java Programs"*, John Wiley & Sons, 1999.

[76] N. Medvidovic, P. Oreizy, J.E. Robbins, and R.N. Taylor, *"Using Object-Oriented Typing to Support Architectural Design in the C2 Style"*, in Proc. of the *4th ACM Int. Symp. on the Foundations of Software Engineering (FSE-4)*, ACM Press, pp. 24–32, San Francisco (CA), 1996.

[77] N. Medvidovic, D.S. Rosenblum, D.F. Redmiles, and J.E. Robbins, *"Modeling Software Architectures in the Unified Modeling Language"*, in ACM Trans. on Software Engineering and Methodology 11:2–57, 2002.

[78] N. Medvidovic and R.N. Taylor, *"A Classification and Comparison Framework for Software Architecture Description Languages"*, in IEEE Trans. on Software Engineering 26:70–93, 2000.

[79] R. Milner, *"A Calculus of Communicating Systems"*, Springer, LNCS 92, 1980.

[80] R. Milner, *"Communication and Concurrency"*, Prentice Hall, 1989.

[81] R. Milner, *"Communicating and Mobile Systems: The π-Calculus"*, Cambridge University Press, 1999.

[82] R. Milner, J. Parrow, and D. Walker, *"A Calculus of Mobile Processes"*, in Information and Computation 100:1–77, 1992.

[83] M. Moriconi, X. Qian, and R.A. Riemenschneider, *"Correct Architecture Refinement"*, in IEEE Trans. on Software Engineering 21:356–372, 1995.

[84] M.F. Neuts, *"Matrix-Geometric Solutions in Stochastic Models – An Algorithmic Approach"*, John Hopkins University Press, 1981.

[85] E.-R. Olderog and C.A.R. Hoare, *"Specification-Oriented Semantics for Communicating Processes"*, in Acta Informatica 23:9–66, 1986.

[86] F. Oquendo, *"π-ADL: An Architecture Description Language Based on the Higher-Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures"*, in ACM SIGSOFT Software Engineering Notes 29:1–14, 2004.

[87] R. Paige and R.E. Tarjan, *"Three Partition Refinement Algorithms"*, in SIAM Journal on Computing 16:973–989, 1987.

[88] D. Park, *"Concurrency and Automata on Infinite Sequences"*, in Proc. of the *5th GI Conf. on Theoretical Computer Science*, Springer, LNCS 104:167–183, Karlsruhe (Germany), 1981.

[89] D.E. Perry and A.L. Wolf, *"Foundations for the Study of Software Architecture"*, in ACM SIGSOFT Software Engineering Notes 17:40–52, 1992.

[90] C.A. Petri, *"Kommunikation mit Automaten"*, Ph.D. Thesis, Technical University of Darmstadt (Germany), 1962.

[91] G.D. Plotkin, *"A Structural Approach to Operational Semantics"*, Tech. Rep. DAIMI-FN-19, Aarhus University (Denmark), 1981 (published in Journal of Logic and Algebraic Programming 60/61:17–139, 2004).

[92] M. Reiser and S.S. Lavenberg, *"Mean-Value Analysis of Closed Multichain Queueing Networks"*, in Journal of the ACM 27:313–322, 1980.

[93] W.H. Sanders and J.F. Meyer, *"A Unified Approach for Specifying Measures of Performance, Dependability, and Performability"*, in Dependable Computing and Fault Tolerant Systems 4:215–237, 1991.

[94] D. Sangiorgi and D. Walker, *"The π-Calculus: A Theory of Mobile Processes"*, Cambridge University Press, 2001.

[95] D.S. Scott, *"Data Types as Lattices"*, in SIAM Journal on Computing 5:522–587, 1976.

[96] M. Shaw, R. De Line, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik, *"Abstractions for Software Architecture and Tools to Support Them"*, in IEEE Trans. on Software Engineering 21:314–335, 1995.

[97] M. Shaw and D. Garlan, *"Software Architecture: Perspectives on an Emerging Discipline"*, Prentice Hall, 1996.

[98] C. Smith, *"Performance Engineering of Software Systems"*, Addison-Wesley, 1990.

[99] B. Spitznagel and D. Garlan, *"Architecture-Based Performance Analysis"*, in Proc. of the *10th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE 1998)*, IEEE-CS Press, pp. 146–151, San Francisco Bay (CA), 1998.

[100] B. Spitznagel and D. Garlan, *"A Compositional Formalization of Connector Wrappers"*, in Proc. of the *25th Int. Conf. on Software Engineering (ICSE 2003)*, IEEE-CS Press, pp. 374–384, Portland (OR), 2003.

[101] W.J. Stewart, *"Introduction to the Numerical Solution of Markov Chains"*, Princeton University Press, 1994.

[102] M. Tivoli and P. Inverardi, *"Failure-Free Coordinators Synthesis for Component-Based Architectures"*, in Science of Computer Programming 71:181–212, 2008.

[103] S. Vestal, *"Mode Changes in a Real-Time Architecture Description Language"*, in Proc. of the *2nd Int. Workshop on Configurable Distributed Systems (CDS 1994)*, IEEE-CS Press, pp. 136–146, Pittsburgh (PA), 1994.

[104] G. Winskel, *"Events in Computation"*, Ph.D. Thesis, University of Edinburgh (UK), 1980.

[105] C.M. Woodside, J.E. Neilson, D.C. Petriu, and S. Majumdar, *"The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software"*, in IEEE Trans. on Computers 44:20–34, 1995.

[106] D.M. Yellin and R.E. Strom, *"Protocol Specifications and Component Adaptors"*, in ACM Trans. on Programming Languages and Systems 19:292–333, 1997.