

A Process Algebraic Approach to Software Architecture Design

Marco Bernardo

University of Urbino – Italy

© October 2008

Table of Contents

<i>Starting Point: On the Usability of Process Algebra</i>	3
<i>Part I: Component-Oriented Modeling</i>	33
<i>Part II: Component-Oriented Verification</i>	122
<i>Part III: Component-Oriented Performance Evaluation</i>	198
<i>Part IV: Methodologies for Integrated Analysis</i>	313
<i>Part V: Architecture-Driven Code Generation</i>	314
<i>Part VI: Architecture-Driven Test Generation</i>	315
<i>References</i>	316

Starting Point:
On the Usability of Process Algebra

Process Algebra

- Calculi for reasoning about the **semantics of concurrent programming** developed since 1980 (CCS, CSP, ACP, ...).
- Understanding the **behavior of communicating concurrent systems** and their various facets (nondeterminism, priority, probability, time, ...).
- Basis of **formal description techniques and tools** for system modeling and verification (LOTOS, CWB, CADP, ...).
- Used to **teach** the foundations of concurrent programming as well as concurrent, distributed and mobile systems.

Running Example: Buffer

- General description:
 - ⊙ A buffer temporarily stores a variable number of items.
 - ⊙ Items can be deposited into the buffer as long as the buffer capacity is not exceeded.
 - ⊙ Stored items can then be withdrawn from the buffer if any.
 - ⊙ Insertions/removals take place according to some predefined discipline.
- Specific scenario:
 - ⊙ Two positions.
 - ⊙ Identical items, hence the discipline is not important.

Syntax for PA

- Based on **actions** and **behavioral operators** ($Name = Name_v \cup \{\tau\}$):

P	$::=$	$\underline{0}$	inactive process	
		B	process constant	$(B \triangleq P)$
		$a.P$	action prefix	$(a \in Name)$
		$P + P$	alternative composition	
		$P \parallel_S P$	parallel composition	$(S \subseteq Name_v)$
		P / H	hiding	$(H \subseteq Name_v)$
		$P \setminus L$	restriction	$(L \subseteq Name_v)$
		$P[\varphi]$	relabeling	$(\varphi : Name \rightarrow Name$ s.t. $\varphi^{-1}(\tau) = \{\tau\})$

- Compositionality** (parallel composition) and **abstraction** (hiding details) are inherently supported.

- Running example (syntax):

- ⊙ Relevant actions: *deposit* and *withdraw*.

- ⊙ Structure-independent process algebraic description:

$$\begin{aligned} Buffer_{0/2} &\triangleq deposit.Buffer_{1/2} \\ Buffer_{1/2} &\triangleq deposit.Buffer_{2/2} + withdraw.Buffer_{0/2} \\ Buffer_{2/2} &\triangleq withdraw.Buffer_{1/2} \end{aligned}$$

- ⊙ Specification to which every implementation should conform.

Semantics for PA

- Each process term P is mapped to a state-transition graph $\llbracket P \rrbracket$ called **labeled transition system** representing all possible computations of 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 corresponding action.
- Derivation of one single transition at a time by applying operational semantic rules inductively defined on the syntactical structure of the process term associated with the source state of the transition.

- $\llbracket 0 \rrbracket$ is a single state with no transitions.
- Basic rule for action prefix and inductive rules for all the other operators.
- **Dynamic operators** (\cdot , $+$) vs. **static operators** (\parallel , $/$, \backslash , $[\]$).
- $a.P$ can execute an action with name a and then behaves as P :

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

- B behaves as the process term P occurring in its defining equation:

$$\frac{B \triangleq P \quad P \xrightarrow{a} P'}{B \xrightarrow{a} P'}$$

- $P_1 + P_2$ behaves as either P_1 or P_2 depending on which of them executes an action first:

$$\boxed{
 \begin{array}{cc}
 \frac{P_1 \xrightarrow{a} P'}{\phantom{P_1 + P_2 \xrightarrow{a} P'}} & \frac{P_2 \xrightarrow{a} P'}{\phantom{P_1 + P_2 \xrightarrow{a} P'}} \\
 \hline
 P_1 + P_2 \xrightarrow{a} P' & P_1 + P_2 \xrightarrow{a} P'
 \end{array}
 }$$

- The choice between P_1 and P_2 can be influenced by the external environment (**nondeterminism**).

- $P_1 \parallel_S P_2$ behaves as P_1 in parallel with P_2 as long as actions are executed whose name does not belong to S :

$$\boxed{
 \begin{array}{c}
 \frac{P_1 \xrightarrow{a} P'_1 \quad a \notin S}{P_1 \parallel_S P_2 \xrightarrow{a} P'_1 \parallel_S P_2} \qquad \frac{P_2 \xrightarrow{a} P'_2 \quad a \notin S}{P_1 \parallel_S P_2 \xrightarrow{a} P_1 \parallel_S P'_2}
 \end{array}
 }$$

- Synchronization is forced between any action executed by P_1 and any action executed by P_2 that have the same name belonging to S :

$$\boxed{
 \frac{P_1 \xrightarrow{a} P'_1 \quad P_2 \xrightarrow{a} P'_2 \quad a \in S}{P_1 \parallel_S P_2 \xrightarrow{a} P'_1 \parallel_S P'_2}
 }$$

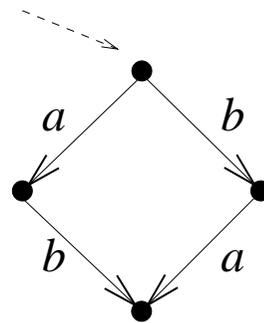
- Parallel composition is given an [interleaving semantics](#).
- The following process terms represent structurally different systems:

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

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

but they are indistinguishable by an external observer.

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



- Truly concurrent semantics are also possible (e.g., via Petri nets).

- Hiding:

$$\boxed{
 \begin{array}{cc}
 \frac{P \xrightarrow{a} P' \quad a \in H}{P / H \xrightarrow{\tau} P' / H} & \frac{P \xrightarrow{a} P' \quad a \notin H}{P / H \xrightarrow{a} P' / H}
 \end{array}
 }$$

- Restriction:

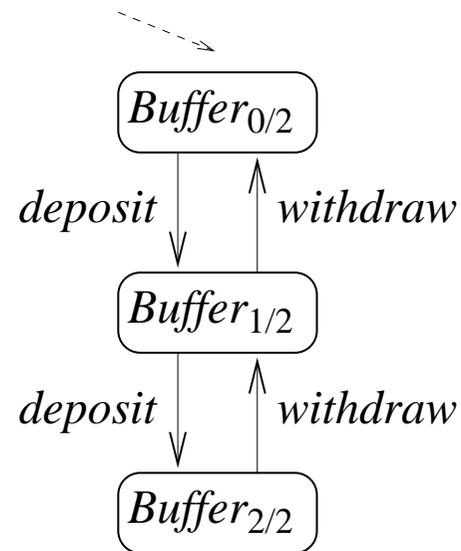
$$\boxed{
 \frac{P \xrightarrow{a} P' \quad a \notin L}{P \setminus L \xrightarrow{a} P' \setminus L}
 }$$

- Relabeling:

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

- Running example (semantics):

- Labeled transition system $\llbracket Buffer_{0/2} \rrbracket$:



- 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**.
- **Compositionality** (parallel subterms) and **abstraction** (from details) 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 system properties are preserved (*state space reduction before analysis*).
- Various approaches: bisimulation, testing, trace.

- **Bisimilarity**: two process terms are equivalent if they are able to *mimic* each other's behavior *stepwise*.
- A relation \mathcal{R} is a bisimulation iff for all $(P_1, P_2) \in \mathcal{R}$ and $a \in \text{Name}$:
 - ◉ whenever $P_1 \xrightarrow{a} P'_1$, then $P_2 \xrightarrow{a} P'_2$ with $(P'_1, P'_2) \in \mathcal{R}$;
 - ◉ whenever $P_2 \xrightarrow{a} P'_2$, then $P_1 \xrightarrow{a} P'_1$ with $(P'_1, P'_2) \in \mathcal{R}$.
- Bisimilarity $\sim_{\mathbf{B}}$ is the union of all the bisimulations.
- $P_1 \sim_{\mathbf{B}} P_2$ can be decided in $O(m \cdot \log n)$ time, where n is the number of states and m is the number of transitions of the disjoint union of $\llbracket P_1 \rrbracket$ and $\llbracket P_2 \rrbracket$.

- \sim_B is a **congruence** with respect to all the behavioral operators.
- Substituting equals for equals does not alter the overall meaning in any process context.
- Useful for compositional state space reduction.
- Whenever $P_1 \sim_B P_2$, then:

	$a.P_1 \sim_B a.P_2$	
$P_1 + P \sim_B P_2 + P$		$P + P_1 \sim_B P + P_2$
$P_1 \parallel_S P \sim_B P_2 \parallel_S P$		$P \parallel_S P_1 \sim_B P \parallel_S P_2$
	$P_1 / H \sim_B P_2 / H$	
	$P_1 \setminus L \sim_B P_2 \setminus L$	
	$P_1[\varphi] \sim_B P_2[\varphi]$	

- \sim_B is characterized by a set of **equational laws** resulting in a sound and complete axiomatization.
- The axioms can be used as rewriting rules for syntactically manipulating process terms in a way that is consistent with the equivalence.
- Helpful to understand the essence of the equivalence.
- Basic laws (alternative composition):

$$\begin{aligned}P_1 + P_2 &= P_2 + P_1 \\(P_1 + P_2) + P_3 &= P_1 + (P_2 + P_3) \\P + \underline{0} &= P \\P + P &= P\end{aligned}$$

- Expansion law (interleaving view of parallel composition):

$$\begin{aligned}
 \sum_{i \in I} a_i \cdot P_i \parallel_S \sum_{j \in J} b_j \cdot Q_j &= \sum_{k \in I, a_k \notin S} a_k \cdot \left(P_k \parallel_S \sum_{j \in J} b_j \cdot Q_j \right) + \\
 &\quad \sum_{h \in J, b_h \notin S} b_h \cdot \left(\sum_{i \in I} a_i \cdot P_i \parallel_S Q_h \right) + \\
 &\quad \sum_{k \in I, a_k \in S} \sum_{h \in J, b_h = a_k} a_k \cdot (P_k \parallel_S Q_h)
 \end{aligned}$$

- Distribution laws (other static operators):

$$\begin{array}{lcl}
 \underline{0} / H & = & \underline{0} \\
 (a.P) / H & = & \tau.(P / H) \quad \text{if } a \in H \\
 (a.P) / H & = & a.(P / H) \quad \text{if } a \notin H \\
 (P_1 + P_2) / H & = & P_1 / H + P_2 / H \\
 \\
 \underline{0} \setminus L & = & \underline{0} \\
 (a.P) \setminus L & = & \underline{0} \quad \text{if } a \in L \\
 (a.P) \setminus L & = & a.(P \setminus L) \quad \text{if } a \notin L \\
 (P_1 + P_2) \setminus L & = & P_1 \setminus L + P_2 \setminus L \\
 \\
 \underline{0}[\varphi] & = & \underline{0} \\
 (a.P)[\varphi] & = & \varphi(a).(P[\varphi]) \\
 (P_1 + P_2)[\varphi] & = & P_1[\varphi] + P_2[\varphi]
 \end{array}$$

- \sim_B has a [modal logic characterization](#).
- Based on logical operators expressing the fact that certain behaviors can/must happen.
- Helpful to understand what behavioral properties are preserved by the equivalence.
- Useful to explain inequivalence when a distinguishing modal formula can be automatically built by the equivalence checking algorithm.
- $P_1 \sim_B P_2$ if and only if P_1 and P_2 satisfy the same subset of formulas of the [Hennessy-Milner logic \(HML\)](#).

- Syntax of HML ($a \in Name$):

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

- Derived logical operators:

false	\equiv	$\neg\text{true}$	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 HML:

P	\models	true	
P	\models	$\neg\phi$	if $P \not\models \phi$
P	\models	$\phi_1 \wedge \phi_2$	if $P \models \phi_1$ and $P \models \phi_2$
P	\models	$\langle a \rangle \phi$	if there exists $P \xrightarrow{a} P'$ such that $P' \models \phi$

- Interpretation of the derived logical operators:

P	$\not\models$	false	
P	\models	$\phi_1 \vee \phi_2$	if $P \models \phi_1$ or $P \models \phi_2$
P	\models	$[a]\phi$	if whenever $P \xrightarrow{a} P'$ then $P' \models \phi$

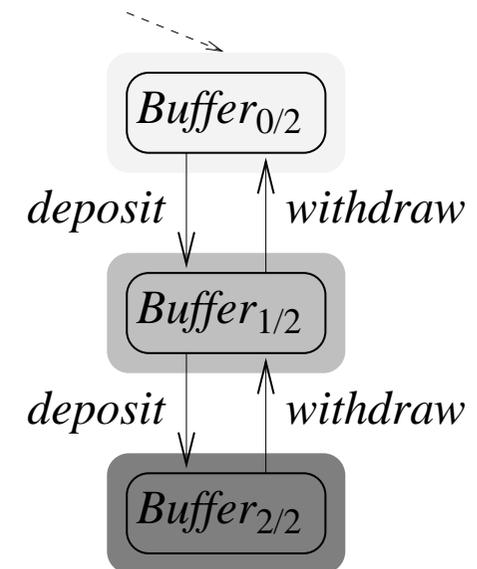
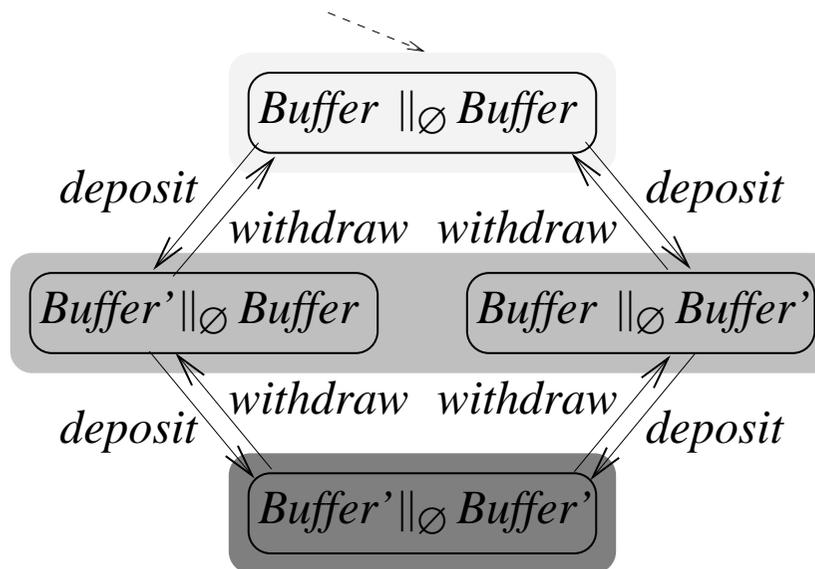
- Running example (bisimilarity):

- Parallel implementation:

$$ParBuffer_2 \triangleq Buffer \parallel_{\emptyset} Buffer$$

$$Buffer \triangleq deposit.withdraw.Buffer$$

- Bisimulation proving $ParBuffer_2 \sim_B Buffer_{0/2}$:



- **Weak bisimilarity**: two process terms are equivalent if they are able to mimic each other's *visible* behavior stepwise.
- A relation \mathcal{R} is a weak bisimulation iff for all $(P_1, P_2) \in \mathcal{R}$ and $a \in \text{Name}_v$:
 - ⊙ whenever $P_1 \xrightarrow{a} P'_1$, then $P_2 \xRightarrow{\tau^* a \tau^*} P'_2$ with $(P'_1, P'_2) \in \mathcal{R}$;
 - ⊙ whenever $P_1 \xrightarrow{\tau} P'_1$, then $P_2 \xRightarrow{\tau^*} P'_2$ with $(P'_1, P'_2) \in \mathcal{R}$;
 - ⊙ whenever $P_2 \xrightarrow{a} P'_2$, then $P_1 \xRightarrow{\tau^* a \tau^*} P'_1$ with $(P'_1, P'_2) \in \mathcal{R}$;
 - ⊙ whenever $P_2 \xrightarrow{\tau} P'_2$, then $P_1 \xRightarrow{\tau^*} P'_1$ with $(P'_1, P'_2) \in \mathcal{R}$.
- Weak bisimilarity \approx_B is the union of all the weak bisimulations.

- \approx_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:

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

- Weak modal operators:

$$\begin{aligned}
 P \models \langle\langle a \rangle\rangle \phi & \quad \text{if there exists } P \xrightarrow{\tau^* a \tau^*} P' \text{ such that } P' \models \phi \\
 P \models [[a]] \phi & \quad \text{if whenever } P \xrightarrow{\tau^* a \tau^*} P' \text{ then } P' \models \phi
 \end{aligned}$$

- Running example (weak bisimilarity):

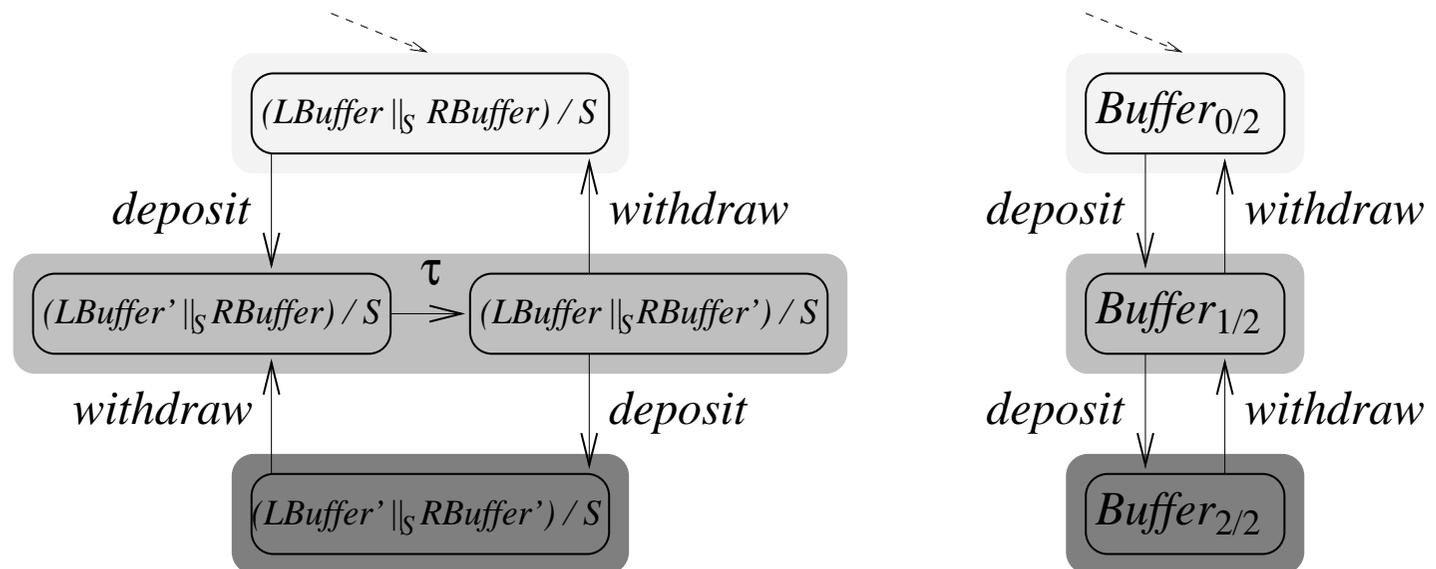
- Pipeline implementation consistent with FIFO discipline:

$$PipeBuffer_2 \triangleq (LBuffer \parallel_{\{pass\}} RBuffer) / \{pass\}$$

$$LBuffer \triangleq deposit.pass.LBuffer$$

$$RBuffer \triangleq pass.withdraw.RBuffer$$

- Weak bisimulation proving $PipeBuffer_2 \approx_B Buffer_{0/2}$:



Strengths of PA

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

Weaknesses of PA

- Only a **very limited adoption** of PA in the software development process is taking place outside the academia.
- PA 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 is compositional but not component-oriented*).
- PA **technicalities obfuscate** the way in which systems are modeled (*parallel composition describes both structure and behavior*).

An Architectural Upgrade

- How to enhance the usability of PA?
- Need to support a friendly component-oriented way of modeling systems with PA \implies *the designer can reason in terms of composable software units while abstracting from PA technicalities.*
- Need to provide an efficient component-oriented way of analyzing functional and non-functional system properties with PA techniques, which returns component-oriented diagnostic information in case of violation \implies *the designer can pinpoint the sources of error.*
- Need to integrate the use of PA in the system development cycle. (Requirement analysis? Architectural design? Design/selection and integration of components? Deployment? Testing? Maintenance?)

- Working at the [architectural design level](#).
- DeRemer & Kron (1976): “... *structuring a large collections 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 ...*”.

- Focus not on algorithms and data structures (programming languages), but on components and connectors (modeling notations).
- Objective: a **document** that describes the **system structure and behavior** at a high level of abstraction, to be used in all the subsequent phases of the system development cycle.
- If an **architectural description language (ADL)** is used to formalize this document, then **system property analyzability** can be achieved in the early stages of the development process (saving time and money).
- How to transform PA into an ADL?
- How to drive the whole development process with the resulting PADL?

Part I:
Component-Oriented Modeling

Transformation Guidelines

1. Separation of behavior description from topology description.
2. Reuse of the specification of components and connectors.
3. Elicitation of the interfaces 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. Separation of dynamic operators from static operators.
8. Provision of support for architectural styles.

Running Example: Client-Server System

- General description:
 - ⊙ A possibly replicated server provides a number 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.
 - ⊙ No buffer for holding incoming requests on the server side.
 - ⊙ Two identically behaving clients.
 - ⊙ After a request, a client cannot proceed until it receives a response.

G1: Separating Behavior from Topology

- Hard to understand which process terms communicate with each other in a system description with several process terms composed in parallel.
- Separate behavior specification from topology specification.
- Two distinct sections in every architectural description:
 - ⊙ Architectural **behavior** section: definition of the types of components and connectors of the system (**AET – architectural element type**).
 - ⊙ Architectural **topology** section: declaration of the instances of components and connectors (**AEI – architectural element instance**) and the way they communicate with each other (**attachment**).

G2: Reusing Component/Connector Specification

- There may be several process terms in a system description that differ only for the names of some of their actions or process constants.
- Define **only one AET** for all of those slightly different process terms in the architectural behavior section.
- Declare as **many instances of the AET** as there are similar process terms in the architectural topology section.

- Running example (G1/G2):
 - ⊙ AETs:
 - * one for the server (**Server_Type**);
 - * one for both clients (**Client_Type**) due to their identical behavior.
 - ⊙ AEs:
 - * one of server type (**S**);
 - * two of client type (**C_1**, **C_2**).
 - ⊙ Each of the two client AEs is attached to the server AE.

G3: Eliciting Component/Connector Interface

- The actions occurring in a process term do not play the same role from the communication viewpoint.
- Distinguish explicitly in any AET definition between:
 - ⊙ **internal actions** forming the component/connector implementation;
 - ⊙ **interactions** forming the component/connector interface:
 - * **input** vs. **output** (AET definition);
 - * **local** vs. **architectural** (architectural topology section).
- Static checks:
 - ⊙ *Any local interaction involved at least in one attachment.*
 - ⊙ *No internal action/architectural interaction involved in attachments.*
 - ⊙ *Attachments only between input interactions and output interactions belonging to different AETs.*

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 counterpart not ready);
 - ⊙ **asynchronous** interactions (decoupling start from end).
- All nine two-by-two combinations allowed in attachments.
- Boolean variable **success** associated with semi-synchronous interactions for catching exceptions within the AET behavior.

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
(or-dep ensures that an output is sent to the same AEI that issued the corresponding input).
- Static checks:
 - ⊙ *Any local uni-interaction attached to only one local interaction.*
 - ⊙ *Any local and-/or-interaction attached to local uni-interactions only.*
 - ⊙ *No output or-interaction occurring before the input or-interaction it depends on, and both of them attached to the same AEIs.*

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

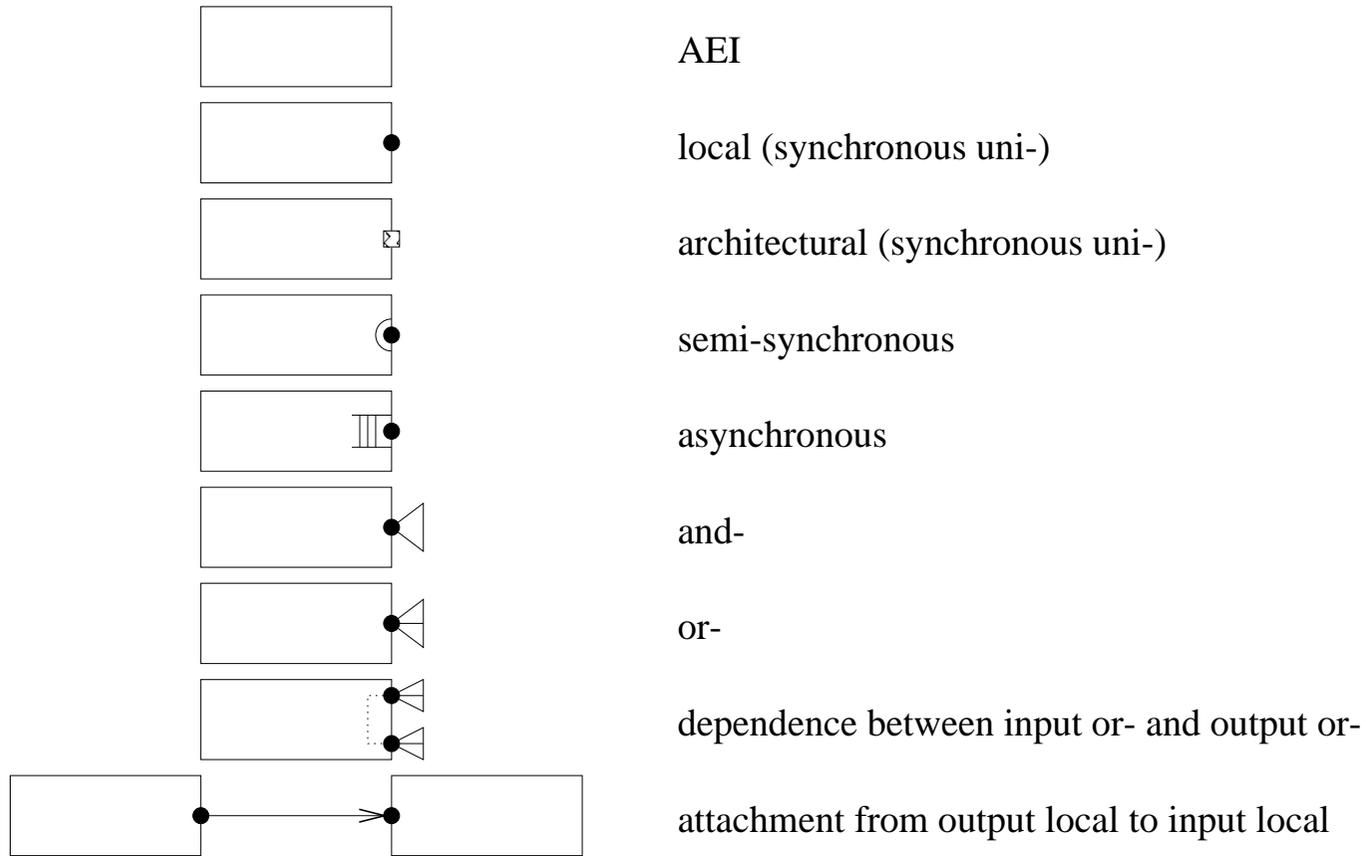
G6: Mixing Textual and Graphical Notations

- Process algebra only provides a textual notation, which may not be as easy to use as desired by software designers.
- Enhance the textual notation with architectural syntactic sugar.
- Make available a graphical notation too.
- Permit an integrated use of the two notations:
 - ⊙ The graphical one for representing the topology.
 - ⊙ The textual one for describing the behavior inside each AEI.

- Structure of a textual architectural description in PADL:

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

- Graphical notation based on [enriched flow graphs](#):



G7: Separating Dynamic and Static Operators

- Static operators (`||`, `/`, `\`, `[]`) are harder to use than dynamic ones (`..`, `+`).
- Make available only the dynamic operators to the designer for the description of the behavior of AETs.
- Make dynamic operators more verbose (`0` becomes `stop`).
- Avoid semantic overloading of dynamic operators (`+` becomes `choice`).
- Hide static operators to the designer.

- Revised syntax of a PA defining equation in an AET definition:

$$B(\textit{formal_data_parameter_list}; \textit{data_variable_list}) = P$$

- Formal pars of the first defining equation have to be initialized.
- Only dynamic operators can occur in P (cond clause is optional):

P	::=	stop	inactive process
		$B(\textit{actual_data_parameter_list})$	process constant
		$\textit{cond}(\textit{bool_expr}) \rightarrow a.P$	action prefix
		$\textit{cond}(\textit{bool_expr}) \rightarrow a?(var_list).P$	input action prefix
		$\textit{cond}(\textit{bool_expr}) \rightarrow a!(expr_list).P$	output action prefix
		choice $\{P, \dots, P\}$	alternative composition

- Data types: **boolean**, **integer**, **real**, **list**, **array**, **record**, **object**.

- 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):

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 AEs 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(void; void) =
```

```
process . send_request .
```

```
choice
```

```
{
```

```
cond(send_request.success = true) ->
```

```
receive_response . Client(),
```

```
cond(send_request.success = false) ->
```

```
keep_processing . Client()
```

```
}
```

```
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
```

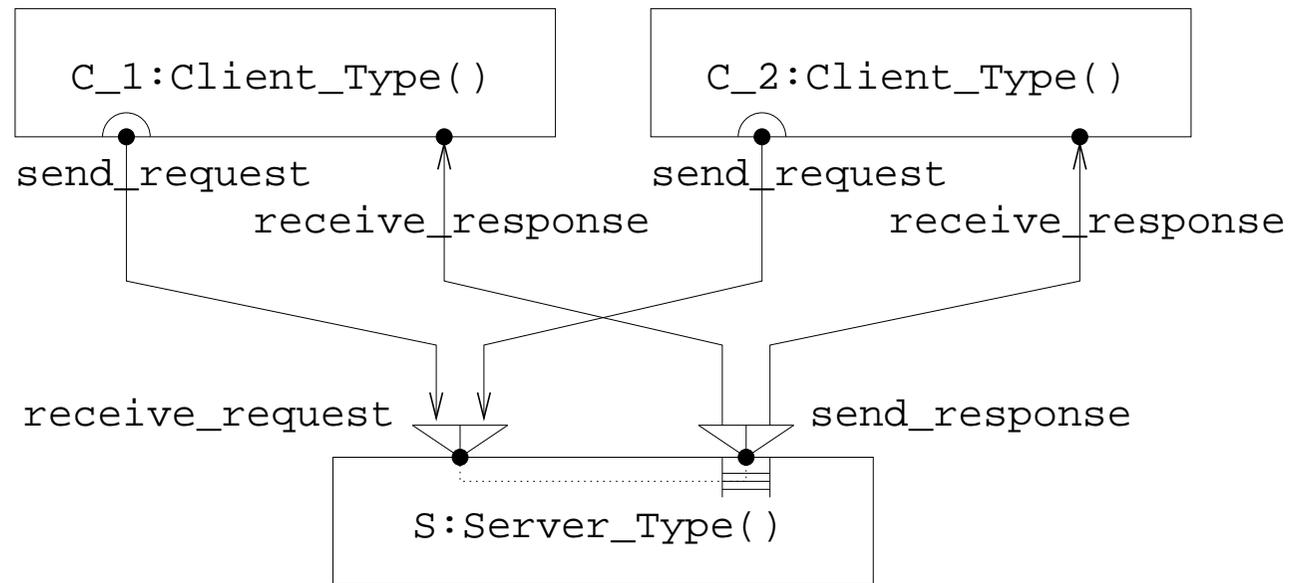
```
FROM C_1.send_request TO S.receive_request;
```

```
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:



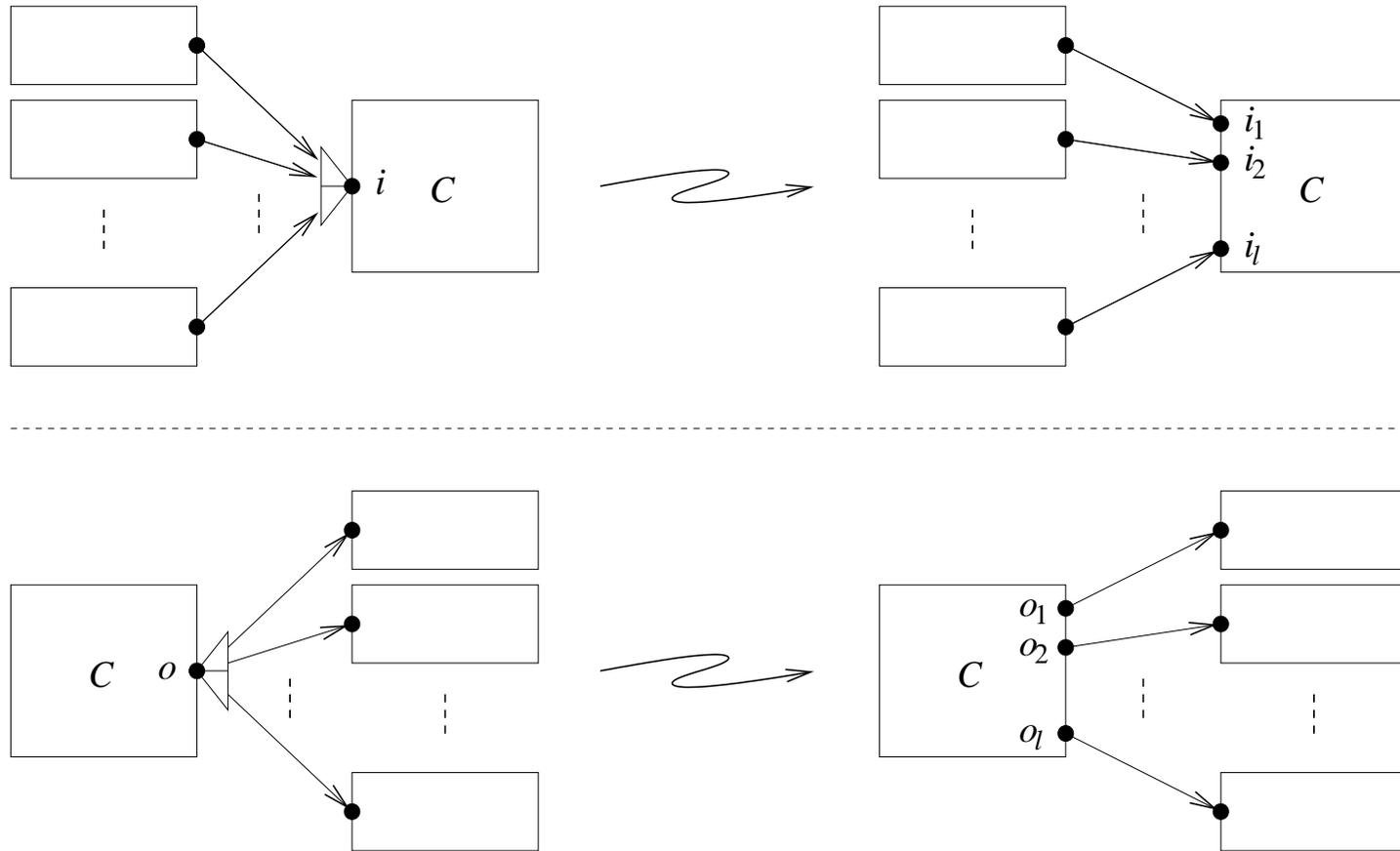
Translation Semantics for PADL

- Every PADL description is translated into a PA specification.
- *First step:*
 - ⊙ The semantics of any AEI is the behavior of the corresponding AET.
 - ⊙ AET formal data parameters replaced by AEI actual ones.
 - ⊙ Local or-interactions replaced by sets of fresh local uni-interactions.
 - ⊙ Implicit buffers for handling asynchronous local interactions.
- *Second step:*
 - ⊙ The semantics of the whole description is the parallel composition of the semantics of the occurring AEIs.
 - ⊙ Synchronization sets determined by the attachments.
 - ⊙ Interactions renamed consistently with the synchronization sets.
 - ⊙ Additional semantic rules for handling exceptions (semi-sync interactions).

First Step of the Translation

- Let \mathcal{C} be an AET with:
 - ⊙ formal data parameters fp_1, \dots, fp_m ;
 - ⊙ behavior given by a sequence \mathcal{E} of PA defining equations containing only dynamic operators.
- Let C be an AEI of type \mathcal{C} with:
 - ⊙ actual data parameters ap_1, \dots, ap_m .
- C may have local or-interactions and asynchronous local interactions.

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



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

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

$$or-rewrite_{FI}(a.P) = a.or-rewrite_{FI}(P)$$

- 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 $attach-no(C.a) = l \geq 2$:

$$or-rewrite_{FI}(a.P) = \mathbf{choice}\{a_1.or-rewrite_{FI}(P), \\ \vdots \\ a_l.or-rewrite_{FI}(P)\}$$

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

$$or-rewrite_{FI}(i.P) = \text{choice}\{i_1.or-rewrite_{FI \cup \{i_1\} - \{i_j | 1 \leq j \leq l \wedge j \neq 1\}}(P), \\ \vdots \\ i_l.or-rewrite_{FI \cup \{i_l\} - \{i_j | 1 \leq j \leq l \wedge j \neq l\}}(P)\}$$

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

$$or-rewrite_{FI}(o.P) = o_j.or-rewrite_{FI}(P)$$

- Rewriting process for the remaining operators allowed within \mathcal{E} :

$$\begin{aligned}
 or\text{-rewrite}_{FI}(\mathbf{stop}) &= \mathbf{stop} \\
 or\text{-rewrite}_{FI}(B(\mathit{actual_data_par_list})) &= B_{FI}(\mathit{actual_data_par_list}) \\
 or\text{-rewrite}_{FI}(\mathbf{choice}\{P_1, \dots, P_n\}) &= \mathbf{choice}\{or\text{-rewrite}_{FI}(P_1), \\
 &\quad \vdots \\
 &\quad or\text{-rewrite}_{FI}(P_n)\}
 \end{aligned}$$

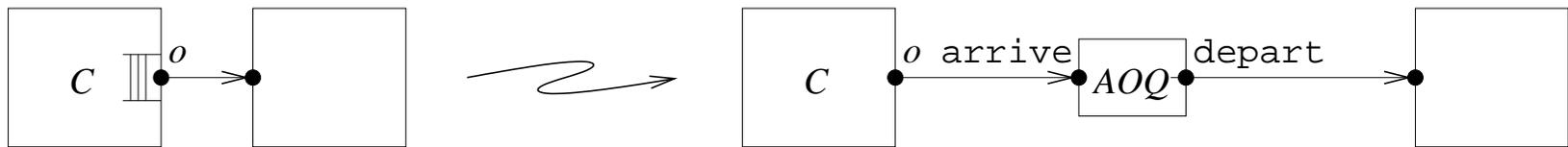
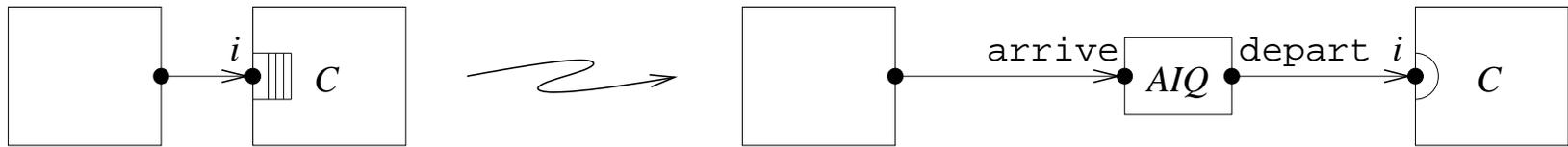
where $B_{FI} \equiv B$ for $FI = \emptyset$.

- For $FI \neq \emptyset$:

$$B_{FI}(\mathit{formal_data_par_list}; \mathit{data_var_list}) = or\text{-rewrite}_{FI}(P)$$

if $B(\mathit{formal_data_par_list}; \mathit{data_var_list}) = P$.

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



- AET for asynchronous input local uni-/and-interactions:

```
ARCHI_ELEM_TYPE Async_Input_Queue(void)
```

```
BEHAVIOR
```

```
Queue(int n := 0; void) =
```

```
choice
```

```
{
```

```
cond(true) -> arrive . Queue(n + 1),
```

```
cond(n > 0) -> depart . Queue(n - 1)
```

```
}
```

```
INPUT_INTERACTIONS SYNC --- arrive /* same multi/attachs */
```

```
OUTPUT_INTERACTIONS SYNC UNI depart
```

- The original asynchronous input interaction of C is converted into a semi-synchronous uni-interaction and $AIQ.depart$ is attached to it.

- AET for asynchronous output local uni-/and-interactions:

```
ARCHI_ELEM_TYPE Async_Output_Queue(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
```

```
OUTPUT_INTERACTIONS SYNC --- depart      /* same multi/attachs */
```

- The original asynchronous output interaction of C is converted into a synchronous uni-interaction and attached to $AOQ.arrive$.

- Semantics of C in isolation (with asynchronous input local interactions $C.i_1, \dots, C.i_h$ and asynchronous output local interactions $C.o_1, \dots, C.o_k$):

$$\begin{aligned}
\llbracket C \rrbracket = & \overbrace{(\text{Queue} \parallel_{\emptyset} \dots \parallel_{\emptyset} \text{Queue})}^h [\varphi_{C, \text{async}}] \\
& \parallel \{AIQ_1.\text{depart}\#C.i_1, \dots, AIQ_h.\text{depart}\#C.i_h\} \\
& \text{or-rewrite}_{\emptyset}(\mathcal{E}\{ap_1/fp_1, \dots, ap_m/fp_m\}) [\varphi_{C, \text{async}}] \\
& \parallel \{C.o_1\#AOQ_1.\text{arrive}, \dots, C.o_k\#AOQ_k.\text{arrive}\} \\
& \underbrace{(\text{Queue} \parallel_{\emptyset} \dots \parallel_{\emptyset} \text{Queue})}_{k} [\varphi_{C, \text{async}}]
\end{aligned}$$

- $\{-/-, \dots, -/-\}$ denotes a syntactical substitution.
- $\varphi_{C, \text{async}}$ transforms $C.i_1, \dots, C.i_h, C.o_1, \dots, C.o_k$ and the related attached interactions of $AIQ_1, \dots, AIQ_h, AOQ_1, \dots, AOQ_k$ into the respective fresh names occurring in the two synchronization sets.

- Running example (first step of the translation):
 - ◉ The semantics of C_1 and C_2 is their defining equation because there are neither local or-interactions nor asynchronous local interactions:

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

- ⊙ The semantics of S requires the application of function *or-rewrite* to its defining equation because $attach-no(S.receive_request) = attach-no(S.send_response) > 1$:

```

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

```

- ⊙ Then we need to add two implicit AEs for the asynchronous output local uni-interactions $S.send_response_1$ and $S.send_response_2$:

```

Server'[S.send_response_1 ↦ S.send_response_1#A0Q_1.arrive,
        S.send_response_2 ↦ S.send_response_2#A0Q_2.arrive]

```

```

|| {S.send_response_1#A0Q_1.arrive,
    S.send_response_2#A0Q_2.arrive}

```

```

(Queue ||∅ Queue)[A0Q_1.arrive ↦ S.send_response_1#A0Q_1.arrive,
                  A0Q_2.arrive ↦ S.send_response_2#A0Q_2.arrive]

```

Second Step of the Translation

- Let $\{C_1, \dots, C_n\}$ be a set of AEs.
- Let $\mathcal{LI}_{C_1}, \dots, \mathcal{LI}_{C_n}$ be the sets of local interactions of C_1, \dots, C_n
(after rewriting local or-interactions and adding implicit AEs for async local interactions).
- Focus on the local interactions of each C_j attached to $\{C_1, \dots, C_n\}$:

$$\mathcal{LI}_{C_j; C_1, \dots, C_n} \subseteq \mathcal{LI}_{C_j}$$

- Rename those interactions so that each C_j can communicate with the other AEs even if attached interactions have been given different names.
- Construct suitable synchronization sets based on the attachments that contain the renamed interactions.
- Dot notation and name concatenation to ensure renaming uniqueness
(e.g., $C_j.o\#C_g.i$ if interaction o of C_j is attached to interaction i of C_g).

- Need a set of fresh action names, one for each pair of attached local uni-interactions within $\{C_1, \dots, C_n\}$ and for each local and-interaction attached to local uni-interactions within $\{C_1, \dots, C_n\}$:

$$\mathcal{S}(C_1, \dots, C_n)$$

- Need an injective relabeling function for each $\mathcal{LI}_{C_j; C_1, \dots, C_n}$:

$$\varphi_{C_j; C_1, \dots, C_n} : \mathcal{LI}_{C_j; C_1, \dots, C_n} \longrightarrow \mathcal{S}(C_1, \dots, C_n)$$

such that:

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

iff $C_j.a_1$ and $C_g.a_2$ attached to each other or to the same and-interaction.

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

$$\llbracket C_j \rrbracket_{C_1, \dots, C_n} = \llbracket C_j \rrbracket[\varphi_{C_j; C_1, \dots, C_n}]$$

- Individual synchronization set of C_j with respect to $\{C_1, \dots, C_n\}$:

$$\mathcal{S}(C_j; C_1, \dots, C_n) = \varphi_{C_j; C_1, \dots, C_n}(\mathcal{LI}_{C_j; C_1, \dots, C_n})$$

- Pairwise synchronization set of C_j and C_g with respect to $\{C_1, \dots, C_n\}$:

$$\mathcal{S}(C_j, C_g; C_1, \dots, C_n) = \mathcal{S}(C_j; C_1, \dots, C_n) \cap \mathcal{S}(C_g; C_1, \dots, C_n)$$

- Interacting semantics of $\{C'_1, \dots, C'_{n'}\} \subseteq \{C_1, \dots, C_n\}$ with respect to $\{C_1, \dots, C_n\}$:

$$\begin{aligned} \llbracket C'_1, \dots, C'_{n'} \rrbracket_{C_1, \dots, C_n} &= \llbracket C'_1 \rrbracket_{C_1, \dots, C_n} \parallel_{\mathcal{S}(C'_1, C'_2; C_1, \dots, C_n)} \\ &\quad \llbracket C'_2 \rrbracket_{C_1, \dots, C_n} \parallel_{\mathcal{S}(C'_1, C'_3; C_1, \dots, C_n) \cup \mathcal{S}(C'_2, C'_3; C_1, \dots, C_n)} \\ &\quad \dots \parallel_{\bigcup_{j=1}^{n'-1} \mathcal{S}(C'_j, C'_{n'}; C_1, \dots, C_n)} \llbracket C'_{n'} \rrbracket_{C_1, \dots, C_n} \end{aligned}$$

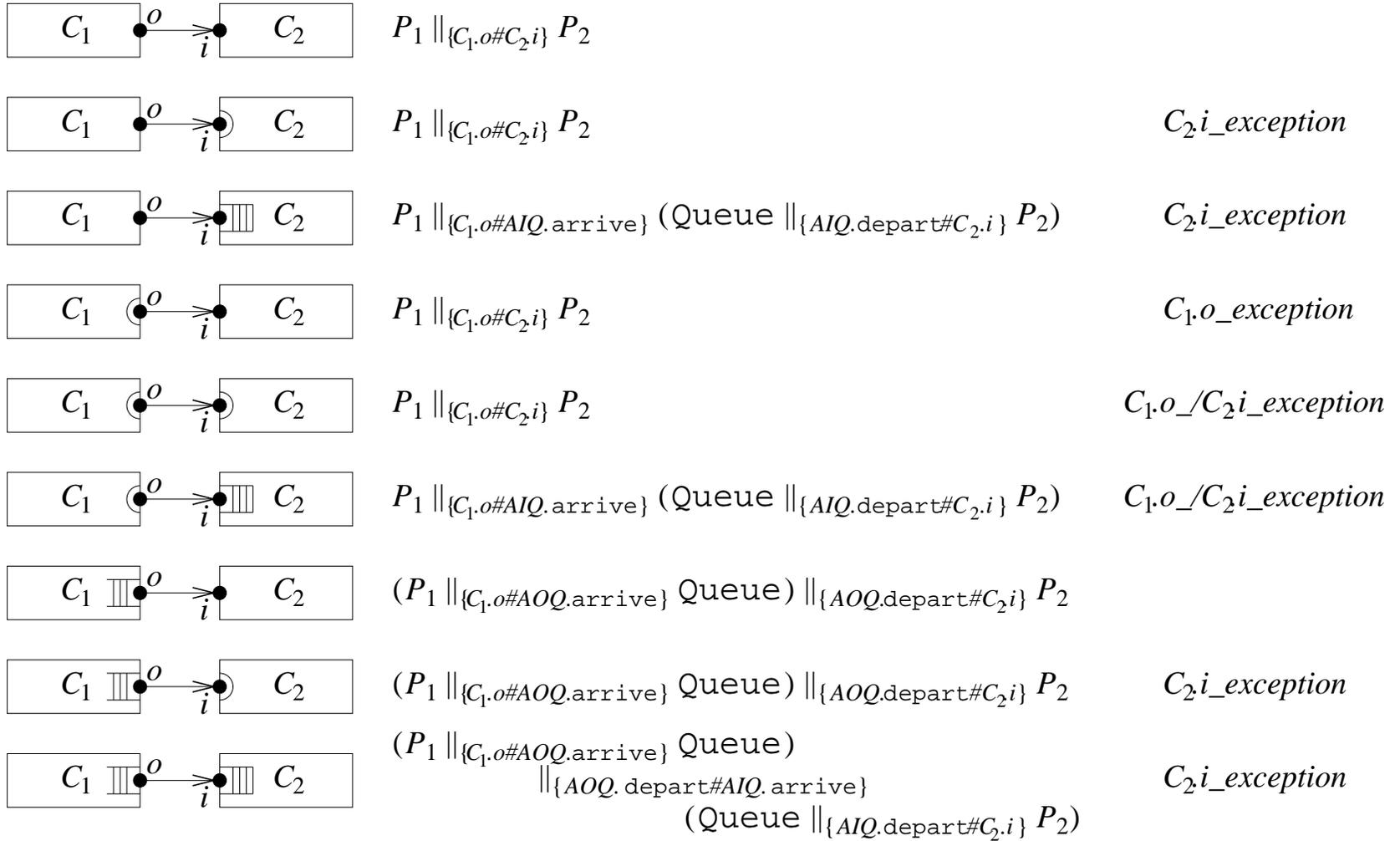
where left associativity of parallel composition is assumed.

- A semi-synchronous interaction a executed by an AEI C results in a transition labeled with a within $\llbracket C \rrbracket$.
- In an interacting context, this transition is relabeled as an exception if a is local and cannot immediately participate in a communication.
- Additional semantic rules for handling exceptions ($C_1.o$ attached to $C_2.i$):

$P_1 \xrightarrow{C_1.o\#C_2.i} /$	$P_2 \xrightarrow{C_1.o\#C_2.i} P_2'$	$C_2.i$ semi-synchronous
$P_1 \parallel_S P_2 \xrightarrow{C_2.i_exception} P_1 \parallel_S P_2'$		
$P_1 \xrightarrow{C_1.o\#C_2.i} P_1'$	$P_2 \xrightarrow{C_1.o\#C_2.i} /$	$C_1.o$ semi-synchronous
$P_1 \parallel_S P_2 \xrightarrow{C_1.o_exception} P_1' \parallel_S P_2$		

where P_j represents the current state of the interacting semantics of C_j .

- Summary of the semantic treatment of communication synchronicity:



- Let \mathcal{A} be an architectural description.
- Let $\{C_1, \dots, C_n\}$ be the set of its AEs.
- Let H , L , 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:

$$\llbracket \mathcal{A} \rrbracket_{\text{bbm}} = \llbracket C_1, \dots, C_n \rrbracket_{C_1, \dots, C_n}$$

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

$$\llbracket \mathcal{A} \rrbracket_{\text{abm}} = \llbracket \mathcal{A} \rrbracket_{\text{bbm}} / H \setminus L[\varphi]$$

- Running example (second step of the translation):

- Semantics of `Client_Server`:

$$\llbracket S \rrbracket [\text{receive_request_1} \mapsto C_1.\text{send_request} \# S.\text{receive_request_1}, \\ \text{A0Q_1.depart} \mapsto \text{A0Q_1.depart} \# C_1.\text{receive_response}, \\ \text{receive_request_2} \mapsto C_2.\text{send_request} \# S.\text{receive_request_2}, \\ \text{A0Q_2.depart} \mapsto \text{A0Q_2.depart} \# C_2.\text{receive_response}]$$

$$\parallel \{C_1.\text{send_request} \# S.\text{receive_request_1}, \\ \text{A0Q_1.depart} \# C_1.\text{receive_response}\}$$

$$\llbracket C_1 \rrbracket [\text{send_request} \mapsto C_1.\text{send_request} \# S.\text{receive_request_1}, \\ \text{receive_response} \mapsto \text{A0Q_1.depart} \# C_1.\text{receive_response}]$$

$$\parallel \{C_2.\text{send_request} \# S.\text{receive_request_2}, \\ \text{A0Q_2.depart} \# C_2.\text{receive_response}\}$$

$$\llbracket C_2 \rrbracket [\text{send_request} \mapsto C_2.\text{send_request} \# S.\text{receive_request_2}, \\ \text{receive_response} \mapsto \text{A0Q_2.depart} \# C_2.\text{receive_response}]$$

Summarizing Example: Pipe-Filter System

- General description:
 - ⊙ Each filter reads streams of row data on its inputs, applies them a transformation, and produces streams of processed data on its outputs (incremental process).
 - ⊙ Each pipe transmits outputs of one filter to inputs of another filter.
- Specific scenario:
 - ⊙ Four identical filters, each equipped with a ten-position buffer.
 - ⊙ One upstream filter and three downstream filters linked by one pipe.
 - ⊙ The pipe forwards each 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:

$$\begin{aligned}
 \textit{Pipe_Filter} &\triangleq \textit{Upstream_Filter}_{0/10} \parallel \{\textit{output_accept_item}\} \\
 &\quad \textit{Pipe} \parallel \{\textit{forward_input_item}_1\} \\
 &\quad \quad \textit{Downstream_Filter}_{0/10}^1 \parallel \{\textit{forward_input_item}_2\} \\
 &\quad \quad \quad \textit{Downstream_Filter}_{0/10}^2 \parallel \{\textit{forward_input_item}_3\} \\
 &\quad \quad \quad \quad \textit{Downstream_Filter}_{0/10}^3
 \end{aligned}$$

- Which software unit communicates with which software unit?
- Are we sure that those synchronization sets are correct?

- Definition of the upstream filter ($1 \leq j \leq 9$):

$$\begin{aligned}
 \textit{Upstream_Filter}_{0/10} &\triangleq \textit{input_item.transform_item.Upstream_Filter}_{1/10} \\
 \textit{Upstream_Filter}_{j/10} &\triangleq \textit{input_item.transform_item.Upstream_Filter}_{j+1/10} + \\
 &\quad \textit{output_accept_item.Upstream_Filter}_{j-1/10} \\
 \textit{Upstream_Filter}_{10/10} &\triangleq \textit{output_accept_item.Upstream_Filter}_{9/10}
 \end{aligned}$$

- What is its interface?
- Not free to choose adequate names for synchronizing actions as they must share the same name (*output_accept_item*).

- Definition of the pipe:

$$\textit{Pipe} \triangleq \textit{output_accept_item} . (\textit{forward_input_item}_1 . \textit{Pipe} + \\ \textit{forward_input_item}_2 . \textit{Pipe} + \\ \textit{forward_input_item}_3 . \textit{Pipe})$$

- Same problem for names of synchronizing actions.
- What if there were 100 downstream filters instead of only 3?

- Definition of the first downstream filter ($1 \leq j \leq 9$):

$$\begin{aligned}
 \textit{Downstream_Filter}_{0/10}^1 &\triangleq \textit{forward_input_item}_1.\textit{transform_item}. \\
 &\qquad\qquad\qquad \textit{Downstream_Filter}_{1/10}^1 \\
 \textit{Downstream_Filter}_{j/10}^1 &\triangleq \textit{forward_input_item}_1.\textit{transform_item}. \\
 &\qquad\qquad\qquad \textit{Downstream_Filter}_{j+1/10}^1 + \\
 &\qquad\qquad\qquad \textit{output_item}.\textit{Downstream_Filter}_{j-1/10}^1 \\
 \textit{Downstream_Filter}_{10/10}^1 &\triangleq \textit{output_item}.\textit{Downstream_Filter}_{9/10}^1
 \end{aligned}$$

- Similar to the definition of the upstream filter: the only difference lies in the names of the process constants and of some actions.

- Definition of the second downstream filter ($1 \leq j \leq 9$):

$$\textit{Downstream_Filter}_{0/10}^2 \triangleq \textit{forward_input_item}_2.\textit{transform_item}.$$

$$\textit{Downstream_Filter}_{j/10}^2 \triangleq \textit{forward_input_item}_2.\textit{transform_item}.$$

$$\textit{Downstream_Filter}_{j+1/10}^2 +$$

$$\textit{output_item}.\textit{Downstream_Filter}_{j-1/10}^2$$

$$\textit{Downstream_Filter}_{10/10}^2 \triangleq \textit{output_item}.\textit{Downstream_Filter}_{9/10}^2$$

- Similar to the definitions of the previous two filters.

- Definition of the third downstream filter ($1 \leq j \leq 9$):

$$Downstream_Filter_{0/10}^3 \triangleq forward_input_item_3.transform_item.$$

$$Downstream_Filter_{j/10}^3 \triangleq forward_input_item_3.transform_item.$$

$$Downstream_Filter_{10/10}^3 \triangleq output_item.Downstream_Filter_{j-1/10}^3 + Downstream_Filter_{j+1/10}^3 + output_item.Downstream_Filter_{9/10}^3$$

- What if there were 100 filters instead of only 4?

- PADL description header:

```
ARCHI_TYPE Pipe_Filter(const integer pf_buffer_size := 10,  
                        const integer pf_init_item_num := 0)
```

- Explicit support for data parameterization.
- Localized at the beginning of the description.
- The initialization values are the default actual values for the formal data parameters of the whole description.

- Definition of the filter AET (one is enough, free choice of names):

```
ARCHI_ELEM_TYPE Filter_Type(const integer buffer_size,  
                             const integer init_item_num)
```

```
BEHAVIOR
```

```
  Filter(integer(0..buffer_size) item_num := init_item_num;  
         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  UNI input_item
```

```
OUTPUT_INTERACTIONS UNI output_item
```

- Definition of the pipe AET (exploits or-interaction mechanism):

```
ARCHI_ELEM_TYPE Pipe_Type(void)
```

```
BEHAVIOR
```

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

```
INPUT_INTERACTIONS  UNI accept_item
```

```
OUTPUT_INTERACTIONS OR  forward_item
```

- Declaration of the topology (architectural interactions for structural extensions):

ARCHI_ELEM_INSTANCES

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

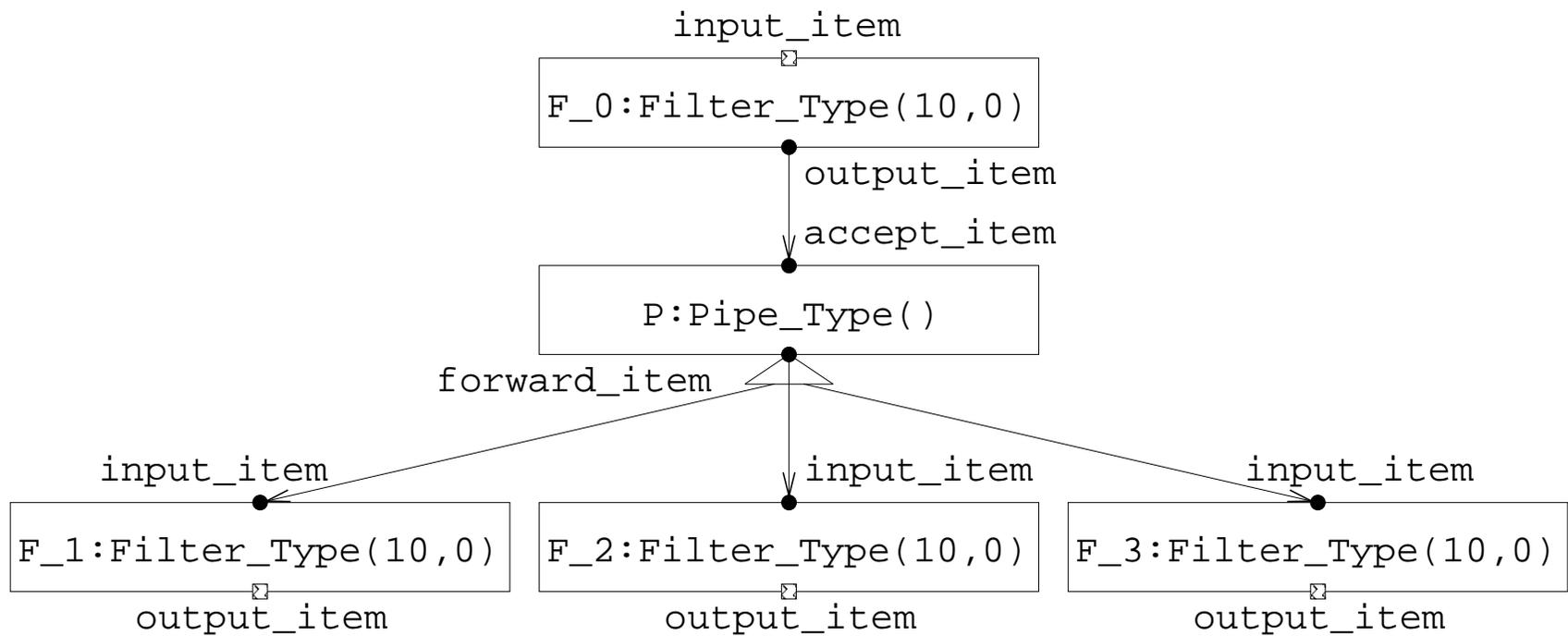
ARCHI_INTERACTIONS

```
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
```

- Enriched flow graph:



- Semantics of the AEs:

[[F_0]] = Filter{10/buffer_size, 0/item_num}

[[F_1]] = Filter{10/buffer_size, 0/item_num}

[[F_2]] = Filter{10/buffer_size, 0/item_num}

[[F_3]] = Filter{10/buffer_size, 0/item_num}

[[P]] = *or-rewrite*_∅(Pipe)

where *or-rewrite*_∅(Pipe) is given by:

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

- Semantics of `Pipe_Filter(10, 0)`:

$\llbracket F_0 \rrbracket [\text{output_item} \mapsto F_0.\text{output_item} \# P.\text{accept_item}]$

$\parallel_{\{F_0.\text{output_item} \# P.\text{accept_item}\}}$

$\llbracket P \rrbracket [\text{accept_item} \mapsto F_0.\text{output_item} \# P.\text{accept_item},$
 $\text{forward_item}_1 \mapsto P.\text{forward_item}_1 \# F_1.\text{input_item},$
 $\text{forward_item}_2 \mapsto P.\text{forward_item}_2 \# F_2.\text{input_item},$
 $\text{forward_item}_3 \mapsto P.\text{forward_item}_3 \# F_3.\text{input_item}]$

$\parallel_{\{P.\text{forward_item}_1 \# F_1.\text{input_item}\}}$

$\llbracket F_1 \rrbracket [\text{input_item} \mapsto P.\text{forward_item}_1 \# F_1.\text{input_item}]$

$\parallel_{\{P.\text{forward_item}_2 \# F_2.\text{input_item}\}}$

$\llbracket F_2 \rrbracket [\text{input_item} \mapsto P.\text{forward_item}_2 \# F_2.\text{input_item}]$

$\parallel_{\{P.\text{forward_item}_3 \# F_3.\text{input_item}\}}$

$\llbracket F_3 \rrbracket [\text{input_item} \mapsto P.\text{forward_item}_3 \# F_3.\text{input_item}]$

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 programs, hierarchical layers, 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 ...”.

- The concept of architectural style is hard to formalize.
- At least two degrees of freedom:
 - ⊙ variability of the component/connector behavior;
 - ⊙ variability of the system topology.
- Approximate an architectural style with an architectural type (AT).
- The behavior of components/connectors 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](#).
- Explicit support for passing data and [architectural parameters](#):
 - ⊙ actual values for formal data parameters (replace default ones);
 - ⊙ actual AETs *preserving the observable behavior of formal AETs*;
 - ⊙ actual topology *complying with the formal topology up to admitted topological variations*;
 - ⊙ actual behavioral modifications;
 - ⊙ actual names for architectural interactions (hierarchical modeling).
- The semantics of an AT instance is built the same way as the semantics of an AT definition (using actual parameters instead of formal ones), with actual architectural interactions relabeled to their actual names.

Hierarchical Modeling

- An AET can be an instance of a previously defined AT.
- AET behavior defined through AT invocation (semantics inheritance).
- AET local interactions given by AT architectural interactions.
- AET instances graphically represented as boxes with a double border.
- Simplest form of AT invocation:
 - ⊙ passing actual data parameters (if any);
 - ⊙ reusing formal AETs, topology, and behavioral modifications;
 - ⊙ unifying AT architectural interactions with AET local interactions
(locals are actual names for architecturals, whose synchronicities/multiplicities are ignored).

- **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 Hier_Client_Server(const integer hcs_buffer_size := 10,  
                               const integer hcs_init_item_num := 0)
```

- Formal data parameters are now necessary.
- The definition of the client AET does not change.
- We assume there is a single client AEI for simplicity.

- Redefinition of the server AET:

```
ARCHI_ELEM_TYPE Server_Type(const integer buffer_size,  
                             const integer init_item_num)
```

```
BEHAVIOR
```

```
Server(void; void) =
```

```
    Pipe_Filter(buffer_size, init_item_num @  
                @      /* reuse formal AETs */  
                @@@@   /* reuse formal topology */  
                @@@    /* no behavioral modifications */  
    UNIFY F_0.input_item WITH receive_request;  
    UNIFY F_1.output_item,  
          F_2.output_item,  
          F_3.output_item WITH send_response)
```

```
INPUT_INTERACTIONS SYNC UNI receive_request
```

```
OUTPUT_INTERACTIONS ASYNC UNI send_response
```

- Redeclaration of the topology:

```
ARCHI_ELEM_INSTANCES
```

```
S : Server_Type(hcs_buffer_size, hcs_init_item_num);
```

```
C : Client_Type()
```

```
ARCHI_INTERACTIONS
```

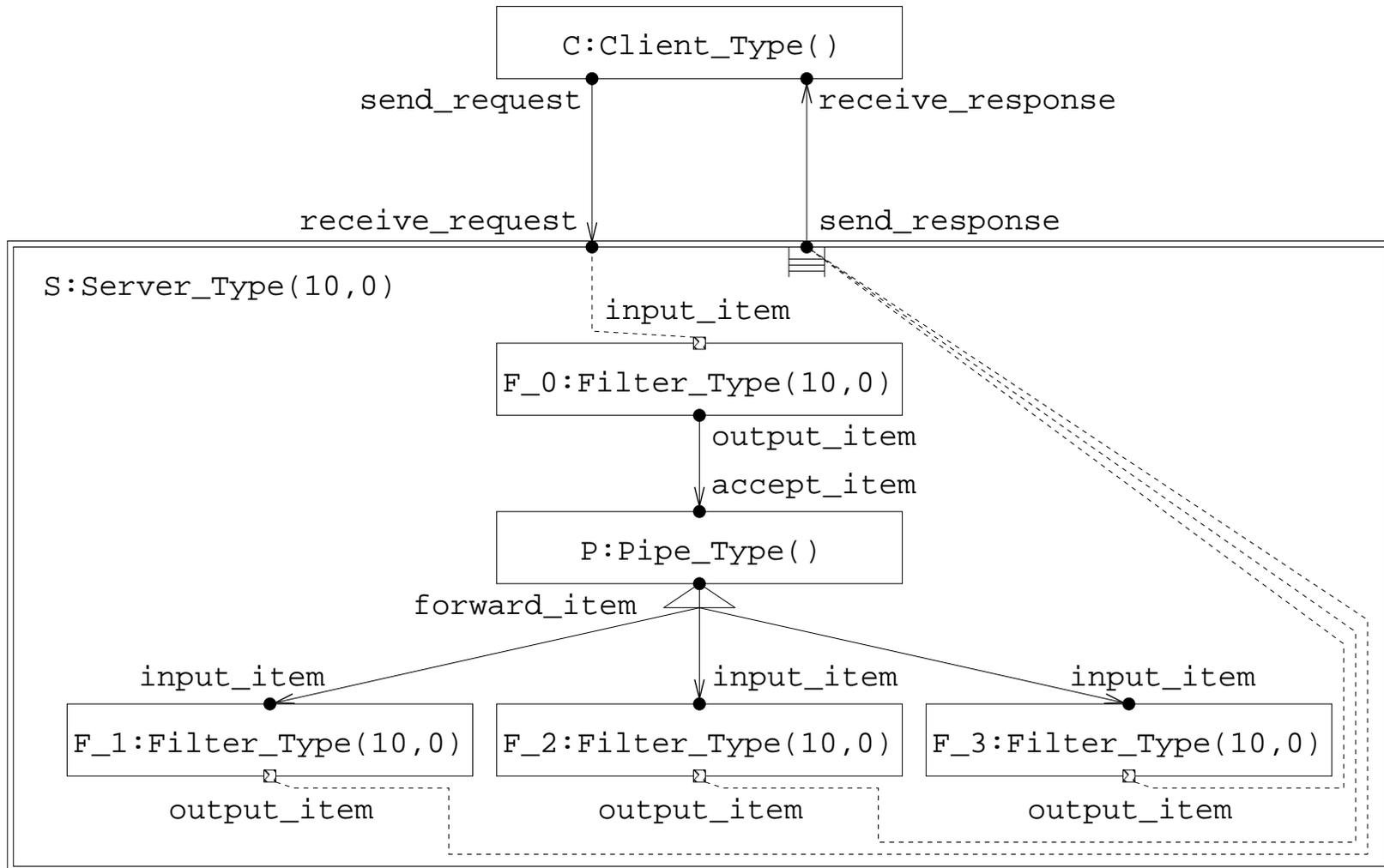
```
void
```

```
ARCHI_ATTACHMENTS
```

```
FROM C.send_request TO S.receive_request;
```

```
FROM S.send_response TO C.receive_response
```

- Enriched flow graph:



Behavioral Conformity

- 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 bisimilarity \approx_B :
 - ◉ Abstraction from internal actions.
 - ◉ Congruence with respect to static operators.
 - ◉ Preservation of many properties of interest.
- Compositionality of \approx_B guarantees AT behavioral conformity from AET behavioral conformity.
- The complexity of the AT behavioral conformity check is thus linear in the number of AETs (instead of exponential in the number of AETs).

- Let $\mathcal{C}_1, \mathcal{C}_2$ be two AETs.
- Let $\mathcal{D}_1, \mathcal{D}_2$ be the sets of their formal data parameters.
- Let $\mathcal{E}_1, \mathcal{E}_2$ be the sequences of their PA defining equations.
- Let $\mathcal{H}_1, \mathcal{H}_2$ be the sets of their internal actions.
- Let $\mathcal{I}_1, \mathcal{I}_2$ be 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 $\mathcal{D}_1, \mathcal{D}_2$:

$$(\mathcal{E}_1 \sigma) / \mathcal{H}_1 [\varphi_1] \approx_B (\mathcal{E}_2 \sigma) / \mathcal{H}_2 [\varphi_2]$$

- Two AT instances \mathcal{A}_1 and \mathcal{A}_2 behaviorally conform to each other iff:
 - ⊙ their actual data parameters are consistent by number, order, type, and value;
 - ⊙ *their AETs are consistent by number, order, and behavioral conformity;*
 - ⊙ their AETs are consistent by number, order, and type and have actual data parameters with the same values;
 - ⊙ their architectural interactions are consistent by number, order, qualifiers, and membership to corresponding AETs;
 - ⊙ their attachments are consistent by number, order, and qualifiers and membership to corresponding AETs of their local interactions.

- Let $\mathcal{A}_1, \mathcal{A}_2$ be two AT instances.
- Let $\mathcal{H}_1, \mathcal{H}_2$ be the sets of internal actions of their AEs.
- Let $\mathcal{I}_1, \mathcal{I}_2$ be the sets of interactions of their AEs.
- Whenever \mathcal{A}_1 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 the local interactions, have the same codomain, and are qualifier-consistent, such that:

$$\llbracket \mathcal{A}_1 \rrbracket_{\text{bbm}} / \mathcal{H}_1 [\varphi_1] \approx_{\text{B}} \llbracket \mathcal{A}_2 \rrbracket_{\text{bbm}} / \mathcal{H}_2 [\varphi_2]$$

- **Example:** instance of the pipe-filter AT with faulty filters.
- Definition of the faulty filter AET (subject to failures and subsequent repairs):

```

ARCHI_ELEM_TYPE Faulty_Filter_Type(const integer buffer_size,
                                   const integer init_item_num)

```

```

BEHAVIOR

```

```

    Faulty_Filter(integer(0..buffer_size) item_num := init_item_num;
                  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  UNI input_item

```

```

OUTPUT_INTERACTIONS UNI output_item

```

- Architectural invocation:

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

- Is the *AT* instance associated with the *AT* invocation a legal instance?
- The *Pipe_Filter* instance coming from the *AT* invocation behaviorally conforms to the *Pipe_Filter* instance coming from the *AT* definition because:
 - ⊙ they have the same data parameters;
 - ⊙ they have the same topology;
 - ⊙ they have the same pipe AET;
 - ⊙ *Filter_Type* and *Faulty_Filter_Type* have the same interactions;
 - ⊙ \approx_B equates *Filter*(*init_item_num*) / {*transform_item*} and *Faulty_Filter*(*init_item_num*) / {*transform_item*,*fail*,*repair*}.

Topological Conformity: Exogenous Variations

- Add further AEs by attaching them to the frontier of the topology, which coincides with the set of architectural interactions.
- An exogenous variation comprises the following parameters:
 - ⊙ set of additional AEs, which must be instances of the actual AEs;
 - ⊙ set of replacements of some of the actual architectural interactions with the additional architectural interactions, which must belong to additional AEs of the same type and must have the same name;
 - ⊙ set of additional attachments among the additional AEs and to the frontier, which must follow the pattern prescribed by the formal topology of the AT (no new kinds of attachments can be created);
 - ⊙ possible nested exogenous variations.
- The addendum must comply with the original topology.

- There must exist an AET-preserving injective function $corr$ defined from the set of additional AETs to the set of actual AETs such that:
 - ⊙ corresponding AETs have the same actual data parameter values;
 - ⊙ for all interactions a of an arbitrary additional AET C :
 - * $C.a$ is local/architectural iff $corr(C).a$ is local/architectural;
 - * there is an additional AET 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 AET 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 frontier is composed of:
 - ⊙ `F_0.input_item`;
 - ⊙ `F_1.output_item`, `F_2.output_item`, `F_3.output_item`.
- Consider an exogenous variation at `F_2.output_item`.
- Must replicate the defined topology by viewing `F_2` as an upstream filter.

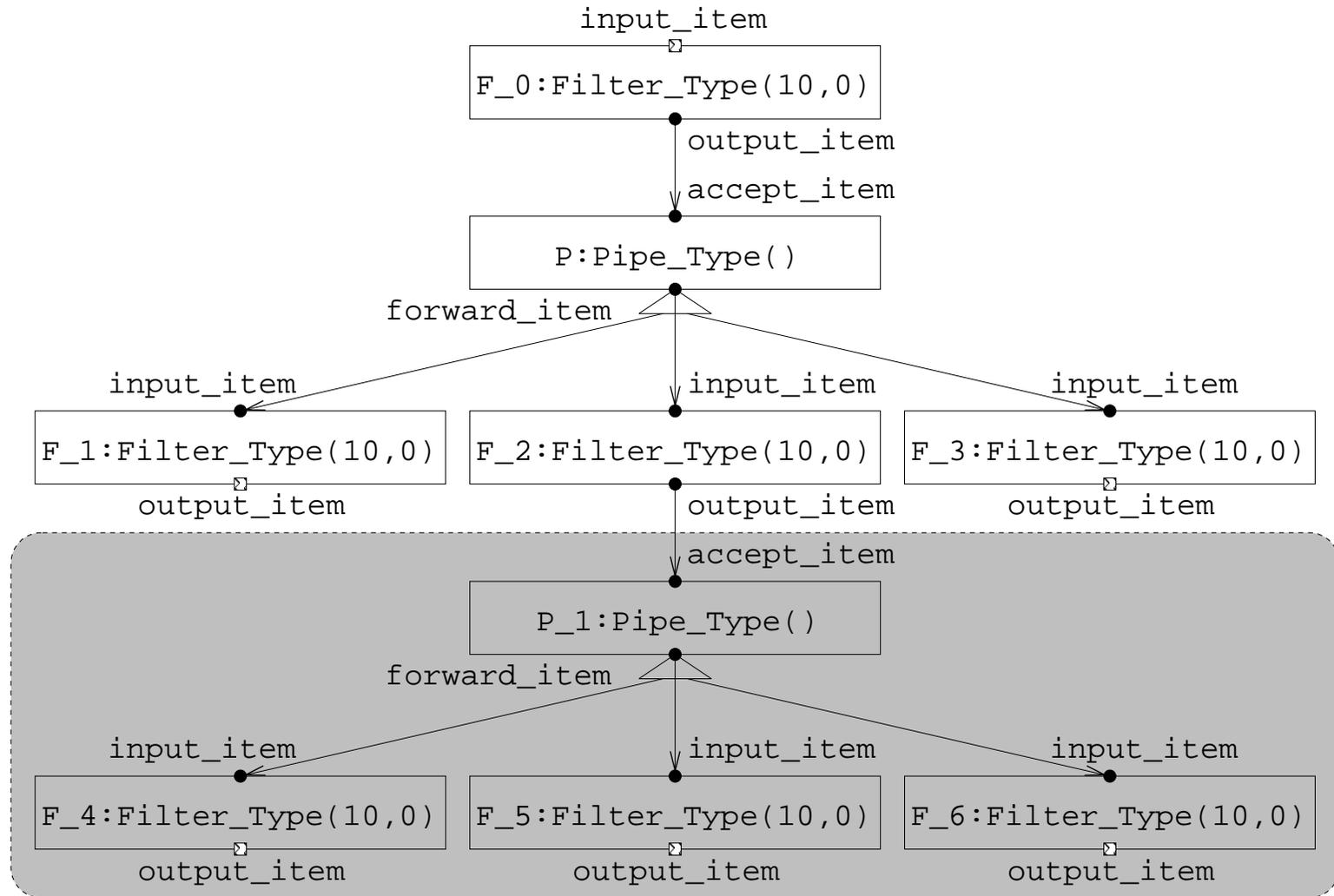
- Architectural invocation:

```

Pipe_Filter(@      /* reuse default values of formal data parameters */
@@@             /* reuse formal AETs and formal topology */
EXO(P_1 : Pipe_Type());
    F_4 : Filter_Type(pf_buffer_size, pf_init_item_num);
    F_5 : Filter_Type(pf_buffer_size, pf_init_item_num);
    F_6 : Filter_Type(pf_buffer_size, pf_init_item_num) @
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:



Topological Conformity: Endogenous Variations

- Vary the number of certain AEs inside the topology.
- Variable number of AEs defined as a data parameter of the AT.
- Indexing mechanism in the topology section of the AT definition for the concise declaration of arbitrarily many AEs and their architectural interactions and attachments.
- New kinds of attachments may be created when varying the number of considered AEs (from one to more than one).
- The variation complies with the original topology by construction.

- **Example:** endogenous variation of a station-ring AT.
- Protocol adopted by each station:
 - ⊙ wait for a message from the previous station in the ring;
 - ⊙ process the received message;
 - ⊙ send the processed message to the next station in the ring.
- Presence of one initial station that starts the system.
- Allowing for a variable number of non-initial stations.
- Header of the textual description:

```
ARCHI_TYPE Station_Ring(const integer sr_normal_station_num := 3)
```

- Definition of the initial station AET:

```
ARCHI_ELEM_TYPE Init_Station_Type(void)
```

```
BEHAVIOR
```

```
Init_Station(void; void) =  
    send_msg . receive_msg . process_msg . Init_Station()
```

```
INPUT_INTERACTIONS UNI receive_msg
```

```
OUTPUT_INTERACTIONS UNI send_msg
```

- Definition of the normal station AET:

```
ARCHI_ELEM_TYPE Station_Type(void)
```

```
BEHAVIOR
```

```
Station(void; void) =
```

```
receive_msg . process_msg . send_msg . Station()
```

```
INPUT_INTERACTIONS UNI receive_msg
```

```
OUTPUT_INTERACTIONS UNI send_msg
```

- Declaration of the topology:

```
ARCHI_ELEM_INSTANCES
```

```
IS : Init_Station_Type();
```

```
FOR_ALL 1 <= j <= sr_normal_station_num
```

```
    S[j] : Station_Type()
```

```
ARCHI_INTERACTIONS
```

```
void
```

```
ARCHI_ATTACHMENTS
```

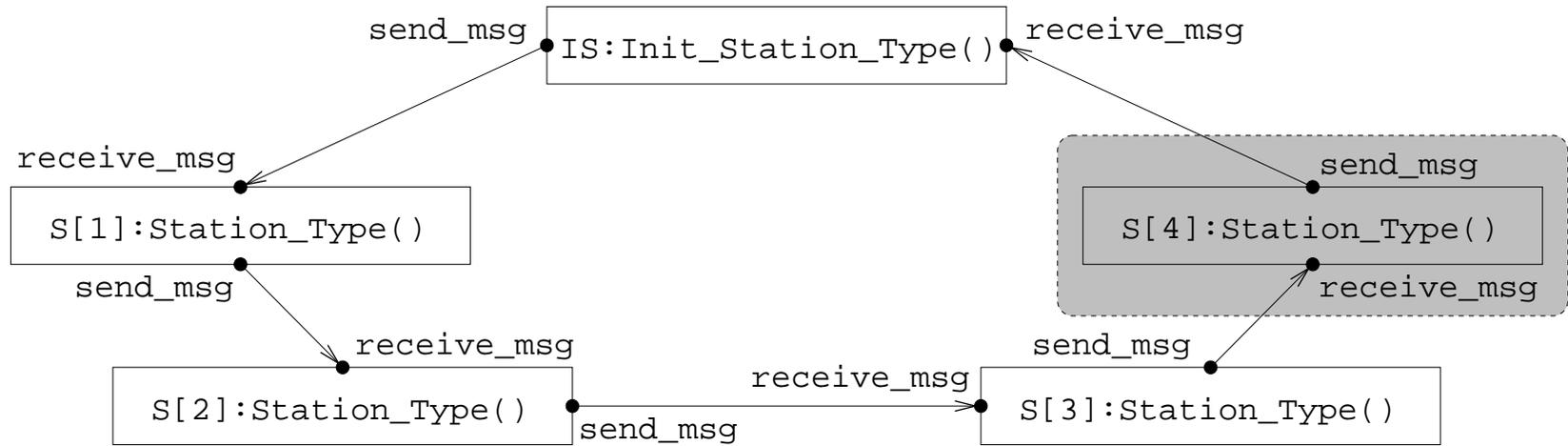
```
FROM IS.send_msg TO S[1].receive_msg;
```

```
FOR_ALL 1 <= j <= sr_normal_station_num - 1
```

```
    FROM S[j].send_msg TO S[j + 1].receive_msg;
```

```
FROM S[sr_normal_station_num].send_msg TO IS.receive_msg
```

- Enriched flow graph of Station_Ring(4 @@@@):



Topological Conformity: And-/Or-Variations

- Vary the number of certain AEs attached to and-/or-interactions.
- Variable number of AEs defined as a data parameter of the AT.
- Same indexing mechanism as for endogenous extensions.
- No new kinds of attachments can be created when varying the number of considered AEs.
- The variation complies with the original topology by construction if the involved and-/or-interactions support variability.
- An and-/or-interaction of an AE supports variability if the AE is not attached with uni-interactions to any of the AEs attached to the considered and-/or-interaction.

- **Example:** or-variation of the pipe-filter AT.
- Allow for a variable number of downstream filters.
- Header of the textual description:

```
ARCHI_TYPE Or_Var_Pipe_Filter(const integer ovpf_ds_filter_num := 3,  
                             const integer ovpf_buffer_size   := 10,  
                             const integer ovpf_init_item_num := 0)
```

- 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, ovpf_init_item_num);
```

```
P    : Pipe_Type();
```

```
FOR_ALL 1 <= j <= ovpf_ds_filter_num
```

```
    F[j] : Filter_Type(ovpf_buffer_size, ovpf_init_item_num)
```

```
ARCHI_INTERACTIONS
```

```
F[0].input_item;
```

```
FOR_ALL 1 <= j <= ovpf_ds_filter_num
```

```
    F[j].output_item
```

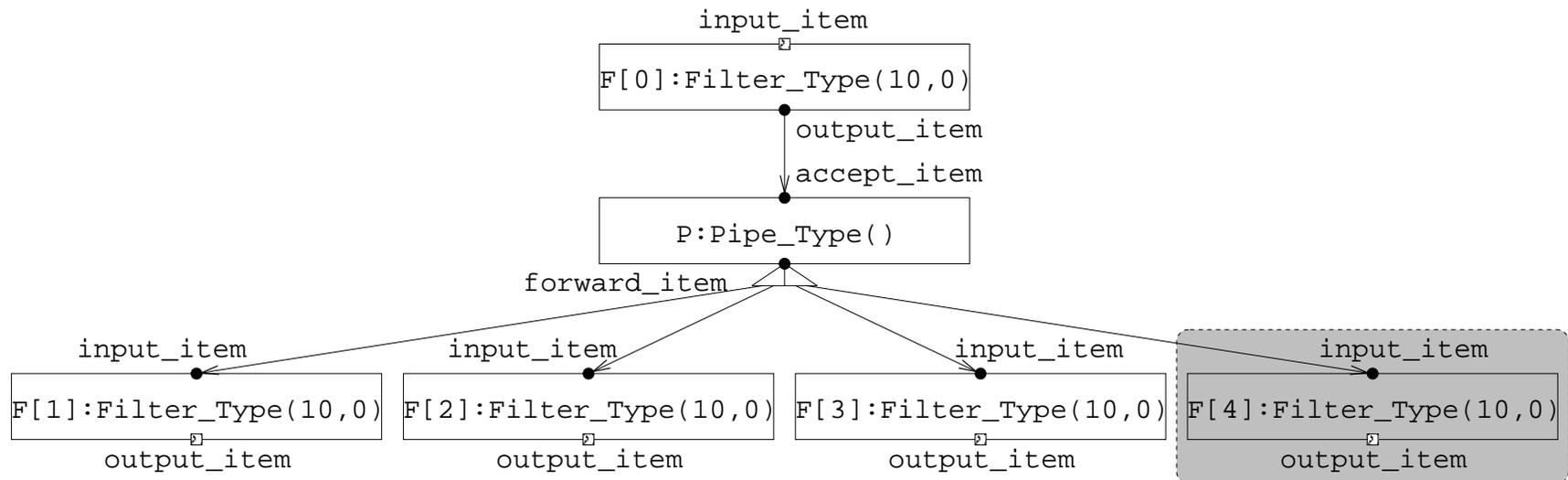
```
ARCHI_ATTACHMENTS
```

```
FROM F[0].output_item TO P.accept_item;
```

```
FOR_ALL 1 <= j <= ovpf_ds_filter_num
```

```
    FROM P.forward_item TO F[j].input_item
```

- Enriched flow graph of `Or_Var_Pipe_Filter(4, 10, 0 @@@@@@@@@@)`:



Comparison with PA

- Behavior description sharply separated from topology description
(no longer both 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 parallel composition operator).
- Communicating interactions explicitly related by means of attachments
(no longer required to have matching names).
- Only simpler-to-use operators available to the designer.
- Error-prone situations easily detected via static checks.
- Higher degree of specification reuse at the component/system level.
- Richer graphical notation.

Comparison with Parallel Composition Operators

- CCS-like:
 - ⊙ Support for two-way synchronizations only.
 - ⊙ Synchronizing actions required to have complementary names.
 - ⊙ Restriction operator necessary to enforce synchronizations.
- CSP-like:
 - ⊙ Support for multi-way synchronizations.
 - ⊙ Synchronizing actions required to have the same name.
 - ⊙ Explicit synchronization sets whose contents depend on the order.
- ACP-like:
 - ⊙ Definition of a communication function over the action name set.
 - ⊙ Restriction operator necessary to enforce synchronizations.
- Non-binary operators with explicit declaration of the process interfaces able to handle m -among- n synchronizations ($2 \leq m \leq n$).

Comparison with Process Algebraic ADLs

- Textual notation and translation semantics similar to those of Wright, Darwin/FSP, and π -ADL (only static architectures, no dynamic/mobile architectures).
- Components and connectors handled uniformly in order to avoid trivial architectural elements within system descriptions.
- Specific qualifiers for eliciting interaction synchronicity and multiplicity (synchronous/semi-synchronous/asynchronous, uni-/and-/or-).
- Behavioral modifications for supporting ad-hoc analyses.
- Architectural types for approximating architectural styles:
 - ⊙ Hierarchical modeling.
 - ⊙ Behavioral conformity (internal behavior variations).
 - ⊙ Topological conformity (exogenous, endogenous, and-/or-variations).

Part II:
Component-Oriented Verification

Architectural Mismatches

- PADL is much easier to use than PA for modeling purposes.
 - △ *What about system property verification?*
- All the analysis techniques developed for PA are inherited by PADL.
 - △ *Are those techniques appropriate 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, they result in lack of coordination among components at run time.

- Inferring system properties from the properties of system components.
- Identifying components responsible for property violations.
- Topological reduction process based on equivalence checking, which:
 - ⊙ scales from basic topological formats to arbitrary topologies;
 - ⊙ scales from single architectures to architectural types;
 - ⊙ returns distinguishing modal logic formulas in case of failure.
- Use of process algebraic machinery only.

Class of Architectural Properties

- Properties expressed in terms of the possibility/necessity of executing *local interactions* in a certain order, as architectural mismatches can only be generated by the wrong interplay of those activities.
- Class of properties \mathcal{P} for each of which there exists $\approx_{\mathcal{P}}$ that is:
 - ◉ \mathcal{P} -preserving, so as to support the topological reduction process;
 - ◉ congruence w.r.t. static operators, so as to achieve efficiency;
 - ◉ weak, so as to be able to abstract from internal actions.
- The action-based modal/temporal logic in which the properties of the class are expressed must not allow the negation to be freely used.
- **Example:** deadlock freedom or any negation-free formula of weak HML and \approx_{B} .

Architectural Checks

- Abstract variant of enriched flow graph:
 - ⊙ vertices correspond to AEs;
 - ⊙ two vertices are linked by an edge iff attachments have been declared among the interactions of their corresponding AEs.
- Basic topological formats: stars and cycles.
- Architectural checks are applied locally to stars and cycles of AEs:
 - ⊙ Objective: verify whether the whole topology can be reduced to a single $\approx_{\mathcal{P}}$ -equivalent AE that satisfies \mathcal{P} .
 - ⊙ Strategy: replace any set of AEs forming a star or a cycle with a single $\approx_{\mathcal{P}}$ -equivalent AE in the set that satisfies \mathcal{P} .
- The congruence property of $\approx_{\mathcal{P}}$ w.r.t. static operators avoids verifying the preservation of the overall behavior at each reduction step.

- Before applying the checks, we need to hide all the internal actions and all the architectural interactions as they cannot result in mismatches but may hamper the reduction process if left visible.
- Likewise we also need to hide all the asynchronous local interactions together with the attached interactions of the related implicit AEs.
- Let $\{C_1, \dots, C_n\}$ be a set of AEs.
- Closed interacting semantics of C_j with respect to $\{C_1, \dots, C_n\}$:

$$\begin{aligned}
\llbracket C_j \rrbracket_{C_1, \dots, C_n}^c &= \llbracket C_j \rrbracket_{C_1, \dots, C_n} / (\text{Name} - \mathcal{LI}_{C_j; C_1, \dots, C_n}) \\
&\quad / \{AIQ_1.\text{depart}\#C_j.i_1, \dots, AIQ_h.\text{depart}\#C_j.i_h, \\
&\quad C_j.o_1\#AOQ_1.\text{arrive}, \dots, C_j.o_k\#AOQ_k.\text{arrive}, \\
&\quad C_j.i_1\text{-exception}, \dots, C_j.i_h\text{-exception}\}
\end{aligned}$$

- Closed interacting semantics of $\{C'_1, \dots, C'_{n'}\} \subseteq \{C_1, \dots, C_n\}$ with respect to $\{C_1, \dots, C_n\}$:

$$\begin{aligned} \llbracket C'_1, \dots, C'_{n'} \rrbracket_{C_1, \dots, C_n}^c &= \llbracket C'_1 \rrbracket_{C_1, \dots, C_n}^c \parallel_{\mathcal{S}(C'_1, C'_2; C_1, \dots, C_n)} \\ &\quad \llbracket C'_2 \rrbracket_{C_1, \dots, C_n}^c \parallel_{\mathcal{S}(C'_1, C'_3; C_1, \dots, C_n) \cup \mathcal{S}(C'_2, C'_3; C_1, \dots, C_n)} \\ &\quad \dots \parallel_{\bigcup_{j=1}^{n'-1} \mathcal{S}(C'_j, C'_{n'}; C_1, \dots, C_n)} \llbracket C'_{n'} \rrbracket_{C_1, \dots, C_n}^c \end{aligned}$$

- Closed semantics of $\mathcal{A} = \{C_1, \dots, C_n\}$ before behavioral modifications:

$$\llbracket \mathcal{A} \rrbracket_{\text{bbm}}^c = \llbracket C_1, \dots, C_n \rrbracket_{C_1, \dots, C_n}^c$$

Architectural Compatibility Check for Stars

- A star is an acyclic portion of the abstract enriched flow graph of an AT instance \mathcal{A} , formed by:
 - ◉ the central AEI K ;
 - ◉ the border $\mathcal{B}_K = \{C_1, \dots, C_n\}$ of all the AEIs attached to K .
- The validity of a property \mathcal{P} can be investigated by analyzing the interplay between the center K of the star and each of the AEIs in the border of the star, as there are no other attachments in the star.
- 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:

$$(\llbracket K \rrbracket_{K, \mathcal{B}_K}^c \parallel_{S(K, C_j; K, \mathcal{B}_K)} \llbracket C_j \rrbracket_{K, \mathcal{B}_K}^c) / H_j \approx_{\mathcal{P}} \llbracket K \rrbracket_{K, \mathcal{B}_K}^c / H_j$$

- H_j includes all the semi-synchronous interactions between K and C_j as well as all the interactions of implicit AEs between K and C_j .
- Each of them has a specific outcome at a specific time when executed by K in isolation.
- But can have a different outcome (if semi-synchronous) or be delayed (if asynchronous) when executed by K in parallel with C_j .
- If not hidden, those interactions may lead to inequivalence even if there are no mismatches.

- Whenever K is \mathcal{P} -compatible with every $C_j \in \mathcal{B}_K$ and is such that $\llbracket K \rrbracket_{K, \mathcal{B}_K}^c / \bigcup_{j=1}^n H_j$ satisfies \mathcal{P} , then $\llbracket K, \mathcal{B}_K \rrbracket_{K, \mathcal{B}_K}^c / \bigcup_{j=1}^n H_j$ satisfies \mathcal{P} provided that $H_j \cap H_g = \emptyset$ for all $j \neq g$ (no non-synchronous and-interaction or non-synchronous interaction attached to an and-interaction).
- If the \mathcal{P} -compatibility check is passed by all $C_j \in \mathcal{B}_K$, then the star can be reduced to its center K .
- If the \mathcal{P} -compatibility check is not passed by some $C_g \in \mathcal{B}_K$, then this reveals a potential violation of \mathcal{P} in the interplay 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 $\llbracket K, \mathcal{B}_K \rrbracket_{K, \mathcal{B}_K}^c$ against \mathcal{P} grows exponentially with the size of $\{K\} \cup \mathcal{B}_K$.

Example: 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 communicating through a function-call-based stream interface that allows an upstream filter to push data into a downstream filter.
- The gzip program is a Unix filter communicating through pipes.
- Objective: design the system so that deadlock freedom is ensured.

- The gzip program explicitly chooses when to get data, while the HTTP server is forced to read when data are pushed to it.
- The gzip program may attempt to output a portion of the compressed data before finishing getting all the input data (internal buffer full).
- Assembling the compressing proxy system from the existing HTTP server and the gzip program without modifications.
- Need for a software adaptor because of the different communication mechanisms of the HTTP server and of the gzip program.
- Textual description header:

`ARCHI_TYPE Compressing_Proxy(void)`

- Definition of the upstream filter AET:

```
ARCHI_ELEM_TYPE UFilter_Type(void)
```

```
BEHAVIOR
```

```
UFilter(void; void) =  
    write_data . UFilter()
```

```
INPUT_INTERACTIONS void
```

```
OUTPUT_INTERACTIONS UNI write_data
```

- Definition of the downstream filter AET:

```
ARCHI_ELEM_TYPE DFilter_Type(void)
```

```
BEHAVIOR
```

```
DFilter(void; void) =  
    read_data . DFilter()
```

```
INPUT_INTERACTIONS  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  UNI receive_from_filter; get_from_gzip; get_eoo_gzip
```

```
OUTPUT_INTERACTIONS 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  UNI get_data; get_eoi
```

```
OUTPUT_INTERACTIONS UNI put_data; put_eoo
```

- Declaration of the topology:

```
ARCHI_ELEM_INSTANCES
```

```
UF : UFilter_Type();
```

```
DF : DFilter_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;
```

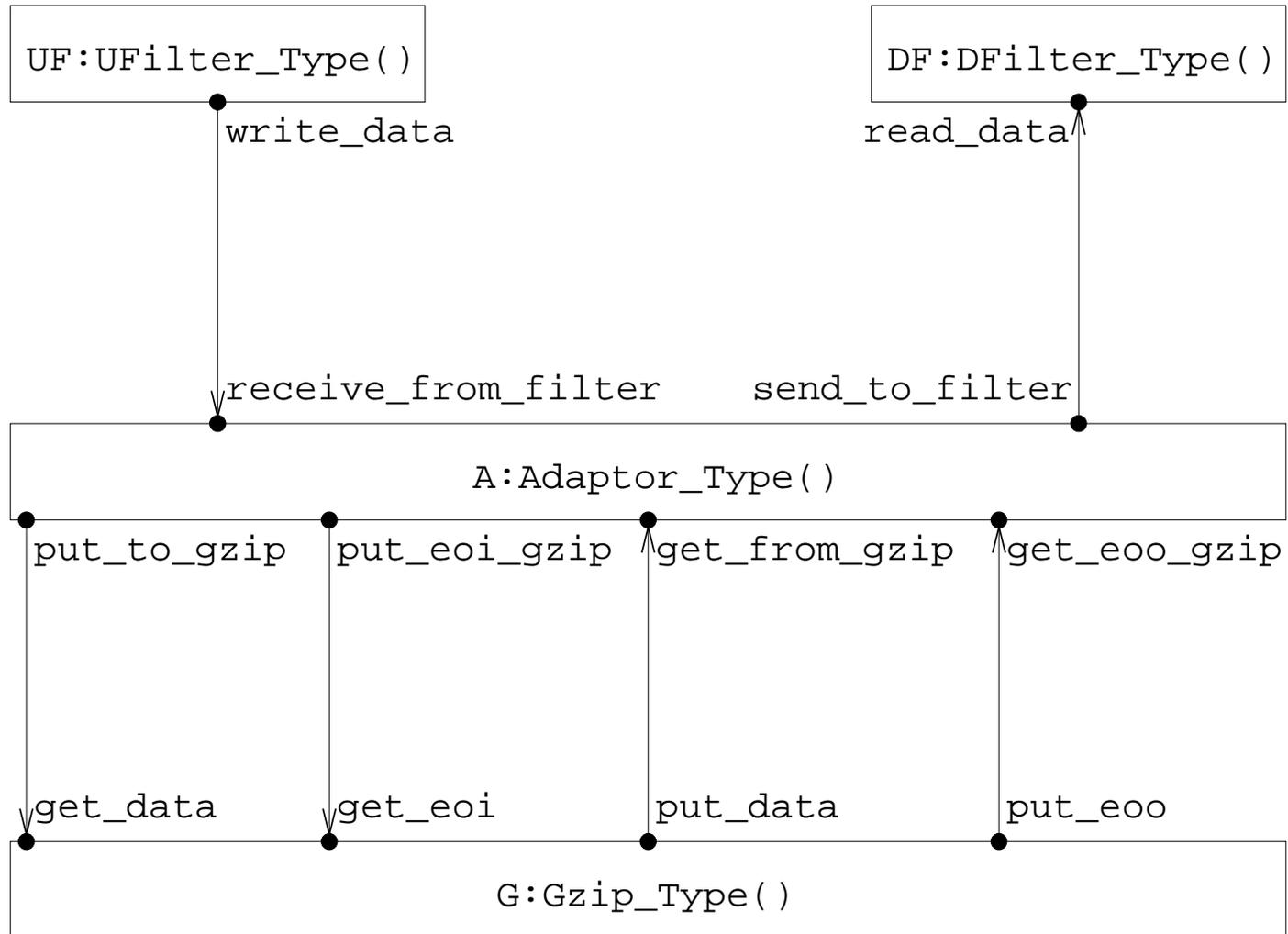
```
FROM A.put_eoi_gzip TO G.get_eoi;
```

```
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 topology is a star whose center is **A**.
- **A** is deadlock free but not deadlock-freedom-compatible with **G** because:

$$\llbracket A \rrbracket_{A,UF,DF,G}^c \parallel_{S(A,G;A,UF,DF,G)} \llbracket G \rrbracket_{A,UF,DF,G}^c \not\approx_B \llbracket A \rrbracket_{A,UF,DF,G}^c$$

- Distinguishing weak HML formula:

```

⟨⟨A.receive_from_filter⟩⟩
  ⟨⟨A.put_to_gzip#G.get_data⟩⟩
    ¬⟨⟨A.put_eoi_gzip#G.get_eoi⟩⟩ 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 iff **A** signalled eoi.
- Solution:
 - ⊙ **A** must be redesigned in order to account for the possibility of receiving compressed data from **G** at any time;
 - ⊙ **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_eoi_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  UNI receive_from_filter;
                    get_from_gzip; get_eoo_gzip;
                    notified_buffer_full
```

```
OUTPUT_INTERACTIONS 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  UNI get_data; get_eoi
```

```
OUTPUT_INTERACTIONS UNI put_data; put_eoo;  
                    notify_buffer_full
```

- Redeclaration of the attachments:

ARCHI_ATTACHMENTS

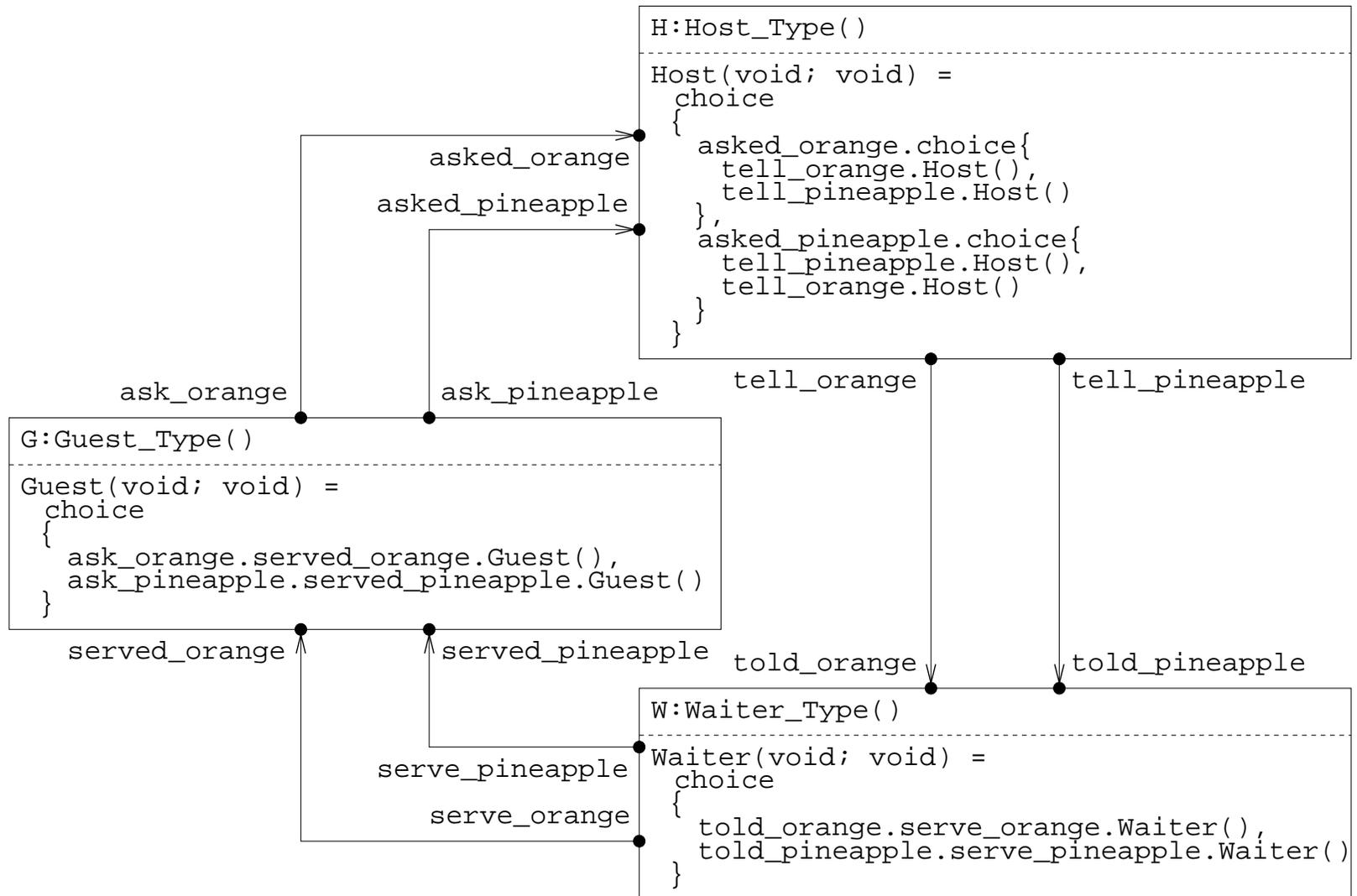
```
FROM UF.write_data      TO A.receive_from_filter;
FROM A.put_to_gzip      TO G.get_data;
FROM A.put_eoi_gzip     TO G.get_eoi;
FROM G.notify_buffer_full TO A.notified_buffer_full;
FROM G.put_data         TO A.get_from_gzip;
FROM G.put_eoo          TO A.get_eoo_gzip;
FROM A.send_to_filter   TO DF.read_data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

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

Architectural Interoperability Check for Cycles

- The compatibility check is not enough to deal with cycles.
- **Example:** guest analogy.
- Party with three actors: 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 textual notation:



- Can the party deadlock?
- H, W, G are deadlock free.
- H is deadlock-freedom-compatible both with W and with G.
- But the overall system is not deadlock free!
- The topology is a cycle, not a star!

- A cycle is a closed simple path in the abstract enriched flow graph of an AT instance \mathcal{A} , formed by the set of AEIs $\{C_1, \dots, C_n\}$ (with $n \geq 3$).
- Each AEI in the cycle may interfere with any of the other AEIs in the cycle, hence when investigating a property \mathcal{P} these AEIs can no longer be considered two-by-two as in the compatibility check.
- Coordination is ensured if the actual observable behavior of any AEI in the cycle coincides with the observable behavior expected by the rest of the cycle.
- An AEI [interoperates](#) with the rest of the cycle if the observable behavior of the AEI is not altered by the insertion of the AEI itself into the cycle.

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

$$\boxed{\llbracket C_1, \dots, C_n \rrbracket_{\mathcal{A}}^c / (\text{Name} - \mathcal{S}(C_j; \mathcal{A})) / H_j \approx_{\mathcal{P}} \llbracket C_j \rrbracket_{\mathcal{A}}^c / H_j}$$

- H_j includes all the semi-synchronous interactions between C_j and the rest of the cycle as well as all the interactions of implicit AEs between C_j and the rest of the cycle.
- Each of them has a specific outcome at a specific time when executed by C_j in isolation.
- But can have a different outcome (if semi-synchronous) or be delayed (if asynchronous) when executed by C_j inside the cycle.
- If not hidden, those interactions may lead to inequivalence even if there are no mismatches.

- Interoperability is an adaptation of compatibility to cycles:

$$\begin{array}{ccc}
 ([C_1]_{\mathcal{A}}^c \parallel_{\bar{S}_1} [C_2, \dots, C_n]_{\mathcal{A}}^c) / \hat{H}_1 / H_1 & \approx_{\mathcal{P}} & [C_1]_{\mathcal{A}}^c / H_1 \\
 \vdots & & \vdots \\
 ([\dots, C_{j-1}]_{\mathcal{A}}^c \parallel_{\tilde{S}_{j-1}} [C_j]_{\mathcal{A}}^c \parallel_{\bar{S}_j} [C_{j+1}, \dots]_{\mathcal{A}}^c) / \hat{H}_j / H_j & \approx_{\mathcal{P}} & [C_j]_{\mathcal{A}}^c / H_j \\
 \vdots & & \vdots \\
 ([C_1, \dots, C_{n-1}]_{\mathcal{A}}^c \parallel_{\tilde{S}_{n-1}} [C_n]_{\mathcal{A}}^c) / \hat{H}_n / H_n & \approx_{\mathcal{P}} & [C_n]_{\mathcal{A}}^c / H_n
 \end{array}$$

where:

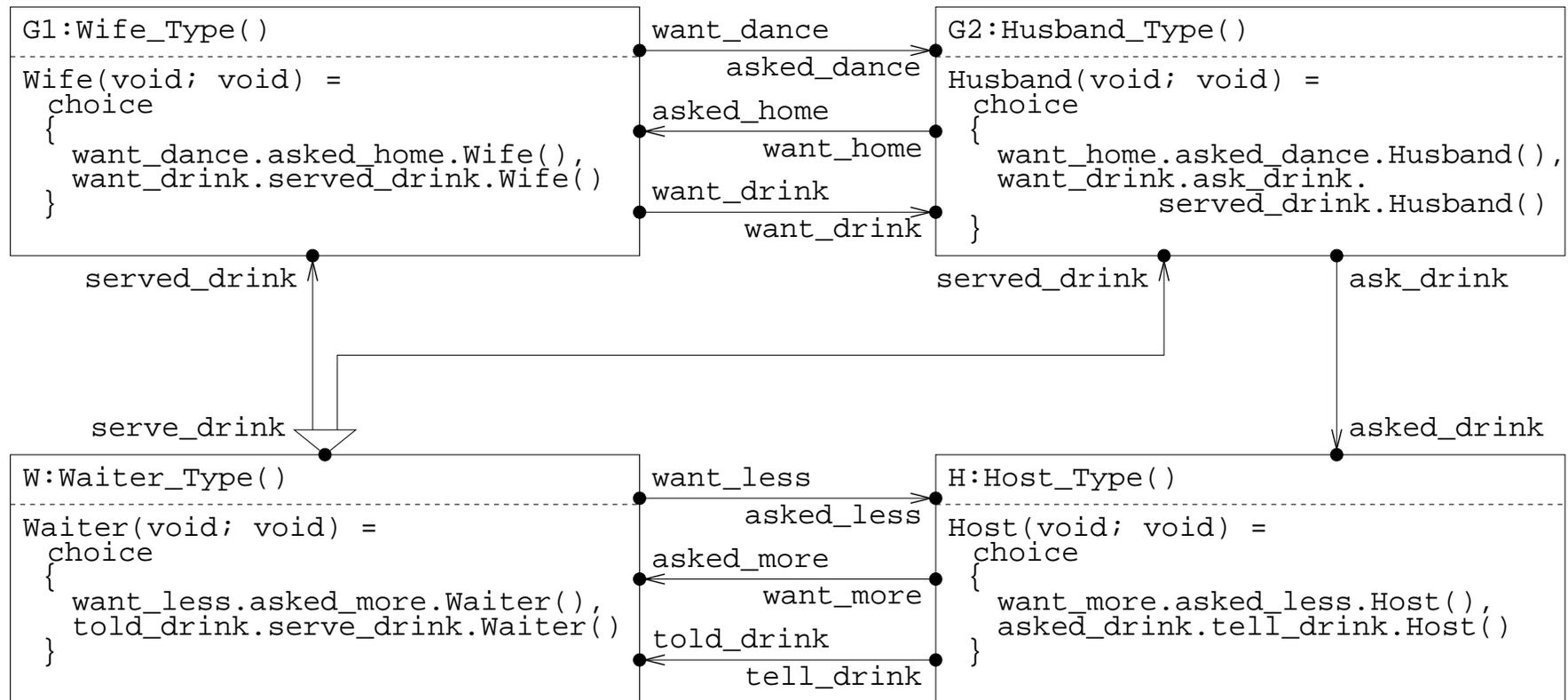
- ⊙ $\bar{S}_j = \bigcup_{f=1}^j \bigcup_{g=j+1}^n \mathcal{S}(C_f, C_g; \mathcal{A});$
- ⊙ $\tilde{S}_j = \bigcup_{g=1}^j \mathcal{S}(C_g, C_{j+1}; \mathcal{A});$
- ⊙ $\hat{H}_j = \text{Name} - \mathcal{S}(C_j; \mathcal{A}).$

- Whenever there exists C_j in the cycle that \mathcal{P} -interoperates with the other AEs in the cycle and is such that $\llbracket C_j \rrbracket_{\mathcal{A}}^c / H_j$ satisfies \mathcal{P} , then $\llbracket C_1, \dots, C_n \rrbracket_{\mathcal{A}}^c / (\text{Name} - \mathcal{S}(C_j; \mathcal{A})) / H_j$ satisfies \mathcal{P} .
- If the \mathcal{P} -interoperability check is passed by some C_j in the cycle, then the cycle can be reduced to C_j .
- If the \mathcal{P} -interoperability check is not passed by any C_g in the cycle, then this reveals a potential violation of \mathcal{P} within the cycle.
- The cost of the \mathcal{P} -interoperability-check-based verification grows exponentially with the size of the cycle.
- Can be mitigated if the state space of the cycle is built compositionally and minimized at each step with respect to $\approx_{\mathcal{P}}$.

- Loop shrinking procedure for investigating potential violations:
 - ◉ Consider an AEI C_g in the cycle that does not \mathcal{P} -interoperate with the other AEs in the cycle.
 - ◉ Study the behavior of C_g together with modal-logic diagnostic information coming from the failure of the \mathcal{P} -interoperability check for C_g in order to determine whether the source of a potential violation of \mathcal{P} is within C_g , the rest of the cycle, or both.
 - ◉ If C_g violates \mathcal{P} , modify it 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 given by the parallel composition of the closed interacting semantics of the three original AEs, then repeat the \mathcal{P} -interoperability check.

- The effectiveness of the interoperability check can be improved by considering sets of adjacent AEs in the cycle instead of single AEs.
- **Example:** guest analogy revisited.
- Party with four actors: host, waiter, couple of guests.
- As before, whenever a guest wants something to drink, the guest asks the host who in turn tells the waiter to serve the requested drink.
- The wife would like to dance with her husband or to get a drink, whereas her husband would like to go home or to get a drink (agreement on drinking).
- The host would like the waiter to work one more hour (successful party), whereas the waiter would like to leave one hour in advance (very tired).

- Enriched flow graph integrated with textual notation:



- Can the party deadlock?
- $G1$, $G2$, H , W are deadlock free.
- None of those AEs deadlock-freedom-interoperates with the others because of the wife-husband and host-waiter conflicts.
- However the overall system is deadlock free.
- The AE resulting from the parallel composition of $G1$ and $G2$ deadlock-freedom-interoperates with the remaining AEs!

- Consider a cycle formed by the set of AEs $\{C_1, \dots, C_n\}$.
- Take a subset of l adjacent AEs $\{C'_1, \dots, C'_l\}$ with $1 \leq l \leq n/2$.
- The l adjacent AEs interoperate with the rest of the cycle if the observable behavior of those AEs is not altered by the insertion of the AEs themselves into the cycle.
- $\{C'_1, \dots, C'_l\}$ \mathcal{P} -interoperates with the other AEs in the cycle iff:

$$\boxed{\llbracket C_1, \dots, C_n \rrbracket_{\mathcal{A}}^c / (\text{Name} - \bigcup_{j=1}^l \mathcal{S}(C'_j; \mathcal{A})) / \bigcup_{j=1}^l H'_j \approx_{\mathcal{P}} \llbracket C'_1, \dots, C'_l \rrbracket_{\mathcal{A}}^c / \bigcup_{j=1}^l H'_j}$$

- Whenever there exists $\{C'_1, \dots, C'_l\}$ in the cycle that \mathcal{P} -interoperates with the other AEs in the cycle and is such that $\llbracket C'_1, \dots, C'_l \rrbracket_{\mathcal{A}}^c / \bigcup_{j=1}^l H'_j$ satisfies \mathcal{P} , then $\llbracket C_1, \dots, C_n \rrbracket_{\mathcal{A}}^c / (\text{Name} - \bigcup_{j=1}^l \mathcal{S}(C'_j; \mathcal{A})) / \bigcup_{j=1}^l H'_j$ satisfies \mathcal{P} .

Example: 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 system is able to accelerate or decelerate the automobile back to the previously recorded speed.
- Objective: design the system so that deadlock freedom is ensured.

- Software units: sensor, speed controller, speed detector, speed actuator.
- The sensor forwards the driver's commands to the speed controller.
- The speed controller triggers the speed actuator on the basis of the driver's commands that are received from the sensor (inactive, active, cruising, suspended).
- The speed detector periodically communicates the number of wheel revolutions per time unit to the speed actuator.
- 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 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  UNI turned_engine_on; turned_engine_off;  
                    pressed_accelerator; pressed_brake;  
                    pressed_on; pressed_off; pressed_resume  
OUTPUT_INTERACTIONS UNI trigger_record; trigger_resume;  
                    trigger_disable
```

- 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  UNI turned_engine_on; turned_engine_off
```

```
OUTPUT_INTERACTIONS UNI signal_speed
```

- Definition of the speed actuator AET:

```
ARCHI_ELEM_TYPE Actuator_Type(void)
```

```
BEHAVIOR
```

```
  Disabled(void; void) =
```

```
    choice
```

```
    {
```

```
      signalled_speed . Disabled(),
```

```
      triggered_record . record_speed . Enabled(),
```

```
      triggered_resume . resume_speed . Enabled()
```

```
    };
```

```
  Enabled(void; void) =
```

```
    choice
```

```
    {
```

```
      signalled_speed . adjust_throttle . Enabled(),
```

```
      triggered_disable . disable_control . Disabled()
```

```
    }
```

```
INPUT_INTERACTIONS  UNI triggered_record; triggered_resume;
```

```
                    triggered_disable;
```

```
                    signalled_speed
```

```
OUTPUT_INTERACTIONS void
```

- Declaration of the topology:

ARCHI_ELEM_INSTANCES

S : Sensor_Type();

C : Controller_Type();

D : Detector_Type();

A : Actuator_Type()

ARCHI_INTERACTIONS

S.detected_engine_on; S.detected_engine_off;

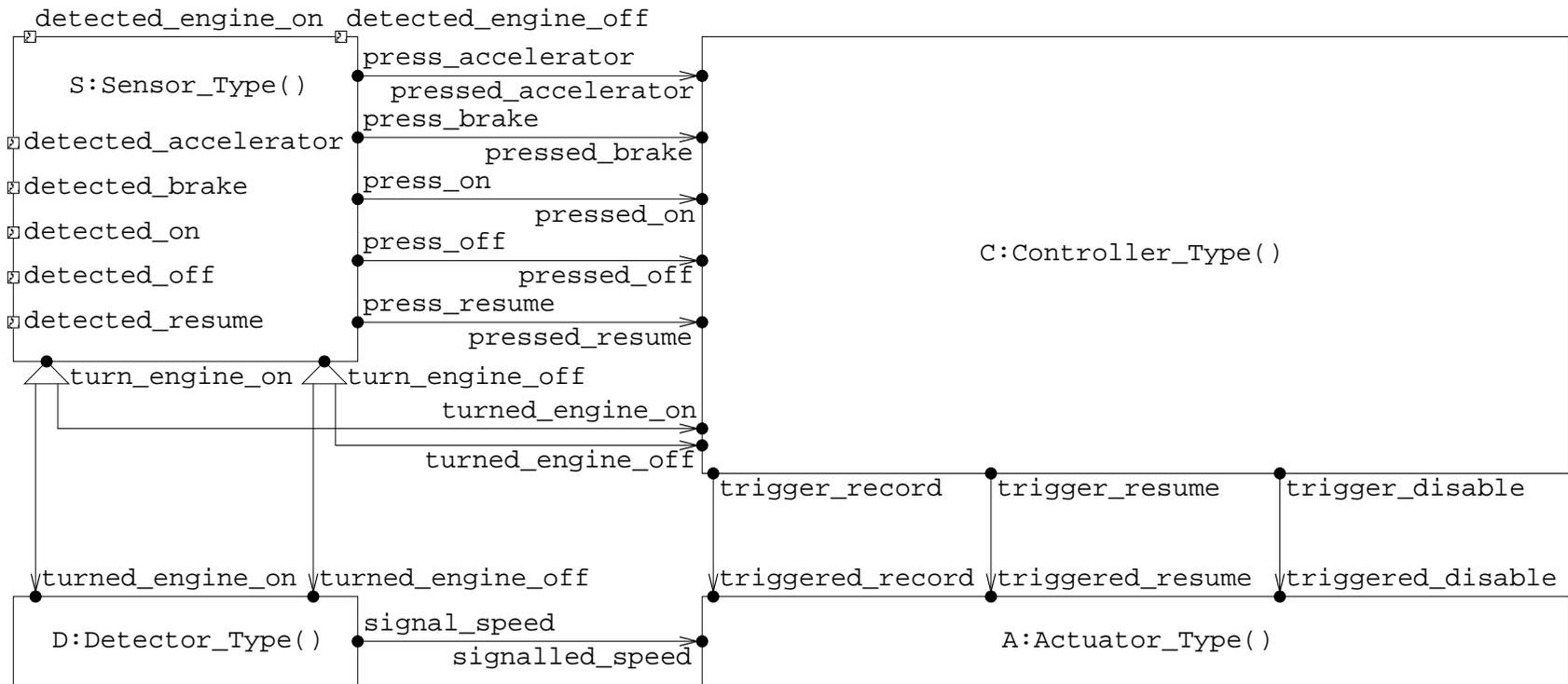
S.detected_accelerator; S.detected_brake;

S.detected_on; S.detected_off; S.detected_resume

ARCHI_ATTACHMENTS

```
FROM S.turn_engine_on      TO C.turned_engine_on;
FROM S.turn_engine_on      TO D.turned_engine_on;
FROM S.turn_engine_off     TO C.turned_engine_off;
FROM S.turn_engine_off     TO D.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 C.trigger_record      TO A.triggered_record;
FROM C.trigger_resume      TO A.triggered_resume;
FROM C.trigger_disable     TO A.triggered_disable;
FROM D.signal_speed        TO A.signalled_speed
```

- Enriched flow graph:



- Cycle formed by S, C, D, A.
- All the AEs 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 signalling activities take place endlessly between D and A as long as the engine is running.
- If we hide those activities, a deadlock shows up.
- They are hidden in the interoperability checks for S and C.

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

$$\llbracket S, C, D, A \rrbracket_{S,C,D,A}^C / (Name - \mathcal{S}(S; S, C, D, A)) \not\approx_B \llbracket S \rrbracket_{S,C,D,A}^C$$

- Distinguishing weak HML formula:

```

⟨⟨S.turn_engine_on#C.turned_engine_on#D.turned_engine_on⟩⟩
  ⟨⟨S.press_on#C.pressed_on⟩⟩
    ⟨⟨S.turn_engine_off#C.turned_engine_off#D.turned_engine_off⟩⟩
      ⟨⟨S.turn_engine_on#C.turned_engine_on#D.turned_engine_on⟩⟩
        ⟨⟨S.press_on#C.pressed_on⟩⟩
          ¬⟨⟨S.press_brake#C.pressed_brake⟩⟩ 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!

- The solution is to modify the cruising behavior of the speed controller 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 rest of the cycle.
- Then **Cruise_Control** is deadlock free even if we hide the speed signalling activities.

Generalization to Arbitrary Topologies

- A topology can contain arbitrarily many stars and cycles.
- Careful treatment of any AEI that belongs to the intersection of a cycle with an acyclic portion of the topology or with another cycle.
- The **frontier** of $\{C_1, \dots, C_n\} \subseteq \mathcal{A}$ is the subset of those AEIs in the set that can interact with the rest of the topology:

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

- The reduction of every cycle into a single $\approx_{\mathcal{P}}$ -equivalent AEI is implemented through a **cycle covering strategy**.
- The cyclic border of $K \in \mathcal{A}$ is the set of all cycles traversing K :

$$\mathcal{CB}_K = \{C \in \mathcal{A} \mid K \text{ and } C \text{ are in the same cycle}\}$$

- A cycle covering strategy σ for \mathcal{A} is defined by the following algorithm:
 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 K .
 - (b) Mark all the AEIs in \mathcal{CB}_K .

- The result of a cycle covering strategy is a set \mathcal{CB}_σ of cyclic borders that involve every AEI belonging to a cycle in the abstract enriched flow graph of \mathcal{A} .
- Such cyclic borders are pairwise connected at most through a single shared AEI or through the attachments between a single AEI of one cyclic border and a single AEI of the other cyclic border.
- A cycle covering strategy σ for \mathcal{A} is **total** iff, after replacing each cyclic border $\mathcal{CB}_K = \{C_1, \dots, C_n\} \in \mathcal{CB}_\sigma$ with an AEI whose behavior is isomorphic to:

$$\llbracket C_1, \dots, C_n \rrbracket_{\mathcal{A}}^c / (\text{Name} - \bigcup_{C_j \in \mathcal{F}_{C_1, \dots, C_n}} \mathcal{S}(C_j; \mathcal{A}))$$

the resulting topology is acyclic.

- Let \mathcal{A} be an architectural type such that the following conditions hold:
 1. For each $K \in \mathcal{A}$, $\llbracket K \rrbracket_{\mathcal{A}}^c / H_K$ satisfies \mathcal{P} .
 2. For each $K \in \mathcal{A}$ that belongs to an acyclic portion or to the frontier of some cycle in the abstract enriched flow graph of \mathcal{A} , K is \mathcal{P} -compatible with any AEI in $\{C \in \mathcal{B}_K \mid C \notin \mathcal{CB}_K\}$.
 3. If \mathcal{A} is cyclic, there exists a total cycle covering strategy σ such that:
 - (a) if \mathcal{CB}_σ has a single cyclic border $\{C_1, \dots, C_n\}$ and $\mathcal{F}_{C_1, \dots, C_n} = \emptyset$, then there exists $C_j \in \{C_1, \dots, C_n\}$ that \mathcal{P} -interoperates with the rest of the cyclic border;
 - (b) otherwise, for each cyclic border $\{C_1, \dots, C_n\} \in \mathcal{CB}_\sigma$, any $C_j \in \mathcal{F}_{C_1, \dots, C_n}$ \mathcal{P} -interoperates with the rest of the cyclic border.

Then $\llbracket \mathcal{A} \rrbracket_{\text{bbm}}^c$ satisfies \mathcal{P} .

- **Example:** applet-based simulator for the cruise control system.
- The applet has seven software buttons and a text area.
- The seven software buttons correspond to turning the engine on/off, pressing accelerator/brake, and pressing on/off/resume.
- The text area shows the sequence of buttons successfully pressed so far.
- Each of the seven buttons can be pressed at any time.
- If pressing one of them succeeds, the applet can interact with the sensor and the text area is updated accordingly.
- If it fails, the applet cannot interact with the sensor and emits a beep (simply due to a delay within the simulator or, e.g., pressing the brake when the engine is off).
- Objective: design the system so that deadlock freedom is ensured.

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

```
ARCHI_TYPE Cruise_Control_Simulator(void)
```

- Definition of the applet AET:

```
ARCHI_ELEM_TYPE Applet_Type(void)
```

```
BEHAVIOR
```

```
Unallocated(void; void) =  
    create_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()  
    };
```


- Definition of the cruise control system AET:

```
ARCHI_ELEM_TYPE Cruise_Control_System_Type(void)
```

```
BEHAVIOR
```

```
Cruise_Control_System(void; void) =  
    Cruise_Control(@      /* no data parameters */  
                  @@@@   /* reuse formal AETs and 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  UNI engine_on; engine_off;  
                    accelerator; brake; on; off; resume
```

```
OUTPUT_INTERACTIONS void
```

- Declaration of the topology:

ARCHI_ELEM_INSTANCES

P : Applet_Type();

CCS : Cruise_Control_System_Type()

ARCHI_INTERACTIONS

P.create_applet; P.destroy_applet;

P.start_applet; P.stop_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;

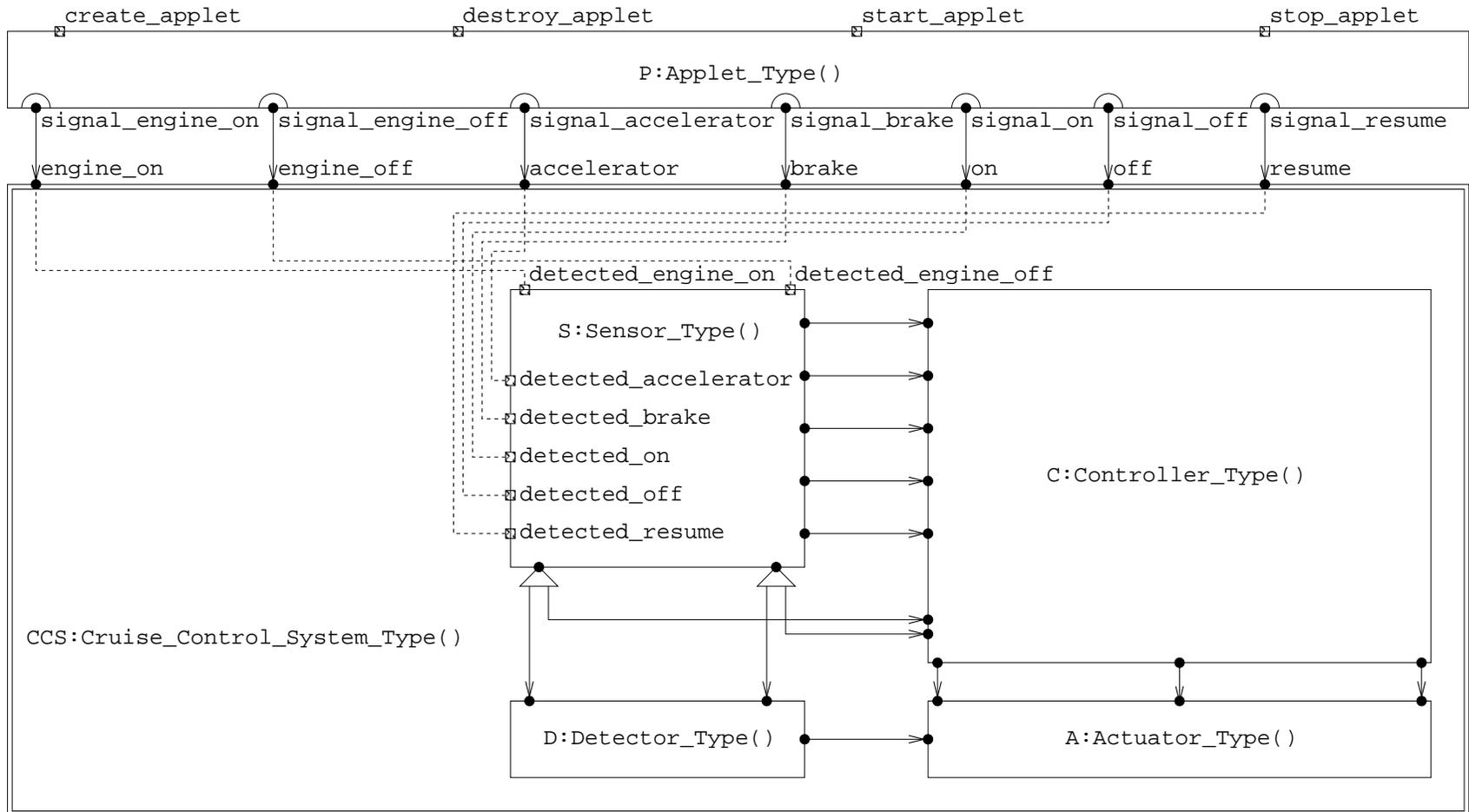
FROM P.signal_brake TO CCS.brake;

FROM P.signal_on TO CCS.on;

FROM P.signal_off TO CCS.off;

FROM P.signal_resume TO CCS.resume

- Enriched flow graph:



- Consider the flattened topology where the architectural interactions provided by **S** have become local.
- **P**, **S**, **C**, **D**, **A** are all deadlock free.
- Intersection of a cycle and a star.
- Cycle formed by **S**, **C**, **D**, **A** within **CCS**.
- **S** deadlock-freedom-interoperates with **C**, **D**, **A**, hence the cycle can be reduced to **S**.
- Only **S** is linked to **P**, hence the frontier of the cycle includes only **S**.
- **P** is deadlock-freedom-compatible with **S** (semi-sync interactions are hidden).
- Then **Cruise_Control_Simulator** is deadlock free.

Generalization to Architectural Types

- Deduce information about the absence of architectural mismatches for an entire architectural type from one of its instances.
- Four cases:
 - ⊙ internal behavior variations;
 - ⊙ exogenous variations;
 - ⊙ endogenous variations;
 - ⊙ and-/or-variations.

- First case: internal behavior variations.
- Let \mathcal{A} be an architectural type such that conditions 1, 2, 3 hold. If \approx_B is contained in $\approx_{\mathcal{P}}$, then all the behaviorally conformant instances of \mathcal{A} satisfy \mathcal{P} .

- Second case: exogenous variations.
- The architectural interactions at which an exogenous variation takes place become local, hence they must be kept visible in the appropriate \mathcal{P} -compatibility and \mathcal{P} -interoperability checks.
- An exogenous variation may change some sets of cyclic borders for the original topology.
- Scalability is not possible whenever an exogenous variation generates kinds of cycles that are not present in the original topology.

- Let $\{C_1, \dots, C_n\}$ be a set of AEs.
- Let $\mathcal{AI}_{C_1}, \dots, \mathcal{AI}_{C_n}$ be the architectural interactions of C_1, \dots, C_n .
- Semi-closed interacting semantics of C_j with respect to $\{C_1, \dots, C_n\}$:

$$\begin{aligned}
\llbracket C_j \rrbracket_{C_1, \dots, C_n}^{\text{sc}} &= \llbracket C_j \rrbracket_{C_1, \dots, C_n} / (\text{Name} - (\mathcal{LI}_{C_j; C_1, \dots, C_n} \cup \mathcal{AI}_{C_j})) \\
&\quad / \{ \text{AIQ}_1.\text{depart}\#C_j.i_1, \dots, \text{AIQ}_h.\text{depart}\#C_j.i_h, \\
&\quad C_j.o_1\#\text{AOQ}_1.\text{arrive}, \dots, C_j.o_k\#\text{AOQ}_k.\text{arrive}, \\
&\quad C_j.i_1\text{-exception}, \dots, C_j.i_h\text{-exception} \}
\end{aligned}$$

- Semi-closed interacting semantics of $\{C'_1, \dots, C'_{n'}\} \subseteq \{C_1, \dots, C_n\}$ with respect to $\{C_1, \dots, C_n\}$:

$$\begin{aligned} \llbracket C'_1, \dots, C'_{n'} \rrbracket_{C_1, \dots, C_n}^{\text{sc}} = & \llbracket C'_1 \rrbracket_{C_1, \dots, C_n}^{\text{sc}} \parallel_{\mathcal{S}(C'_1, C'_2; C_1, \dots, C_n)} \\ & \llbracket C'_2 \rrbracket_{C_1, \dots, C_n}^{\text{sc}} \parallel_{\mathcal{S}(C'_1, C'_3; C_1, \dots, C_n) \cup \mathcal{S}(C'_2, C'_3; C_1, \dots, C_n)} \\ & \dots \parallel_{\bigcup_{i=1}^{n'-1} \mathcal{S}(C'_i, C'_{n'}; C_1, \dots, C_n)} \llbracket C'_{n'} \rrbracket_{C_1, \dots, C_n}^{\text{sc}} \end{aligned}$$

- Semi-closed semantics of $\mathcal{A} = \{C_1, \dots, C_n\}$ before behav. modifications:

$$\llbracket \mathcal{A} \rrbracket_{\text{bbm}}^{\text{sc}} = \llbracket C_1, \dots, C_n \rrbracket_{C_1, \dots, C_n}^{\text{sc}}$$

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

$$(\llbracket K \rrbracket_{K, \mathcal{B}_K}^{\text{sc}} \parallel_{\mathcal{S}(K, C_j; K, \mathcal{B}_K)} \llbracket C_j \rrbracket_{K, \mathcal{B}_K}^c) / H_j \approx_{\mathcal{P}} \llbracket K \rrbracket_{K, \mathcal{B}_K}^{\text{sc}} / H_j$$

- C_j belonging to a cycle \mathcal{P}^{sc} -interoperates with the other AEs in the cycle iff:

$$\llbracket C_1, \dots, C_n \rrbracket_{\mathcal{A}}^{\text{sc}} / (\text{Name} - (\mathcal{S}(C_j; \mathcal{A}) \cup \mathcal{AI}_{C_j})) / H_j \approx_{\mathcal{P}} \llbracket C_j \rrbracket_{\mathcal{A}}^{\text{sc}} / H_j$$

- \mathcal{P}^{sc} -compatibility and \mathcal{P}^{sc} -interoperability imply \mathcal{P} -compatibility and \mathcal{P} -interoperability, respectively, after hiding architectural interactions and exploiting the congruence property of $\approx_{\mathcal{P}}$.
- Semi-closed frontier of $\{C_1, \dots, C_n\}$:

$$\mathcal{F}_{C_1, \dots, C_n}^{\text{sc}} = \{C_j \in \{C_1, \dots, C_n\} \mid \mathcal{LI}_{C_j; C_1, \dots, C_n} \neq \mathcal{LI}_{C_j} \vee \mathcal{AI}_{C_j} \neq \emptyset\}$$

- Let σ be a total cycle covering strategy for \mathcal{A} and let \mathcal{A}' be an exogenous variation of \mathcal{A} .
- The exogenous variation of σ for \mathcal{A}' is defined by the following algorithm:
 1. All the AEIs of \mathcal{A}' are initially unmarked.
 2. For each $\mathcal{CB}_K^{\mathcal{A}} \in \mathcal{CB}_\sigma^{\mathcal{A}}$, pick out K and mark all the AEIs in $\mathcal{CB}_K^{\mathcal{A}'}$.
 3. While there is an unmarked additional AEI C in the cycles of \mathcal{A}' such that there exists $\mathcal{CB}_{C'}^{\mathcal{A}} \in \mathcal{CB}_\sigma^{\mathcal{A}}$ with $C' = \text{corr}(C)$ and $\mathcal{CB}_C^{\mathcal{A}'}$ topologically conforming to $\mathcal{CB}_{C'}^{\mathcal{A}}$:
 - (a) Pick out C .
 - (b) Mark all the AEIs in $\mathcal{CB}_C^{\mathcal{A}'}$.
- \mathcal{A}' is exo-coverable by σ iff all the AEIs in the cycles of \mathcal{A}' are marked, the exogenous variation of σ is total, and for each $K \in \mathcal{A}$ such that $\mathcal{CB}_K^{\mathcal{A}} \in \mathcal{CB}_\sigma^{\mathcal{A}}$ it holds $\mathcal{CB}_K^{\mathcal{A}} = \mathcal{CB}_K^{\mathcal{A}'}$.

- Let \mathcal{A} be an architectural type such that conditions 1, 2, 3 hold and let \mathcal{A}' be an exogenous variation of \mathcal{A} such that the following additional conditions hold:

4.^{ex} \mathcal{A}' is exo-coverable by σ (cycle covering strategy of condition 3).

5.^{ex} For each $K \in \mathcal{A}$ of the same type as an AEI having architectural interactions at which the exogenous variation takes place, K is \mathcal{P}^{sc} -compatible with any AEI in $\{C \in \mathcal{B}_K \mid C \notin \mathcal{CB}_K^{\mathcal{A}}\}$.

6.^{ex} If \mathcal{A} is cyclic, then for each cyclic border $\{C_1, \dots, C_n\} \in \mathcal{CB}_\sigma^{\mathcal{A}}$, any $C_j \in \mathcal{F}_{C_1, \dots, C_n}^{\text{sc}}$ that is of the same type as an AEI having architectural interactions at which the exogenous variation takes place \mathcal{P}^{sc} -interoperates with the rest of the cyclic border.

Then $[[\mathcal{A}']]_{\text{bbm}}^c$ satisfies \mathcal{P} .

- Third case: endogenous variations.
- An endogenous variation may change some sets of cyclic borders for the original topology.
- Scalability is not possible whenever an endogenous variation generates kinds of attachments that are not present in the original topology.
- Scalability is not possible whenever an endogenous variation generates kinds of cycles that are not present in the original topology.

- Let σ be a total cycle covering strategy for \mathcal{A} and let \mathcal{A}' be an endogenous variation of \mathcal{A} .
- The endogenous variation of σ for \mathcal{A}' is defined by the following algorithm:
 1. All the AEIs of \mathcal{A}' are initially unmarked.
 2. For each $CB_K^A \in CB_\sigma^A$, pick out K and mark all the AEIs in $CB_K^{A'}$.
 3. While there is an unmarked additional AEI C in the cycles of \mathcal{A}' such that there exists $CB_{C'}^A \in CB_\sigma^A$ with C' of the same type as C and $CB_C^{A'}$ topologically conforming to $CB_{C'}^A$:
 - (a) Pick out C .
 - (b) Mark all the AEIs in $CB_C^{A'}$.
- \mathcal{A}' is endo-coverable by σ iff all the AEIs in the cycles of \mathcal{A}' are marked, the endogenous variation of σ is total, and for each $K \in \mathcal{A}$ such that $CB_K^A \in CB_\sigma^A$ it holds $CB_K^A = CB_K^{A'}$ up to the additional AEIs.

- Let \mathcal{A} be an architectural type such that conditions 1, 2, 3 hold and let \mathcal{A}' be an endogenous variation of \mathcal{A} such that the following additional conditions hold:

4.^{en} \mathcal{A}' is endo-coverable by σ (cycle covering strategy of condition 3).

5.^{en} For each attachment in \mathcal{A}' from an AEI K_1 to another AEI K_2 , there exists an attachment in \mathcal{A} from an AEI of the same type as K_1 to another AEI of the same type as K_2 .

6.^{en} For each $CB_K^{\mathcal{A}} \in CB_{\sigma}^{\mathcal{A}}$ it holds $[[CB_K^{\mathcal{A}'}]]_{CB_K^{\mathcal{A}'}}^c / \mathcal{LI} \approx_{\mathcal{P}} [[CB_K^{\mathcal{A}}]]_{CB_K^{\mathcal{A}}}^c$, where \mathcal{LI} is the set of local interactions of the additional AEIs.

Then $[[\mathcal{A}']]_{\text{bbm}}^c$ satisfies \mathcal{P} .

- Fourth case: and-/or-variations.
- No need to introduce a notion of and/or-variation for a cycle covering strategy as the sets of cyclic borders cannot change.
- Scalability is not possible whenever an and-/or-variation generates kinds of cycles that are not present in the original topology.

- Let \mathcal{A} be an architectural type such that conditions 1, 2, 3 hold and let \mathcal{A}' be an and-/or-variation of \mathcal{A} such that the following additional conditions hold:

4.^{ao} Each or-interaction involved in the variation is enabled infinitely often.

5.^{ao} No additional AEI belongs to a cycle in \mathcal{A}' .

Then $[[\mathcal{A}']]_{\text{bbm}}^c$ satisfies \mathcal{P} .

Comparison with Related Techniques

- Focus on classes of properties rather than single properties in order to gain generality.
- Use of property-specific behavioral equivalences in place of general ones.
- Exploiting the hiding operator instead of specifying ports/roles thus lightening the modeling task.
- Generalization w.r.t. topological formats considered in the past:
 - Two software units projected onto a pair of attached port/role.
 - Two entire software units communicating with each other through arbitrarily many interactions.
 - Set of software units forming a star or a cycle.
- Scalability from a single architecture to an entire architectural type.

Part III:
Component-Oriented Performance Evaluation

Performance Aspects

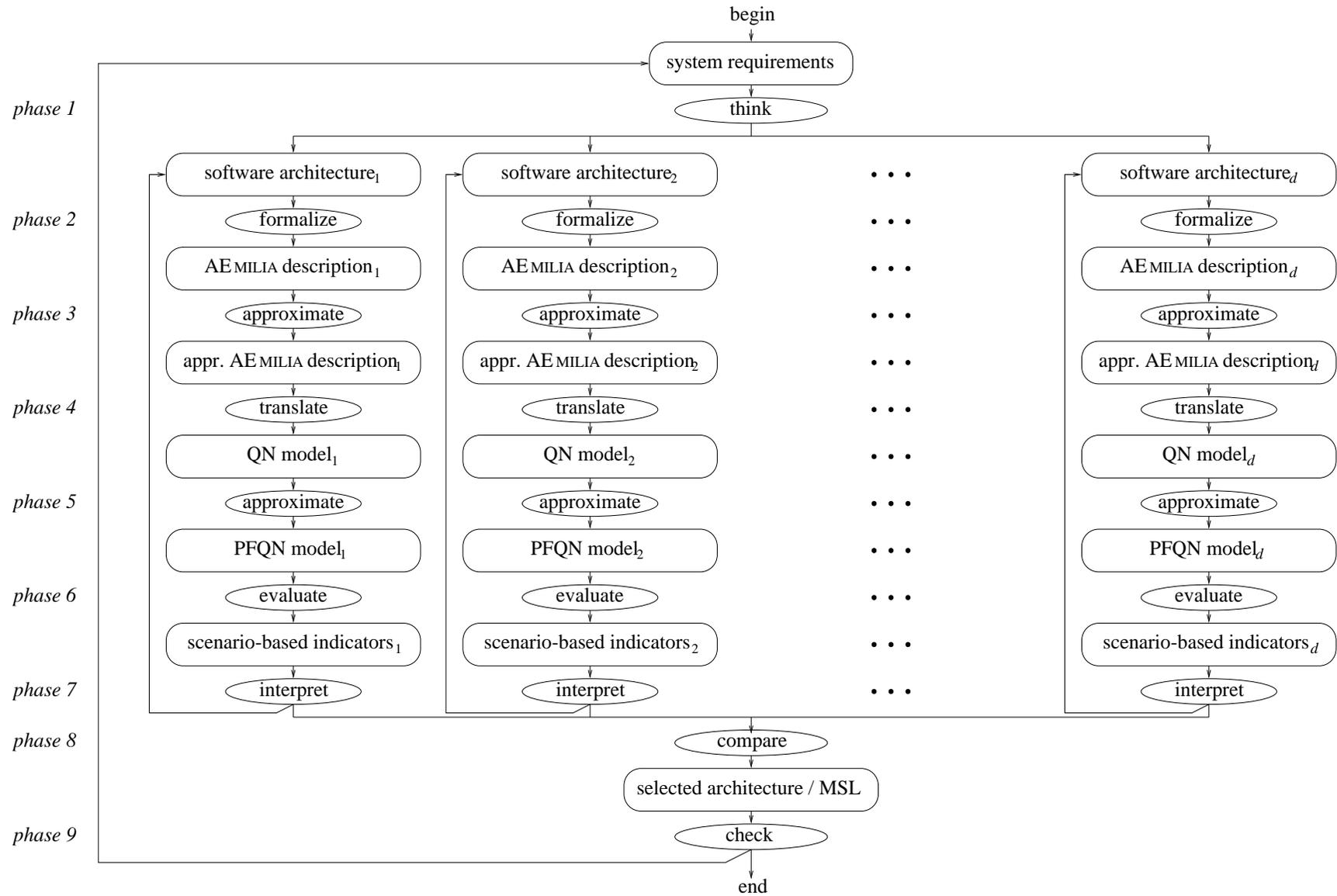
- Integrated view of functional and **non-functional aspects** in the software development process.
- △ *How to choose among a number of alternative architectural designs (that are functionally correct)?*
- △ *How to choose among a number of alternative components off-the-shelf (that provide a given 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.
- Combining 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).

Performance-Aware Methodology

- Multi-phase methodology for an arbitrary number of alternative designs.
- Focus on typical average indicators that can be efficiently computed and give insights in the overall system 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 generic indicators.

- **Throughput**: productivity of system components.
- Useful to single out those components that are bottlenecks and hence must be redesigned.
- **Utilization**: relative usage of computational resources by components.
- Useful at deployment time for a balanced distribution of the workload among computational resources.
- **Mean queue length**: average size of data repositories.
- Useful to avoid component execution blocking due to under-sized buffers as well as waste of memory due to over-sized buffers.
- **Mean response time**: average running time of components.
- Useful to predict the quality of service that will be perceived by system users on average.



ÆMILIA: Performance Modeling

- ÆMILIA is a performance-aware variant of PADL.
- Each of its actions is composed of a name and a **duration**.
- Actions are divided into **exponentially timed**, **immediate**, and **passive**.
- Semantics by translation into extended Markovian process algebra with generative-reactive synchronizations (EMPA_{gr}).
- Labeled multitransition 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 $\{X(t) \mid t \in \mathbb{R}_{\geq 0}\}$ such that for all $n \in \mathbb{N}$, time instants $t_0 < t_1 < \dots < t_n < t_{n+1}$, and states $s_0, s_1, \dots, s_n, s_{n+1} \in S$:

$$\Pr\{X(t_{n+1}) = s_{n+1} \mid X(t_0) = s_0 \wedge X(t_1) = s_1 \wedge \dots \wedge X(t_n) = s_n\} = \Pr\{X(t_{n+1}) = s_{n+1} \mid X(t_n) = s_n\}$$

- 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** $\pi()$ at an arbitrary time instant.

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

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

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

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

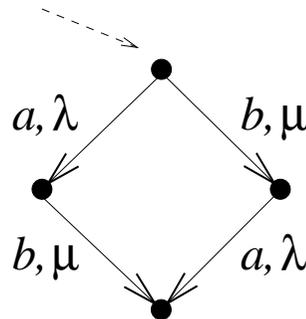
- Exponentially distributed random variables are the only continuous random variables satisfying the **memoryless property**:

$$\Pr\{X \leq t + t' \mid X > t'\} = \Pr\{X \leq t\}$$

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

- Classical [interleaving semantics](#) for concurrent exp. 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 prob. of simultaneous termination of two concurrent exponentially timed actions is zero.
- Labeled (multi)transition system of $\langle a, \lambda \rangle . \underline{0} \parallel_{\emptyset} \langle b, \mu \rangle . \underline{0}$:



- Interleave concurrent exp. timed actions without adjusting their rates inside transition labels.

- Apply the **race policy** if several exponentially timed actions with rates $\lambda_1, \dots, \lambda_h$ 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:

$$\min(X_{\lambda_1}, \dots, X_{\lambda_h}) = X_{\lambda_1 + \dots + \lambda_h}$$

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

- An **immediate action** $\langle a, \infty_{l,w} \rangle$ takes place at an unbounded rate.
- The duration of $\langle a, \infty_{l,w} \rangle$ 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, \dots, w_h .
- The execution probability of immediate action with weight w_i is $w_i / (w_1 + \dots + w_h)$.

- A **passive action** $\langle a, *_{l,w} \rangle$ takes place at an unspecified rate.
- The duration of $\langle a, *_{l,w} \rangle$ 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 level $l \in \mathbb{N}_{>0}$ 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 level.
- Assume that their weights are w_1, \dots, w_h .
- The execution probability of passive action with weight w_i is $w_i / (w_1 + \dots + w_h)$.

- **Generative-reactive synchronizations:** in EMPA_{gr} exponentially timed and immediate actions can synchronize only with passive actions.
- Among all the enabled non-passive 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 that have the same name as the proposed one *react* by performing a selection based on their priorities and weights.
- The non-passive action winning the generative selection and the passive action winning the reactive selection synchronize with each other.

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

$$\begin{array}{c}
 \frac{P_1 \xrightarrow{a, \lambda} P'_1 \quad P_2 \xrightarrow{a, *l, w} P'_2 \quad a \in S}{P_1 \parallel_S P_2 \xrightarrow{a, \lambda \cdot \frac{w}{\text{weight}(P_2, a, l)}} P'_1 \parallel_S P'_2} \quad (\text{plus symmetrical rule}) \\
 \\
 \frac{P_1 \xrightarrow{a, \infty l', w'} P'_1 \quad P_2 \xrightarrow{a, *l, w} P'_2 \quad a \in S}{P_1 \parallel_S P_2 \xrightarrow{a, \infty l', w' \cdot \frac{w}{\text{weight}(P_2, a, l)}} P'_1 \parallel_S P'_2} \quad (\text{plus symmetrical rule}) \\
 \\
 \frac{P_1 \xrightarrow{a, *l, w_1} P'_1 \quad P_2 \xrightarrow{a, *l, w_2} P'_2 \quad a \in S}{P_1 \parallel_S P_2 \xrightarrow{a, *l, \frac{w_1}{\text{weight}(P_1, a, l)} \cdot \frac{w_2}{\text{weight}(P_2, a, l)} \cdot (\text{weight}(P_1, a, l) + \text{weight}(P_2, a, l))} P'_1 \parallel_S P'_2}
 \end{array}$$

- Weight of a process term P with respect to passive actions of name a :

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

- 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).
 - ⊙ Rates/priorities/weights can be parameterized (AT and AET headers).
 - ⊙ Every set of local interactions attached to each other can contain at most one non-passive local interaction.
 - ⊙ The occurrences of an action name within an AET must be all exponentially timed, all immediate, or all passive.

- **Example:** performance-aware variant of the pipe-filter AT.
- Item transformation is the only system activity that introduces a non-negligible delay.
- 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 downstream forward probabilities.

- Textual description header:

```
ARCHI_TYPE Perf_Pipe_Filter(const integer ppf_ds_filter_num := 3,  
                             const integer ppf_buffer_size   := 10,  
                             const integer ppf_init_item_num  := 0,  
                             const rate    ppf_transf_rate_0  := 60,  
                             const weight  ppf_forward_prob_0 := 1,  
                             const array(ppf_ds_filter_num, rate)  
                               ppf_transf_rate := array_cons(70, 45, 30),  
                             const array(ppf_ds_filter_num, weight)  
                               ppf_forward_prob := array_cons(0.5, 0.3, 0.2))
```

- Definition of the filter buffer AET:

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

BEHAVIOR

```
Filter_Buffer(integer(0..buffer_size) item_num := init_item_num;  
              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  UNI input_item
```

```
OUTPUT_INTERACTIONS UNI pass_item
```

- Definition of the filter core AET:

```
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 UNI select_item
```

```
OUTPUT_INTERACTIONS UNI output_item
```

- Definition of the pipe AET:

ARCHI_ELEM_TYPE Pipe_Type(void)

BEHAVIOR

Pipe(void; void) =

<accept_item, _> . <forward_item, inf> . Pipe()

INPUT_INTERACTIONS UNI accept_item

OUTPUT_INTERACTIONS OR forward_item

- Declaration of the topology:

ARCHI_ELEM_INSTANCES

```
FB[0] : Filter_Buffer_Type(ppf_buffer_size, ppf_init_item_num, ppf_forward_prob_0);
FC[0] : Filter_Core_Type(ppf_transf_rate_0);
P      : Pipe_Type();
FOR_ALL 1 <= j <= ppf_ds_filter_num
    FB[j] : Filter_Buffer_Type(ppf_buffer_size, ppf_init_item_num, ppf_forward_prob[j]);
FOR_ALL 1 <= j <= ppf_ds_filter_num
    FC[j] : Filter_Core_Type(ppf_transf_rate[j])
```

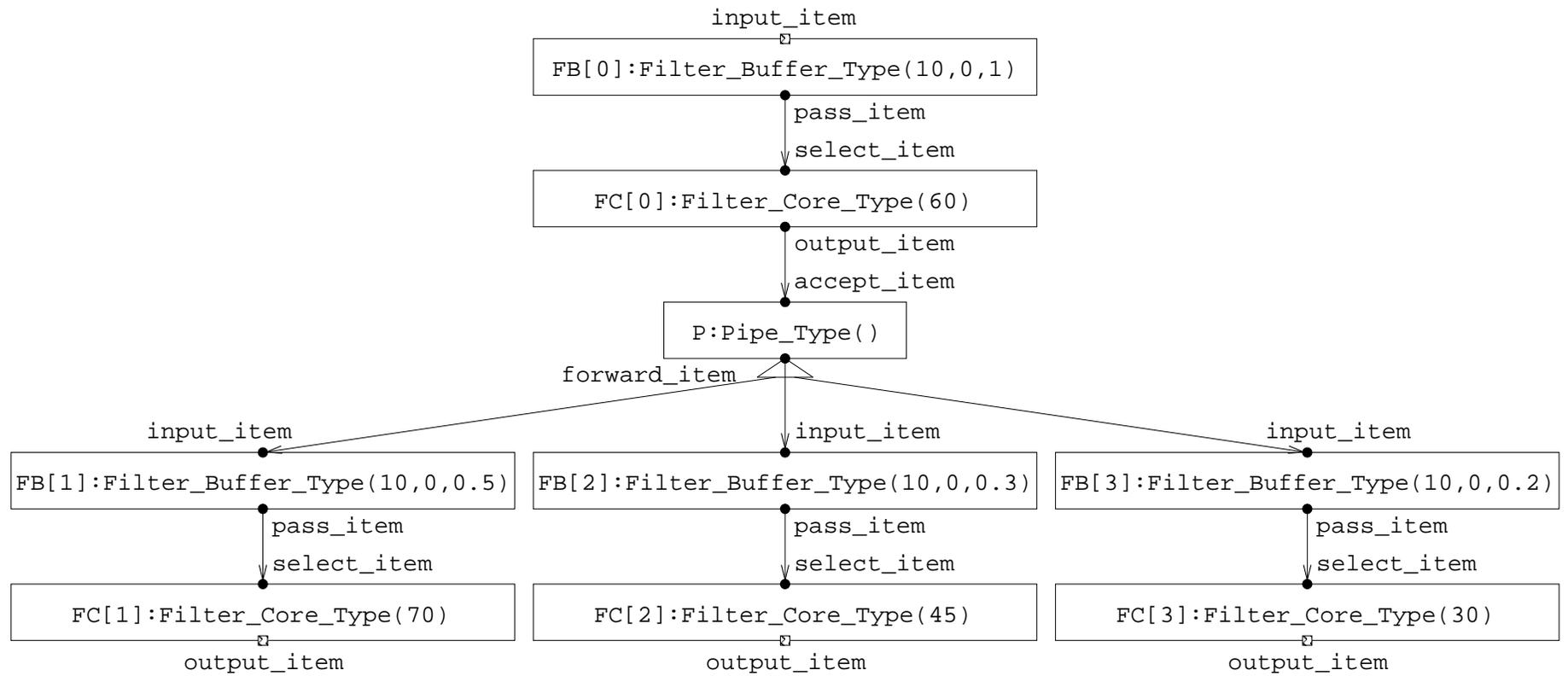
ARCHI_INTERACTIONS

```
FB[0].input_item;
FOR_ALL 1 <= j <= ppf_ds_filter_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 <= ppf_ds_filter_num
    FROM P.forward_item TO FB[j].input_item;
FOR_ALL 1 <= j <= ppf_ds_filter_num
    FROM FB[j].pass_item TO FC[j].select_item
```

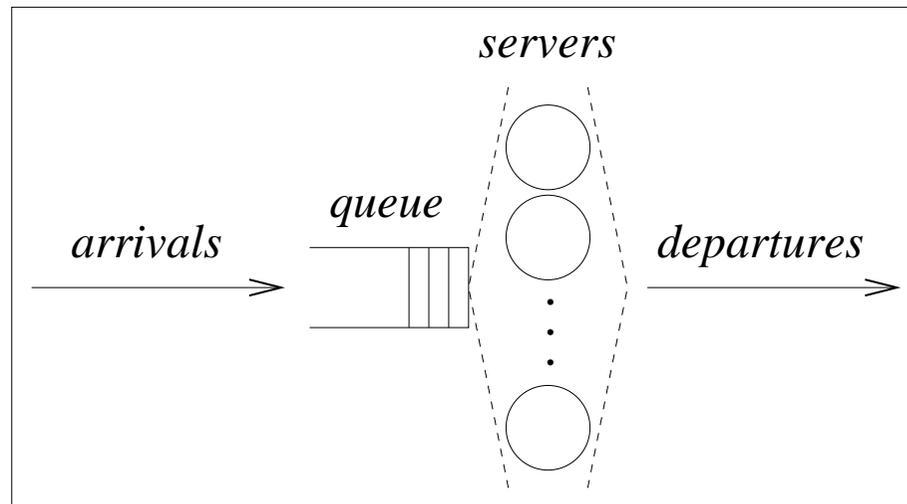
- Enriched flow graph:



PFQN: Performance Analysis

- CTMC models derivable from \mathcal{A} EMILIA descriptions are state based, hence they are not suited for the architectural design level.
- Better resorting to QN models, which are collections of interacting service centers that represent resources shared by classes of customers.
- The customer competition for resources corresponds to queueing into the service centers.
- QNs are **structured performance models**:
 - ⊙ System components and their connectivity are elucidated.
 - ⊙ Typical average performance indicators for both system level and 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 consists of a source of arrivals, a queue, and a set of servers:



- Single-class/multi-class depending on the number of classes of customers
(different arrival processes and service demands).
- A QS $A/B/m/c/p$ is characterized by the following 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 (default value FCFS).

- First come first served (**FCFS**): customers are served in the order of their arrival.
- 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 the customers that 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.
- Non-preemptive 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 its service demand is satisfied; the waiting customers are served in the order in which they last entered the queue.
- Processor sharing (**PS**): customers receive service simultaneously with equal shares of the service rate.
- Infinite server (**IS**): no queueing takes place as each arriving customer always finds an available server.

- 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).
- Steady-state analysis requires that the customer arrival rate is 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 , service rate μ , and FCFS queueing discipline is geometrically distributed with parameter $\rho_1 = \lambda/\mu < 1$ (same with 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:

$$\bar{T}_1 = \mu \cdot \Pr\{N_1 > 0\} = \mu \cdot \rho_1 = \lambda$$

- Utilization of M/M/1:

$$\bar{U}_1 = \Pr\{N_1 > 0\} = \rho_1$$

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

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

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

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

- Stability condition for M/M/m:

$$\rho_m = \lambda / (m \cdot \mu) < 1$$

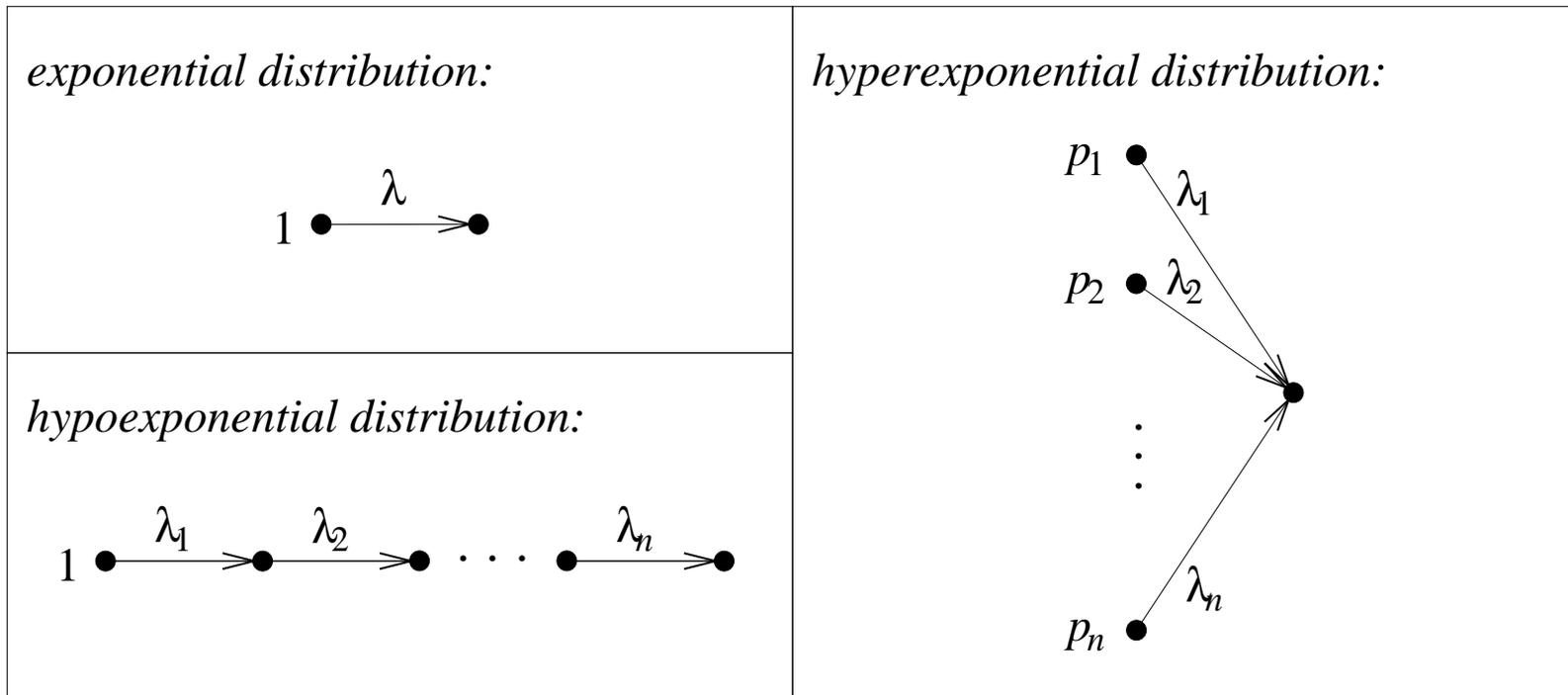
- Average performance indicators for M/M/m:

$$\begin{aligned} \bar{T}_m &= \sum_{k=1}^{m-1} k \cdot \mu \cdot \Pr\{N_m = k\} + m \cdot \mu \cdot \Pr\{N_m \geq m\} = \lambda \\ \bar{U}_m &= 1 - \Pr\{N_m = 0\} = 1 - \left(\sum_{k=0}^{m-1} \frac{(m \cdot \rho_m)^k}{k!} + \frac{(m \cdot \rho_m)^m}{m! \cdot (1 - \rho_m)} \right)^{-1} \\ \bar{N}_m &= \sum_{k=0}^{\infty} k \cdot \Pr\{N_m = k\} = m \cdot \rho_m + \frac{\Pr\{N_m=0\} \cdot \rho_m \cdot (m \cdot \rho_m)^m}{m! \cdot (1 - \rho_m)^2} \\ \bar{R}_m &= \bar{N}_m / \lambda = \frac{1}{\mu} \cdot \left(1 + \frac{\Pr\{N_m=0\} \cdot \rho_m \cdot (m \cdot \rho_m)^{m-1}}{m! \cdot (1 - \rho_m)^2} \right) \end{aligned}$$

- In general a QN is a network of service centers, which are QNs when considered in isolation.
- Topology described through a matrix of routing probabilities.
- Single-class/multi-class depending on the number of classes of customers (different arrival processes, service demands, and routings).
- Open/closed/mixed depending on whether external arrivals and departures are allowed or not (when closed a fixed number of customers circulate indefinitely among the service centers).
- 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, \dots, k_n) is the product of the probabilities that each service center i contains k_i customers (up to a constant for closed QNs).
- Solve each service center in isolation, then combine their solutions.
- **BCMP theorem**: any open/closed/mixed single-class/multi-class QN with Poisson arrivals having possibly state-dependent rates and with Markovian routing is product form if each of its service centers is characterized by a combination of the following:
 - ⊙ FCFS with the same exponential service time for all the classes of customers;
 - ⊙ LCFS-PR, PS, IS with phase-type service time possibly different for the various classes of customers.

- Under the second condition, the average performance indicators do not change if the phase-type service times are replaced by exponential service times with the same expected values:



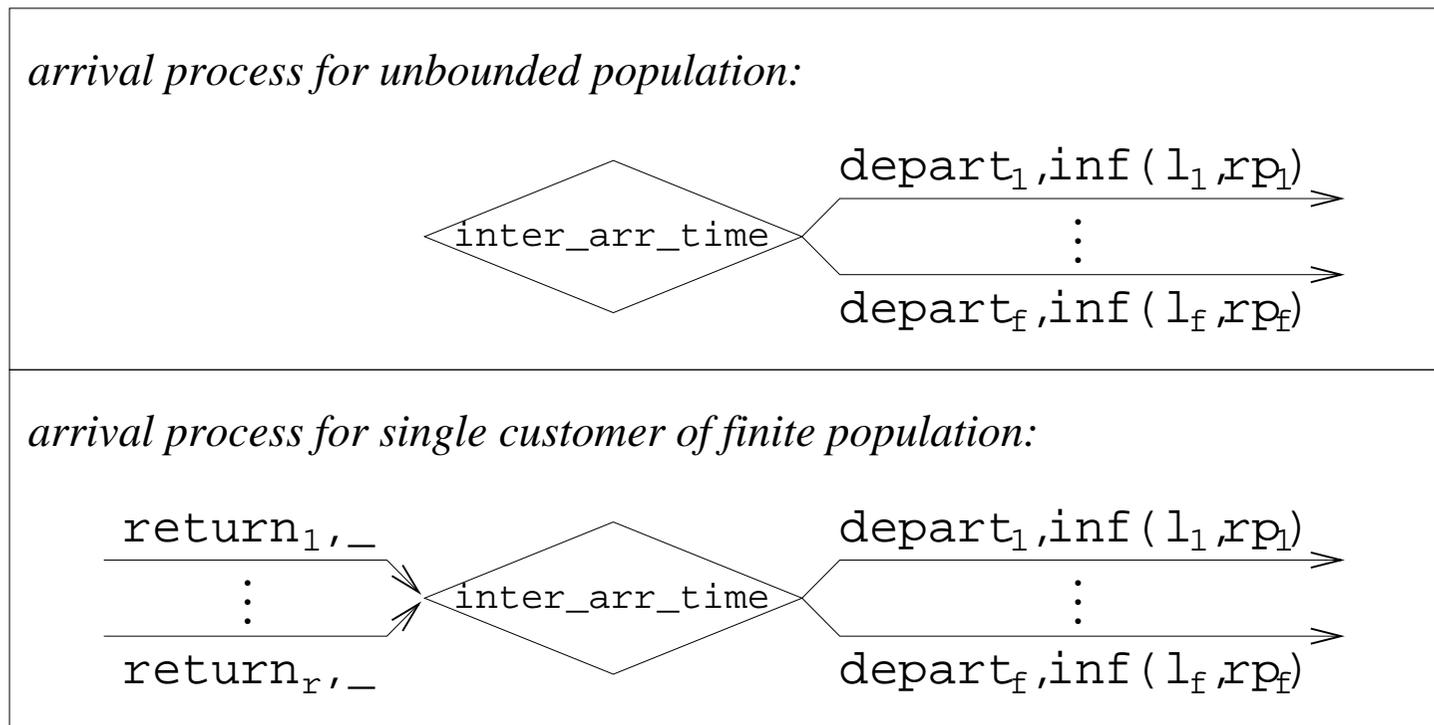
Translating ÆMILIA Descriptions into QN Models

- The performance-aware methodology needs to derive QN models from ÆMILIA descriptions of alternative designs.
- Several differences between the two formalisms:
 - ◉ ÆMILIA is a general-purpose, process algebraic, component-oriented, action-based language.
 - ◉ QNs are special-purpose, resource-oriented, queue-based models with some details expressed in natural language.
- Only some ÆMILIA descriptions will be translatable into QN models, depending on whether they follow a queue-like pattern or not.
- Impose general syntax restrictions that single out a reasonably wide family of ÆMILIA descriptions from which QN models can be derived.

- Then the basic idea is to map each AEI to a QS PH/PH/ m / c / p .
- This does not work in general because AEIs are usually finer than QSs.
- Better try to map groups of AEIs to QSs.
- Equivalently, try to map each AEI to a QN basic element:
 - ⊙ arrival process;
 - ⊙ buffer;
 - ⊙ service process;
 - ⊙ fork process;
 - ⊙ join process;
 - ⊙ routing process.
- Impose specific syntax restrictions to single out those AEIs that can be mapped to QN basic elements.

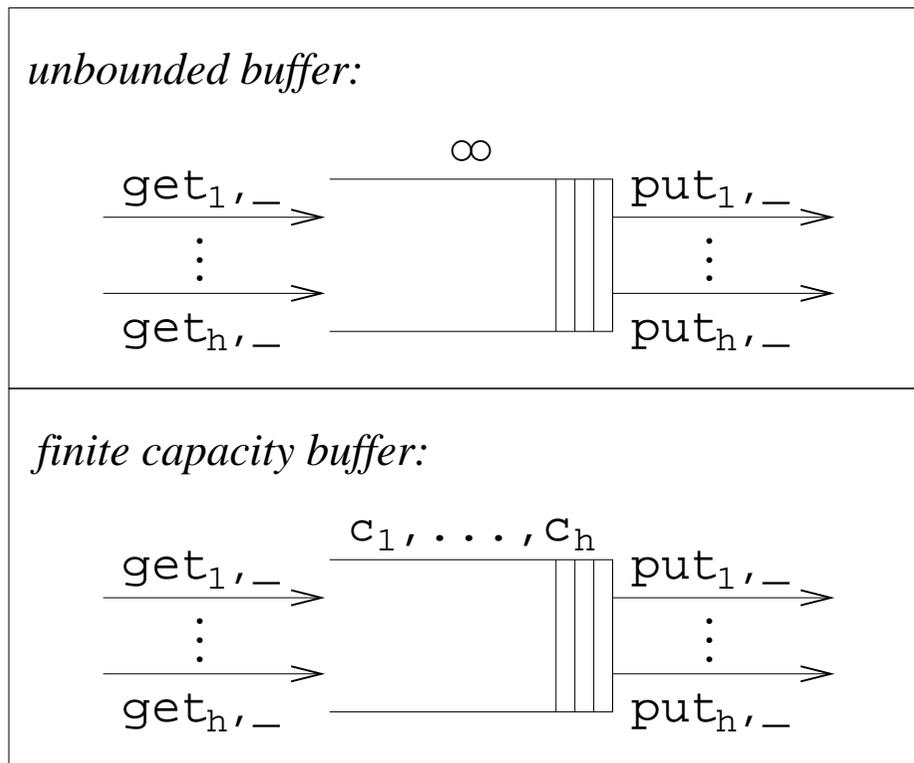
- General syntax restrictions:
 - ⊙ Every AEI must conform to a QN basic element and be suitably connected to the other AEIs in order to yield a well-formed QN.
 - ⊙ Delays associated with arrival or service processes must be confined to internal actions, hence no interaction can be exponentially timed.
 - ⊙ Since arrival and service processes are sequential, there cannot be exponentially timed actions alternative to each other.
 - ⊙ In order to detect the phase-type distributed delays associated with arrival or service processes:
 - * no exponentially timed action can be alternative to an immediate or passive action;
 - * no immediate action can be alternative to a passive action;
 - * no interaction can be alternative to an internal action.

- Arrival process:
 - ⊙ Generator of arrivals of customers of a given class.
 - ⊙ Interarrival times follow a phase-type distribution.
 - ⊙ Two different kinds depending on the customer population size:

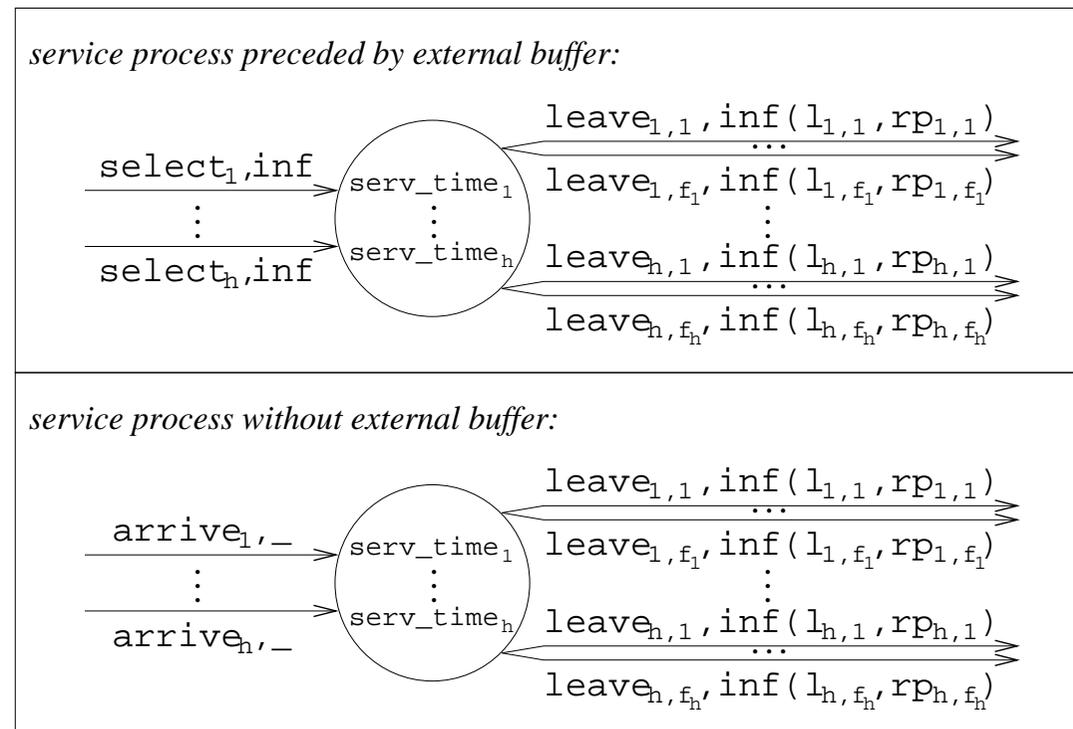


- Buffer:

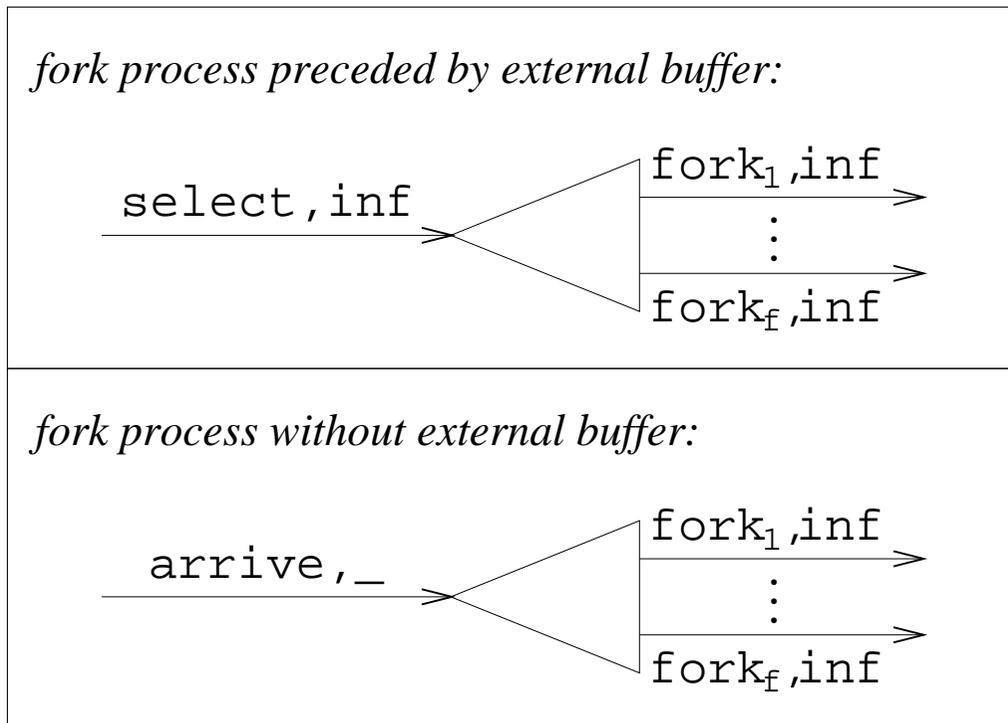
- Repository of waiting customers of h different classes.
- Any noninterruptable queueing discipline (FCFS, LCFS, SIRO, NP).
- Two different kinds depending on the buffer capacity:



- Service process:
 - Server for customers of h 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 (leaving may be absent):

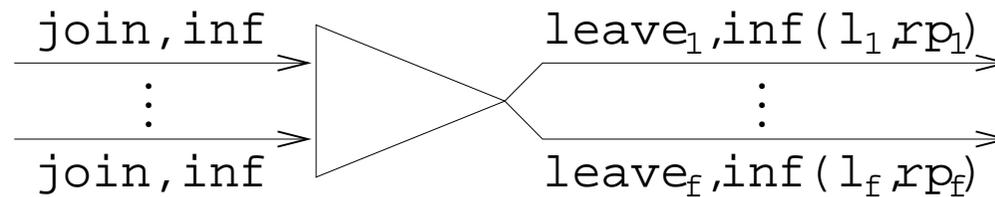


- Fork process:
 - ⊙ Splitter of requests coming from customers of a given class into subrequests directed to different service centers (potential and-interaction).
 - ⊙ Two different kinds depending on the presence of an external buffer where requests can wait before being split:

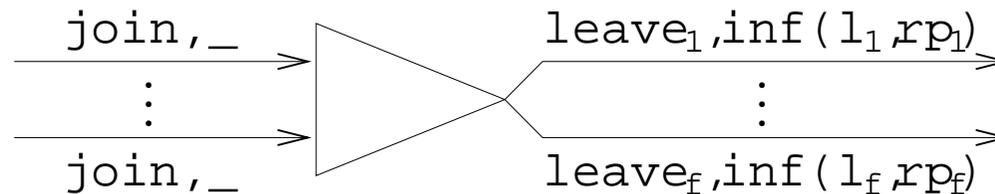


- Join process:
 - ◉ Merger of subrequests coming from customers of a given class after they have been served at different service centers (*and-interaction*).
 - ◉ Two different kinds depending on the presence of external buffers where subrequests can wait before being merged (*leaving may be absent*):

join process preceded by external buffers:



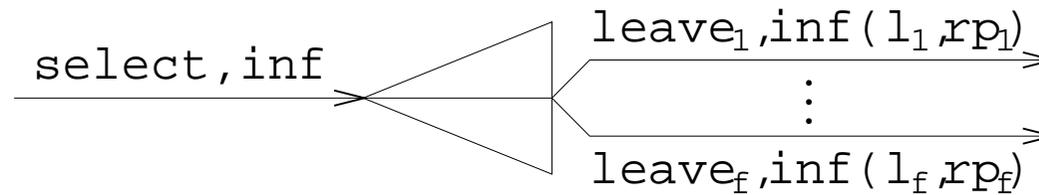
join process without external buffers:



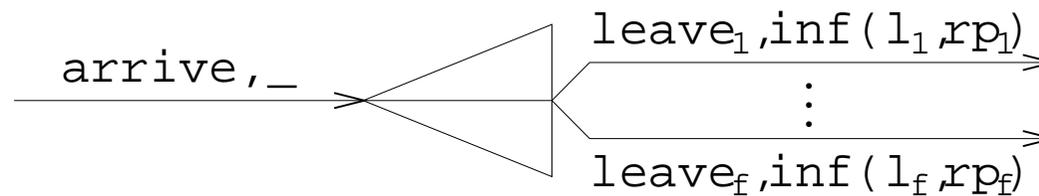
- Routing process:

- ⊙ Router of customers of a given class (potential or-interaction).
- ⊙ Two different kinds depending on the presence of an external buffer where customers can wait before being routed:

routing process preceded by external buffer:



routing process without external buffer:



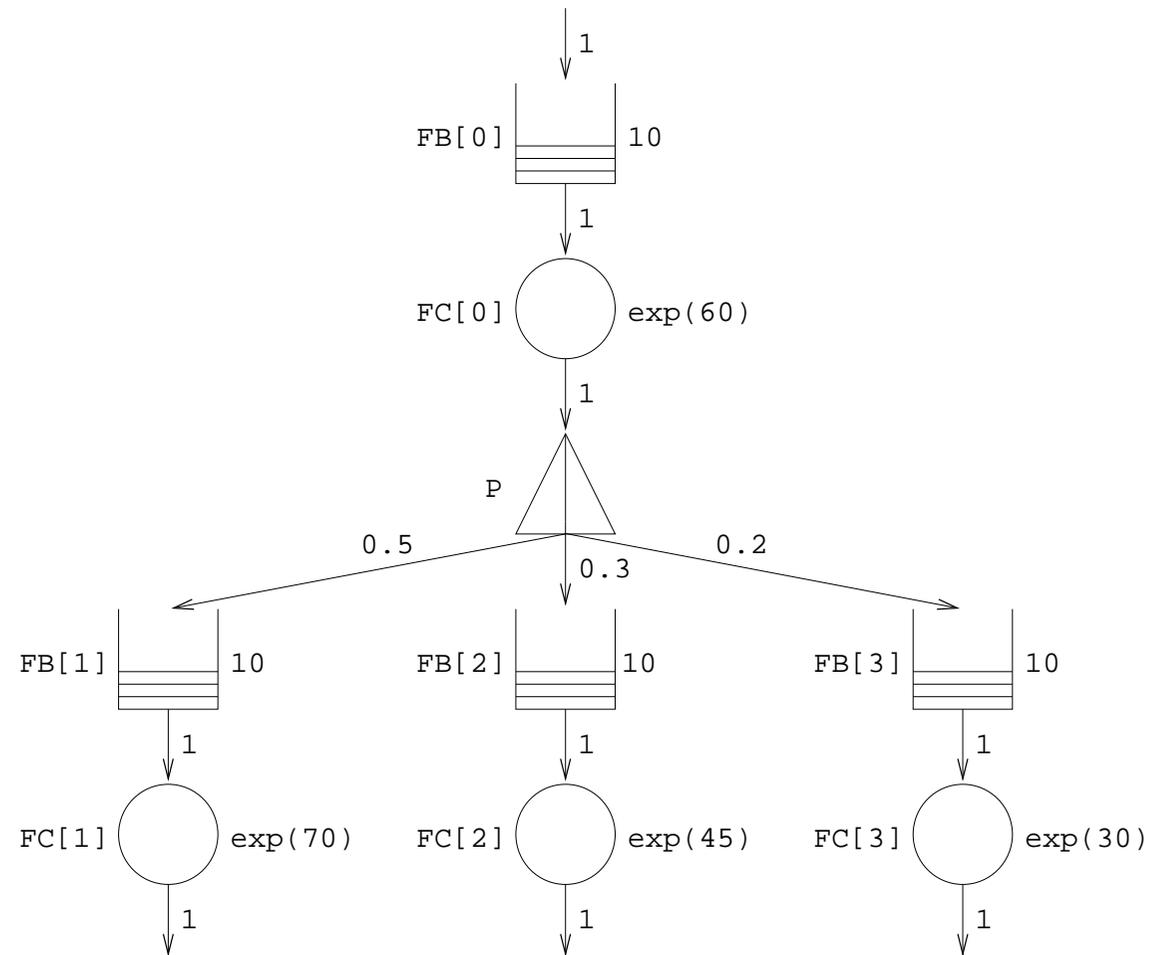
- Every AEI satisfying all syntax restrictions is translated into a QN basic element by applying a set of [documental and charactering functions](#).
- Attachment-driven composition of the obtained QN basic elements:
 - ⊙ An arrival process can only be followed by a service or fork process possibly preceded by a buffer.
 - ⊙ A buffer can only be followed by a service, fork, join, or routing process.
 - ⊙ A service process can be followed by any element.
 - ⊙ A fork process can only be followed by a service process or another fork process possibly preceded by a buffer.
 - ⊙ A join process can be followed by any element.
 - ⊙ A routing process can be followed by any element.
- The complexity of the translation is linear in the number of AEIs.

- Documental functions:
 - ⊙ *qnbe* specifies the kind of QN basic element into which an AEI is translated;
 - ⊙ *name* associates the name of an AEI with the corresponding QN basic element;
 - ⊙ *input* associates the input local interactions of an AEI with the incoming arcs of the corresponding QN basic element;
 - ⊙ *output* associates the output local interactions of an AEI with the outgoing arcs of the corresponding QN basic element.

- Characterizing functions:
 - ⊙ *inter_arr_time* computes the phase-type distribution governing interarrival times of customers of a given class for an AEI translated into an arrival process;
 - ⊙ *capacity* computes the capacity for an AEI translated into a buffer;
 - ⊙ *queueing_disc* computes the queueing discipline for an AEI translated into a buffer;
 - ⊙ *serv_time* computes the phase-type distribution governing service times of customers of a given class for an AEI translated into a service process;
 - ⊙ *routing_prob* computes the routing probabilities of customers of a given class for an AEI translated into an arrival, service, join, or routing process.

- **Example:** QN model for the performance-aware pipe-filter AT.
- All syntax restrictions are satisfied by the $\mathcal{A}EMILIA$ description.
- $FB[0]$, $FB[1]$, $FB[2]$, $FB[3]$ are buffers.
- $FC[0]$, $FC[1]$, $FC[2]$, $FC[3]$ are service processes preceded by buffers.
- P is a routing process without external buffer.
- An additional arrival process is necessary to characterize the workload, so that the value of the typical average performance indicators can then be derived.

- QN model:



Example: Comparing 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, non to be optimized.
- Several alternative architectures:
 - ⊙ sequential;
 - ⊙ pipeline;
 - ⊙ concurrent.
- Objective: choose the best one from the performance viewpoint.

- 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 UNI deliver_prog
```

- Definition of the two-classes program buffer AET:

```
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 UNI get_prog_1; get_prog_2
```

```
OUTPUT_INTERACTIONS UNI put_prog_1; put_prog_2
```

- Definition of the compiler AET:

```
ARCHI_ELEM_TYPE 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

```
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)> . 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)> . Compiler()  
  }
```

INPUT_INTERACTIONS UNI select_prog_1; select_prog_2

OUTPUT_INTERACTIONS UNI 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   : 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

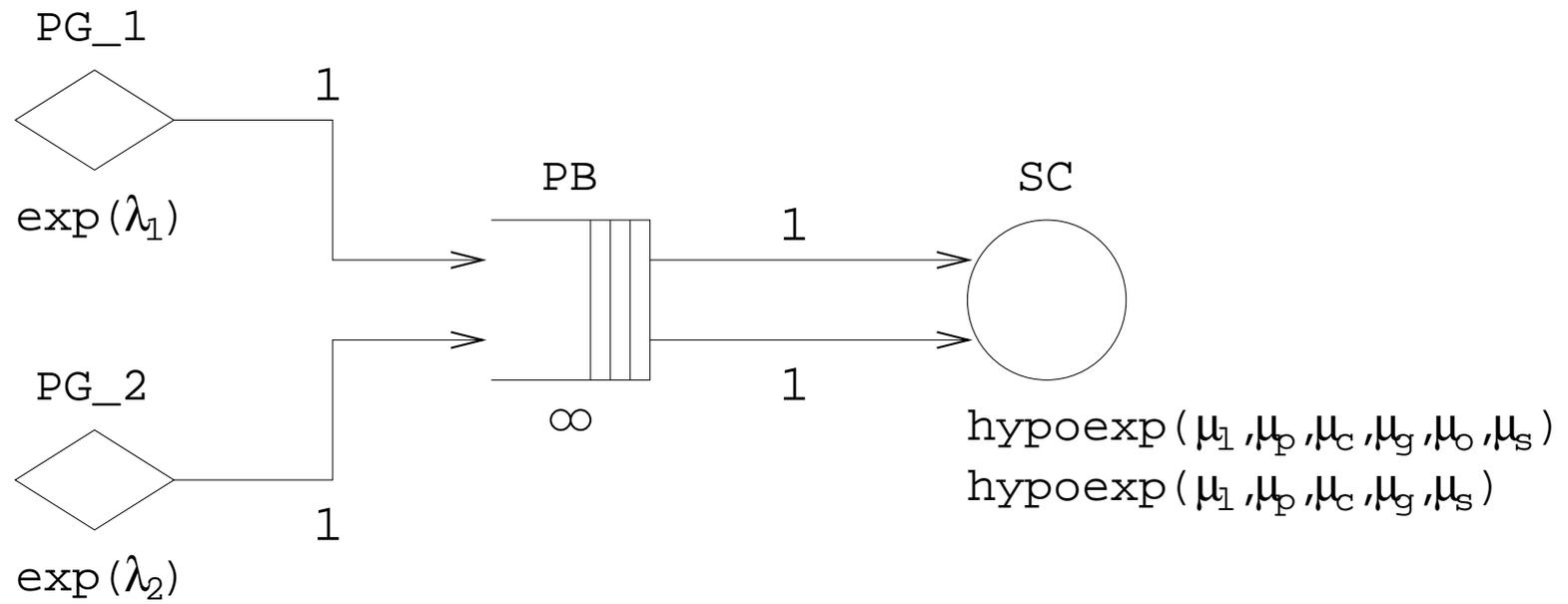
```
FROM PG_1.deliver_prog TO PB.get_prog_1;
```

```
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 (similar to a QS M/M/1):



- In order to derive a (single-class) QS M/M/1 and exploit its formulas, we have to merge the two classes into a single one.
- Aggregated arrival rate:

$$\lambda = \lambda_1 + \lambda_2$$

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

$$\begin{aligned} 1/\mu_1 &= 1/\mu_1 + 1/\mu_p + 1/\mu_c + 1/\mu_g + 1/\mu_o + 1/\mu_s \\ 1/\mu_2 &= 1/\mu_1 + 1/\mu_p + 1/\mu_c + 1/\mu_g + 1/\mu_s \end{aligned}$$

- Convert the aggregated hyperexponential service time into an average-preserving exponential service time:

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

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

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

- Sequential compiler throughput:

$$\bar{T}_{\text{seq}} = \lambda$$

- Sequential compiler utilization:

$$\bar{U}_{\text{seq}} = \rho_{\text{seq}}$$

- Mean number of programs in the sequential compiler system:

$$\bar{N}_{\text{seq}} = \frac{\rho_{\text{seq}}}{1 - \rho_{\text{seq}}}$$

- Mean sequential compilation time:

$$\bar{R}_{\text{seq}} = \frac{1}{\mu \cdot (1 - \rho_{\text{seq}})}$$

- Pipeline compiler: simultaneous compilation of several programs at different phases.
- Phases are decoupled by means of intermediate buffers.
- 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.

- Definition of the one-class program buffer AET:

```
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  UNI get_prog
```

```
OUTPUT_INTERACTIONS 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  UNI select_prog_1; select_prog_2
```

```
OUTPUT_INTERACTIONS 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  UNI select_tokens_1; select_tokens_2
```

```
OUTPUT_INTERACTIONS 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  UNI select_phrases_1; select_phrases_2
```

```
OUTPUT_INTERACTIONS 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  UNI select_cphrases_1; select_cphrases_2
```

```
OUTPUT_INTERACTIONS UNI deliver_icode_1; deliver_icode_2
```

- Definition of the intermediate code optimizer AET:

```
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  UNI select_icode
```

```
OUTPUT_INTERACTIONS 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  UNI select_oicode_1; select_icode_2
```

```
OUTPUT_INTERACTIONS UNI 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_1);
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_0 : Program_Buffer_1C_Type();
O     : Optimizer_Type(pc_mu_o);
PB_S : Program_Buffer_2C_Type();
S     : Synthesizer_Type(pc_mu_s)
```

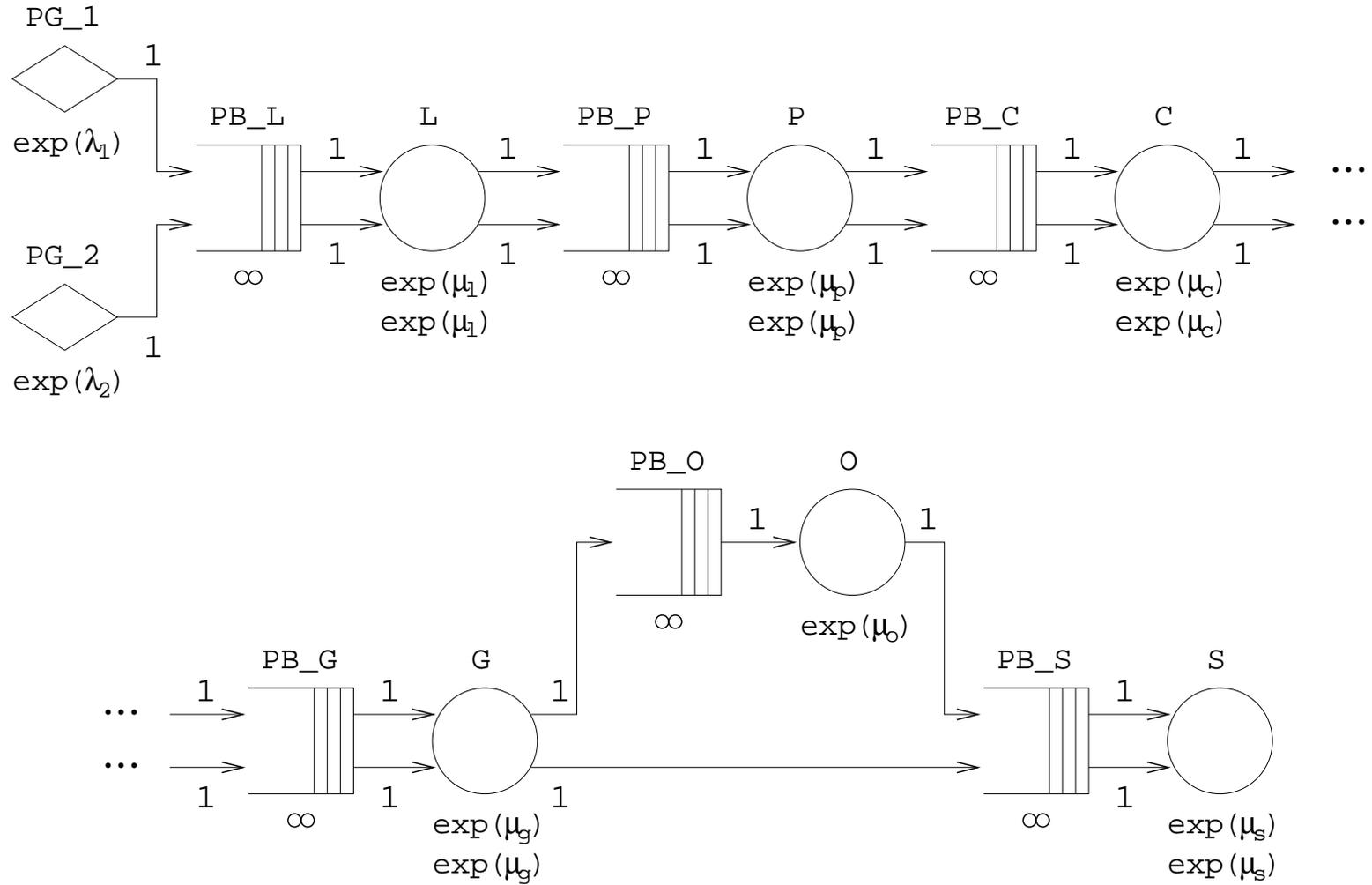
ARCHI_INTERACTIONS

```
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_0.get_prog;
FROM G.deliver_icode_2 TO PB_S.get_prog_2;
FROM PB_0.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 QNs each similar to a QN M/M/1):



- In order to exploit the BCMP theorem and QS M/M/1 formulas, 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:

$$\lambda = \lambda_1 + \lambda_2$$

- 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 QNs:

$$\begin{array}{l} \rho_1 = \lambda/\mu_1 < 1 \\ \rho_p = \lambda/\mu_p < 1 \\ \rho_c = \lambda/\mu_c < 1 \\ \rho_g = \lambda/\mu_g < 1 \\ \rho_o = \lambda_1/\mu_o < 1 \\ \rho_s = \lambda/\mu_s < 1 \end{array}$$

- Overall stability condition for the resulting QN:

$$\lambda < \min(\mu_1, \mu_p, \mu_c, \mu_g, \mu_o \cdot \frac{\lambda}{\lambda_1}, \mu_s)$$

- Throughput of phase i :

$$\begin{aligned}\bar{T}_i &= \lambda & i \in \{1, p, c, g, s\} \\ \bar{T}_o &= \lambda_1\end{aligned}$$

- Utilization of phase i :

$$\bar{U}_i = \rho_i$$

- Mean number of programs in phase i :

$$\bar{N}_i = \frac{\rho_i}{1-\rho_i}$$

- Mean response time of phase i :

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

- Pipeline compiler throughput:

$$\bar{T}_{\text{pipe}} = \bar{T}_s$$

- Pipeline compiler utilization:

$$\bar{U}_{\text{pipe}} = 1 - \prod_i (1 - \bar{U}_i)$$

- Mean number of programs in the pipeline compiler system:

$$\bar{N}_{\text{pipe}} = \sum_i \bar{N}_i$$

- Mean pipeline compilation time:

$$\bar{R}_{\text{pipe}} = \frac{\lambda_1}{\lambda} \cdot \sum_i \bar{R}_i + \frac{\lambda_2}{\lambda} \cdot \sum_{i \neq 0} \bar{R}_i$$

- Concurrent compiler: simultaneous compilation of several programs by as many replicas of the sequential compiler.
- All the replicas share the same buffer.
- Textual description header:

```

ARCHI_TYPE Concurrent_Compiler(const integer cc_compiler_num := 2,
                               const rate   cc_lambda_1     := ...,
                               const rate   cc_lambda_2     := ...,
                               const rate   cc_mu_1         := ...,
                               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_compiler_num
    SC[j] : Compiler_Type(cc_mu_l, cc_mu_p, cc_mu_c, cc_mu_g, cc_mu_o, cc_mu_s)
```

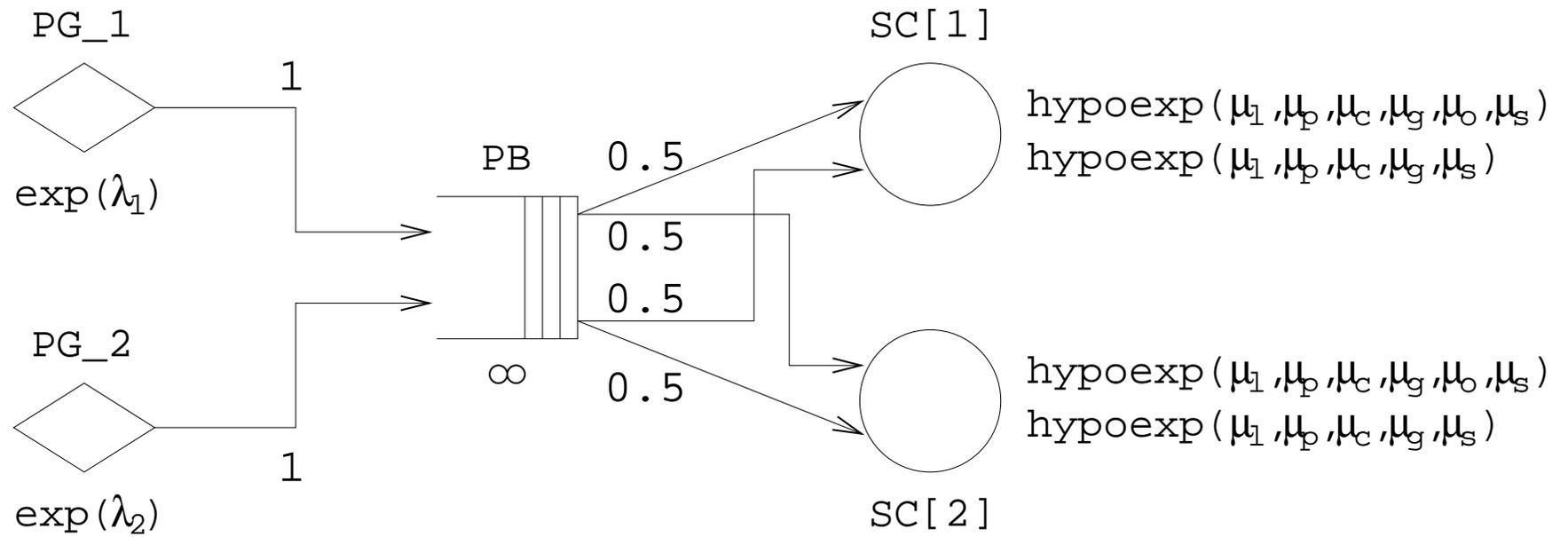
ARCHI_INTERACTIONS

```
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_compiler_num
    FROM PB.put_prog_1 TO SC[j].select_prog_1;
FOR_ALL 1 <= j <= cc_compiler_num
    FROM PB.put_prog_2 TO SC[j].select_prog_2
```

- QN model (similar to a QS M/M/2):



- In order to derive a (single-class) QS M/M/2 and exploit its formulas, we have to merge the two classes into a single one.
- Operations similar to those for the sequential compiler:

$$\begin{aligned}\lambda &= \lambda_1 + \lambda_2 \\ 1/\mu &= \frac{\lambda_1}{\lambda} \cdot (1/\mu_1 + 1/\mu_p + 1/\mu_c + 1/\mu_g + 1/\mu_o + 1/\mu_s) + \\ &\quad \frac{\lambda_2}{\lambda} \cdot (1/\mu_1 + 1/\mu_p + 1/\mu_c + 1/\mu_g + 1/\mu_s)\end{aligned}$$

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

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

- Concurrent compiler throughput:

$$\bar{T}_{\text{conc}} = \lambda$$

- Concurrent compiler utilization:

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

- Mean number of programs in the concurrent compiler system:

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

- Mean concurrent compilation time:

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

- Scenario-based comparison of the three compiler architectures.
- The comparison must be fair:
 - ⊙ identical actual values of service rates for each compilation phase across the three architectures;
 - ⊙ actual values of arrival rates for each class of programs such 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 aggregated arrival rates.
- Consider heavy load when comparing.

- Each of the three architectures works close to its maximum throughput (aggregated arrival rates arbitrarily close to corresponding overall service rates).

- From the stability conditions we derive:

$$\begin{aligned}\bar{T}_{\text{seq,max}} &= \mu \\ \bar{T}_{\text{pipe,max}} &= \min(\mu_1, \mu_p, \mu_c, \mu_g, \mu_o/p_1, \mu_s) \\ \bar{T}_{\text{conc,max}} &= 2 \cdot \mu\end{aligned}$$

- Three scenarios:

- ⊙ All compilation phases have the same average duration μ_{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 durations of all phases range between μ_{max}^{-1} and μ_{min}^{-1} , which may be several orders of magnitude apart.

- First scenario:

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

- Ratios:

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

- In this case the pipeline architecture wins.

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

$$\begin{array}{l} \bar{T}_{\text{seq,max}} = \mu_1 \\ \bar{T}_{\text{pipe,max}} = \mu_1 \\ \bar{T}_{\text{conc,max}} = 2 \cdot \mu_1 \end{array}$$

- Ratios:

$$\begin{array}{l} \bar{T}_{\text{conc,max}} / \bar{T}_{\text{pipe,max}} = 2 \\ \bar{T}_{\text{conc,max}} / \bar{T}_{\text{seq,max}} = 2 \\ \bar{T}_{\text{pipe,max}} / \bar{T}_{\text{seq,max}} = 1 \end{array}$$

- In this case the concurrent architecture wins.

- Third scenario:

$$\begin{aligned}
 (5 + p_1)^{-1} \cdot \mu_{\min} &\leq \bar{T}_{\text{seq,max}} \leq (5 + p_1)^{-1} \cdot \mu_{\max} \\
 \mu_{\min} &\leq \bar{T}_{\text{pipe,max}} \leq \mu_{\min} \\
 2 \cdot (5 + p_1)^{-1} \cdot \mu_{\min} &\leq \bar{T}_{\text{conc,max}} \leq 2 \cdot (5 + p_1)^{-1} \cdot \mu_{\max}
 \end{aligned}$$

- Ratios:

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

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

MSL: Performance Measure Specification

- The last phase of the performance-aware methodology checks whether the selected ÆMILIA description satisfies the performance requirements (possible approximations, generic indicators).
- Notations for performance modeling are accompanied by notations for specifying performance measures.
- 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 or cumulative measures refer to a time interval.
- Both kinds of measures can 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:

$$\sum_{s \in S} R_r(s) \cdot \pi[s] + \sum_{s \xrightarrow{a, \lambda} s'} R_i(s, a, \lambda, s') \cdot \phi(s, a, \lambda, s')$$

where:

- ⊙ $R_r(s)$ is the rate reward associated with s ;
- ⊙ $\pi[s]$ is the probability of being in s at the considered instant of time;
- ⊙ $R_i(s, a, \lambda, s')$ 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, which is given by $\pi[s] \cdot \lambda$.

- 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 $[z_1, \dots, z_n]$ representing the current behavior of each AEI.
- Based on a set of first-order predicates concerned with:
 - ⊙ either the local states in a set $Z \subseteq S_{\text{local}}$ relevant to the measure;
 - ⊙ or the activities in a set $A \subseteq \text{Name}$ 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:

$$\forall a \in A (is_trans(s, a, \lambda, s') \Rightarrow eq(act_contrib(s, a, \lambda, s'), act_rew(a, \lambda))) \Rightarrow eq(state_rew(s), sum_act_contrib(s, A))$$

- Every 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:

$$\forall a \in A (is_trans(s, a, \lambda, s') \Rightarrow eq(trans_rew(s, a, \lambda, s'), act_rew(a, \lambda)))$$

- Every 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:

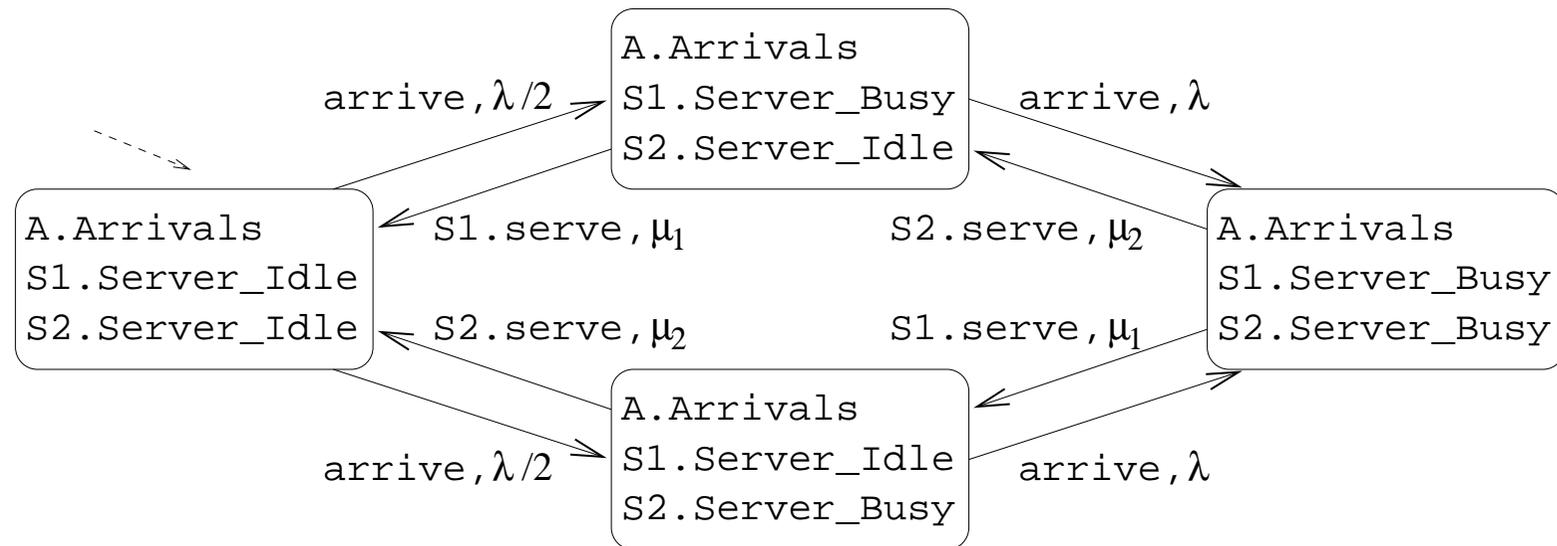
MEASURE $\langle name \rangle$ ($\langle formal\ component\ oriented\ parameters \rangle$) **IS** $\langle body \rangle$

- 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 and 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 non-experts 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 servers.
- Unbounded population of customers arriving at the service center at rate $\lambda \in \mathbb{R}_{>0}$.
- Whenever both servers are idle, an incoming customer has the same probability to be served by the two servers.
- There is no buffer, hence an incoming customer cannot be accepted and is lost whenever both servers are busy.
- The two servers process requests at rate $\mu_1, \mu_2 \in \mathbb{R}_{>0}$ respectively.

- Underlying action-labeled CTMC model:



- First formula schema:
 - ⊙ System throughput:

$$\begin{aligned}Z &= \{S1.Server_Busy, S2.Server_Busy\} \\lstate_rew(S1.Server_Busy) &= \mu_1 \\lstate_rew(S2.Server_Busy) &= \mu_2\end{aligned}$$

- ⊙ Throughput of S1:

$$\begin{aligned}Z &= \{S1.Server_Busy\} \\lstate_rew(S1.Server_Busy) &= \mu_1\end{aligned}$$

- Second formula schema:

- ⊙ System throughput:

$$\begin{aligned} A &= \{S1.serve, S2.serve\} \\ act_rew(S1.serve, any) &= \mu_1 \\ act_rew(S2.serve, any) &= \mu_2 \end{aligned}$$

- ⊙ Throughput of S1:

$$\begin{aligned} A &= \{S1.serve\} \\ act_rew(S1.serve, any) &= \mu_1 \end{aligned}$$

- Third formula schema:
 - ⊙ System throughput:

$$\begin{aligned} A &= \{S1.serve, S2.serve\} \\ act_rew(S1.serve, any) &= 1 \\ act_rew(S2.serve, any) &= 1 \end{aligned}$$

- ⊙ Throughput of S1:

$$\begin{aligned} A &= \{S1.serve\} \\ act_rew(S1.serve, any) &= 1 \end{aligned}$$

- Fourth formula schema:

- ⊙ System utilization:

$$\begin{aligned} Z &= \{S1.Server_Busy, S2.Server_Busy\} \\ lstate_rew(S1.Server_Busy) &= 1 \\ lstate_rew(S2.Server_Busy) &= 1 \\ cf &= \min \end{aligned}$$

- ⊙ Utilization of S1:

$$\begin{aligned} Z &= \{S1.Server_Busy\} \\ lstate_rew(S1.Server_Busy) &= 1 \\ cf &= \min \end{aligned}$$

- Fifth formula schema:

- System utilization:

$$\begin{aligned}A &= \{S1.serve, S2.serve\} \\act_rew(S1.serve, any) &= 1 \\act_rew(S2.serve, any) &= 1 \\cf &= \min\end{aligned}$$

- Utilization of S1:

$$\begin{aligned}A &= \{S1.serve\} \\act_rew(S1.serve, any) &= 1 \\cf &= \min\end{aligned}$$

- Sixth formula schema:

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

$$\begin{aligned} A &= \{\text{arrive}\} \\ \text{act_rew}(\text{arrive}, \text{any}) &= 1 \\ cf &= \text{min} \end{aligned}$$

- **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:

MEASURE $throughput(C_1.a_1, \dots, C_n.a_n)$ IS

$$\forall a \in \{C_1.a_1, \dots, C_n.a_n\}$$

$$(is_trans(s, a, \lambda, s') \Rightarrow$$

$$eq(act_contrib(s, a, \lambda, s'), act_rew(a, \lambda))) \Rightarrow$$

$$eq(state_rew(s), sum_act_contrib(s, \{C_1.a_1, \dots, C_n.a_n\}))$$

where $act_rew(a, \lambda) = \lambda$ for all $a \in \{C_1.a_1, \dots, C_n.a_n\}$.

- Alternative definition of throughput:

MEASURE $throughput(C_1.a_1, \dots, C_n.a_n)$ IS

$$\forall a \in \{C_1.a_1, \dots, C_n.a_n\}$$

$$(is_trans(s, a, \lambda, s') \Rightarrow$$

$$eq(trans_rew(s, a, \lambda, s'), 1))$$

- Definition of utilization:

MEASURE $utilization(C.a_1, \dots, C.a_n)$ IS

$\exists a \in \{C.a_1, \dots, C.a_n\}$

$(is_trans(s, a, \lambda, s')) \Rightarrow$

$eq(state_rew(s), choose_act_rew(s, \{C.a_1, \dots, C.a_n\}, min))$

where $act_rew(a, any) = 1$ for all $a \in \{C.a_1, \dots, C.a_n\}$.

- Definition of mean queue length:

MEASURE $mean_queue_length(C.B_1(k_1), \dots, C.B_n(k_n))$ IS

$\exists z \in \{C.B_1, \dots, C.B_n\}$

$(is_local(z, s)) \Rightarrow$

$eq(state_rew(s), choose_lstate_rew(s, \{C.B_1, \dots, C.B_n\}, min))$

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

- Definition of mean response time:

MEASURE $mean_response_time(C.B_1(k_1/\lambda), \dots, C.B_n(k_n/\lambda))$ IS
 $\exists z \in \{C.B_1, \dots, C.B_n\}$
 $(is_local(z, s)) \Rightarrow$
 $eq(state_rew(s), choose_lstate_rew(s, \{C.B_1, \dots, C.B_n\}, min))$

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):

$$\begin{aligned}
 \text{MEASURE } & \textit{total_mean_queue_length}(C_1.B_{1,1}(k_{1,1}), \dots, C_1.B_{1,n_1}(k_{1,n_1}), \\
 & \qquad \qquad \qquad C_2.B_{2,1}(k_{2,1}), \dots, C_2.B_{2,n_2}(k_{2,n_2}), \\
 & \qquad \qquad \qquad \vdots \\
 & \qquad \qquad \qquad C_m.B_{m,1}(k_{m,1}), \dots, C_m.B_{m,n_m}(k_{m,n_m})) \text{ IS} \\
 & \textit{mean_queue_length}(C_1.B_{1,1}(k_{1,1}), \dots, C_1.B_{1,n_1}(k_{1,n_1})) + \\
 & \textit{mean_queue_length}(C_2.B_{2,1}(k_{2,1}), \dots, C_2.B_{2,n_2}(k_{2,n_2})) + \\
 & \qquad \qquad \qquad \vdots \\
 & \textit{mean_queue_length}(C_m.B_{m,1}(k_{m,1}), \dots, C_m.B_{m,n_m}(k_{m,n_m}))
 \end{aligned}$$

Comparison with Related Techniques

- Generation of QN models from component-oriented descriptions instead of labeled transition systems in order to gain effectiveness.
- Focus on typical average performance indicators instead of specific performance measures in order to quicken the comparison among alternative architectural designs.
- Component-oriented specification of performance measures in order to strengthen usability.

Part IV:
Methodologies for Integrated Analysis

(in preparation)

Part V:
Architecture-Driven Code Generation

(in preparation)

Part VI:
Architecture-Driven Test Generation

(in preparation)

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] 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*, LNCS 3465:155-189, 2005.
- [3] A. Aldini and M. Bernardo, “*On the Usability of Process Algebra: An Architectural View*”, in *Theoretical Computer Science* 335:281-329, 2005.
- [4] 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.
- [5] 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.
- [6] 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)*, LNCS 1382:21-37, Lisbon (Portugal), 1998.
- [7] R. Allen and D. Garlan, “*A Formal Basis for Architectural Connection*”, in *ACM Trans. on Software Engineering and Methodology* 6:213-249, 1997.
- [8] F. Aquilani, S. Balsamo, and P. Inverardi, “*Performance Analysis at the Software Architectural Design Level*”, in *Performance Evaluation* 45:205-221, 2001.
- [9] J.C.M. Baeten and W.P. Weijland, “*Process Algebra*”, Cambridge University Press, 1990.
- [10] 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*, LNCS 2459:261-289, 2002.

- [11] S. Balsamo, M. Bernardo, and M. Simeoni, “*Performance Evaluation at the Software Architecture Level*”, in [22]:207-258.
- [12] 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.
- [13] J.A. Bergstra, A. Ponse, and S.A. Smolka (eds.), “*Handbook of Process Algebra*”, Elsevier, 2001.
- [14] M. Bernardo, “*Symbolic Semantic Rules for Producing Compact STGLA from Value Passing Process Descriptions*”, in *ACM Trans. on Computational Logic* 5:436-469, 2004.
- [15] M. Bernardo, “*Two Towers 5.1 User Manual*”, <http://www.sti.uniurb.it/bernardo/twotowers/>, 2006.
- [16] 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.
- [17] 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)*, LNCS 3454:188-203, Namur (Belgium), 2005.
- [18] M. Bernardo and E. Bontà, “*Non-Synchronous Communications in Process Algebraic Architectural Description Languages*”, in *Proc. of the 2nd European Conf. on Software Architecture (ECSA 2008)*, LNCS 5292:3-18, Paphos (Cyprus), 2008.
- [19] M. Bernardo and M. Bravetti, “*Performance Measure Sensitive Congruences for Markovian Process Algebras*”, in *Theoretical Computer Science* 290:117-160, 2003.
- [20] 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.
- [21] 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*, LNCS 2459:236-260, 2002.
- [22] M. Bernardo and P. Inverardi (eds.), “*Formal Methods for Software Architectures*”, LNCS 2804, 2003.

- [23] M. Bernardo and C.A. Middelburg (eds.), “*Special Issue on Process Algebra and System Architecture*”, *Journal of Logic and Algebraic Programming* 63, 2005.
- [24] T. Bolognesi and E. Brinksma, “*Introduction to the ISO Specification Language LOTOS*”, in *Computer Networks and ISDN Systems* 14:25-59, 1987.
- [25] 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)*, LNCS 4038:28-43, Bologna (Italy), 2006.
- [26] C. Canal, E. Pimentel, and J.M. Troya, “*Compatibility and Inheritance in Software Architectures*”, in *Science of Computer Programming* 41:105-138, 2001.
- [27] G. Clark, S. Gilmore, J. Hillston, and M. Ribaud, “*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)*, LNCS 1786:247-261, Schaumburg (IL), 2000.
- [28] 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.
- [29] R. Cleaveland and O. Sokolsky, “*Equivalence and Preorder Checking for Finite-State Systems*”, in [13]:391-424.
- [30] T.M. Cover and J.A. Thomas, “*Elements of Information Theory*”, John Wiley & Sons, 1991.
- [31] T.R. Dean and J.R. Cordy, “*A Syntactic Theory of Software Architecture*”, in *IEEE Trans. on Software Engineering* 21:302-313, 1995.
- [32] F. DeRemer and H.H. Kron, “*Programming-in-the-Large Versus Programming-in-the-Small*”, in *IEEE Trans. on Software Engineering* 2:80-86, 1976.
- [33] 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)*, LNCS 4590:158-163, Berlin (Germany), 2007.

- [34] 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.
- [35] R.J. van Glabbeek, “The Linear Time - Branching Time Spectrum I”, in [13]:3-99.
- [36] 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.
- [37] 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)*, LNCS 2852:443-466, Leiden (The Netherlands), 2002.
- [38] 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.
- [39] 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.
- [40] C.A.R. Hoare, “*Communicating Sequential Processes*”, Prentice Hall, 1985.
- [41] R.A. Howard, “*Dynamic Probabilistic Systems*”, John Wiley & Sons, 1971.
- [42] 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)*, LNCS 2029:60-75, Genova (Italy), 2001.
- [43] 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.
- [44] 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.
- [45] L. Kleinrock, “*Queueing Systems*”, John Wiley & Sons, 1975.

- [46] 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.
- [47] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, “*Specifying Distributed Software Architectures*”, in Proc. of the *5th European Software Engineering Conf. (ESEC 1995)*, LNCS 989:137-153, Barcelona (Spain), 1995.
- [48] J. Magee and J. Kramer, “*Exposing the Skeleton in the Coordination Closet*”, in Proc. of the *2nd Int. Conf. on Coordination Models and Languages (COORDINATION 1997)*, LNCS 1282:18-31, Berlin (Germany), 1997.
- [49] J. Magee and J. Kramer, “*Concurrency: State Models & Java Programs*”, John Wiley & Sons, 1999.
- [50] 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.
- [51] 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.
- [52] R. Milner, “*Communication and Concurrency*”, Prentice Hall, 1989.
- [53] M. Moriconi, X. Qian, and R.A. Riemenschneider, “*Correct Architecture Refinement*”, in IEEE Trans. on Software Engineering 21:356-372, 1995.
- [54] 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.
- [55] D.E. Perry and A.L. Wolf, “*Foundations for the Study of Software Architecture*”, in ACM SIGSOFT Software Engineering Notes 17:40-52, 1992.
- [56] 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.

- [57] M. Shaw, R. DeLine, 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.
- [58] M. Shaw and D. Garlan, “*Software Architecture: Perspectives on an Emerging Discipline*”, Prentice Hall, 1996.
- [59] C. Smith, “*Performance Engineering of Software Systems*”, Addison-Wesley, 1990.
- [60] W.J. Stewart, “*Introduction to the Numerical Solution of Markov Chains*”, Princeton University Press, 1994.