Compositionality, Interaction, and Abstraction in Complex Computing Systems: Process Calculi and Behavioral Equivalences

Marco Bernardo

University of Urbino – Italy

© July 2013
Organization of the Lectures

Part I: Concurrency and Communication 3

Part II: Process Calculi 19

Part III: Behavioral Equivalences 36

References 66
Part I:
Concurrency and Communication
Evolution in Computing

- **Sequential computing** (1940): a single instruction at a time is executed.
- Imperative models: Turing machines and less expressive automata (finite-state automata, push-down automata).
- Declarative models: Church $\lambda$-calculus, first-order logic.
- Programming languages (Fortran, Cobol, Algol, C, ..., Lisp, Prolog, ...).
- **Concurrent and distributed computing** (1980): several instructions can be simultaneously executed.
- Shared-memory model vs. message-passing model.
- Primitives for programs synchronization (semaphores, monitors, ...).
- Concurrent programming languages (Ada, Occam, Java, ...).
• **Global computing** (2000): computation over infrastructures available globally and able to provide uniform services with variable guarantees, with particular regard to exploiting their universal scale as well as the programmability of their services.

• Development of large-scale general-purpose computer systems that have dependably predictable behavior, for the needs of a distributed world.

• Support for resource sharing (grids), Internet commerce (web services), ambient intelligence (e.g. via UMTS), ...

• Addressing issues that go beyond concurrent and distributed systems: mobility, ubiquity, dynamicity, interactivity, ...

• “*Computing is interaction!*”
Communicating Concurrent Systems

- Concurrency and communication are essential in complex computing systems such as global computers.

- Any such system is composed of many interconnected parts that may interact by exchanging information or simply synchronizing.

- Concurrency and communication are complementary notions.

- Diversity: each part acts concurrently with (independently of) other parts.

- Unity: achieved through communication among the various parts.
• How can we model and analyze complex computing systems?

• Making no distinction of kind between systems and their parts enables uniform reasoning at different abstraction levels: notion of process.

• A process may be decomposed into subprocesses for a certain purpose or may be viewed as being atomic for other purposes.

• A process is a series of actions/events divided into:
  – Internal actions, possibly due to subprocesses interaction.
  – Interactions with the neighboring processes/external environment.
• A computing system is characterized by its structure and its behavior.

• A process is an abstraction of the behavior of a computing system.

• The behavior of a complex computing system can be defined as its entire capability of communication.

• Black-box view: the behavior of a system is exactly what is observable, and to observe a system is exactly to communicate with it.

• The notion of process focusses on the behavioral aspects of a system while neglecting its structural and physical attributes.
Consider a sequential system that:
- either performs action \( a \) followed by action \( b \) and then terminates;
- or performs action \( b \) followed by action \( a \) and then terminates.

Consider another system that performs actions \( a \) and \( b \) in parallel:
- either action \( a \) terminates first and then action \( b \) terminates;
- or action \( b \) terminates first and then action \( a \) terminates.

Interleaving view: for an external observer, a concurrent system behaves like a sequential one obtained by interleaving the actions of its parts.
Observational Semantics

- Computational systems featuring concurrency and communication are often required to possess a high degree of reactivity to external stimuli and are usually nonterminating.

- They are typically structured into a set of autonomous components that can evolve independently of each other and from time to time can exchange information or simply synchronize.

- Nondeterminism in the final result or in the computation can arise due to the different speeds of the components, the interaction scheme among the components, and the scheduling policies that are adopted.
• The behavior of a sequential system can be defined as a mathematical function that associates a finishing state with every possible start state.

• This input-output transformation approach for sequential systems is no longer applicable in the case of communicating concurrent systems.

• Consider the following two fragments of sequential program:

  (1) X := 1;  
  (2) X := 0; X := X + 1;

• In the absence of interference, they have the same effect (X becomes 1).

• If the two fragments are executed concurrently, the final value of X is not necessarily 1, but can be either 1 or 2 (cannot be deterministically predicted).
• We cannot abstract from the intermediate states of the computation when dealing with concurrent systems.

• Under the interleaving view of concurrency, the behavior of a concurrent system can be defined as:
  – A set of traces, with a trace being the sequence of actions performed by the system during a computation (linear-time semantics).
  – A computation tree or graph including all traces and selection points in the behavior of the system (branching-time semantics).

• Need for behavioral relations to compare structurally different systems.
Process Algebra

- Start in the late 1970’s / early 1980’s: Robin Milner, Tony Hoare, …
- **Process**: series of actions or events.
- **Algebra**: calculus of symbols combining according to certain laws.
- **Calculus**: system or method of calculation.
- Generalization of formal languages and automata theory focusing on system behavior rather than language recognition and generation.
- “Is there a calculus for processes as basic as λ-calculus for functions?”
• Conceived for understanding communicating concurrent systems and their various aspects (nondeterminism, priority, probability, time, mobility, ...).

• Compositional modeling by means of behavioral operators expressing concepts like the sequential composition, the alternative composition, and the parallel composition of processes (represent the behavior of systems).

• Abstraction from certain details of system behavior by distinguishing between visible and invisible actions (represent system activities).

• Behavioral comparison through equivalences (notion of same behavior) and preorders (notion of behavior refinement).
• Means for reasoning about the **semantics of concurrent programming** (basic formalisms such as CCS, CSP, ACP, π-calculus, ...).

• Scientific impact witnessed by a rich literature and the development of **formal description techniques** and **automated software tools** for system modeling and analysis (ISO language LOTOS; tools CADP, CWB, FDR, μCRL, ...).

• Process calculi and behavioral equivalences yielding process algebra used for **teaching**:
  - Foundations of concurrent programming.
  - Model-based design of concurrent, distributed, and mobile systems.
• Process algebra is the linguistic counterpart of computational models developed for communicating concurrent systems.

• Support for the (automata-based) interleaving view of concurrency:
  ○ Trace model.
  ○ Synchronization tree model.

• Support for the true concurrency view.
  ○ Petri net model.
  ○ Event structure model.
• Process algebra constitutes a different approach to dynamical systems.

• Basic elements:
  behaviors instead of numerically quantified characteristics.

• Combinators:
  behavioral operators instead of the arithmetical operators +, −, ×, /.

• Relations:
  behavioral equivalences and preorders instead of the usual =, ≤.

• Dynamics:
  behavioral equations instead of differential equations.
Running Example: Producer-Consumer System

- General description:
  - Three components: producer, finite-capacity buffer, consumer.
  - The producer deposits items into the buffer as long as the buffer capacity is not exceeded.
  - Stored items are then withdrawn by the consumer according to some predefined discipline (like FIFO or LIFO).

- Specific scenario:
  - The buffer has only two positions.
  - Items are identical, hence the discipline is not important.
Part II: Process Calculi
Process Calculi Syntax

• Specification language for communicating concurrent systems.

• Support for compositionality (building complex models by combining simpler models).

• Capability of abstraction (neglecting certain details of a model).

• Based on actions and behavioral operators.

• $Name_v$: set of visible action names.

• $Name = Name_v \cup \{\tau\}$: set of all action names.

• $Relab = \{\varphi : Name \rightarrow Name | \varphi^{-1}(\tau) = \{\tau\}\}$: set of visibility-preserv. relabeling functions.

• $Var$: set of process variables ($Const$: set of process constants).
• Process term syntax for process language $\mathcal{PL}$:

\[
P ::= 0 \quad \text{inactive process}
\]

\[
| \quad a \cdot P \quad \text{action prefix} \quad (a \in \text{Name})
\]

\[
| \quad P + P \quad \text{alternative composition}
\]

\[
| \quad P \parallel_S P \quad \text{parallel composition} \quad (S \subseteq \text{Name}_v)
\]

\[
| \quad P / H \quad \text{hiding} \quad (H \subseteq \text{Name}_v)
\]

\[
| \quad P \setminus L \quad \text{restriction} \quad (L \subseteq \text{Name}_v)
\]

\[
| \quad P[\varphi] \quad \text{relabeling} \quad (\varphi \in \text{Relab})
\]

\[
| \quad X \quad \text{process variable} \quad (X \in \text{Var})
\]

\[
| \quad \text{rec } X : P \quad \text{recursion} \quad (X \in \text{Var})
\]

(recursion is alternatively expressed by means of equations of the form $B \triangleq P$).
• $P_1 + P_2$ behaves as $P_1$ or $P_2$ depending on which executes first.

• The choice among several enabled actions is solved nondeterministically.

• The choice is internal if the enabled actions are all invisible, otherwise the choice can be influenced by the external environment.

• $P_1 \parallel S P_2$ behaves as $P_1$ in parallel with $P_2$ under synchronization set $S$.

• Actions whose name does not belong to $S$ are executed autonomously by $P_1$ and by $P_2$.

• Synchronization is forced between any action enabled by $P_1$ and any action enabled by $P_2$ that have the same name belonging to $S$, in which case the resulting action has the same name as the two original actions ($S = \emptyset$ implies $P_1$ and $P_2$ fully independent, $S = Name_v$ implies $P_1$ and $P_2$ fully synchronized).
• $\emptyset$ is a terminated process and hence cannot execute any action.

• $a \cdot P$ can perform $a$ and then behaves as $P$ (action-based sequential composition).

• $P / H$ behaves as $P$ but every action belonging to $H$ is turned into $\tau$ (abstraction mechanism; can be used for preventing a process from communicating).

• $P \setminus L$ behaves as $P$ but every action belonging to $L$ is forbidden (same effect as $P \parallel_L \emptyset$).

• $P[\varphi]$ behaves as $P$ but every action is renamed according to function $\varphi$ (redundance avoidance; encoding of the previous two operators if $\varphi$ is non-visib.-pres./partial).

• Operator precedence: unary operators $>$ $+$ $>$ $\parallel$.

• Operator associativity: $+$ and $\parallel$ are left associative.
• **rec** $X : P$ behaves as $P$ with every free occurrence of process variable $X$ being replaced by **rec** $X : P$ (same as $B \triangleq P$ with $B$ being a process constant).

• A process variable is said to occur **free** in a process term if it is not in the scope of a **rec** binder for that variable, otherwise it is said to be **bound** in that process term.

• A process term is said to be **closed** if all of its occurrences of process variables are bound, otherwise it is said to be **open**.

• A process term is said to be **guarded** iff all of its occurrences of process variables are in the scope of action prefix operators.

• $\mathcal{P}$: set of closed and guarded process terms (fully defined, finitely branching).
• **Running example** (process syntax):
  
  ○ Conventions: action names are verbs composed of lower-case letters, process constant names are nouns starting with an upper-case letter.
  ○ The only observable activities are deposits and withdrawals.
  ○ Visible actions: *deposit* and *withdraw*.
  ○ Structure-independent process algebraic description:

    \[
    \begin{align*}
    ProdCons_{0/2} & \triangleq deposit \cdot ProdCons_{1/2} \\
    ProdCons_{1/2} & \triangleq deposit \cdot ProdCons_{2/2} + withdraw \cdot ProdCons_{0/2} \\
    ProdCons_{2/2} & \triangleq withdraw \cdot ProdCons_{1/2}
    \end{align*}
    \]

  ○ Specification to which every correct implementation should conform.
Process Calculi Semantics

- Mathematical model in the form of a state transition graph representing all computations and branching points (synchronization tree if unwound).
- Keller transition systems (instead of Kripke structures) to elicit interaction.
- Every process term $P \in \mathcal{P}$ is mapped to a labeled transition system $[P]$:  
  - Each state corresponds to a process term into which $P$ can evolve.
  - The initial state corresponds to $P$.
  - Each transition from a source state to a target state is labeled with the action that determines the corresponding state change.

- The transition relation $\xrightarrow{P}$ of $[P]$ is contained in the smallest subset of $\mathcal{P} \times Name \times \mathcal{P}$ that satisfies some Plotkin style operational semantic rules defined by induction on the syntactical structure of process terms.
• Derivation of one single transition at a time by applying the operational semantic rules to the source state of the transition.

• Basic rule for action prefix, inductive rules for all the other operators.

• Different formats: dynamic operators (\( . + \)), static operators (\( \parallel / \backslash [ ] \)).

• No rule for 0: \([0]\) has a single state and no transitions.

• Operational semantic rule for action prefix:

\[
\begin{align*}
\text{a} \cdot P & \xrightarrow{a} P
\end{align*}
\]

• Operational semantic rule for recursion:

\[
\begin{align*}
P\{\text{rec } X : P \leftrightarrow X\} & \xrightarrow{a} P' \\
\text{rec } X : P & \xrightarrow{a} P'
\end{align*}
\]

\[
\begin{align*}
\left( B \triangleq P \quad P \xrightarrow{a} P' \right) \\
B \xrightarrow{a} P'
\end{align*}
\]
• Operational semantic rules for alternative composition:

\[
\begin{align*}
&P_1 \xrightarrow{a} P'_1 \\
&P_1 + P_2 \xrightarrow{a} P'_1 \\
&P_2 \xrightarrow{a} P'_2 \\
&P_1 + P_2 \xrightarrow{a} P'_2
\end{align*}
\]

• If several actions are initially enabled, the choice among them is solved nondeterministically due to the absence of precise criteria or quantitative information (if-then-else, priority, probability, time).

• The choice is internal if the initially enabled actions are all invisible.

• Otherwise the choice can be influenced by the external environment.
• Operational semantic rules for parallel execution:

\[
\begin{align*}
&P_1 \xrightarrow{a} P'_1 \quad a \notin S \\
&P_2 \xrightarrow{a} P'_2 \quad a \notin S \\
&P_1 \parallel S \ P_2 \xrightarrow{a} P'_1 \parallel S \ P_2 \\
&P_1 \parallel S \ P_2 \xrightarrow{a} P_1 \parallel S \ P'_2
\end{align*}
\]

• Operational semantic rule for synchronization:

\[
\begin{align*}
&P_1 \xrightarrow{a} P'_1 \quad P_2 \xrightarrow{a} P'_2 \quad a \in S \\
&P_1 \parallel S \ P_2 \xrightarrow{a} P'_1 \parallel S \ P'_2
\end{align*}
\]

• Parallel composition is thus given an **interleaving semantics**.
• The following process terms represent structurally different systems:

\[ a \cdot b \cdot 0 + b \cdot a \cdot 0 \]

\[ a \cdot 0 \| \emptyset b \cdot 0 \]

but they are indistinguishable by an external observer.

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

![Diagram](image)

• Truly concurrent semantics are possible (via Petri net or event structure models) in which causally independent activities can also occur simultaneously.
- Operational semantic rules for hiding, restriction, relabeling:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \xrightarrow{a} P'$</td>
<td>$a \in H$</td>
</tr>
<tr>
<td>$P \xrightarrow{\tau} P'$</td>
<td></td>
</tr>
<tr>
<td>$P / H \xrightarrow{a} P' / H$</td>
<td>$a \notin H$</td>
</tr>
<tr>
<td>$P \xrightarrow{a} P'$</td>
<td>$a \notin L$</td>
</tr>
<tr>
<td>$P \xrightarrow{\phi(a)} P'$</td>
<td>$P[\phi] \xrightarrow{\phi(a)} P'[\phi]$</td>
</tr>
</tbody>
</table>

- $[P]$ is **finite state** if no recursive definition in $P$ contains static operators.
- **Running example** (process semantics):
  - Labeled transition system \([\textit{ProdCons}_{0/2}]\) with explicit states:

    ![Diagram](diagram.png)

    - Obtained by mechanically applying the operational semantic rules for process constant, alternative composition, and action prefix.
Computational Power of Process Calculi

- Process calculi with the considered operators (in particular, recursion) have full computational power.

- A Turing machine can be simulated by two stacks together with a finite-state control mechanism.

- In a process calculus, the interaction of the control mechanism with the two stacks can be represented through parallel composition.

- The finite-state control mechanism can be represented by making use of action prefix, alternative composition, and recursion:

\[ Q_i \triangleq a_{i,j_1} \cdot Q_{j_1} + a_{i,j_2} \cdot Q_{j_2} + \ldots + a_{i,j_{n_i}} \cdot Q_{j_{n_i}} \]
• A stack for a set $V$ of values can be \emph{inductively specified} as follows:

$$Stack(\varepsilon) \triangleq \sum_{v \in V} push_v . Stack(v) + signal_{\text{empty}} . Stack(\varepsilon)$$

$$Stack(v :: \sigma) \triangleq \sum_{w \in V} push_w . Stack(w :: v :: \sigma) + pop_v . Stack(\sigma)$$

where $\varepsilon$ is the empty sequence and $\sigma$ belongs to $V^*$.

• \emph{Infinitely many} equations to be implemented because $V^*$ is countable.

• \emph{Infinitely branching} summations to be implemented if $V$ is infinite.

• $V$ must be finite in a Turing machine.
• **Finite implementation** for a finite set $V$ of values based on as many cells as there are values, which are linked together as needed ($|V| + 2$ equations):

$$
Cell_v \triangleq \sum_{w \in V} push_w \cdot (Cell_v \uparrow Cell_v) + pop_v \cdot Inactive
$$

$$
Inactive \triangleq \sum_{u \in V} new_{\text{top}_u} \cdot Cell_u + last_{\text{cell}} \cdot Empty
$$

$$
Empty \triangleq \sum_{w \in V} push_w \cdot (Cell_w \uparrow Empty) + signal_{\text{empty}} \cdot Empty
$$

• Definition of the auxiliary linking operator:

$$
P \uparrow Q \triangleq P^{\{\text{dec} / \text{new}_{\text{top}}, \text{e} / \text{last}_{\text{cell}}\}} \parallel \{\text{inc}, \text{dec}, \text{e}\}
Q^{\{\text{inc} / \text{push}, \text{dec} / \text{pop}, \text{e} / \text{signal}_{\text{empty}}\}}
$$
Part III:
Behavioral Equivalences
Behavioral Equivalences for PA

- Establishing whether two process terms are equivalent amounts to establishing whether the systems they represent behave the same.

- Preservation of compositionality and abstraction should be achieved.

- Useful for theoretical and applicative purposes:
  - Comparing syntactically different process terms on the basis of the behavior they exhibit.
  - Relating process algebraic descriptions of the same system at different abstraction levels (top-down modeling).
  - Manipulating process algebraic descriptions in a way that preserves certain properties (state space reduction before analysis).
• Features of a good behavioral equivalence:
  ○ Being a congruence with respect to all the behavioral operators, so as to support compositional reasoning.
  ○ Having a sound and complete axiomatization, which elucidates the fundamental equational laws of the equivalence with respect to the behavioral operators (rewriting rules for syntactical manipulation).
  ○ Having a logical characterization, which shows the behavioral properties preserved by the equivalence (diagnostic information).
  ○ Being equipped with an efficient verification algorithm, which runs in polynomial time in the worst case (finite-state systems – undecidable o.w.).
  ○ Being able to abstract from invisible actions.

• Three fundamental approaches: trace, bisimulation, testing.
- **Trace approach** (Hoare et al.): two processes are equivalent if they are able to execute the same sequences of actions ($\approx_{Tr}$).

- Total abstraction from branching points leads to **deadlock insensitivity**:
  \[ \text{rec } X : a \cdot X + a \cdot 0 \approx_{Tr} \text{ rec } X : a \cdot X \]  but the first one can deadlock.

- Deadlock-sensitive (hence finer) variants of trace equivalence:
  - **Completed-trace equivalence**: it compares process terms also with respect to traces that lead to deadlock ($\approx_{Tr,c}$).
  - **Failure equivalence**: it takes into account the sets of visible actions that can be refused after executing a trace ($\approx_{F}$).
  - **Readiness equivalence**: it takes into account the sets of visible actions that are enabled after executing a trace ($\approx_{R}$).
  - **Failure-trace equivalence**: it takes into account the sets of visible actions that can be refused at each step of a trace ($\approx_{FTr}$).
  - **Ready-trace equivalence**: it takes into account the sets of visible actions that are enabled at each step of a trace ($\approx_{RTr}$).
• **Bisimulation approach** (*Milner, Park*): two processes are equivalent if they are able to mimic each other’s behavior stepwise ($\sim_B$).

• Faithful account of branching points leads to overdiscrimination: $a \cdot b \cdot c \cdot 0 + a \cdot b \cdot d \cdot 0 \not\sim_B a \cdot (b \cdot c \cdot 0 + b \cdot d \cdot 0)$ is hardly justifiable.

• Coarser variants of bisimulation equivalence:
  - **Simulation equivalence**: it is the intersection of two preorders, each of which considers the capability of stepwise behavior mimicking in one single direction ($\sim_S$).
  - **Failure-simulation equivalence**: same as simulation equivalence, with in addition the fact that each of the two preorders checks for the equality of the sets of actions that can be stepwise refused ($\sim_{FS}$).
  - **Ready-simulation equivalence**: same as simulation equivalence, with in addition the fact that each of the two preorders checks for the equality of the sets of actions that are stepwise enabled ($\sim_{RS}$).
• **Testing approach** (*De Nicola & Hennessy*): two processes are equivalent if their reaction to tests is the same ($\approx_{Te}$).

• Tests formalized as processes extended with a success action/state.

• Interaction between process and test formalized as parallel composition.

• It holds that $a \cdot b \cdot c \cdot 0 + a \cdot b \cdot d \cdot 0 \approx_{Te} a \cdot (b \cdot c \cdot 0 + b \cdot d \cdot 0)$.

• Intersection of may-testing equivalence (at least one computation leads to success) and must-testing equivalence (all computations lead to success).

• May-testing equivalence coincides with trace equivalence.

• Testing equivalence coincides with failure equivalence for nondiverging, finitely-branching processes.

• Checking whether every test may/must be passed.
- Linear-time/branching-time spectrum for finitely-branching processes with no $\tau$-actions (Van Glabbeek):

```
\sim_B \sim \sim_{RS} \sim_{FS} \sim_{RTr} \sim_{R} \sim_{FTr} \sim_{F} \sim_{Te} \sim_{Tr,c} \sim_{Tr}
```
Bisimulation Equivalence

• Whenever a process term can perform a certain action, then any process
term equivalent to the given one has to be able to perform that action,
and the derivative process terms must still be equivalent to each other.

• A binary relation $B$ over $P$ is a bisimulation iff, whenever $(P_1, P_2) \in B$,
then for all actions $a \in Name$:
  - Whenever $P_1 \xrightarrow{a} P'_1$, then $P_2 \xrightarrow{a} P'_2$ with $(P'_1, P'_2) \in B$.
  - Whenever $P_2 \xrightarrow{a} P'_2$, then $P_1 \xrightarrow{a} P'_1$ with $(P'_1, P'_2) \in B$.

• Bisimulation equivalence $\sim_B$ is the union of all the bisimulations.

• Coinductive definition.
• In order for $P_1 \sim_B P_2$, it is necessary that for all $a \in \text{Name}$ there exist $P'_1, P'_2 \in \mathbb{P}$ such that:

\[
\begin{align*}
P_1 \xrightarrow{a} P'_1 & \iff P_2 \xrightarrow{a} P'_2
\end{align*}
\]

• A binary relation $\mathcal{B}$ over $\mathbb{P}$ is a bisimulation up to $\sim_B$ iff, whenever $(P_1, P_2) \in \mathcal{B}$, then for all actions $a \in \text{Name}$:

\[\circ \text{ Whenever } P_1 \xrightarrow{a} P'_1, \text{ then } P_2 \xrightarrow{a} P'_2 \text{ with } P'_1 \sim_B Q_1 \mathcal{B} Q_2 \sim_B P'_2.\]

\[\circ \text{ Whenever } P_2 \xrightarrow{a} P'_2, \text{ then } P_1 \xrightarrow{a} P'_1 \text{ with } P'_1 \sim_B Q_1 \mathcal{B} Q_2 \sim_B P'_2.\]

• Focus on important pairs of process terms that form a bisimulation.

• In order for $P_1 \sim_B P_2$, it is sufficient to find a bisimulation up to $\sim_B$ that contains $(P_1, P_2)$. 
• $\sim_B$ is a congruence with respect to all the dynamic and static operators as well as recursion.

• Substituting equals for equals does not alter the overall meaning in any process context.

• Let $P_1, P_2 \in \mathbb{P}$. Whenever $P_1 \sim_B P_2$, then:

\[
\begin{array}{c}
a \cdot P_1 \sim_B a \cdot P_2 \\
P_1 + P \sim_B P_2 + P \\
P_1 \parallel_S P \sim_B P_2 \parallel_S P \\
P_1 / H \sim_B P_2 / H \\
P_1 \setminus L \sim_B P_2 \setminus L \\
P_1[\varphi] \sim_B P_2[\varphi]
\end{array}
\]
• Recursion: extend $\sim_B$ to open process terms by replacing all variables freely occurring outside rec binders with every closed process term.

• Let $P_1, P_2 \in \mathcal{PL}$ be guarded process terms containing free occurrences of $k \in \mathbb{N}$ process variables $X_1, \ldots, X_k \in Var$ at most.

• We define $P_1 \sim_B P_2$ iff:

\[
P_1\{Q_i \leftarrow X_i \mid 1 \leq i \leq k\} \sim_B P_2\{Q_i \leftarrow X_i \mid 1 \leq i \leq k\}
\]

for all $Q_1, \ldots, Q_k \in \mathbb{P}$.

• Whenever $P_1 \sim_B P_2$, then:

\[
\text{rec } X : P_1 \sim_B \text{ rec } X : P_2
\]
• $\sim_B$ has a **sound and complete axiomatization** over the set $\mathbb{P}_{nrec}$ of nonrecursive process terms of $\mathbb{P}$.

• **Basic laws** (*commutativity, associativity, and neutral element of +*):

\[
\begin{align*}
(A_{B,1}) & \quad P_1 + P_2 = P_2 + P_1 \\
(A_{B,2}) & \quad (P_1 + P_2) + P_3 = P_1 + (P_2 + P_3) \\
(A_{B,3}) & \quad P + 0 = P
\end{align*}
\]

• **Characterizing laws** (*idempotency of +, not valid for synchronization tree isomorphism*):

\[
\begin{align*}
(A_{B,4}) & \quad P + P = P
\end{align*}
\]
- **Expansion law** (interleaving view of concurrency; $I$ and $J$ nonempty and finite):

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
</table>
| \((\mathcal{A}_B,5)\) | \[
\sum_{i \in I} a_i \cdot P_i \parallel_s \sum_{j \in J} b_j \cdot Q_j = \sum_{k \in I, a_k \notin S} a_k \cdot \left( P_k \parallel_s \sum_{j \in J} b_j \cdot Q_j \right) + \sum_{h \in J, b_h \notin S} b_h \cdot \left( \sum_{i \in I} a_i \cdot P_i \parallel_s Q_h \right) + \sum_{k \in I, a_k \in S} \sum_{h \in J, b_h = a_k} a_k \cdot (P_k \parallel_s Q_h) \]
| \((\mathcal{A}_B,6)\) | \[
\sum_{i \in I} a_i \cdot P_i \parallel_s 0 = \sum_{k \in I, a_k \notin S} a_k \cdot P_k \]
| \((\mathcal{A}_B,7)\) | \[
0 \parallel_s \sum_{j \in J} b_j \cdot Q_j = \sum_{h \in J, b_h \notin S} b_h \cdot Q_h \]
| \((\mathcal{A}_B,8)\) | \[
0 \parallel_s 0 = 0 \]
• Distribution laws (for unary static operators):

\begin{align*}
(\mathcal{A}_B,9) & \quad 0 / H = 0 \\
(\mathcal{A}_B,10) & \quad (a \cdot P) / H = \tau \cdot (P / H) \quad \text{if } a \in H \\
(\mathcal{A}_B,11) & \quad (a \cdot P) / H = a \cdot (P / H) \quad \text{if } a \notin H \\
(\mathcal{A}_B,12) & \quad (P_1 + P_2) / H = P_1 / H + P_2 / H \\
(\mathcal{A}_B,13) & \quad 0 \setminus L = 0 \\
(\mathcal{A}_B,14) & \quad (a \cdot P) \setminus L = 0 \quad \text{if } a \in L \\
(\mathcal{A}_B,15) & \quad (a \cdot P) \setminus L = a \cdot (P \setminus L) \quad \text{if } a \notin L \\
(\mathcal{A}_B,16) & \quad (P_1 + P_2) \setminus L = P_1 \setminus L + P_2 \setminus L \\
(\mathcal{A}_B,17) & \quad 0[\varphi] = 0 \\
(\mathcal{A}_B,18) & \quad (a \cdot P)[\varphi] = \varphi(a) \cdot (P[\varphi]) \\
(\mathcal{A}_B,19) & \quad (P_1 + P_2)[\varphi] = P_1[\varphi] + P_2[\varphi]
\end{align*}
• $DED(A_B)$: deduction system based on all the previous axioms plus:
  ○ Reflexivity: $A_B \vdash P = P$.
  ○ Symmetry: $A_B \vdash P_1 = P_2 \implies A_B \vdash P_2 = P_1$.
  ○ Transitivity: $A_B \vdash P_1 = P_2 \land A_B \vdash P_2 = P_3 \implies A_B \vdash P_1 = P_3$.
  ○ Substitutivity: $A_B \vdash P_1 = P_2 \implies A_B \vdash a \cdot P_1 = a \cdot P_2 \land \ldots$

• The deduction system $DED(A_B)$ is sound and complete for $\sim_B$ over $\mathbb{P}_{nrec}$; i.e., for all $P_1, P_2 \in \mathbb{P}_{nrec}$:

\[
A_B \vdash P_1 = P_2 \iff P_1 \sim_B P_2
\]
• $\sim_B$ has a modal logic characterization based on Hennessy-Milner logic.

• Basic truth values and propositional connectives, plus modal operators expressing how to behave after executing certain actions.

• Syntax of the modal language $\text{HML}$ ($a \in \text{Name}$):

$$
\phi ::= \text{true} \quad \text{basic truth value} \\
| \neg\phi \quad \text{negation} \\
| \phi \land \phi \quad \text{conjunction} \\
| \langle a \rangle \phi \quad \text{possibility}
$$

plus derived logical operators:

$$
\text{false} \equiv \neg\text{true} \quad \text{basic truth value} \\
\phi_1 \lor \phi_2 \equiv \neg(\neg\phi_1 \land \neg\phi_2) \quad \text{disjunction} \\
[a]\phi \equiv \neg\langle a \rangle \neg\phi \quad \text{necessity}
$$
• Interpretation of HML over $\mathbb{P}$:

<table>
<thead>
<tr>
<th>$P$</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>$\neg \phi$ if $P \nmid \phi$</td>
</tr>
<tr>
<td>$P$</td>
<td>$\phi_1 \land \phi_2$ if $P \models \phi_1$ and $P \models \phi_2$</td>
</tr>
<tr>
<td>$P$</td>
<td>$\langle a \rangle \phi$ if there exists $P' \in \mathbb{P}$ such that $P \xrightarrow{a} P'$ and $P' \models \phi$</td>
</tr>
</tbody>
</table>

plus derived logical operators:

<table>
<thead>
<tr>
<th>$P$</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>$\phi_1 \lor \phi_2$ if $P \models \phi_1$ or $P \models \phi_2$</td>
</tr>
<tr>
<td>$P$</td>
<td>$[a] \phi$ if for all $P' \in \mathbb{P}$, whenever $P \xrightarrow{a} P'$, then $P' \models \phi$</td>
</tr>
</tbody>
</table>

• For all $P_1, P_2 \in \mathbb{P}$:

$$P_1 \sim_B P_2 \iff (\forall \phi \in \text{HML}. \ P_1 \models \phi \iff P_2 \models \phi)$$
• \( \sim_B \) has a **temporal logic characterization** based on \( \text{CTL}^* \).

• State formulae with atomic propositions and propositional connectives, plus path formulae including temporal operators about the future.

• A path \( \pi \) is a sequence of states \( s_0s_1s_2 \ldots \) such that \( s_i \) reaches \( s_{i+1} \).

• \( \text{CTL}^* \) is interpreted over Kripke structures, hence we redefine \( \sim_B \)
  (propositions labeling states instead of actions labeling transitions).

• A binary relation \( \mathcal{B} \) over Kripke structure \( (S, \mathcal{L}, \rightarrow) \) is a **bisimulation** iff, whenever \( (s_1, s_2) \in \mathcal{B} \), then \( \mathcal{L}(s_1) = \mathcal{L}(s_2) \) and:
  
  - Whenever \( s_1 \rightarrow s'_1 \), then \( s_2 \rightarrow s'_2 \) with \( (s'_1, s'_2) \in \mathcal{B} \).
  - Whenever \( s_2 \rightarrow s'_2 \), then \( s_1 \rightarrow s'_1 \) with \( (s'_1, s'_2) \in \mathcal{B} \).

• Bisimulation equivalence \( \sim_B \) is the union of all the bisimulations.
• Syntax of the temporal language CTL*:

| φ  ::=  ap          | atomic proposition             |
|     | ¬φ            | state formula negation         |
|     | φ ∧ φ         | state formula conjunction      |
|     | Eψ            | existential path quantifier    |

| ψ  ::=  φ          | state formula             |
|     | ¬ψ            | path formula negation        |
|     | ψ ∧ ψ         | path formula conjunction     |
|     | Xψ            | next operator               |
|     | ψ Uψ          | until operator              |

plus derived logical operators:

| Aψ  ≡  ¬E¬ψ   | universal path quantifier          |
| Fψ  ≡  true Uψ | eventually operator                |
| Gψ  ≡  ¬F¬ψ   | globally operator                  |
• Interpretation of CTL* over Kripke structure \((S, \mathcal{L}, \longrightarrow)\):

| \( s \models ap \) | if \( ap \in \mathcal{L}(s) \) |
| \( s \models \neg \phi \) | if \( s \not\models \phi \) |
| \( s \models \phi_1 \land \phi_2 \) | if \( s \models \phi_1 \) and \( s \models \phi_2 \) |
| \( s \models E \psi \) | if there exists a path \( \pi \) starting from \( s \) such that \( \pi \models \psi \) |

| \( \pi \models \phi \) | if \( \pi[0] \models \phi \) |
| \( \pi \models \neg \psi \) | if \( \pi \not\models \psi \) |
| \( \pi \models \psi_1 \land \psi_2 \) | if \( \pi \models \psi_1 \) and \( \pi \models \psi_2 \) |
| \( \pi \models X \psi \) | if \( \pi^1 \models \psi \) |
| \( \pi \models \psi_1 U \psi_2 \) | if \( \pi^k \models \psi_2 \) for some \( k \geq 0 \) and \( \pi^i \models \psi_1 \) for all \( 0 \leq i < k \) |

• For all \( s_1, s_2 \in S \):

\[
s_1 \sim_B s_2 \iff (\forall \phi \in \text{CTL}^*. s_1 \models \phi \iff s_2 \models \phi)
\]
• $\sim_B$\ is decidable in polynomial time\ over the set $\mathbb{P}_{\text{fin}}$\ of finite-state process terms\ of $\mathbb{P}$\ with Paige-Tarjan partition refinement algorithm.

• Based on the fact that $\sim_B$\ can be characterized as the limit of a sequence of successively finer equivalence relations (original definition of bisimilarity):

\[
\sim_B = \bigcap_{i \in \mathbb{N}} \sim_{B,i}
\]

• $\sim_{B,0} = \mathbb{P} \times \mathbb{P}$\ hence it induces the trivial partition $\{\mathbb{P}\}$.

• Whenever $P_1 \sim_{B,i} P_2$, $i \in \mathbb{N}_{\geq 1}$, then for all actions $a \in \text{Name}$:
  
  $\circ$ \ Whenever $P_1 \xrightarrow{a} P_1'$, then $P_2 \xrightarrow{a} P_2'$ with $P_1' \sim_{B,i-1} P_2'$.
  
  $\circ$ \ Whenever $P_2 \xrightarrow{a} P_2'$, then $P_1 \xrightarrow{a} P_1'$ with $P_1' \sim_{B,i-1} P_2'$.

• $\sim_{B,1}$\ refines $\{\mathbb{P}\}$\ by creating an equivalence class for each set of process terms that satisfy the necessary condition for $\sim_B$. 

Steps of the algorithm for checking whether $P_1 \sim_B P_2$:

1. Build an initial partition with a single class including all the states of $[P_1]$ and all the states of $[P_2]$.
2. Initialize a list of splitters with the above class as its only element.
3. While the list of splitters is not empty, select a splitter and remove it from the list after refining the current partition for all $a \in Name_{P_1,P_2}$:
   a. Split each class of the current partition by comparing its states when executing actions of name $a$ that lead to the selected splitter.
   b. For each class that has been split, insert its smallest subclass into the list of splitters.
4. Return yes/no depending on whether the initial states of $[P_1]$ and $[P_2]$ belong to the same class of the final partition or not.

The time complexity is $O(m \cdot \log n)$, where $n$ is the number of states and $m$ is the number of transitions of $[P_1]$ and $[P_2]$ (also for minimization).
• **Running example** (bisimulation equivalence):

  ○ Concurrent implementation (with two independent one-position buffers):
    
    \[
    PC_{conc,2} \triangleq \text{Prod} \parallel\{\text{deposit}\}(\text{Buff} \parallel\emptyset \text{Buff}) \parallel\{\text{withdraw}\} \text{Cons}
    \]
    \[
    \text{Prod} \triangleq \text{deposit} \cdot \text{Prod}
    \]
    \[
    \text{Buff} \triangleq \text{deposit} \cdot \text{withdraw} \cdot \text{Buff}
    \]
    \[
    \text{Cons} \triangleq \text{withdraw} \cdot \text{Cons}
    \]

  ○ Bisimulation proving \(PC_{conc,2} \sim_B \text{ProdCons}_{0/2}\):

  ![Diagram](image_url)
Weak Bisimulation Equivalence

- $\sim_B$ does not abstract from invisible actions: $a\cdot b\cdot 0 \not\sim_B a\cdot \tau\cdot b\cdot 0$.

- Two process terms should be considered equivalent in the bisimulation approach if they are able to mimic each other’s visible behavior stepwise.

- Extending the transition relation $\quad \rightarrow \quad$ to action sequences.

- $P \xrightarrow{a_1\ldots a_n} P'$ iff:
  - either $n = 0$ and $P \equiv P'$, meaning that $P$ stays idle;
  - or $n \geq 1$ and there exist $P_0, P_1, \ldots, P_n \in \mathbb{P}$ such that:
    - $P \equiv P_0$;
    - $P_{i-1} \xrightarrow{a_i} P_i$ for all $1 \leq i \leq n$;
    - $P_n \equiv P'$.
• A binary relation $\mathcal{B}$ over $\mathbb{P}$ is a weak bisimulation iff, whenever $(P_1, P_2) \in \mathcal{B}$, then:
  
  - Whenever $P_1 \xrightarrow{\tau} P'_1$, then $P_2 \xrightarrow{\tau^*} P'_2$ with $(P'_1, P'_2) \in \mathcal{B}$.
  - Whenever $P_2 \xrightarrow{\tau} P'_2$, then $P_1 \xrightarrow{\tau^*} P'_1$ with $(P'_1, P'_2) \in \mathcal{B}$.

  and for all visible actions $a \in \text{Name}_v$:
  
  - Whenever $P_1 \xrightarrow{a} P'_1$, then $P_2 \xrightarrow{\tau^* a \tau^*} P'_2$ with $(P'_1, P'_2) \in \mathcal{B}$.
  - Whenever $P_2 \xrightarrow{a} P'_2$, then $P_1 \xrightarrow{\tau^* a \tau^*} P'_1$ with $(P'_1, P'_2) \in \mathcal{B}$.

• Weak bisimulation equivalence $\cong_B$ is the union of all the weak bisimulations.

• Coinductive definition.
• \( \approx_B \) is a congruence with respect to all the behavioral operators except for alternative composition (not a problem in practice).

• Additional \( \tau \)-laws highlighting abstraction capabilities:

\[
\begin{align*}
\tau \cdot P &= P \\
\alpha \cdot \tau \cdot P &= \alpha \cdot P \\
P + \tau \cdot P &= \tau \cdot P \\
\alpha \cdot (P_1 + \tau \cdot P_2) + \alpha \cdot P_2 &= \alpha \cdot (P_1 + \tau \cdot P_2)
\end{align*}
\]

• Weak modal operators replacing those of HML \((\alpha \in \text{Name}_v)\):

\[
\begin{align*}
P &\models \langle \tau \rangle \phi \quad \text{if there exists } P' \in \mathbb{P} \text{ such that } P \xrightarrow{\tau^*} P' \text{ and } P' \models \phi \\
P &\models \langle \alpha \rangle \phi \quad \text{if there exists } P' \in \mathbb{P} \text{ such that } P \xrightarrow{\tau^* \alpha \tau^*} P' \text{ and } P' \models \phi
\end{align*}
\]

• Temporal logic characterization based on \( \text{CTL}^* \) without \( \text{X} \).
• $P_1 \approx_B P_2$ can be decided in $O(n^2 \cdot m \cdot \log n)$ time with the verification algorithm for $\sim_B$ preceded by the following preprocessing step:

0. Build the reflexive and transitive closure of $\tau \rightarrow$ in $[P_i]$ for $i = 1, 2$:
   a. Add a looping $\tau$-transition to each state.
   b. Add a $\tau$-transition between the initial state and the final state of any sequence of at least two $\tau$-transitions, if the two states are distinct and all the transitions in the sequence are distinct and nonlooping.
   c. Add an $a$-transition, $a \in Name_v$, between the initial state and the final state of any sequence of at least two transitions in which one is labeled with $a$, if all the other transitions in the sequence are labeled with $\tau$, distinct, and nonlooping.
• The fact that \( \approx_B \) is not a congruence with respect to the alternative composition operator stems from \( \tau \cdot P = P \) (abstraction from initial \( \tau \)-actions).

• This \( \tau \)-law cannot be freely used in all contexts when \( P \) is stable: \( \tau \cdot a \cdot 0 \approx_B a \cdot 0 \) but \( \tau \cdot a \cdot 0 + b \cdot 0 \not\approx_B a \cdot 0 + b \cdot 0 \).

• Congruence w.r.t. the alternative composition operator can be restored by enforcing a matching on initial \( \tau \)-actions in the definition.

• \( P_1 \in \mathbb{P} \) is weakly bisimulation congruent to \( P_2 \in \mathbb{P} \), written \( P_1 \approx^c_B P_2 \), iff for all actions \( a \in \text{Name} \) (hence including \( \tau \)):
  
  - Whenever \( P_1 \xrightarrow{a} P'_1 \), then \( P_2 \xrightarrow{\tau^* a \tau^*} P'_2 \) with \( P'_1 \approx_B P'_2 \).
  - Whenever \( P_2 \xrightarrow{a} P'_2 \), then \( P_1 \xrightarrow{\tau^* a \tau^*} P'_1 \) with \( P'_1 \approx_B P'_2 \).

• \( \approx^c_B \) is the largest congruence contained in \( \approx_B \).
• $\approx_B$ does not fully retain the property possessed by $\sim_B$ of respecting the branching structure of process terms.

• Given $P_1 \approx_B P_2$, when $P_1 \xrightarrow{a} P'_1$ then $P_2 \xrightarrow{\tau^*} Q \xrightarrow{a} Q' \xrightarrow{\tau^*} P'_2$ with $P'_1 \approx_B P'_2$ . . .

• . . . but we do not know whether any relation exists between $P_1$ and $Q$ and between $P'_1$ and $Q'$.

• The property can be restored by requiring that $P_1$ be equivalent to $Q$ and that $P'_1$ be equivalent to $Q'$.

• The resulting equivalence is branching bisimulation equivalence ($\approx_{B,b}$).

• Characterized by a single $\tau$-law: $a \cdot (\tau \cdot (P_1 + P_2) + P_1) = a \cdot (P_1 + P_2)$.

• $\approx_{B,b}$ coincides with $\approx_B$ on any pair of process terms with at most one of them reaching unstable process terms.
• **Running example** (weak bisimulation equivalence):

  ○ Pipeline implementation (with two communicating one-position buffers):

    $$PC_{\text{pipe},2} \triangleq \text{Prod} \parallel \{\text{deposit}\} (\text{LBuff} \parallel \{\text{pass}\} \text{RBuff}) / \{\text{pass}\} \parallel \{\text{withdraw}\} \text{Cons}$$

    $\text{Prod} \triangleq \text{deposit} \cdot \text{Prod}$

    $\text{LBuff} \triangleq \text{deposit} \cdot \text{pass} \cdot \text{LBuff}$

    $\text{RBuff} \triangleq \text{pass} \cdot \text{withdraw} \cdot \text{RBuff}$

    $\text{Cons} \triangleq \text{withdraw} \cdot \text{Cons}$

  ○ Weak bisimulation proving $PC_{\text{pipe},2} \approx_B \text{ProdCons}_{0/2}$:

    ![Diagram showing weak bisimulation proving](image)
References


