Teaching Concepts of Compositional
Concurrency with State Machines

Antti Valmari
Tampere University of Technology, Institute of Software Systems
PO Box 553, FI-33101 Tampere, FINLAND
Antti.Valmari@tut.fi

Abstract. This paper discusses an approach towards teaching concurrency emphasising process-algebraic compositionality while avoiding the parts of process algebra theory that engineering students find most difficult. Process-algebraic languages are replaced by a formalism of interacting state machines with local variables. Before introducing full abstraction, the behaviour of systems is discussed extensively at the operational, non-abstracted level. This connects the material to non-process-algebraic theories of concurrency, such as Petri nets or temporal logic verification of Promela-style parallel processes with Kripke structures, widening the usefulness of the material to the students.

1 Introduction

Process algebras such as Calculus of Communicating Systems (CCS) [7] and Communicating Sequential Processes (CSP) [9] are a major class of theories for concurrency. Their perhaps most attractive feature is full abstraction: the behaviour of a system, subsystem or individual process can be represented in an abstract way, so that precisely the information is preserved that is needed to determine the contribution of the subsystem to the behaviour of any larger system it is part of. This makes it possible to analyse the behaviour of the system compositionally and hierarchically, having applications in the design and verification of systems. Furthermore, distinguishing between "what" and "how" level descriptions is considered desirable in software engineering. Abstract behaviour is an ideal example of a "what"-level description of (some aspect of) a system. At a more philosophical level, abstract behaviour — or, actually, the existence of many different abstract behaviours — has improved our understanding on what we mean by the notion of "behaviour".

Process algebras have been popular among theoretical computer scientists, and there is even a standardised process-algebraic language [2], but practicing software engineers do not use them a lot. One reason for this was already alluded to: there are several process algebras and several semantics, and no clear guideline how to choose one. Furthermore, most process-algebraic languages use recursion as the main means of creating cyclic behaviour and manipulate data in the functional programming style. Most engineers find this unfamiliar, as they are used to doing the same things with loops and assignments.
Worse still, semantic theories of process algebras contain complicated issues whose relevance for system design is difficult to see. Such issues discourage practical engineers. For instance, in CSP, the semantics of recursive process definitions, and thus of loops, is defined via a highly mathematical fixed point theory. As another example, the most important way of representing program branches in process-algebraic languages is the so-called choice operator. The main notion of abstracted behaviour in CCS is “observational equivalence” or “weak bisimilarity”. This equivalence is not a congruence with respect to the choice operator. As a consequence, CCS literature introduces “observational congruence”, but talks very little of it. The reader is left confused about which one to use.

Yet another issue is that in process algebras, behaviour is talked about in terms of actions or state changes, whereas engineers often want to use state information, such as “in critical section”. The same information can be represented with actions “entering critical section” and “leaving critical section”, but it may be less natural. Furthermore, the most popular temporal logics are state-based.

On the other hand, software engineers are used to state machines. They have been extensively used for specifying reactive behaviour for decades, and they have been included in the Unified Modelling Language (UML) [8]. Furthermore, object-oriented thinking is very common, and an object can be naturally thought of as a state machine, with data attributes storing its state and methods implementing transitions. State machines also have an intuitive visual representation.

Unfortunately, the engineers’ notion of the interaction of state machines is usually ambiguous and far from clear. Often state machine notations do not tell with sufficient detail what happens when two state machines interact and, when they do, the chosen interaction primitives may be unsuitable for the application. As a consequence, engineers usually trust their intuition, use whatever interaction primitives they want, and ignore the formal semantics, if there is any. However, concurrency phenomena are difficult and intuition often fails.

This paper discusses an attempt to make process-algebraic full abstraction accessible to engineers by teaching it using interacting state machines. A second goal of the approach is to combine major ideas from several concurrency theories into one course in a natural way, thus emphasising what unifies different approaches to concurrency, and giving the students as widely applicable a background as possible.

The first part of the course focuses on the detailed (non-abstracted) operational semantics of interacting state machines. In this part, concepts such as unfolding of variables, deadlocks, livelocks, state spaces, Kripke structures, isomorphism and strong bisimilarity are introduced. Also compositionality results that hold at the non-abstracted level are presented. In addition to being useful in itself, this material is more concrete and therefore most likely easier for the students than the material on full abstraction. Presenting as much as possible in this part of the course is believed to ensure that the students have a strong basis when starting to study full abstraction.

The second part of the course is a fairly typical exposition of certain process-algebraic congruences and precongruences. In the implemented course, the focus
has been on the Chaos-Free Failures Divergence (CFFD) semantics [11], but
many other semantics could be used as well. Some comments on this part are
presented in Section 5. The majority of this paper concentrates on the less tra-
iditional first part of the course.

Section 2 presents first state machines with local variables and then their
interaction. Section 3 formalises them and gives their operational semantics.
Section 4 concentrates on the behaviour at the operational level. Please notice
that the discussion in these sections is not what the course presents to the
students. The purpose of the discussion is to list the material that is presented
to the students, and reveal the motivation for presenting — and in some cases
for not presenting — that material and in that form.

The course has never been taught in precisely the described form. The ap-
proach has been developing over a decade from a real-life course on the CFFD
semantics that originally used Lotos [2] and labelled transition systems for mod-
elling systems. The course is still changing. Its two most recent implementations
used state machines. In Tampere University of Technology, 12 students passed
the course in 1997, 13 in 1999, 6 in 2001, 6 in 2004, and 5 in 2006. The course
has also been taught in the University of Helsinki, with 6 passing in 2000 and 9
in 2002. Grading has been based on varying combinations of examinations and
weekly exercises. Statistics on admittance of new students to the Information
Technology Curriculum show a significant decrease in their mathematic skills
since 2000.

Experiences with the course are thus too heterogeneous and too few to make
firm conclusions. Even so, some observations on teaching it are mentioned in the
paper.

2 State Machine Notation

2.1 State Machines

Figures 1 and 2 show three examples of state machines. (It may be helpful to
have first a quick look at Figure 3 on page 6, to grasp the context of these state
machines.) We see the familiar convention of representing states as circles or
ellipses, and transitions as arrows. Initial states are denoted with arrows that
start from no state.

Although the figures show no example, multiple initial states are allowed.
This is because uninitialised variables are common in systems, and the unfold-
ing construction (Subsection 3.1) creates multiple initial states from uninitialised
variables. This decision did not come for free. The choice operator that is com-
monly used in process algebras for creating branching behaviour introduces a
congruence problem to many semantics. In CFFD, this problem was solved by
adding one bit to the semantics, recording whether the initial state can execute a
τ-transition. This solution breaks up when there are multiple initial states. As
a consequence, the traditional choice operator had to be discarded. Fortunately,
it is not an important operator in this framework, because branching behaviour
can be freely created by drawing the transitions of a state machine in the desired way. Furthermore, an action-guarded version of choice can be kept.

It is required that there is at least one initial state. Zero initial states would introduce a mathematically sound but from the real world point of view absurd “universal” state machine that would satisfy all linear temporal logic specifications!

All state machines in Figures 1 and 2 have local variables. The variables are listed and their types are given somewhere near the top of the figures. Excluding balance, variable names are repeated in some states. This is to indicate that the value of the variable only matters when the state machine is in those states. The variable balance is relevant in every state of BANK. Therefore, to avoid cluttering the drawing, it is not mentioned in the states. (This implies that it is not possible to indicate that a variable is not relevant in any state. However, such a variable would be superfluous as it has no effect to the behaviour of the system.) That a variable may be specified to exist only in a subset of states makes sometimes the operation of a state machine much easier to understand. As we will see in Subsection 3.1, it does not complicate much the theory.

The variable balance is initialised to 0 next to the initial state arrow. Other variables are not initialised there, because they do not exist in the initial states. They get their values when the state machine moves from a state without the variable to a state with the variable.

The values of variables may be input, output, computed with, and replaced by new values during the transitions. For instance, “request ?amount” denotes that
the value of amount is input while CASH DISPENSER executes a “request”-action. Similarly, CASH DISPENSER outputs the value during the subsequent “enquiry !amount”-transition. The transition labelled with “salary ?amount balance += amount” of BANK adds the input value to balance. “[amount ≤ balance]→” is an example of a guard, that is, a Boolean condition that must evaluate to True for the transition to be enabled. Transitions may also have postconditions that are evaluated with the values of variables after the transition. With postconditions one may specify, for instance, that the input value must be within a certain range.

FIFO2 uses the special notation “in? TYPE” to denote that a value is input but not stored anywhere. This is used to model loss of a message in the fifo. It also uses the notation “(TYPE)” to denote that it is actually not a single state machine but a state machine template, from which individual state machines are obtained by writing a concrete type in between the “(“ and “)”. Two examples of this are shown in Figure 3, using the natural numbers and {0, 1} as the types.

It is a useful exercise for the students to invent rules that specify appropriate handling of variables in transitions. For instance, if a variable is not defined in the start state of a transition but is defined in its end state, then it is reasonable to require that the transition either inputs to or assigns to the variable. It may also be reasonable to stipulate that altogether at most one input or assignment is applied to the variable in the same transition. Or perhaps one may allow several, the semantics being that all values that try to enter the variable must be the same, or the transition is not enabled?

For the purposes of analysing and verifying the behaviour of a state machine, it may have state propositions. For each state proposition and each state, there is a Boolean expression over the variables of that state that specifies when the state proposition evaluates to True. If the expression is just True, then it suffices to write the name of the state proposition in the state. If it is False, then it suffices to not write the name in the state. Otherwise one writes name[expression] in the state.

State propositions naturally turn state spaces into Kripke structures amenable to verification with typical temporal logic model checking techniques. State

```
balance : N
amount : N

BANK
balance := 0

salary?amount
balance += amount

enquiry

[amount ≤ balance]→
balance_ok

[amount > balance]→
bad_balance

balance -= amount

amount
```

Fig. 2. The BANK
propositions are second-class citizens in that they cannot be used for communication within the system. This decision was made because allowing the state machines to read each others’ state propositions would complicate the notation and the theory a lot, it is unclear how the details of such communication should be chosen (for instance, how to ensure that a proposition value does not change before other state machines have had enough time to see it?), and the corresponding communication needs can be easily met with the general mechanism explained in the next subsection.

The precise notation for representing guards, postconditions, initial and transition-time assignments, and state propositions is not important. The same holds for the template mechanism (which can be thought of as notational convenience that needs not be discussed further). In the implemented course, C++-style pseudocode has been used, because the students were familiar with C++. The pseudocode is only for showing examples and to offer a gentle start. In Subsection 3.1 the pseudocode will be replaced by an abstract mathematical construction.

2.2 Synchronous Interaction

Figure 3 shows how the example system is put together. (The gray texts are not part of the “official” notation. They are included to make the figure more informative.) Each state machine has a set of gates. For instance, the gates of BANK are “salary”, “enquiry”, “balance_ok” and “bad_balance”.

All transitions in the state machines on Figures 1 and 2 contain a gate name somewhere in their label. It is also possible to write the special symbol $\tau$ instead of a gate name. It specifies that the transition is internal to a state machine. Neither other state machines nor the environment of the system sees executions of internal transitions. Such transitions are called invisible.
If a line connects a gate of a state machine to a gate of another state machine, then the state machines must execute the corresponding transitions simultaneously. For instance, when BANK executes “balance-ok”, FIFO2(\{0,1\}) executes “in(0)” at the same time. Connections may branch, connecting more than two state machines together. If all ends of a connection have the same name, then it suffices to write the name once anywhere along the connection. If one end of a connection extends to the outside of the dashed line that encircles the state machines of the system, then also the environment of the system participates the transition. Otherwise the transition is invisible to the environment, and is labelled with $\tau$.

These graphical rules are just a means of specifying what we call synchronisation rules. They resemble the synchronisation vectors of Arnold [1], but allow labelling the resulting transitions. The idea will be made mathematically precise in Subsection 3.2. In Subsection 4.4 it will turn out that the same effect as with synchronisation rules can be obtained with the conventional hiding, relational renaming, and alphabet-based parallel composition operators of process algebras. Synchronisation rules were adopted in the course because they and their graphical representation are easier to write and understand than the corresponding structures built with conventional operators. Furthermore, the graphical representation highlights naturally that parallel composition is commutative and associative (modulo isomorphism).

The meaning of “$\exists$”- and “?$”-attributes in transition labels is the one typically used in process-algebraic languages. They denote that data is transported (or at least shown) during the transition. A transition may have any number and combination of “$\exists$”- and “?$”-attributes. An “$\exists$”-attribute is of the form “expression”, and it denotes that the state machine insists on the value of the expression (evaluated using the values of the variables at the start state of the transition) to occur in that position. “?$” is followed by a variable name. It tells that the state machine accepts any value from the type of the variable (unless the value is restricted with a postcondition), and stores the actual value to the variable.

All state machines that participate the transition must agree on the total number of the attributes (except where the attribute mechanism has been overridden by a “(\cdots)” in the synchronisation rule). Furthermore, for each attribute position, they must agree on the value. However, one state machine may use “$\exists$” in a position, while another one uses “?$” in the same position. As a matter of fact, the typical situation is that one state machine determines — that is, outputs — the value with “$\exists$”, while all others input the value with “?$”. Even so, any combination of “$\exists$” and “?$” in the same attribute position is allowed. If there is no value that fits all the attributes in the same position, then the transition is not enabled. If there are several fitting values (and if also all other attribute positions have a match), then an arbitrary one of them is chosen.

One can mix input and output in the same transition by using both a “?$”- and an “$\exists$”-attribute. Furthermore, with postconditions one can create an interaction which confuses input and output in the same attribute position. If a transition of one state machine is labelled with “negotiate?$n \geq 1$” and a transition of
another with “negotiate?m [m \leq 4]”, then a “negotiate”-transition occurs with a value between 1 and 4, and that value is stored to both n and m. The situation is so symmetric that neither state machine can be said to “output” nor “input” the value. Input and output are thus just restricted roles in an interaction, and there are forms of interaction where they are not meaningful terms.

A state machine that is ready for an interaction need not commit to it. That is, it may have alternative transitions from the same state. There are no restrictions in this respect. Therefore, a state machine may be simultaneously attempting interaction via several gates, or even several alternative interactions via the same gate. The semantics is that of all enabled transitions of the system as a whole, an arbitrary one is chosen.

This interaction mechanism is very general and quite different from those used in real systems. It is next to impossible to implement in its full generality, if the interacting state machines are physically separated. Why was it chosen for the course?

It was chosen for the course, because extensive experience by several researchers has shown that all real-life communication and interaction mechanisms can be easily modelled with it. For instance, Figure 3 and the right hand side of Figure 1 show how an unreliable fifo with capacity two can be modelled. As another example, a global variable can be interpreted as a state machine in its own right, and synchronisation rules can be written to link its operations “+1”, “= 0”, etc., with the appropriate actions of the state machines that use it. Broadcasting is not a problem, thanks to the generality of synchronisation rules. On the other hand, if any of the commonly used mechanisms were chosen for the course (such as the unbounded fifos of the SDL language [4]), then it would have been difficult or impossible to apply the theory to other mechanisms (such as bounded fifos with sending blocked when the fifo is full). Synchronous interaction thus appears to be the most fundamental form of interaction.

Unrestricted synchronous interaction in concurrency theory is like atoms and molecules in hardware engineering: although practical engineering almost always operates with coarser structures, atoms and molecules are the theoretical basis on which fundamental understanding of the exploited phenomena lays. This is an instance of one of the design principles of this course in action:

---

**Course design principle**

The goal of the course is more in understanding concurrency phenomena than in learning to design concurrent systems.

The motivation behind this principle is that the field of concurrent programming is rather unorganised at the moment, with various programming principles (semaphores, monitors, remote procedure calls, rendez-vous, fifo queues, threads and signals, ...). This is remarkably different to sequential programming, where the ideas of sequence, choice (if, switch) and bounded and unbounded repetition (for, while) are ubiquitous and have almost put the once so common goto-statement into oblivion. Similar development is clearly going on in object-oriented programming and design. In the absence of uniformly accepted concurrent programming concepts, it is probably fruitful to take a more theoretical
view and concentrate on concepts and phenomena that are more or less explicitly found in many concurrency theories.

--- Grand ideas of concurrency ---

Synchronous interaction is the fundamental form of interaction, in terms of which apparently all forms of communication and interaction can be modelled. Input and output are only roles in certain forms of interaction, and are not always meaningful concepts.

The little experience that there is of teaching the course indicates that students have no problems in understanding general synchronous interaction and its universality.

In addition to synchronisation rules, there may be state proposition rules. They will be described towards the end of Subsection 3.2.

3 Formalising the Notation

3.1 Arc Relations

Up to this point the presentation in the course has been informal. The state machine notation has been presented as a kind of a pseudocode. Mathematical definitions start in this subsection. The first task is to replace the programming-language like notation that was used for specifying variables and their manipulation by something more suitable for the development of the theory.

During a transition, there are three sets of values: old variable values at the beginning of the transition, new variable values at the end, and the values communicated in “!”- and “?”-attributes. The guard and postcondition constitute a Boolean condition that determines whether the transition is enabled. It may first seem that the values output in “!”-attributes are determined as partial functions of the old variable values (where the partial function is defined precisely when the transition is enabled), and the new variable values are partial functions of old variable values and the values input in “?”-attributes. However, sooner or later the need arises for assigning a randomly generated value.

For instance, a random number can be indirectly assigned to a variable by inputting it with “?” and restricting its range with a suitable postcondition, and then hiding the action in question. As a consequence, functions and partial functions do not suffice for fully explaining the manipulation of variables in transitions of combined state machines. To support the idea that any combination of state machines can be represented as a single state machine, random assignments must be allowed in the formalism.

Therefore, variable manipulation is modelled mathematically as a relation. Let \( V \) denote the number of variables. Let the variables be listed in some arbitrary order, and let their types be \( T_1, T_2, \ldots, T_V \). To model that a variable is not used in a state, a special symbol “⊥” that does not belong to any type is used as the value of unused variables. For \( 1 \leq i \leq V \), let \( T'_i = \emptyset \) if the \( i \)th variable is used in all states, and \( T'_i = T_i \cup \{⊥\} \) otherwise. Then \( T'_1 \times \cdots \times T'_V \)
is (a superset of) the set of all possible combinations of variable values. We will denote it with $VC$. Let $U$ be the union of all types used in the attribute positions of the transitions of the state machines in question. The manipulation of variables in a transition corresponds to a relation $R \subseteq VC \times U^k \times VC$, where $k$ is the number of attribute positions in the transition. We call $R$ the arc relation.

Now transitions can be defined as 4-tuples $(s, a, R, s')$, where $s$ and $s'$ are the start and end state of the transition, $a$ is its gate name or $\tau$, and $R$ is the arc relation. To complete the construction, for each initial state $\hat{s}$ a predicate $Init^{\hat{s}}$ is introduced that specifies the combinations of values that variables may initially have. The same information can be equivalently represented as a relation $Init \subseteq \hat{S} \times VC$. It is required that $Init^{\hat{s}}$ holds for at least one variable value combination. Furthermore, it is required that arc relations and $Init^{\hat{s}}$ insist that the value of a variable is “$\perp$” in the states where the variable is not defined, and it is not “$\perp$” in the remaining states.

Arc relations make it possible to combine transitions. If there are two transitions $(s, a, R, s')$ and $(s, a, R_2, s')$ between the same states with the same gate name or $\tau$, and if $R_1$ and $R_2$ have the same arity, then the behaviour stays the same if they are combined to the one transition $(s, a, R_1 \cup R_2, s')$. Combination of synchronising transitions of parallel state machines will be discussed in Subsection 4.1.

We have completely liberated our theory from concrete syntax of variables and their manipulation, and replaced it all by abstract relations. We have thus fully separated the theory of programming language syntax and semantics from our theory of interacting state machines. This is good, because issues on programming languages do not belong to this course but other courses. We can still use the pseudocode in examples, though. For that purpose, any programming notation can be used which can be understood as representing relations between old and new variable values and the values in attribute positions.

---

**Course design principle**

Don’t burden this course with issues that belong to other courses. Inherit those issues from the other courses, preferably via a clean interface.

Arc relations are a clean interface for connecting one’s favourite sequential programming notation to state machines.

A state machine can now be formally defined as the tuple

$$(S, VC, \Sigma, \Delta, \hat{S}, Init, II, val)$$

where $S$ is the set of states, $VC$ is the set of variable value combinations (see above), $\Sigma$ is the set of gates, and $\Delta$ is the set of transitions. Each transition is of the form $(s, a, R, s')$, where $s \in S$, $s' \in S$, $a \in \Sigma \cup \{\tau\}$, and $R$ is a relation over $VC \times U^k \times VC$ where $U$ is the set of all data values (see above) and $k$ is some natural number (if $a = \tau$ then $k = 0$). $\hat{S} \subseteq S$ is the set of initial states and $\hat{S} \neq \emptyset$. $Init \subseteq \hat{S} \times VC$ specifies the initial values of variables in each initial state, and for each initial state, and $\forall \hat{s} \in \hat{S}: \exists (v_1, \ldots, v_Y) \in VC : (\hat{s}, (v_1, \ldots, v_Y)) \in Init$, that is, at least one value combination is possible for each initial state. $II$ is the set of state propositions, and $val \subseteq II \times S \times VC$. 
If there are no variables, then the state machine reduces to a 6-tuple 
$(S, \Sigma, \Delta, S, \Pi, \text{val})$ such that $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$, $\emptyset \neq S \subseteq S$ and $\text{val} \subseteq \Pi \times S$.

A state machine may execute a transition from state $s$ with variable values $v_1, \ldots, v_V$ to state $s'$ with variable values $v'_1, \ldots, v'_V$, showing attribute values $d_1, \ldots, d_k$ in gate $a$ if and only if there is a transition $(s, a, R, s')$ such that $R(v_1, \ldots, v_V, d_1, \ldots, d_k, v'_1, \ldots, v'_V)$ holds. This is written as

$$s(v_1, \ldots, v_V) \rightarrow a(d_1, \ldots, d_k) \rightarrow s'(v'_1, \ldots, v'_V)$$

or, using a vector notation, $s(v) \rightarrow a(d) \rightarrow s'(v')$. If $k = 0$ (as is always the case when $a = \tau$) then the “$|$” may be discarded, resulting in $s(v) \rightarrow \rightarrow s'(v')$, and similarly if $V = 0$.

Each state machine with variables defines an equivalently behaving state machine without variables via a procedure that is known as unfolding. The initial states $S_0$ of the result are those $s(v)$ for which $(s, v) \in \text{Init}$. The alphabet of the result is $\Sigma_U = \Sigma \cup \tau^*$, that is, gate names extended with all possible sequences of communicated values. The reachable states $S_U$ (that are of the form $s(v)$ where $s \in S$ and $v \in V^C$) and the transitions $\Delta_U$ (that are of the form $s \rightarrow a \rightarrow s'$ where $s \in S_U$, $a \in \Sigma_U \cup \{\tau\}$ and $s' \in S_U$) are determined as was just described. $\Pi_U$ is simply $\Pi$, and $\text{val}_U = \{ (\tau, s(v)) \mid (\tau, s, v) \in \text{val} \}$. The results is a 6-tuple $(S_U, \Sigma_U, \Delta_U, S_U, \Pi_U, \text{val}_U)$ that satisfies the definition of a state machine without variables presented above.

---

**Grand ideas of concurrency**

The behaviour of a state machine with variables is a state machine without variables.

The importance of this observation is that state machines and their behaviours may be interchanged in many contexts. This is the first in a series of observations that make it possible to do the unfolding and parallel composition in almost any ordering.

### 3.2 Synchronisation Rules

Synchronisation rules were described informally in Subsection 2.2. Mathematically, a system is described by giving its component state machines, a set of gates (these will be the gates of the system as a whole), a set of synchronisation rules and a set of proposition rules.

A synchronisation rule for state machines $P_1, \ldots, P_n$ is a vector $\langle x_1, \ldots, x_n; x \rangle$. Each $x_i$ is either the special symbol “$|$” or of the form $g_i$ or $g_i(d_i)$, where $g_i$ is a gate of $P_i$, and $d_i$ is any sequence of data values. Likewise, $x$ is either $\tau$ or of the form $g$ or $g(d)$, where $g$ is a gate of the system as a whole. To avoid pathological situations, $x = g$ is only allowed when at least one $x_i$ is $g_i$.

Let first $x = g$. Let $d'$ be any sequence of data values. The system may make the transition

$$\left(s_1(v_1), \ldots, s_n(v_n)\right) - g(d') \rightarrow \left(s'_1(v'_1), \ldots, s'_n(v'_n)\right)$$
according to the synchronisation rule if and only if $s_i' = s_i$ and $v_i' = v_i$ for those $1 \leq i \leq n$ for which $x_i = \sim$, $s_i(v_i) - \delta_i(d_i) \rightarrow s_i'(v_i')$ for those $i$ for which $x_i = g_i(d_i)$, and $s_i(v_i) - \delta_i(d') \rightarrow s_i'(v_i')$ for the remaining $i$. In other words, those state machines that have $x_i = g_i$ participate the transition with common data values, facilitating communication. Those that have $x_i = g_i(d_i)$ use their own data values (which means no data values at all when $d_i$ is empty), and do not insist that others use the same data values. This facilitates the moving of information from data values to gate names and vice versa. It was used with the lower FIFO2 in Figure 3.

If $x = g(d)$ or $x = \tau$, then the rule is otherwise the same, but the label of the global transition is not $g(d')$ but $g(d)$ or $\tau$. The transition exists if a suitable $d'$ exists.

In addition to the transitions created by synchronisation rules, the system may make the transition

$$(s_1(v_1), \ldots, s_n(v_n)) \rightarrow \tau \rightarrow (s'_1(v'_1), \ldots, s'_n(v'_n))$$

such that $s'_i(v'_i) \rightarrow \tau$ for one $1 \leq i \leq n$, and $s'_i = s_i$ and $v'_i = v_i$ for the remaining $i$.

State proposition rules are of the form $name := expression$. The expression is a Boolean expression whose variable symbols are of the form $\pi_i$; where $\pi$ is a state proposition of $P_i$. The "$i$" distinguishes the state proposition from state propositions of other state machines that possibly have the same name. The rule introduces a new state proposition for the system and specifies its value as a function of the values of state propositions of the component state machines.

4 Detailed Behaviour of State Machines

4.1 Global State and State Space

The concept of the state space is presented at this point of the course. The construction of the state space is similar to typical constructions in temporal logic and, e.g., SPIN literature [5], with the exception of an additional $\Sigma$ component. The initial global states $\hat{S}_G$ of the system are those $(\hat{s}_1(v_1), \ldots, \hat{s}_n(v_n))$ for which $\hat{s}_i \in S_i$ and $(\hat{s}_i, v_i) \subseteq Init_i$ holds for every $1 \leq i \leq n$. The alphabet is $\Sigma_G = \Sigma \times U'$, where $\Sigma$ is the (explicitly given) set of gates of the system as a whole. The transitions $\Delta_G$ and reachable states $S_G$ are determined by the rules described in Subsection 3.2. Likewise, $H_G$ and $val_G$ are determined by state proposition rules. The result is a 6-tuple $(S_G, \Sigma_G, \Delta_G, \hat{S}_G, H_G, val_G)$. Again, it is a state machine without variables.

---

**Grand ideas of concurrency**

The behaviour of a system of interacting state machines (which may have local variables) is a state machine without variables.

An important observation is that the state space (or an isomorphic entity) can also be obtained in two stages, and even in two different orderings. First,
one may first unfold each component state machine and then combine them into a system and compute its state space. The global set of gates and the synchronisation rules have to be adapted to the absence of data attributes and presence of the same information in gate names, but this is not difficult.

Second, and this seems little known, it is also possible to first combine the state machines according to the synchronisation rules, and then do the unfolding. The \( VC \) of the result of the first stage is \( VC_1 \times \cdots \times VC_n \), modelling the idea that the variables of the global system are obtained by taking all local variables of the components. The \( Init \) of the result is the conjunction of the \( Init \) (interpreted as predicates). Transitions \( (s_i, a_i, R_i, s'_i) \in \Delta_i \) are combined according to the synchronisation rules such that the \( R_i \) components are combined as follows. Consider the synchronisation rule \( \langle x_1, \ldots, x_n; x \rangle \). Let \( R'_i(v_i, d'_i, v'_i) \) be \( v'_i = v_i \), if \( x_i = x \); \( R_i(v_i, d'_i, v'_i) \), if \( x_i = g_i \); and \( R_i(v_i, d_i, v'_i) \), if \( x_i = g_i(d_i) \). In the latter two cases \( R'_i = false \), if the middle parameter of \( R_i \) does not match the arity. We have \( R(v_1, \ldots, v_n, d', v'_1, \ldots, v'_n) \) is \( \bigwedge_{i=1}^n R'_i(v_i, d'_i, v'_i) \), if \( x = g \); \( d' = d \land \exists d'^0 : \bigwedge_{i=1}^n R'_i(v_i, d'^0, v'_i) \), if \( x = g(d) \); and \( \exists d'^0 : \bigwedge_{i=1}^n R'_i(v_i, d'^0, v'_i) \), if \( x = \tau \). The construction of the remaining components is more or less obvious.

Thus any combination of state machines with variables can be represented as a single state machine with variables. This result makes it possible, at least in principle, to compute parallel compositions at the structural or program code level, reducing the number of parallel processes in a system design. Although the formulae for combining the \( R_i \) may seem daunting, students of the course have found it easy to do the same at the level of the pseudocode used in Section 2. This indicates that the construction is actually intuitive.

---

**Grand ideas of concurrency**

\[ Unfold(P_1) \parallel \cdots \parallel Unfold(P_n) = Unfold(P_1) \parallel \cdots \parallel P_n \]

The fundamental reason behind this fact is that variables contribute to the global state mathematically in the same way as parallel processes. It may seem that there is a big difference in how a state machine skeleton interacts with — we usually say “uses” — its local variable and how it interacts with another state machine, but the generality of our synchronisation mechanism has more or less faded this difference away. Therefore, unfolding is actually the act of computing the parallel composition of the state machine skeleton with the variables in question. Parallel composition is commutative and associative (modulo isomorphism). Therefore, the ordering in which parallel composition is computed is insignificant. The above formula is just an instance of changing the ordering.

---

**Grand ideas of concurrency**

Use of a variable is parallel composition with it (with appropriate synchronisation rules).

4.2 Concepts of Behaviour

In this subsection deadlocks, livelocks, stuttering, and similar concepts are introduced and their significance is discussed.
4.3 When Are Two State Spaces Equivalent?

In this subsection strong bisimilarity of state machines without variables is defined and compared to identity and isomorphism. The idea that strong bisimilarity is a much better candidate for the notion of “same behaviour” than isomorphism has not been immediately obvious to the students. However, students accept it when the effect of remnant variable values to state spaces is shown. Consider the CASH DISPENSER of Figure 1, and its variant where amount is defined in every state.

Most students immediately agree that the value of amount in the initial and three other states is irrelevant, because the next thing that is done with amount is that a new value is read into it, destroying the old value. Therefore, the original and mutated CASH DISPENSER behave in the same way. However, their state spaces are not isomorphic. On the other hand, they are strongly bisimilar.

4.4 Simpler System Composition Operators

The synchronisation mechanism of Subsection 3.2 is strong and easy to use when modelling systems, but it is also complicated to analyse mathematically. Fortunately, it can be returned to three simple operators that are commonly used in process algebras: alphabet-based parallel composition “||”, hiding “\”, and relational renaming (also known as multiple renaming). The idea is to give a unique action name to each (unfolded) synchronisation rule, rename the gates of the local state machines to match the appropriate rules, and finally hide or rename the rule name to its resulting action. The details can be found in [6].

---------- Course design principle ----------

When presenting theory, prefer simple ideas that may be impractical as such, but in terms of which practical ideas may be easily represented.

Next, a series of theorems of the type “\((P \setminus A) \setminus B = P \setminus (A \cup B)\)" and “if \(A\) has no common elements with the alphabet of \(Q\), then \((P || Q) \setminus A = (P \setminus A) || Q\)" are given. It is shown how these theorems and earlier results of the course can be used to re-organise the structure of the system such that hiding, parallel composition and renaming — and unfolding! — alternate in the desired manner. This prepares for the compositional analysis later in the course.

5 Continuing the Course

By this point a strong foundation has been laid, emphasising compositionality at the level of isomorphic behaviours. The teacher can continue with process-algebraic equivalences, preorders, full abstraction and reduction algorithms, or with Kripke-structure-based temporal-logic-type verification — or with both! A state machine without variables can be converted to a Kripke structure by removing the \(\Sigma_x\)-component and the labels of transitions (and by adding to each deadlock state a local transition to itself, because most temporal logics
require that deadlocks are replaced by infinite stuttering). Alternatively, it can be converted to a labelled transition system by removing the $II_{\xi}$- and $\nu_{\xi}$-components.

Stable failures become more understandable to the students if they are represented as the minimal extension to the notion of “deadlock” that has to be made to guarantee compositionality.

Process-algebraic reduction of state spaces of subsystems is useful also in Kripke-structure-based verification. The easiest way of doing it is to divide the component state machines to two categories, those with and those without state propositions, and only apply reductions to the latter [10]. If the congruence preserved in the reductions is a certain divergence-preserving variant of branching bisimilarity, then the validity of stuttering-insensitive CTL formulae is preserved [3]. If it is the CFFD or the NDFD equivalence [11], then the validity of stuttering-insensitive LTL formulae is preserved. What is more, NDFD is the weakest congruence with this property. If $P \leq_{\text{NDFD}} Q$ and $Q \models \varphi$, then $P \models \varphi$.

There are also results on adding state propositions to the semantics. The “$\equiv_{\varphi}$”-relation can be made to keep track of the values of state propositions by replacing the actions $a$ with pairs $(a, D)$, where $D$ is the set of state propositions whose values change during the transition. The role of invisible action is taken by $(\tau, \emptyset)$. The notion of trace is extended with the initial values of the state propositions. In stable failures, the refusal set is still a subset of visible actions, because the $D$-component does not affect the refusal or acceptance. This idea seems fundamentally healthy, but it has not yet worked fully satisfactorily in the course, because the mathematics has become slightly too complicated.

References