External Behaviour of Systems of State Machines with Variables

Antti Valmari

Tampere University of Technology, Department of Mathematics
P.O. Box 553, FI-33101 Tampere, Finland
Antti.Valmari@tut.fi

Abstract. This tutorial is an introduction to compositionality and externally observable behaviour. To make it easier to understand system descriptions, traditional process-algebraic languages have been replaced by state machines represented as annotated directed graphs. Emphasis is on a novel way of treating local variables, and on the Chaos-Free Failures Divergences semantics. Even so, big themes that are not tied to any particular semantics are pointed out where possible. Other semantic models are introduced briefly. Most important verification methods facilitated by compositionality are mentioned with pointers to literature. Mathematical details are given less attention but not left out altogether. Throughout the tutorial, important principles are summarized in framed pieces of text.

1 Introduction

External behaviour, or *externally observable behaviour*, is the behaviour of an entity as seen at its interface, without seeing inside the entity or its other interfaces. For instance, when withdrawing cash from an automated teller machine (ATM), the user enters her bank card into the slot, types something, gets or does not get money, and gets the card back, and later she sees from the statement that the balance of her account has reduced accordingly. The user does not see the telecommunication between the ATM and the central computer of the bank, and she does not see that the computer checked whether her account had enough money. For another instance, a C++ program sees the C++ standard library `std::map` as something to which (key, value)-pairs can be added, accessed via the key, and removed, but the program does not see the red/black-tree operations that take place inside.

Computer science and software engineering favour abstraction and the description of things in implementation-independent ways. In the case of data structures, this desire has led to the development of abstract data types and specification methods for them, such as algebraic data types. Things become much more difficult with concurrent systems (we will see in Sect. 5.7 that nondeterminism is the culprit). This has led to the development of hundreds of different notions of external behaviour of concurrent systems. Fortunately, there are many common ideas and unifying themes. This tutorial is an introduction

K. Jensen et al. (Eds.): ToPNoC VII, LNCS 7480, pp. 255–299, 2013.
© Springer-Verlag Berlin Heidelberg 2013

to central insights in theories of the externally observable behaviour of concurrent systems. Many of the presented ideas are very well-known, but we will also discuss some that seem less well-known.

Every theory and every tutorial has its limits. This one concentrates on interleaving concurrency and lacks the aspects of real-time and probabilities. One might also complain that only the process-algebraic view is presented. However, this is because there seem to be no alternatives: the vast majority of articles on the external behaviour of concurrent systems are either openly or implicitly based on process-algebraic ideas. It seems to the present author that although the process-algebraic languages are definitely artificial and can be replaced by other notations (as we will do), at the semantic level process algebras have found something universally valid. On the other hand, although the theory of recursive process expressions has received a lot of attention in the literature, it is essentially absent from this tutorial. This is because it is not needed for discussing compositionality and external behaviour. It is also quite difficult.

There are many extensive treatments on process algebras, including [13,20,23,25]. Also the present author has published two tutorials [30,34] and touched the topic in [32]. It is reasonable to ask whether the world needs yet another one. After reading many papers and submissions over the years, it seems to the present author that many researchers try to re-invent results in the field, without realizing that a lot exists already. Perhaps this is because external behaviour, and its close friend compositional analysis, are very natural and desirable goals; but much of the literature on process algebras seems, at the first sight, to present hard theories with a narrow scope instead of material that the reader could apply to her own situation.

This suggests that there is a need for a tutorial that presents the big picture or roadmap in an easily accessible way, without *unnecessarily* requiring its readers to dwell in tricky details that process-algebraic theories abound. (This does not mean that tricky details could be avoided altogether.) This tutorial tries to be such. The readers will decide whether it succeeds.

Writing this tutorial also gave the chance to discuss some ideas that have received little attention although the present author believes that they are fundamental. Finally, the treatment of local variables of state machines in this tutorial has not been published before, excluding some sketchy preliminary versions. It is largely similar to well-known approaches, but uses so-called “data manipulation relations” to separate semantics from syntax. It is therefore given a lot of attention, while material that can be found in earlier tutorials is discussed more briefly.

Big themes in a research field tend to become apparent gradually. Often there is no well-defined first paper, where some idea has first been presented in a clear form. For instance, after many enough papers it has just become more or less common knowledge that alphabet-based synchronization is “the” fundamental notion of parallel composition, because it is simple and (as we will see in Sect. 4) universal in the sense that other common parallel composition operators can be constructed from it, but not always vice versa. For this reason, this tutorial does

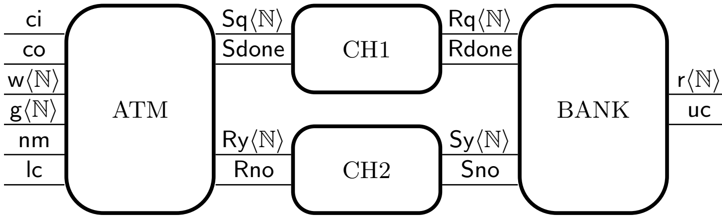


Fig. 1. A simplified cash dispenser system

not contain many references. Some of the given references point to recommended further reading and not necessarily to the original publications.

Section 2 presents the formalism that we will use for describing concurrent systems: state machines with local variables. It also presents a running example that will be used throughout this tutorial. The behaviour of a state machine is the topic of Sect. 3. The section also covers what it means for two behaviours to be equivalent at a detailed level. In Sect. 4 we will discuss the composition of a system from interacting state machines and the behaviour of the result. Behaviour at an abstract level is a central topic in process algebras. It is discussed in Sect. 5. The most well-known semantic models are introduced and CFFD-semantics is discussed in more detail. The section also briefly introduces verification techniques related to compositionality. Section 6 comments on the state of the art. Throughout the tutorial, important principles are informally summarized in boxes.

2 State Machines

In this tutorial, systems are presented as collections of interacting state machines. In this section we concentrate on individual state machines. We first illustrate them with the aid of an example system and then present a formal definition. In the meantime, we also comment on subtleties in the operation of the example system. Finally, we briefly introduce an issue that is of secondary importance for this tutorial but very important in the verification of concurrent systems in general: state propositions.

2.1 A Cash Dispenser System

Our example system is a simple cash dispenser system. Let us first discuss its overall design and operation. The system is shown in Fig. 1. It consists of an automated teller machine (ATM), a bank computer, and telecommunication channels between them. It models how a user can withdraw money and how that affects the balance of her account. To keep the example simple, we only model one user and account, and leave out many details such as user authentication.

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ denote the set of natural numbers. The operation starts when the user puts in her bank card (ci for card in). Then she types the amount

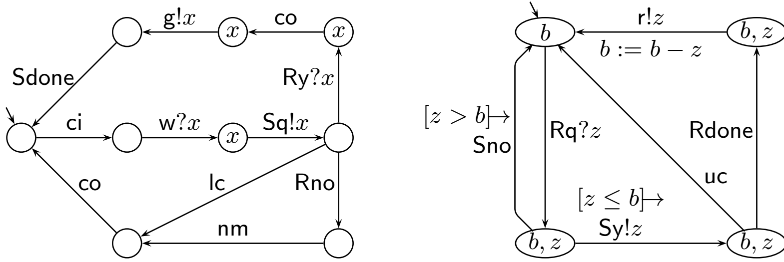


Fig. 2. The ATM and BANK state machines

of money she wants to withdraw (w⟨ℕ⟩). The amount is represented as a natural number parameter of w. Such parameters are called *event parameters* in this tutorial. Event parameters are thus the values that are communicated between state machines and/or the environment of the system during an event.

ATM sends the bank a query whether the user can withdraw that much money (Sq⟨ℕ⟩). CH1 either delivers the query to the bank (Rq⟨ℕ⟩) or loses it. The bank computer checks the balance of the account and sends back the answer "yes" or "no" via CH2. The yes-answer carries a number indicating the amount of money for a reason that we will explain in Sect. 2.3. Also CH2 may lose messages.

Depending on the answer, ATM either gives the money to the user (g⟨ℕ⟩) or replies that the user does not have enough money (nm). To prepare for losses of messages, ATM has a timeout mechanism that may make it abort the transaction and tell the user that connection was lost (lc). In any case, ATM gives the user the card back (co). If ATM gave the user the money, it informs the bank by sending the message "done". When the bank receives it, it reduces the balance of the account accordingly (r⟨ℕ⟩). If the bank waits for "done" in vain, a timer triggers and makes the bank record that the outcome of the transaction is uncertain (uc) and someone must go to the real physical ATM to check its transcript. The rationale of these details will be discussed in Sect. 2.3.

2.2 State Machines of the Cash Dispenser System

Figure 2 shows the ATM and BANK state machines. Our notion of a state machine is pretty much like a coloured Petri net with precisely one token and, consequently, precisely one input and output arc for each Petri net transition. The states (drawn as circles or ovals) correspond to Petri net places and the transitions (arrows) correspond to Petri net transitions and arcs. Together they show when the ci, w⟨20⟩, etc., events can take place. The initial state is indicated with a short arrow that does not start at a state. It corresponds to the Petri net place where the only token is initially.

Figure 1 has w⟨ℕ⟩, because it does not tell who decides the amount of money that is withdrawn, while Fig. 2 has w?x to indicate that ATM is ready for just any amount, and that amount will be stored in variable x of the next state.

Fig. 3. The CH1 and CH2 state machines

The execution of a transition is called *event*. In an event the former syntax is used and $\mathbb{N}$ has been replaced by a natural number, like in $\mathsf{w}\langle 20\rangle$. In the case of $\mathsf{g}!x$ the amount of money is determined by ATM and is, of course, the same as the value of x in the preceding state. In terms of coloured Petri nets, roughly speaking, in the case of ! the variable is present in the inscription of the input arc of the Petri net transition, while ? corresponds to the output arc. We will discuss ! and ? in more detail in Sect. 4.1.

The variable x only exists in some states. This is not fundamental, because we could extend the type of x by an additional value "undefined" and let x have that value in the remaining states, or we could just let x keep its most recent value when its value does not matter. On the other hand, drawing x into only some states makes it explicit when the value of x is and is not significant. This helps sometimes understanding systems. The issue is similar to a coloured Petri net token having some data component in some places and not having it in some other places. We will explain in Sect. 2.3 why x is not present in the start state of the lc-transition.

BANK has two variables, b for the balance and z for temporary storage. The notation $[z > b]\!\mapsto$ indicates that the transition is possible only if $z > b$, that is, the user asked for more money than her bank account has. Such conditions are called *guards*, and they correspond to guards in coloured Petri nets. Obviously $b := b - z$ models the updating of the balance. In coloured Petri nets, the same effect is obtained by specifying the value of a component of an outgoing token with a function on the output arc.

Figure 3 shows the channels. Here our notation is not optimal, because we had to draw the same theme twice in CH1, once for $\mathsf{q}\langle\mathbb{N}\rangle$ and once for done. However, this is notational inconvenience and not important for our goal that emphasizes semantics. The label τ is a special label that denotes that the execution of the transition is not directly observable by anything outside CH1. We say that τ is the *invisible action*. Here τ-transitions model losses of messages. CH2 is similar. It could be made from CH1 with the renaming operator of Sect. 4.3.

The examples demonstrate that state machines may have local variables and sequential computation with them. There are no shared variables in our formalism. This does not imply loss of generality, because one can represent shared variables as state machines with local variables. Indeed, CH1 and CH2 are examples of this. We will see an even more direct example in Fig. 7.

Interaction and communication between state machines belongs to Sect. 4.1, but the basic principle must be told here to be able to discuss the behaviour of the cash dispenser system as a whole. We say that ci, w, and so on are *gates*. Every state machine has an associated set of gates, and many state machines

may share the same gate. In Fig. 1, the gates of the cash dispenser system are shown by lines and their labels (excluding the "$\langle \mathbb{N} \rangle$"-parts).

To execute a transition whose label is or starts with a gate name, all state machines that are connected to that gate must execute a transition with that gate name simultaneously, and the numbers and values of event parameters must match. For instance, if ATM wants to execute $\mathsf{Sq}\langle 10 \rangle$, then also CH1 must execute $\mathsf{Sq}\langle 10 \rangle$. If CH1 is not ready to execute it, then ATM cannot execute it either. On the other hand, if a transition is labelled with τ, then it is executed by that state machine alone.

In terms of coloured Petri nets, this resembles the fusion of transitions that have the same gate name, except that τ-transitions are not fused. Furthermore, fusion is made in all combinations where precisely one transition with the given gate name is picked from each participating state machine. So, if A has two a-transitions, B has four, and C has three, then there are $2 \cdot 4 \cdot 3 = 24$ fused a-transitions. Finally, event parameters do not have a direct counterpart, although similar effects may be obtained by using the same coloured Petri net variable name in more than one fused transition.

The set of gates of a state machine must contain all gates referred to by the transitions of the state machine, and it may contain more. The presence of unused gates in the set of gates is significant, because it prevents the neighbour state machines from executing transitions with those gate names. One may argue whether this is an undesirable feature of the theory, but at least it is so difficult to change that it is better to leave it like that. This is because if the set of gates were defined as the set of used gates, one could cheat by adding an extra, always disabled transition with any desired gate name. It is more transparent to allow the user put extra gates to the set of gates if she wishes.

2.3 Remarks about the Behaviour of the Cash Dispenser System

We already discussed the two main sequences of events of the cash dispenser system: successful withdrawal of money and failure because of not having enough money. We also pointed out that messages may be lost, but ATM recovers from that by executing lc and BANK recovers with uc. The latter deserves a comment.

Ideally, we would like BANK to always get the right idea about whether the money was given to the user. Unfortunately, sending "yes" to ATM does not guarantee that the money is given, because "yes" may be lost in the channel, causing ATM to report loss of connection and not give the money. The "done" message prevents BANK from incorrectly reasoning that the money is given, but introduces the possibility of the opposite type of error, that is, incorrectly reasoning that the money was not given. It is common knowledge in telecommunication that in this kind of a situation, wrong conclusions cannot be fully avoided if a protocol that always eventually terminates is used. Wrong conclusions can be fully avoided if BANK may ask ATM about the outcome again and again until it receives an answer, but that may lead to a never-ending sequence of messages.

Fortunately, we can rule out one of the two types of wrong conclusions. We chose to prevent the one where ATM does not give the money but the account is charged. Furthermore, when the system gives the money without charging the account, the warning uc is always issued, so the people in the bank can go to the physical ATM, check the situation, and fix the balance afterwards. Unfortunately, there will also be false uc alarms.

Another subtle issue is the presence of the amount in the "yes" message and its absence in the start state of the lc-transition of ATM. Their purpose is to prevent the following and similar sequences of events. The user starts a transaction. The BANK computer is busy, and sends "yes" so slowly that the timer in ATM triggers and ATM executes lc before the "yes" arrives. The user reads the reply by ATM carelessly and thinks that she tried to withdraw too much money. Therefore, after getting the card back, she puts the card in again and types a new, smaller amount to withdraw. ATM sends the corresponding query, but it is lost. Next the delayed "yes" message arrives. If it lacked the amount information, ATM would interpret it as a "yes" to the new amount. So it gives the card and money, and sends "done". BANK receives "done" and charges the original amount from the account, which is different from what was given to the user.

So the purpose of the presence of the amount in the "yes" messages is to guarantee that the given amount is always the same as the charged amount (if it is charged). The start state of the lc-transition does not need the amount, because it is picked from the "yes" message.

The subtleties discussed above may make the reader wonder whether we have ruled out all errors. We have not, but fortunately there are verification tools.

2.4 Formal Definition of State Machines

In this subsection we will define state machines formally. To avoid having to define the language used in the guards and assignments of transitions, and to get a much more general theory than allowed by the simple notation that we have discussed, we will introduce abstract *data manipulation relations*. The !-, ?-, $[\cdots]\!\!\mapsto$-, and :=-notation will be interpreted as a handy practical representation for a subset of the data manipulation relations, useful for presenting examples.

Each state machine has the set of *types* used by it, denoted with Θ. Formally, a type is just a nonempty set. For instance, the set of natural numbers is a commonly used type. Each variable of each state has a type from Θ.

It would seem natural to give a type also to each event parameter of each transition, and we often do so in examples. However, it will become evident in Sect. 4 that it is better that the formal definition does not pay attention to the internal structure of the labels of events. Therefore, labels of events will be formalized as arbitrary symbols called *actions*, and types will not be needed for that. We have already mentioned that the invisible action τ has a special role. The remaining actions are *visible*. The set of them will be called the *alphabet* and denoted with Σ. We stipulate that $\tau \notin \Sigma$.

For instance, the alphabet of BANK could usually in practice be chosen as $\{ \mathsf{r}\langle 0\rangle, \mathsf{r}\langle 1\rangle, \mathsf{r}\langle 2\rangle, \ldots, \mathsf{uc}, \mathsf{Rq}\langle 0\rangle, \mathsf{Rq}\langle 1\rangle, \ldots, \mathsf{Rdone}, \mathsf{Sy}\langle 0\rangle, \mathsf{Sy}\langle 1\rangle, \ldots, \mathsf{Sno} \}$. However, this relies on the assumption that all transitions of all state machines have the correct number and types of event parameters. To see what might go wrong if this does not hold, assume that there is also a state machine that represents the user of the cash dispenser system. The user can execute $\mathsf{g}\langle 20\rangle$, that is, get 20 units of money, only when ATM is ready for that. However, if $\mathsf{g}\langle 20.5\rangle$ is in the alphabet of the user but not in the alphabets of the other state machines, then the user can get 20.5 units of money any time at will! Therefore, in the presence of event parameters, the precise alphabet of a state machine is $\Gamma \times \mathcal{U}^*$, where Γ denotes the set of gates and $\mathcal{U}$ denotes the set of all data values used by the system.

Referring to the names of variables in the formal definition would be clumsy. Therefore, for each state s, we assume that its variables are listed in some order and rely on their positions in this order. The types of the variables of s can now be specified as a Cartesian product $\mathcal{T}(s)$ of types. For instance, if s has variables n, b, and x of types `int`, `bool`, and `float`, and if we choose to list them in this order, then $\mathcal{T}(s) = \mathtt{int} \times \mathtt{bool} \times \mathtt{float}$. To handle states that have no variables, we denote the empty list of variable values with $\langle\rangle$. Thus, if s has no variables, then $\mathcal{T}(s) = \{\langle\rangle\}$. With $\Theta^\times$ we denote the set of all finite Cartesian products of types, that is, the set of all $T_1 \times \cdots \times T_n$ where n is a natural number and $T_i \in \Theta$ for $1 \leq i \leq n$. So $\mathcal{T}(s) \in \Theta^\times$, and $\{\langle\rangle\}$ is the element of $\Theta^\times$ that results from $n = 0$.

Of course, the state machine has a *set of states* S and an *initial state* $\hat{s}$. The *initial values* of the variables of the initial state are listed by $\hat{v}$. So $\hat{v} \in \mathcal{T}(\hat{s})$. In some applications, more than one initial state or more than one possible initial value for a variable would be useful, but we do not present that possibility in our formal definition, to avoid making it more complex.

The most complicated part is the set Δ of *transitions*. Each transition is a tuple (s, R, s'), where s is the start state and s' is the end state of the transition. R is the data manipulation relation and will be discussed next.

Let $v_1, \ldots, v_n$ denote the values of the variables of s before executing the transition. Similarly, let $w_1, \ldots, w_m$ be the values of the variables of s' after the transition. So $\langle v_1, \ldots, v_n\rangle \in \mathcal{T}(s)$ and $\langle w_1, \ldots, w_m\rangle \in \mathcal{T}(s')$. Let a be the action, that is, the label of the event. When the ?-, etc., notation is used, a is the gate name together with the values of the event parameters. The purpose of R is to express the dependency between $v_1, \ldots, v_n$, a, and $w_1, \ldots, w_m$. For instance, in the case of $\mathsf{a}!(v_2 + 1)?w_5$, R must say that $a = \mathsf{a}\langle p_1, p_2\rangle$, where $p_1 = v_2 + 1$ and $w_5 := p_2$. If the transition also has the guard $[v_1 \neq 1]\!\!\rightarrow$ and the assignment $w_4 := 2v_1$, then R must also reflect their effect.

A natural first idea would be to let R be a collection of partial functions of the gate name, $v_1, \ldots, v_n$, and those event parameters that are specified with ?. The functions would yield the values of all w_i and the remaining event parameters. They would be partial because of guards. However, sometimes modellers need nondeterministic operations, like the assignment of a random value to a variable.

Furthermore, some process-algebraic languages also feature conditions that test the inputted values. Adding these facilities to partial functions would make the formalism complicated. It is easier — and also more general — to let R be an arbitrary relation.

Therefore, R is defined as a relation on $\mathcal{T}(s) \times (\Sigma \cup \{\tau\}) \times \mathcal{T}(s')$. Assume that the state machine is in state s, and the values of the variables of s are $v_1, \ldots, v_n$. The state machine is ready to execute the transition with event name a to state s' yielding its variables the values $w_1, \ldots, w_m$ if and only if $R(v_1, \ldots, v_n, a, w_1, \ldots, w_m)$ holds. To avoid clumsy formulas, we abbreviate $v_1, \ldots, v_n$ and $w_1, \ldots, w_m$ to $\bar{v}$ and $\bar{w}$. There may be many a and $\bar{w}$ with which $R(\bar{v}, a, \bar{w})$ holds with the current values of the v_i. Most importantly, $?w_5$ at event parameter position 2 allows many values for the second event parameter and w_5, as long as they both have the same value. The transition may thus have many instances. Many instances of the same transition is the same thing as many bindings of a coloured Petri net transition.

The same behaviour can often be expressed as one transition or as many alternative transitions between the same states. That is, the transitions (s, R_1, s') and (s, R_2, s') yield together the same behaviour as the transition $(s, (R_1 \vee R_2), s')$. For instance, the bottommost transition of BANK could be replaced by two transitions labelled $[z < b] \!\!\mapsto \mathsf{Sy}!z$ and $[z = b] \!\!\mapsto \mathsf{Sy}!b$. It is not significant whether some behaviour is represented by one or more transitions.

We say that a data manipulation relation R is *empty* if and only if $R = \emptyset$, that is, $R \Leftrightarrow \mathsf{False}$. A transition with an empty R is never enabled. It could as well be removed from the state machine.

When expressing data manipulation relations mathematically using variable names, there is a problem: the previous and the next state may have a variable with the same name. For instance, each state of BANK has a variable called b. Therefore, to make it explicit whether we mean a $\bar{v}$-variable or a $\bar{w}$-variable, we write : after the name of the latter. The effect of the topmost transition of BANK on b can thus be written as $b - z = b{:}$, or, equivalently, $b{:} = b - z$. Considering : as a separate token we can use spaces differently and write $b := b - z$. This looks familiar and has the expected meaning. This is why we chose : as the "afterwards" specifier. In many other notations, the "afterwards" specifier is $'$, like in $b' = b - z$.

It is common that a variable inherits its value from a variable with the same name in the previous state. This could be expressed as $x{:} = x$, or as $x := x$. However, having to write and read a lot of that would be clumsy. Therefore, the ?-, etc., notation has an implicit assumption that if the value of a variable is not specified with ? or :=, and if also the previous state has a variable with the same name, then the variable gets the value of its namesake.

We are ready to present the formal definition.

Definition 1. *A* state machine *is a tuple* $(S, \Theta, \mathcal{T}, \Sigma, \Delta, \hat{s}, \hat{v})$*, where* S *is a set;* Θ *is a set of nonempty sets;* $\mathcal{T}$ *is a function from* S *to* $\Theta^{\times}$*;* $\hat{s} \in S$*;* $\hat{v} \in \mathcal{T}(\hat{s})$*;* Σ *is a set such that* $\tau \notin \Sigma$*; and* Δ *is a set of tuples of the form* (s, R, s')*, where* $s \in S$*,* $s' \in S$*, and* $R \subseteq \mathcal{T}(s) \times (\Sigma \cup \{\tau\}) \times \mathcal{T}(s')$*.*

In the sequel, we will have to talk about the *reachable part* of a state machine. It means the state machine induced by the states and transitions to which there are paths from the initial state. It is obvious that only the reachable part is relevant for the behaviour of the state machine. However, the reachable part is only an upper approximation to the relevant part, because, for instance, the guard of some transition may be equivalent to False. The precise relevant part is not necessarily easy to recognize. For instance, adding the guard $[z = b + 1]\!\mapsto$ to the rightmost transition of BANK would make the top right corner state of BANK irrelevant, but that is not immediately obvious.

Definition 2. *The* reachable part *of a state machine* $(S, \Theta, \mathcal{T}, \Sigma, \Delta, \hat{s}, \hat{v})$ *is the state machine* $(S', \Theta, \mathcal{T}', \Sigma, \Delta', \hat{s}, \hat{v})$*, where* $\mathcal{T}'$ *is the restriction of* $\mathcal{T}$ *to* S'*, and* S' *and* Δ' *are the smallest sets such that*

- $\hat{s} \in S'$*, and*
- *if* $s \in S'$ *and* $(s, R, s') \in \Delta$*, then* $s' \in S'$ *and* $(s, R, s') \in \Delta'$*.*

We will see in later sections that data manipulation relations are handy for the development of the theory. They liberate the concurrency part of the theory almost completely from concrete syntax and similar issues. They are usually naturally obtained from inscriptions written in languages for sequential computation. Their idea is more or less implicitly present in many concurrency formalisms, including coloured Petri nets. The following observation is at least implicitly known by many researchers. It is worth emphasizing.

Data manipulation relations are a handy way of combining the language for sequential computation to the theory of the behaviour of concurrent systems.

2.5 State Propositions

Many concurrency formalisms (especially temporal logics) associate logical propositions to states, such as "light is on" or "execution is in a critical section". They are called *state propositions*. We chose not to have state propositions in our definition, again to avoid complexity. However, we will comment on them in a couple of places.

In the absence of variables, they could be defined by associating to the state machine a set Π of state propositions, and a function $val : S \to 2^{\Pi}$ so that $val(s)$ is the set of state propositions that hold on s. In the presence of variables the definition becomes more complicated, because also their values may affect the truth value of a state proposition.

3 Concrete Behaviour

In this section we define the behaviour of a single state machine and demonstrate that it is essentially a state machine without variables. We also introduce bisimilarity and justify that it is a good notion for two behaviours to be equivalent at a detailed level.

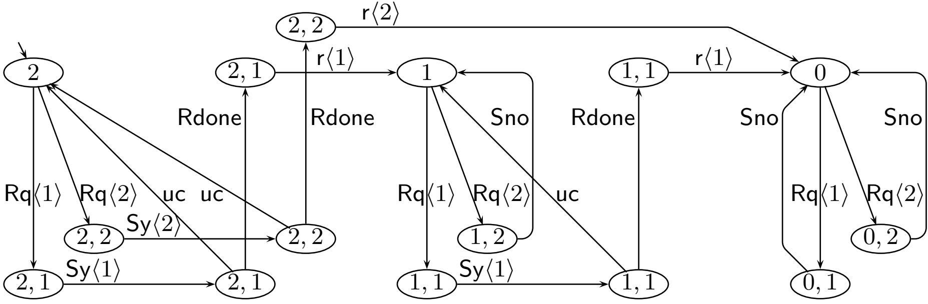


Fig. 4. BANK unfolded assuming that the types of b and z are $\{0, 1, 2\}$ and $\{1, 2\}$, and initially $b = 2$

3.1 Formal Definition of Behaviour: Unfolding

Mathematically, the behaviour of a state machine or system of state machines is represented as a *labelled transition system*, abbreviated *LTS*.

Definition 3. *A* labelled transition system *is a tuple* $(S, \Sigma, \Delta, \hat{s})$*, where* S *is a set,* Σ *is a set such that* $\tau \notin \Sigma$*,* $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$*, and* $\hat{s} \in S$*.*

The reachable part *of an LTS* $(S, \Sigma, \Delta, \hat{s})$ *is the LTS* $(S', \Sigma, \Delta', \hat{s})$*, where* S' *and* Δ' *are the smallest sets such that*

- $\hat{s} \in S'$*, and*
- *if* $s \in S'$ *and* $(s, a, s') \in \Delta$*, then* $s' \in S'$ *and* $(s, a, s') \in \Delta'$*.*

If Δ *is obvious from the context, then* $(s, a, s') \in \Delta$ *can also be written as* $s -a\rightarrow s'$*.*

The definition is thus otherwise the same as the definition of state machines, but there are no Θ, $\mathcal{T}$, and $\hat{v}$ components, and the elements of Δ have an a component instead of the R component. The components of the LTS have the same names as with state machines, that is, $\hat{s}$ is the initial state, and so on.

The behaviour of a state machine is obtained by *unfolding*. It resembles the unfolding of a coloured Petri net to a place/transition net, and similar operations are found also elsewhere in theoretical computer science. The states of the result could be represented as $(s, \bar{v})$, where s is a state of the state machine and $\bar{v}$ is the values of the variables of s. To avoid confusion with other tuple notation, we write $s\langle\bar{v}\rangle$ instead. We also treat s and $s\langle\rangle$ as synonyms. If the same s is encountered with different variable values $\bar{v}$ and $\bar{v}'$, then the result will have different states $s\langle\bar{v}\rangle$ and $s\langle\bar{v}'\rangle$. The initial state of the result is $\hat{s}\langle\hat{v}\rangle$. Only those $s\langle\bar{v}\rangle$ are included in the result that may be reached by executing the state machine starting at the initial state.

Figure 4 shows the behaviour of BANK with the types of variables replaced by so small sets that the figure can be drawn. To help reading, the states are

labelled with the $\bar{v}$ — that is, the values of b or b, z. For instance, the three end states of Rdone-transitions originate from the same state of BANK.

Let $s\langle\bar{v}\rangle$ be a state in the behaviour and (s, R, s') a transition of the state machine. It generates a transition from $s\langle\bar{v}\rangle$ to $s'\langle\bar{w}\rangle$ with action a to the behaviour if and only if $R(\bar{v}, a, \bar{w})$ holds. This transition is written as $(s\langle\bar{v}\rangle, a, s'\langle\bar{w}\rangle)$ or $s\langle\bar{v}\rangle -a\!\rightarrow s'\langle\bar{w}\rangle$. If there are event parameters and we want to show them, we write the transition as $(s\langle\bar{v}\rangle, a\langle\bar{p}\rangle, s'\langle\bar{w}\rangle)$ or $s\langle\bar{v}\rangle -a\langle\bar{p}\rangle\!\rightarrow s'\langle\bar{w}\rangle$, where $\bar{p}$ denotes the event parameters.

For instance, the Sy-transition of BANK is enabled only if $z \leq b$, so corresponding transitions start in Fig. 4 at those end states of Rq-transitions that are labelled with "$2, 1$", "$2, 2$", and "$1, 1$", but not at those labelled with "$1, 2$", "$0, 1$", and "$0, 2$". The transitions are labelled with $\mathsf{Sy}\langle 1\rangle$ and $\mathsf{Sy}\langle 2\rangle$.

Let us define the unfolding formally.

Definition 4. *Let $M = (S, \Theta, \mathcal{T}, \Sigma, \Delta, \hat{s}, \hat{v})$ be a state machine. Its* behaviour *$\mathcal{B}(M)$ is the reachable part of the LTS $(S', \Sigma, \Delta', \hat{s}')$, where*

- $S' = \{\, s\langle\bar{v}\rangle \mid s \in S \wedge \bar{v} \in \mathcal{T}(s) \,\}$;
- $\Delta' = \{\, (s\langle\bar{v}\rangle, a, s'\langle\bar{w}\rangle) \mid \exists R : R(\bar{v}, a, \bar{w}) \wedge (s, R, s') \in \Delta \,\}$; *and*
- $\hat{s}' = \hat{s}\langle\hat{v}\rangle$.

It follows almost immediately from the definitions that if a state machine has no variables, it is essentially its own behaviour. The biggest difference is that while the state machine formalism allows to represent a set of transitions in one bunch as (s, R, s'), the LTS formalism requires to present each of the transitions individually. In the absence of variables we can let $\Theta = \emptyset$. For convenience, we denote also the void $\mathcal{T}$ and $\hat{v}$ with $\emptyset$.

Proposition 5. *Let $(S, \emptyset, \emptyset, \Sigma, \Delta, \hat{s}, \emptyset)$ be a state machine without variables. Its behaviour is isomorphic to the reachable part of $(S, \Sigma, \Delta', \hat{s})$, where $\Delta' = \{\, (s, a, s') \mid \exists R : R(a) \wedge (s, R, s') \in \Delta \,\}$.*

Proof. In the absence of variables, $\hat{v}$ and each $\bar{v}$ collapse to the empty list of values. The states of the behaviour are thus of the form $s\langle\rangle$, where s is a state of the state machine. In particular, $\hat{s}' = \hat{s}\langle\rangle$. The data manipulation relations only have a-components. Therefore, the transition (s, R, s') introduces precisely the transitions $(s\langle\rangle, a, s'\langle\rangle)$ to the behaviour, where $R(a)$ holds. Finally, also the definition of behaviour has restriction to the reachable part. □

We also want to demonstrate that the behaviour of any state machine (possibly with variables) is essentially a state machine without variables. By definition, it is an LTS. It suffices to show that for each LTS, there is a state machine without variables whose behaviour is isomorphic to the reachable part of the LTS. We do that next. In the construction, a separate state machine transition is made from each LTS transition. The data manipulation relation of that transition is $\{a\}$, that is, the relation R such that $R(a)$ holds and $R(x)$ does not hold if $x \neq a$. The correctness of the proposition is obvious.

Proposition 6. *Let* $(S, \Sigma, \Delta, \hat{s})$ *be an LTS. Its reachable part is isomorphic to the behaviour of the state machine* $(S, \emptyset, \emptyset, \Sigma, \Delta', \hat{s}, \emptyset)$*, where* $\Delta' = \{ (s, \{a\}, s') \mid (s, a, s') \in \Delta \}$*.*

So the behaviours of state machines can be treated as state machines. This unification of state machines with their behaviours gives the theory a lot of power that we will enjoy in later sections.

The behaviour of a state machine is essentially a state machine without variables, and the behaviour of a state machine without variables is essentially the state machine itself.

3.2 Equivalence of Detailed Behaviours

From the point of view of externally observable behaviour, labels of events (i.e., actions) are important, but names of states are not. If one wants to look at properties of states during verification, then one must either use the state propositions of Sect. 2.5, or reason the necessary properties from visible events. One may, for instance, introduce the actions enter and leave to verify that two state machines are not in their critical sections at the same time. We already utilised the insignificance of state names in the previous subsection, by considering two LTSs essentially the same if there is an isomorphism between their states.

However, isomorphism often fails to unify intuitively obvious instances of "same behaviour". For instance, consider Fig. 4. It seems clear that fusing the two states at bottom right corner (the start states of Sno-transitions) does not change the externally observable behaviour. Isomorphism cannot reflect that, because isomorphic LTSs always have the same number of states, and state fusion changes the number of states.

Bisimilarity is an equivalence notion that is much better than isomorphism in unifying intuitively equivalent behaviours, while it still avoids unifying behaviours when it should not unify. It is strictly weaker than isomorphism, that is, isomorphic LTSs are always bisimilar, but bisimilar LTSs are not always isomorphic. Many researchers consider bisimilarity as *the* notion of "same behaviour at the detailed level". On the other hand, from the point of view of externally observable behaviour, the detailed level is much less important than the abstract level that we will discuss in Sect. 5. Therefore, bisimilarity is not a goal in itself but only a useful technical tool on the way to the real goal. However, it is a tool that one must master to understand concurrency.

Bisimilarity is sometimes called *strong bisimilarity*, to distinguish it from "weak bisimilarity" that we will meet in Sect. 5.6.

Bisimilarity is defined between states. It could be defined between the sets of states of two LTSs, but it is more flexible to define it between the states of a single LTS, and apply it to two LTSs by first combining them into one LTS by taking their disjoint union. Intuitively, taking the disjoint union simply means drawing the two LTSs on the same sheet of paper and pretending that they are two isolated regions of the same LTS (whose initial state is chosen

arbitrarily). The formal definition of disjoint union is well-known and we skip it. Two LTSs are bisimilar if and only if their initial states are and they have the same alphabet.

To discuss the basic idea of bisimilarity, consider two bisimilar states s_1 and s_2. An essential phenomenon in concurrency is *nondeterminism*, that is, a state may have more than one output transition with the same action. The goal of the definition of bisimilarity is to guarantee that whatever output transitions a state has, the bisimilar state is able to simulate. That is, for each output transition $s_1 -a\rightarrow s_1'$ of s_1, s_2 must have an output transition with the same action a and, furthermore, the futures after the original and simulating transitions must be somehow the same. Let $s_2 -a\rightarrow s_2'$ be that transition. The equivalence of futures is ensured by building the definition so that also s_1' and s_2' will be bisimilar. Of course, it is also required that whatever output transitions s_2 has, s_1 must be able to simulate. However, it is not required that the mapping between transitions and their simulating transitions is one-to-one; that is, a transition may simulate and be simulated by many transitions.

The description above is not precise enough to qualify as a definition of bisimilarity. For instance, the description would allow us to declare that s_1 and s_2 are bisimilar if and only if $s_1 = s_2$, which would obviously be against our goal. Therefore, the formal definition of bisimilarity relies on the auxiliary notion of *bisimulation*. Bisimulation is any relation on states that satisfies the above description. It can be proven that there is a unique weakest bisimulation (it is the union of all bisimulations). This unique weakest bisimulation is the bisimilarity.

Definition 7. *Let $(S, \Sigma, \Delta, \hat{s})$ be an LTS. The relation "$\sim$" $\subseteq S \times S$ is a* bisimulation, *if and only if for every $s_1 \in S$, $s_2 \in S$, $s \in S$, and $a \in \Sigma \cup \{\tau\}$ such that $s_1 \sim s_2$ the following hold:*

- *If $s_1 -a\rightarrow s$, then there is $s' \in S$ such that $s_2 -a\rightarrow s'$ and $s \sim s'$.*
- *If $s_2 -a\rightarrow s$, then there is $s' \in S$ such that $s_1 -a\rightarrow s'$ and $s' \sim s$.*

We say that $s_1 \in S$ and $s_2 \in S$ are bisimilar, *if and only if there is a bisimulation "$\sim$" such that $s_1 \sim s_2$.*

We say that the LTSs $(S_1, \Sigma_1, \Delta_1, \hat{s}_1)$ and $(S_2, \Sigma_2, \Delta_2, \hat{s}_2)$ are bisimilar *if and only if $\Sigma_1 = \Sigma_2$, and $\hat{s}_1$ and $\hat{s}_2$ are bisimilar in the disjoint union of the LTSs.*

The definition of the bisimilarity of LTSs requires that their alphabets must be the same. This is because the alphabet determines whether the state machine may prevent other state machines from executing transitions, as was discussed in Sect. 2.2. It is thus essential from the point of view of neighbour LTSs.

If the formalism is extended with multiple initial states, then the condition that $\hat{s}_1$ and $\hat{s}_2$ are bisimilar must be replaced with the following condition: every initial state of the first LTS has a bisimilar initial state in the second LTS, and vice versa. If the formalism is extended with state propositions, then the requirement must be added that bisimilar states give the same truth values to state propositions. For simplicity, we will not take these extensions into account in the subsequent discussion.

The following result could be easily proven with induction. It says that the notion of simulation of single transitions given by the definition of bisimilarity extends to arbitrary sequences of transitions.

Proposition 8. *Let "$\sim$" denote bisimilarity. If* $s_0 -a_1\rightarrow s_1 -a_2\rightarrow \cdots -a_n\rightarrow s_n$ *and* $s_0 \sim s'_0$*, then there are* s'_1, s'_2, $\ldots$, s'_n *such that* $s_1 \sim s'_1$, $s_2 \sim s'_2$, $\ldots$, $s_n \sim s'_n$ *and* $s'_0 -a_1\rightarrow s'_1 -a_2\rightarrow \cdots -a_n\rightarrow s'_n$*. A similar result holds for infinite sequences* $s_0 -a_1\rightarrow s_1 -a_2\rightarrow \cdots$.

It is not difficult to prove that bisimilarity indeed is an equivalence. To prove that two states are bisimilar, it suffices to present a bisimulation that relates them. Bisimulations are not necessarily equivalences. In particular, the empty relation (the one where $s_1 \sim s_2$ never holds, no matter what s_1 and s_2 are) is a bisimulation. When checking whether two LTSs are bisimilar, only the reachable parts matter, because, so to speak, the empty relation can be used elsewhere. The empty relation cannot be used in the reachable parts because of the requirement that $\hat{s}_1 \sim \hat{s}_2$. From this, Proposition 8 implies that the relation must be nonempty throughout the reachable parts.

For any given finite LTS, there is a unique (modulo isomorphism) smallest bisimilar LTS. It can be found by leaving out the unreachable parts and fusing equivalent states in the reachable part. This can be done (at least in a mathematical sense) also to infinite LTSs, but the result cannot be called "smallest", because it is not a well-defined concept with infinite objects. The fusion operation is defined below. The Δ'-part of the definition goes through all transitions in Δ, but it would suffice to take one state in each $[[s]]$ and go through just their output transitions. This is because the definition of bisimilarity guarantees that if one state in an equivalence class has an a-transition to an equivalence class, then every state in the former equivalence class has an a-transition to the latter equivalence class.

Definition 9. *Let* $(S, \Sigma, \Delta, \hat{s})$ *be an LTS, and let "$\sim$" denote bisimilarity. Its* quotient *modulo bisimilarity is the LTS* $(S', \Sigma, \Delta', \hat{s}')$*, where*

- $[[s]] = \{ s' \mid s \sim s' \}$,
- $S' = \{ [[s]] \mid s \in S \}$,
- $\Delta' = \{ ([[s]], a, [[s']]) \mid (s, a, s') \in \Delta \}$, *and*
- $\hat{s}' = [[\hat{s}]]$.

In the case of finite LTSs, the above construction can be done in $O(|S| + |\Delta| \log |S|)$ time [35]. (The algorithm in [8] has been influential but does not meet this time bound.) This is fast enough for almost all practical purposes. The construction can also be used for checking whether two states or LTSs are bisimilar. This is a big difference from isomorphism, which is believed not to be checkable in worst-case polynomial time.

Bisimilarity is a most appropriate notion of "same behaviour" at the detailed level of behaviour.

4 Putting State Machines Together

In this section we discuss how a system is put together from state machines. We introduce three operators for that and then combine them into one flexible operator. We discuss how the behaviour of the system is determined as a function of the behaviours of its parts, and point out that the same behaviour can also be obtained by first putting the state machines together. Then we comment on the notions of input and output, and mention some other operators used in process algebras.

4.1 Parallel Composition

Many different parallel composition operators have been defined in the literature. The most suitable for our purpose can be called *alphabet-based synchronization.* We define it first for LTSs.

The operator works as we discussed towards the end of Sect. 2.2. Each state of the result is a tuple consisting of the states of the components. So the joint LTS keeps track of the states of the component LTSs. The joint LTS executes a transition labelled with τ when precisely one of the component LTSs executes a τ-transition. If $a \neq \tau$, the joint LTS executes an a-transition when precisely those component LTSs execute simultaneously an a-transition which have a in their alphabets. The components that do not participate the execution stay in their current states. The result is restricted to the reachable part, because the unreachable part is often big and, as we have pointed out, it is irrelevant for the behaviour.

Definition 10. *Let $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$ be LTSs for $1 \leq i \leq n$. Their* parallel composition *$L_1 \,||\, \cdots \,||\, L_n$ is the reachable part of the LTS $(S, \Sigma, \Delta, \hat{s})$, where*

- *$S = S_1 \times \cdots \times S_n$;*
- *$\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$;*
- *if $(s_1, \ldots, s_n) \in S$ and there is $1 \leq j \leq n$ such that for every $1 \leq i \leq n$*
  - *either $i = j$ and $(s_i, \tau, s'_i) \in \Delta_i$*
  - *or $i \neq j$ and $s'_i = s_i$,*

  then $((s_1, \ldots, s_n), \tau, (s'_1, \ldots, s'_n)) \in \Delta$;
- *if $a \in \Sigma$, $(s_1, \ldots, s_n) \in S$, and for every $1 \leq i \leq n$*
  - *either $a \in \Sigma_i$ and $(s_i, a, s'_i) \in \Delta_i$*
  - *or $a \notin \Sigma_i$ and $s'_i = s_i$,*

  then $((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \in \Delta$;
- *Δ has no other elements; and*
- *$\hat{s} = (\hat{s}_1, \ldots, \hat{s}_n)$.*

If the ordering of the components is changed (e.g., from $L_1 \,||\, L_2$ to $L_2 \,||\, L_1$), the names of states change accordingly (from (s_1, s_2) to (s_2, s_1)), but the result is isomorphic to the original. Therefore, it is appropriate to say that $||$ is commutative. It is also associative, because $(L_1 \,||\, L_2) \,||\, L_3$ differs from $L_1 \,||\, (L_2 \,||\, L_3)$ and

even from $L_1 \,||\, L_2 \,||\, L_3$ only by the names of states: $((s_1, s_2), s_3)$ vs. $(s_1, (s_2, s_3))$ vs. (s_1, s_2, s_3).

With $||$, the alphabets determine synchronization. A widely used alternative is to list the synchronizing actions in the operator, e.g., $L_1 \,|[a, b]|\, L_2$. In this example, if $c \neq a$ and $c \neq b$, then c-transitions of L_1 and L_2 do not synchronize, even if both L_1 and L_2 can execute them. Obviously $L_1 \,||\, L_2$ is obtained as $L_1 \,|[a_1, \ldots, a_n]|\, L_2$, where $\{a_1, \ldots, a_n\}$ is the intersection of the alphabets of L_1 and L_2. On the other hand, $|[\cdots]|$ can be easily built from $||$ and the renaming operator of Sect. 4.3. So, from the point of view of expressivity, it does not matter which one we choose. However, $|[\cdots]|$ is not associative, which makes its mathematics more complicated and may also confuse users.

The widely used CCS parallel composition operator $|$ does not allow more than two components to synchronize to the same event. (CCS is *Calculus of Communicating Systems* [20].) It is a significant disadvantage compared to $||$. Using the ideas in Sect. 4.4, $|$ can be easily constructed from $||$ and the operators in Sect. 4.2 and 4.3. On the other hand, it is far from obvious how to construct $||$ from $|$ and the other operators of CCS and this tutorial. There is also a complexity-theoretic sense [36] in which $||$ is simpler than $|[\cdots]|$ and $|$.

Bisimilarity is a *congruence* with respect to $||$. That is, if L_i and L'_i are bisimilar for $1 \leq i \leq n$, then $L_1 \,||\, \cdots \,||\, L_n$ is bisimilar to $L'_1 \,||\, \cdots \,||\, L'_n$. The importance of the congruence property will be discussed in Sect. 5.2. Bisimilarity is a congruence also with respect to the variants that we briefly discussed above, the operators that will be discussed in the remainder of this section, and, indeed, with respect to almost all operators that have been suggested in the process algebra literature.

Let M be a state machine and $\mathcal{B}(M)$ its behaviour. We could now define the behaviour of a parallel composition of state machines as the parallel composition of the behaviours of the state machines: $\mathcal{B}(M_1) \,||\, \cdots \,||\, \mathcal{B}(M_n)$. However, computing the behaviours is often very expensive, as it involves unfolding. So we would like to be able to compute a state machine $M_1 \,||\, \cdots \,||\, M_n$ such that $\mathcal{B}(M_1 \,||\, \cdots \,||\, M_n) = \mathcal{B}(M_1) \,||\, \cdots \,||\, \mathcal{B}(M_n)$. To do that, we must first discuss the joint effect of data manipulation relations of parallel state machines.

For example, assume that there are three state machines, and they have transitions labelled with $[n > 0]\!\rightarrow \mathsf{a}!n?n$, $[i \leq 3]\!\rightarrow \mathsf{a}!i?j$, and $\mathsf{a}?k!0$, respectively. The inscriptions in the first parameter position imply that i and n must have the same value, and that value will be stored into k. It is also the value of the first event parameter, which we denote with p_1. The guards imply that the value must be 1, 2, or 3. The inscriptions in the second parameter position imply that the value of the second event parameter p_2 is 0, and n and j will be 0. Assuming that there are no other variables, the individual data manipulation relations are $n = p_1 > 0 \wedge n := p_2$, $i = p_1 \leq 3 \wedge i := i \wedge j := p_2$, and $k := p_1 \wedge p_2 = 0$. The joint relation is $0 < n = i = p_1 \leq 3 \wedge p_2 = 0 \wedge n := 0 \wedge i := i \wedge j := 0 \wedge k := n$.

When formalizing the joint effect, we must make it precise how variables and actions are treated in the conjunction of data manipulation relations. Variables of different state machines must be treated as different variables even if they have

the same name, but actions must be shared. Not necessarily all state machines participate in a transition. We must specify that those who do not, do not change the values of their variables. For that purpose, for each list of variables we define the *identity relation* $I(\bar{v}, a, \bar{w}) \Leftrightarrow \bar{w} = \bar{v}$, that is, the variable values stay the same and the action does not matter. By $(R_1 \wedge \cdots \wedge R_n)(a)$ we mean the relation $R(\bar{v}_1, \ldots, \bar{v}_n, a, \bar{w}_1, \ldots, \bar{w}_n) \Leftrightarrow R_1(\bar{v}_1, a, \bar{w}_1) \wedge \cdots \wedge R_n(\bar{v}_n, a, \bar{w}_n)$.

We can now define $M_1 \,||\, \cdots \,||\, M_n$.

Definition 11. *Let* $M_i = (S_i, \Theta_i, \mathcal{T}_i, \Sigma_i, \Delta_i, \hat{s}_i, \hat{v}_i)$ *be state machines for* $1 \leq i \leq n$. *Their* parallel composition $M_1 \,||\, \cdots \,||\, M_n$ *is the reachable part of the state machine* $(S, \Theta, \mathcal{T}, \Sigma, \Delta, \hat{s}, \hat{v})$*, where*

- $S = S_1 \times \cdots \times S_n$;
- $\Theta = \Theta_1 \cup \cdots \cup \Theta_n$;
- $\mathcal{T}(s) = \mathcal{T}_1(s_1) \times \cdots \times \mathcal{T}_n(s_n)$ *for every* $s = (s_1, \ldots, s_n) \in S$;
- $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$;
- I_i *is the identity relation for the variables of* s_i;
- *if* $(s_1, \ldots, s_n) \in S$ *and there is* $1 \leq j \leq n$ *such that for every* $1 \leq i \leq n$
  - *either* $i = j$, $(s_i, R_i, s'_i) \in \Delta_i$, *and* $\exists \bar{v}_i, \bar{w}_i : R_i(\bar{v}_i, \tau, \bar{w}_i)$
  - *or* $i \neq j$, $R_i \Leftrightarrow I_i$, *and* $s'_i = s_i$,

  then $((s_1, \ldots, s_n), (R_1 \wedge \cdots \wedge R_n)(\tau), (s'_1, \ldots, s'_n)) \in \Delta$;
- *if* $a \in \Sigma$, $(s_1, \ldots, s_n) \in S$, *and for every* $1 \leq i \leq n$
  - *either* $a \in \Sigma_i$, $(s_i, R_i, s'_i) \in \Delta_i$, *and* $\exists \bar{v}_i, \bar{w}_i : R_i(\bar{v}_i, a, \bar{w}_i)$
  - *or* $a \notin \Sigma_i$, $R_i \Leftrightarrow I_i$, *and* $s'_i = s_i$,

  then $((s_1, \ldots, s_n), (R_1 \wedge \cdots \wedge R_n)(a), (s'_1, \ldots, s'_n)) \in \Delta$;
- Δ *has no other elements;*
- $\hat{s} = (\hat{s}_1, \ldots, \hat{s}_n)$; *and*
- $\hat{v} = (\hat{v}_1, \ldots, \hat{v}_n)$.

The purpose of the conditions $\exists \bar{v}_i, \bar{w}_i : R_i(\bar{v}_i, \tau, \bar{w}_i)$ and $\exists \bar{v}_i, \bar{w}_i : R_i(\bar{v}_i, a, \bar{w}_i)$ is to avoid creating transitions with obviously empty data manipulation relations. Without them, the definition would, for instance, create a dead τ-transition from the ci-transition of ATM. If the conditions are too difficult to check precisely in a practical situation, then one may use an upper approximation and accept that the result may have extra dead transitions. For instance, one may generate all transitions where the gate names match.

Figure 5 shows the parallel composition of a modified ATM and BANK. To get a readable figure, the channels have been left out, and ATM and BANK interact directly. Losses of messages in the channels are modelled by additional transitions in ATM and BANK that read the incoming message but do not change the state nor the values of local variables. To keep the figure readable, the balance variable b has been removed.

The horizontal and vertical arrows correspond to transitions that either ATM or BANK executes alone, and so does the arrow labelled with nm. There are only four transitions in which both participate simultaneously. In two of them, labelled q!x $z := x$ and y!z $x := z$, data is passed, which is shown by the assignment in the inscription of the arrow. The data manipulation relations of

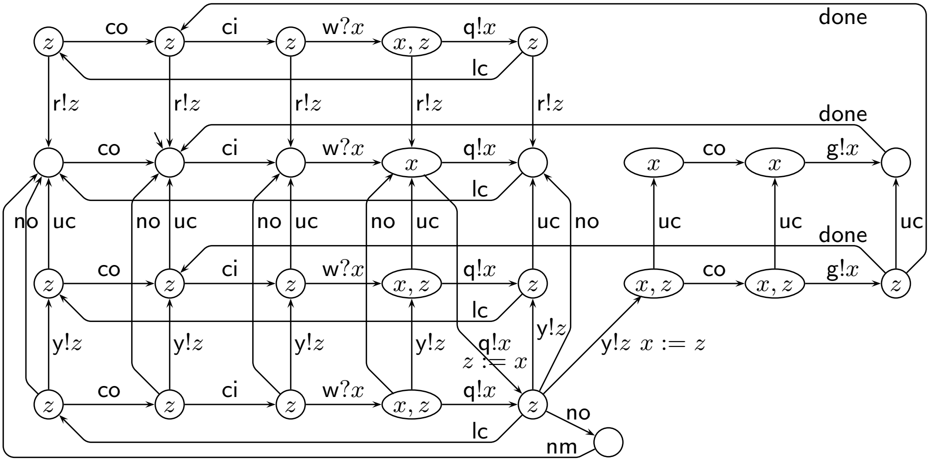


Fig. 5. The parallel composition of ATM and BANK with variable b removed and channels replaced by direct synchronization that can lose messages

the synchronizing q-transitions are $p_1 = x$ and $z := p_1$. Their conjunction is equivalent to $p_1 = x \wedge z := x$. In the figure, this is represented by $!x$ and $z := x$.

We are ready to show that Definition 11 produces what it should.

Proposition 12. *$\mathcal{B}(M_1 \mid\mid \cdots \mid\mid M_n)$ and $\mathcal{B}(M_1) \mid\mid \cdots \mid\mid \mathcal{B}(M_n)$ are isomorphic.*

Proof. The states of $\mathcal{B}(M_1 \mid\mid \cdots \mid\mid M_n)$ are of the form $(s_1, \ldots, s_n)\langle \bar{v}_1, \ldots, \bar{v}_n \rangle$, while the states of $\mathcal{B}(M_1) \mid\mid \cdots \mid\mid \mathcal{B}(M_n)$ are of the form $(s_1\langle \bar{v}_1 \rangle, \ldots, s_n\langle \bar{v}_n \rangle)$. In both ways of computing the result, τ implies that precisely one state machine participates. In both ways, if $a \neq \tau$, each a-transition is participated by precisely those state machines which have a in their alphabets. In both ways, the state machines that do not participate keep their states and variable values. In both ways, the state machines that do participate change their states similarly, and change their variable values according to their data manipulation relations. In both ways, the result is restricted to the reachable part. □

Of course, it would be possible to define an operation that both puts the state machines together and unfolds the result. Such an operation corresponds to traditional state space construction. It is an advantage of process algebras that the behaviour need not be constructed in one big step but it can be constructed in many sub-steps, and there is freedom in the order of the sub-steps. We will see in Sect. 5.8 that useful things can be done between the sub-steps. It would not be possible, if the behaviour had to be computed in one big batch.

The behaviour of a parallel composition is the parallel composition of the behaviours of its components. It can be computed in one big batch or divided to parallel composition and unfolding steps in many different ways.

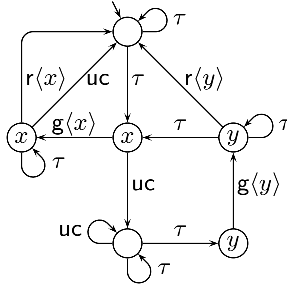


Fig. 6. The user's money view to the simplified cash dispenser system in Fig. 5

4.2 Hiding

The hiding operator converts visible actions to invisible. With it one can choose a view to a system. The behaviour of the system can be projected to the chosen view with techniques discussed in Sect. 5.

Figure 6 shows the projection of Fig. 5 to the view of the user's money. The view was chosen by leaving g, r, and uc visible, and hiding everything else. The figure has been produced manually but, excluding the variables x and y, it is the same as an automatically generated figure with the data type restricted to a singleton set. Most details of the figure cannot be explained before discussing the theory in Sect. 5.4, but some observations can be made already now. For instance, every "reduce balance" transition (r) is preceded by a "give money" transition (g) with the same amount. So the system never charges the balance without giving the money.

If the channels in Fig. 3 are used, then new phenomena emerge. For instance, the system can charge a wrong amount of money from the account! The following sequence of events leads to the error. The user tries to withdraw x units of money. The transaction progresses successfully up to $\mathsf{Sy}\langle x\rangle$, but then both ATM and BANK give up and return to their initial states. While doing so, BANK executes uc. The user tries again with a new amount y. After executing $\mathsf{Sq}\langle y\rangle$, ATM gets the "yes"-answer that was left over in CH2 from the previous attempt, and gives the user x units of money with $\mathsf{g}\langle x\rangle$. BANK reads the second query by executing $\mathsf{Rq}\langle y\rangle$ and replies "yes" to it. ATM sends "done". BANK receives it and reduces y units of money from the account.

Instead of fixing the cash dispenser system, we continue discussing the theory. The definition of the hiding operator on LTSs is simple. The hidden actions are removed from the alphabet, because they become internal to the LTS. This makes it possible to use their names as action names in other parts of the system.

Definition 13. *Let $L = (S, \Sigma, \Delta, \hat{s})$ be an LTS, and A be a set. The result of* hiding *A in L is the LTS $L \setminus A = (S, \Sigma', \Delta', \hat{s})$, where*

- $\Sigma' = \Sigma \setminus A$;
- *if* $(s, a, s') \in \Delta$ *and* $a \notin A$, *then* $(s, a, s') \in \Delta'$;

- *if* $(s, a, s') \in \Delta$ *and* $a \in A$, *then* $(s, \tau, s') \in \Delta'$; *and*
- Δ' *has no other elements.*

It might seem natural to require that $A \subseteq \Sigma$. However, extra elements in A do not affect the result (even $\tau \in A$ is harmless), and the operator is easier to use if one need not ensure that A indeed is a subset of Σ. So we do not make the requirement.

In the definition of the hiding operator of state machines, hiding must be defined for the data manipulation relations, because the same relation may induce transitions with both hidden and unhidden actions. The first part of the definition of $R \setminus A$ keeps transitions with unhidden actions, and the second part generates τ-transitions from transitions with hidden actions.

Definition 14. *Let* $M = (S, \Theta, \mathcal{T}, \Sigma, \Delta, \hat{s}, \hat{v})$ *be a state machine, and* A *be a set. The result of* hiding A *in* M *is the state machine* $M \setminus A = (S, \Theta, \mathcal{T}, \Sigma', \Delta', \hat{s}, \hat{v})$, *where*

- $\Sigma' = \Sigma \setminus A$;
- $(R \setminus A)(\bar{v}, a, \bar{w}) \Leftrightarrow a \notin A \wedge R(\bar{v}, a, \bar{w}) \vee a = \tau \wedge \exists b \in A : R(\bar{v}, b, \bar{w})$; *and*
- $\Delta' = \{ (s, R \setminus A, s') \mid (s, R, s') \in \Delta \}$.

Also hiding has the property that it does not matter whether it is done before or after unfolding, that is, $\mathcal{B}(M \setminus A) = \mathcal{B}(M) \setminus A$. (This time "=" is equality and not just isomorphism.)

A number of properties obviously hold, such as

- $(M \setminus A) \setminus B = M \setminus (A \cup B)$.
- If $A \cap \Sigma_2 = \emptyset$, then $(M_1 \setminus A) \mid\mid M_2 = (M_1 \mid\mid M_2) \setminus A$.

4.3 Relational Renaming

Renaming means changing gate names or actions. As such, it makes it possible to specify a state machine once and use it in more than one place. For instance, one may specify the dining philosophers' system by specifying a single dining philosopher with actions take_left, take_right, release_left, and release_right and a single chop stick with actions take and release, and taking several copies of them, renaming the actions to take_1, take_2, and so on.

Simple renaming and || suffice for the philosophers' system, but they run into trouble in the following kind of a situation. There are many servers and even more clients. Any server can serve any client. When a client needs service, it sends a general call that precisely one free server synchronizes to, but that server can be any of the free servers. If no server is free, the call transition is blocked. We want the outside world to see which client and server synchronized.

This situation can be modelled with a more general form of renaming, where one may map a single action to more than one action. It is called *multiple renaming* or *relational renaming*. We skip the definition for LTSs and only show the definition for state machines.

Definition 15. *Let* $M = (S, \Theta, \mathcal{T}, \Sigma, \Delta, \hat{s}, \hat{v})$ *be a state machine. A* renaming relation *for* M *is any set* Φ *of pairs such that the domain of* Φ *is precisely* Σ*, that is,* $\{ a \mid \exists b : (a,b) \in \Phi \} = \Sigma$*, and* τ *is not in the range of* Φ*, that is,* $\neg\exists a : (a, \tau) \in \Phi$*. The result of applying* Φ *to* M *is the state machine* $M\Phi = (S, \Theta, \mathcal{T}, \Sigma', \Delta', \hat{s}, \hat{v})$*, where*

- $\Sigma' = \{ b \mid \exists a : (a,b) \in \Phi \}$;
- $(R\Phi)(\bar{v}, b, \bar{w}) \Leftrightarrow \exists a : (a,b) \in \Phi \wedge R(\bar{v}, a, \bar{w}) \vee b = \tau \wedge R(\bar{v}, \tau, \bar{w})$; *and*
- $\Delta' = \{ (s, R\Phi, s') \mid (s, R, s') \in \Delta \}$.

The purpose of the restriction on the domain of Φ is to simplify the theory by ruling out unnecessary special cases. Without it, one could remove some transitions altogether by leaving their action a without a pair $(a,b) \in \Phi$; and one could add extra members to Σ' by having $(a,b) \in \Phi$ such that $a \notin \Sigma$. The restriction does not imply loss of generality, because one can have the same effects with $||$. Let $\mathbf{stop}_A$ be the state machine with one state, no variables, no transitions, and the alphabet A. One can remove the a-transitions of M for every $a \in A$ by writing $(M \,||\, \mathbf{stop}_A) \setminus A$. One can add B to the alphabet by writing $M \,||\, \mathbf{stop}_{B \setminus \Sigma}$.

The client–server example can now be modelled as

$$C\Phi_1 \,||\, \cdots \,||\, C\Phi_n \,||\, S\Phi'_1 \,||\, \cdots \,||\, S\Phi'_m \ ,$$

where

- C runs in a loop $s_1 - \mathsf{call} \rightarrow s_2 - \mathsf{reply} \rightarrow s_1$;
- S runs in a loop $s_1 - \mathsf{call}?i \rightarrow s_2\langle i \rangle - \mathsf{reply}!i \rightarrow s_1$;
- $\Phi_{i,j} = \{ (\mathsf{call}, \mathsf{call}\langle i,j \rangle), (\mathsf{reply}, \mathsf{reply}\langle i,j \rangle) \}$;
- $\Phi_i = \Phi_{i,1} \cup \cdots \cup \Phi_{i,m}$;
- $\Phi'_{i,j} = \{ (\mathsf{call}\langle i \rangle, \mathsf{call}\langle i,j \rangle), (\mathsf{reply}\langle i \rangle, \mathsf{reply}\langle i,j \rangle) \}$; and
- $\Phi'_j = \Phi'_{1,j} \cup \cdots \cup \Phi'_{n,j}$.

The variable i in the server makes it reply to the right client. Φ_i adds the identity i of the client to its actions. It also takes one copy of each action for each server, so that the client can synchronize with any server. Φ'_j adds the identity of the server to its actions.

4.4 Synchronization Rules

With the operators introduced this far, one can write complicated expressions such as $((M_1\Phi_1 \,||\, M_2\Phi_2) \setminus A) \,||\, M_3$. However, intuitively each transition of any such system consists of some state machines participating via some actions, other state machines not participating, and the result having some action. The resulting action may be τ even if none of the original actions is, but if any of the original actions is τ, then only that state machine participates, and the resulting action is τ.

We now make this idea precise. Let - be a symbol that is not in any alphabet. It will denote that the state machine does not participate the transition. (We could have used τ for that purpose but felt it confusing, because not participating is not the same thing as participating via a τ-transition.)

Definition 16. *Let $M_i = (S_i, \Theta_i, \mathcal{T}_i, \Sigma_i, \Delta_i, \hat{s}_i, \hat{v}_i)$ be state machines for $1 \leq i \leq n$. Let Σ be a set such that $\tau \notin \Sigma$. A* synchronization rule *for them is a tuple $r = (a_1, \ldots, a_n; a)$, where $a \in \Sigma \cup \{\tau\}$ and $a_i \in \Sigma_i \cup \{\text{-}\}$ for $1 \leq i \leq n$, and at least one $a_i \neq$ -. We let $r[0] = a$ and $r[i] = a_i$ for $1 \leq i \leq n$.*

Let $\mathcal{Y}$ be a set of synchronization rules for $M_1, \ldots, M_n$, and Σ. Then $\mathcal{Y}(M_1, \ldots, M_n)$ is the reachable part of the state machine $(S, \Theta, \mathcal{T}, \Sigma, \Delta, \hat{s}, \hat{v})$, where

- $S = S_1 \times \cdots \times S_n$;
- $\Theta = \Theta_1 \cup \cdots \cup \Theta_n$;
- $\mathcal{T}(s) = \mathcal{T}_1(s_1) \times \cdots \times \mathcal{T}_n(s_n)$ *for every* $s = (s_1, \ldots, s_n) \in S$;
- I_i *is the identity relation for the variables of* s_i;
- *if* $(s_1, \ldots, s_n) \in S$ *and there is* $1 \leq j \leq n$ *such that for every* $1 \leq i \leq n$
  - *either* $i = j$, $(s_i, R_i, s'_i) \in \Delta_i$, *and* $\exists \bar{v}_i, \bar{w}_i : R_i(\bar{v}_i, \tau, \bar{w}_i)$
  - *or* $i \neq j$, $R_i \Leftrightarrow I_i$, *and* $s'_i = s_i$,

  then $((s_1, \ldots, s_n), (R_1 \wedge \cdots \wedge R_n)(\tau), (s'_1, \ldots, s'_n)) \in \Delta$;
- *if* $(a_1, \ldots, a_n; a) \in \mathcal{Y}$, $(s_1, \ldots, s_n) \in S$, *and for every* $1 \leq i \leq n$
  - *either* $a_i \in \Sigma_i$, $(s_i, R_i, s'_i) \in \Delta_i$, *and* $\exists \bar{v}_i, \bar{w}_i : R_i(\bar{v}_i, a_i, \bar{w}_i)$
  - *or* $a_i =$ -, $R_i \Leftrightarrow I_i$, *and* $s'_i = s_i$,

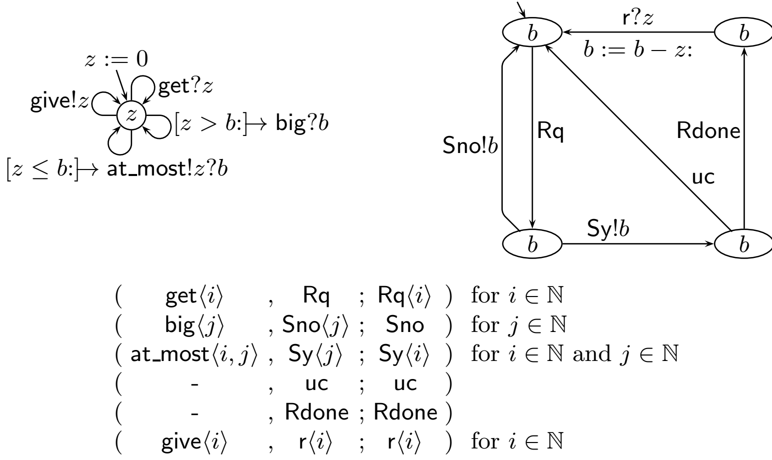
  then $((s_1, \ldots, s_n), R, (s'_1, \ldots, s'_n)) \in \Delta$, *where* $R(\bar{v}_1, \ldots, \bar{v}_n, a, \bar{w}_1, \ldots, \bar{w}_n) \Leftrightarrow R_1(\bar{v}_1, a_1, \bar{w}_1) \wedge \cdots \wedge R(\bar{v}_n, a_n, \bar{w}_n)$;
- Δ *has no other elements;*
- $\hat{s} = (\hat{s}_1, \ldots, \hat{s}_n)$; *and*
- $\hat{v} = (\hat{v}_1, \ldots, \hat{v}_n)$.

This definition is not much more difficult to understand than the previous ones, and it makes it possible to specify arbitrary synchronization patterns. Most, or perhaps all, major parallel composition operators in process algebras can be constructed with synchronization rules.

Synchronization rules can also be used to represent local variables as parallel state machines. For instance, Fig. 7 shows how the variable z of BANK could be replaced by a state machine that is synchronized with the rest of BANK. In this example, b of BANK-MAIN is kept as a local variable. $z := 0$ specifies the initial value of z. b: refers to the afterwards value of b. So, e.g., $[z > b{:}] \mapsto$ big?b means that big$\langle j \rangle$ is available with those values of j that satisfy $z > j$. The third rule and the inscriptions of the transitions say that at_most has two event parameters, the first carrying the current value of z and the second carrying the same or bigger value; this latter value is also the event parameter of Sy of BANK-MAIN and thus equal to b; and the outside world sees Sy with the value of z.

This implementation of BANK is not isomorphic to Fig. 2. The difference is that while BANK-MAIN is in its initial state, z has no value in Fig. 2, but in Fig. 7 it keeps its previous value. This difference does not affect the behaviour significantly, because the value of z in the initial state is not used and is overwritten by the next transition. Indeed, the two models of BANK are bisimilar.

The example has a synchronization rule for each possible combination of event parameter values for each gate. There are thus infinitely many rules. This is not



(	get⟨i⟩	,	Rq	;	Rq⟨i⟩	)	for $i \in \mathbb{N}$
(	big⟨j⟩	,	Sno⟨j⟩	;	Sno	)	for $j \in \mathbb{N}$
(	at_most⟨i, j⟩	,	Sy⟨j⟩	;	Sy⟨i⟩	)	for $i \in \mathbb{N}$ and $j \in \mathbb{N}$
(	-	,	uc	;	uc	)	
(	-	,	Rdone	;	Rdone	)	
(	give⟨i⟩	,	r⟨i⟩	;	r⟨i⟩	)	for $i \in \mathbb{N}$

Fig. 7. BANK made of Z and BANK-MAIN state machines with synchronization rules

a problem for developing theoretical results. In a practical situation, one can use suitable notation for representing rules in bunches. Indeed, Fig. 7 has only six such bunches.

Analogously to earlier operators, we should prove that $\mathcal{B}(\mathcal{Y}(M_1, \ldots, M_n))$ is isomorphic to $\mathcal{Y}(\mathcal{B}(M_1), \ldots, \mathcal{B}(M_n))$. Instead of doing that directly, we will get the result for free from another result: the effect of synchronization rules can be built from the other operators discussed so far. This implies that synchronization rules can be considered shorthand notation, and it suffices to develop the theory for the other operators.

Proposition 17. *Let $M_i = (S_i, \Theta_i, \mathcal{T}_i, \Sigma_i, \Delta_i, \hat{s}_i, \hat{v}_i)$ be state machines for $1 \leq i \leq n$. Let Σ be a set such that $\tau \notin \Sigma$, and let $\mathcal{Y}$ be a set of synchronization rules for them. Then $\mathcal{Y}(M_1, \ldots, M_n) =$*

$$\Big(\Big(\big((M_1 \,||\, \mathbf{stop}_{A_1}) \setminus A_1\big)\Phi_1 \,||\, \cdots \,||\, \big((M_n \,||\, \mathbf{stop}_{A_n}) \setminus A_n\big)\Phi_n\Big) \setminus A\Big)\Phi \,||\, \mathbf{stop}_B \;,$$

where

- $A_i = \{\, a \in \Sigma_i \mid \neg\exists r \in \mathcal{Y} : a = r[i] \,\}$ *for* $1 \leq i \leq n$*;*
- $\Phi_i = \{\, (a, r) \mid a \in \Sigma_i \wedge r \in \mathcal{Y} \wedge a = r[i] \,\}$ *for* $1 \leq i \leq n$*;*
- $A = \{\, r \in \mathcal{Y} \mid r[0] = \tau \,\}$*;*
- $\Phi = \{\, (r, a) \mid r \in \mathcal{Y} \wedge a = r[0] \neq \tau \,\}$*; and*
- $B = \{\, a \in \Sigma \mid \neg\exists r \in \mathcal{Y} : a = r[0] \,\}$*.*

Proof. The part using A_i removes those non-τ-transitions of M_i that have no matching rule. Φ_i renames the remaining non-τ-transitions of M_i so that $||$ synchronizes transitions that match the same rule. Then $\setminus A$ ensures that the final name is τ if the rule requires so. Φ fixes the final names that must not be τ, and $\mathbf{stop}_B$ adds to Σ the elements that are still missing. □

Together with the modelling of channels as state machines in the cash dispenser system, the results and examples in this subsection justify the following informal claims (however, please see also Sect. 5.3).

> All communication between state machines can be expressed in terms of $||$, hiding, and relational renaming. Synchronization rules are a handy shorthand notation for that.

> Use of a local or shared variable is essentially parallel composition with a state machine that directly represents the variable.

Earlier versions of synchronization rules were presented in [1,17].

4.5 Input and Output

In the ?-, etc., notation, ? denotes input and ! output. If transitions labelled $\mathsf{a}!x$ and $\mathsf{a}?y$ synchronize, then it is appropriate to say that the former is an output and the latter input transition. The one who outputs determines the value of the event parameter, and the one who inputs is ready for just any value. The one who inputs usually stores the value in a variable, but this should not be seen as a fundamental property of input, because one may cheat by storing the value to a variable that is not used later.

The roles of input and output are not clear at the level of transitions, but are still at the level of individual event parameters, when transitions labelled $\mathsf{a}!1?x$ and $\mathsf{a}?y!3$ synchronize. The situation may be unclear even if there is only one event parameter, like with $[n\colon \geq 0] \mapsto \mathsf{a}?n$ and $[i\colon < 1] \mapsto \mathsf{a}?i$. Here the first transition determines that the value is at least 0 and the second that it is less than 1. So it is 0. However, neither transition determines the value alone, so neither can be called output. An even more confusing example is $[x := x] \mapsto \mathsf{a}?x$, because, although it seems to read the afterwards value of x from the event parameter, the guard forces the value to be the same as the original value. So it means precisely the same as $\mathsf{a}!x$.

In conclusion, process algebras allow forms of interaction where the notions of input and output do not make sense. Some interaction patterns may be very hard to implement in practice, especially in a distributed setting, but it is better to allow them in the theory than to rule out useful forms of interaction.

> Input and output are roles in interaction. Often interaction can be understood in terms of input and output, but not always.

4.6 Other Operators

Process-algebraic languages (such as [3,13,20]) have other operators in addition to variants of what we have already discussed. In this subsection we briefly discuss the most common. They are not necessary for most of the rest of this tutorial, but are central in many other writings on process algebras. In Sect. 5.3

they will be used to demonstrate that despite their great modelling power, synchronization rules do not cover all reasonable ways of building systems.

Action prefix $a; P$ (also written as $a.P$ and $a \rightarrow P$) means a system that first executes an a-transition and then behaves like P.

There are variants of the *choice* operator with somewhat different meanings. $P \mathbin{\Box} Q$, also written as $P + Q$, has initially the capability of behaving like P and like Q. Its first transition is either an initial transition of P or of Q, and then it continues like P or Q according to with whose transition it started. The environment does not directly see whether P or Q was chosen. Of course, if the action of the initial transition is visible and only used by Q, then it is possible to reason that Q was chosen. As an example of the variants of the choice operator, in *nondeterministic choice* $\sqcap$ the choice is done silently even if the initial transitions have different actions.

The *interrupt* operator $P \mathbin{[\!>} Q$ is otherwise like choice, but Q has the ability to start until P has terminated successfully. Successful termination is indicated by executing a transition with a special action that has been reserved for this purpose. When Q has started, P cannot continue. A *divergence* is an infinite sequence of invisible events. It corresponds to a livelock. Of the operators that we have discussed, interrupt is the only one that can, roughly speaking, stop a divergence.

We used graphs to specify individual state machines, but many process-algebraic languages express everything in terms of textual expressions. Cyclic behaviour is specified by letting expressions call themselves recursively. In this setting, choice operators are the main means of specifying branching behaviour. For instance, BANK can be specified with the following expression.

$$
\begin{array}{rcl}
\mathrm{BANK}(b) & = & \mathsf{Rq}?z;\ (\ \ [z > b] \mapsto \mathsf{Sno}; \mathrm{BANK}(b) \\
 & & \qquad\ \Box\ [z \leq b] \mapsto \mathsf{Sy}!z;\ (\ \ \mathsf{uc}; \mathrm{BANK}(b) \\
 & & \qquad\qquad\qquad\qquad\qquad \Box\ \mathsf{Rdone}; \mathsf{r}!z; \mathrm{BANK}(b-z) \\
 & & \qquad\qquad\qquad\qquad\qquad) \\
 & & \qquad\ )
\end{array}
$$

This way of specifying systems makes it easy to model some situations that cannot be modelled easily or at all with our state machine formalism, like on-the-fly creation of new state machines. However, recursion complicates significantly the development of semantic theories like the ones in Sect. 5. As a consequence, some theories give unintuitive meanings to expressions like $P = \tau; P$. Furthermore, modellers of systems may find the notation cryptic and laborious to use. The present author believes that this is one of the reasons why process-algebraic methods have received much less attention than they deserve.

5 Abstract Behaviour

The detailed behaviour of a system with many hidden actions has typically few visible transitions and many τ-transitions. It cannot be drawn as a readable picture because of the many τ-transitions. In this section we discuss theories of

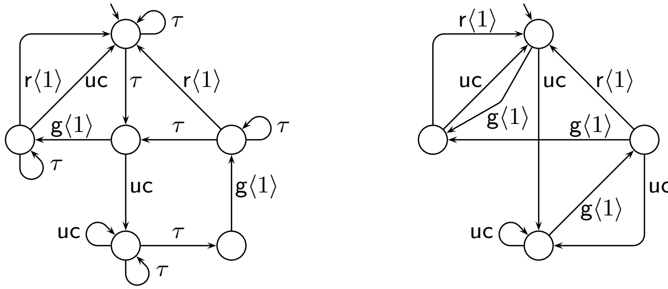


Fig. 8. The CFFD- and trace semantics version of the user's money view in Fig. 6 restricted to data type $\{1\}$

abstract behaviour, with which one can get rid of most τ-transitions and produce pictures such as Fig. 6. There are numerous such theories, so we concentrate on one and briefly mention some others. As a by-product we get a proof that the interrupt operator is fundamental instead of a shorthand. We also study the notion of determinism in the context of abstract behaviour. Finally we mention some verification techniques that exploit abstract behaviour.

5.1 Trace Semantics

A *trace* of an LTS is the sequence of visible actions that is obtained from any finite path that starts in the initial state. For instance, both LTSs in Fig. 8 have the traces ε, $\mathsf{g}\langle 1\rangle$, uc, $\mathsf{g}\langle 1\rangle\,\mathsf{r}\langle 1\rangle$, $\mathsf{g}\langle 1\rangle\,\mathsf{uc}$, $\mathsf{uc}\,\mathsf{uc}$, $\mathsf{uc}\,\mathsf{g}\langle 1\rangle$, $\mathsf{g}\langle 1\rangle\,\mathsf{r}\langle 1\rangle\,\mathsf{g}\langle 1\rangle$, and so on. ε denotes the empty sequence of visible actions. It is a trace of every LTS. A trace of a system is a trace of its behaviour. Figure 8 left has been made from Fig. 6 by restricting the type of x and y to $\{1\}$.

To make it easier to talk about traces and related things, let $s =\sigma\Rightarrow s'$ denote that there is a path from s to s' such that its sequence of visible actions is σ. By $s =\sigma\Rightarrow$ we mean the same thing but do not mention the end state of the path. With this notation, the set of traces of $L = (S, \Sigma, \Delta, \hat{s})$ is $Tr(L) = \{\, \sigma \mid \hat{s} =\sigma\Rightarrow \}$.

The *trace semantics* of L is the pair $(\Sigma(L), Tr(L))$, where $\Sigma(L)$ is the alphabet of L. Two systems are *trace equivalent* if and only if they have the same trace semantics, that is, the same alphabet and the same set of traces, that is, $\Sigma(L) = \Sigma(L') \wedge Tr(L) = Tr(L')$. Every semantics induces an equivalence — $L \simeq L'$ if and only if they have the same semantics. On the other hand, the equivalence classes of any equivalence can be thought of as a semantics. Therefore, we will sometimes use the words "semantics" and "equivalence" interchangeably.

The set of traces of a finite LTS is essentially the same thing as the language accepted by a finite automaton whose every state is a final state. As a consequence, well-known algorithms from automata theory can be used for manipulating finite LTSs so that the trace semantics is preserved. One may, for instance, construct the smallest deterministic LTS that has the same trace semantics as a given finite LTS. Figure 8 right shows the result of doing that to Fig. 8 left.

When we say that an equivalence $\simeq$ *preserves* a property we mean that for every L and L', $L \simeq L'$ implies that L and L' give the same value to the property. For instance, if $\simeq$ preserves deadlocks and $L \simeq L'$, then either none or both of L and L' may deadlock.

The major drawback of the trace equivalence is that it does not preserve deadlocks and livelocks. In those applications where this does not matter, trace semantics is excellent. What the trace equivalence preserves is called *stuttering-insensitive safety properties.* Safety properties are the properties whose violation can be detected after a finite execution, without knowing the future. Deadlock and livelock cannot be detected so, because if an external observer who only sees the visible events did not see anything happen, she does not know whether that was because she did not wait long enough or because nothing is going to ever happen. She cannot assume that if something is going to happen, it will happen in, say, 1000 time units, because there may be 1001 τ-events before the next visible event.

Stuttering-insensitive means that the number of τ-events before a visible event or deadlock does not matter. Bisimilarity is not stuttering-insensitive, but all semantics in this section are. Indeed, the usefulness of the abstract semantics comes from throwing away unnecessary information, and the number of τ-events is almost always unnecessary information.

Trace equivalence can preserve any stuttering-insensitive safety property, and does not preserve deadlock-freedom. Temporal logic researchers sometimes classify deadlock-freedom as a stuttering-insensitive safety property [19, p. 309]. We have a paradox!

Deadlock-freedom is expressed in temporal logics as $\Box(E_1 \vee \cdots \vee E_n)$, where $\Box$ means "always", and the E_i are equivalent to the enabling conditions of the atomic statements of the system. The formula can also be modelled in trace semantics. However, if one more statement is added to the system but not to the formula, the formula is still meaningful, but does not any more express deadlock-freedom. We see that classification of deadlock-freedom as a safety property assumes that the program code of the system is available, to know which formula expresses deadlock-freedom. Process algebra researchers do not make that assumption, so they can model the formula but cannot know if it expresses deadlock-freedom.

State propositions can be taken into account in traces by adding $val(\hat{s})$ to the semantics and replacing the actions by pairs $\langle a, P \rangle$, where $a \in \Sigma \cup \{\tau\}$ and $P \subseteq \Pi$ [11]. P lists the propositions whose truth values change during the transition. The pair $\langle \tau, \emptyset \rangle$ plays the role of the invisible action. In this formalism, a trace is a sequence of pairs $\langle a, P \rangle$, where $a \neq \tau$ or $P \neq \emptyset$.

5.2 Stable Failures

A deadlock-preserving equivalence is obtained by extending the trace semantics with the set of *stable failures.* We say that a state *refuses* a set A of actions, if none of its output transitions is labelled with an element from A. Thus a deadlock is a state that refuses $\Sigma \cup \{\tau\}$, where Σ is the alphabet of the system.

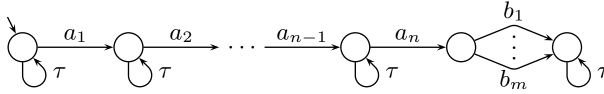


Fig. 9. Showing that stable failures are necessary to preserve deadlocks

A stable failure is a pair (σ, A), where σ is a trace and $A \subseteq \Sigma$. The system has (σ, A) as its stable failure if and only if it has an execution whose trace is σ and that ends in a state that refuses $A \cup \{\tau\}$. The set of stable failures of L is thus

$$SFail(L) = \{ (\sigma, A) \mid \exists s : \hat{s} =\sigma\Rightarrow s \wedge A \subseteq \Sigma(L) \wedge \forall a \in A \cup \{\tau\} : \neg(s -a\rightarrow) \} \ .$$

Here $s -a\rightarrow$ means that there is an s' such that $s -a\rightarrow s'$.

The motivation for this complicated-looking notion is that its presence in the semantics either explicitly or implicitly is necessary to preserve deadlocks, if we use $||$. Let $\sigma = a_1 a_2 \cdots a_n \in \Sigma(L)^*$ and $A = \{b_1, \ldots, b_m\} \subseteq \Sigma(L)$. Consider the LTS L_A^σ in Fig. 9 with the alphabet $\Sigma(L)$. If $L \,||\, L_A^\sigma$ has executed some other trace than σ, then τ is enabled. If σ has been executed and L can execute τ or any of $b_1, \ldots, b_m$, then that action is enabled. Otherwise nothing is enabled. So $L \,||\, L_A^\sigma$ deadlocks if and only if both execute σ and then L refuses τ and $b_1, \ldots, b_m$. That is possible if and only if (σ, A) is a stable failure of L.

We mentioned in Sect. 4.1 that an equivalence $\simeq$ is a congruence with respect to an operator f for putting state machines or behaviours together, if and only if for every $L_1, \ldots, L_n, L'_1, \ldots, L'_n$ we have that $L_1 \simeq L'_1, \ldots, L_n \simeq L'_n$ imply $f(L_1, \ldots, L_n) \simeq f(L'_1, \ldots, L'_n)$. Let $L_1 \simeq L_2$, where $\simeq$ preserves deadlocks and the alphabet, and is a congruence with respect to $||$. Because it is a congruence, $L_1 \,||\, L_A^\sigma \simeq L_2 \,||\, L_A^\sigma$. Then $(\sigma, A) \in SFail(L_1)$ if and only if $L_1 \,||\, L_A^\sigma$ has a deadlock if and only if $L_2 \,||\, L_A^\sigma$ has a deadlock if and only if $(\sigma, A) \in SFail(L_2)$. We have proven the following.

Proposition 18. *Any congruence with respect to $||$ that preserves the alphabet and deadlocks also preserves stable failures.*

This, actually simple, result is from [28]. The semantics consisting of the alphabet and stable failures (but not traces) is a congruence with respect to $||$, hiding, relational renaming, and action prefix. If also the choice operator $\Box$ is used, then the so-called “initial stability” bit that we will discuss a bit later must be added to the semantics, to retain the congruence property. If, furthermore, the interrupt operator $\rhd$ is used, then also traces must be added to the semantics. These emphasize that the congruence property is sensitive to the set of operators in use.

The discussion above can be summarized by saying that even if we could directly observe only deadlocks, we could get information on stable failures and perhaps also other things by putting the system to a suitable environment, and observing the deadlocks of the result. The semantic model consisting of the alphabet, traces, and deadlocks is not *fully abstract*, because we could get additional information about the system by using it as a component in a bigger

Fig. 10. Illustrating a congruence problem with failures

system. On the other hand, assuming that only $||$, hiding, relational renaming, and action prefix are allowed for connecting the system to its environment, the semantic model consisting of the alphabet and stable failures is fully abstract: it contains precisely the information that we can get by putting the system under test in a suitable environment and then observing deadlocks.

Many advanced process-algebraic verification methods are based on replacing components of a system by equivalent components that are better suited for continuing the verification. For instance, an LTS may be replaced by a smaller but equivalent LTS. The correctness of this relies on the assumption that the equivalence in use is a congruence, implying that the semantics of the system as a whole does not change in the replacement. Therefore, the congruence property is central.

When $\neg(s \,{-\tau\!\rightarrow})$ and $a \in \Sigma$, then $s \,{=\!a\!\Rightarrow}$ and $s \,{-a\!\rightarrow}$ mean the same. As a consequence, stable failures can be defined equivalently as

$$\{\ (\sigma, A) \mid \exists s : \hat{s} \,{=\!\sigma\!\Rightarrow}\, s \wedge A \subseteq \Sigma \wedge \forall a \in A : \neg(s \,{=\!a\!\Rightarrow}) \wedge \neg(s \,{-\tau\!\rightarrow})\ \}\ .$$

Historically there was great interest in *failures*, defined as

$$\{\ (\sigma, A) \mid \exists s : \hat{s} \,{=\!\sigma\!\Rightarrow}\, s \wedge A \subseteq \Sigma \wedge \forall a \in A : \neg(s \,{=\!a\!\Rightarrow})\ \}\ .$$

This line of research ran into trouble because of the congruence problem that is illustrated in Fig. 10. The LTSs in the figure have the same alphabet, traces, and failures. However, if a is hidden in both, then $(\varepsilon, \{b\})$ becomes a failure of the one on the right but not of the other. The semantics consisting of the alphabet, traces, and failures is thus not a congruence with respect to hiding.

This is why failures had to be replaced by stable failures. A state is called *stable* if and only if it cannot execute τ. The difference of failures and stable failures is that in the latter, the state after the trace must be stable.

The desire that the equivalence must be a congruence has led to numerous small variants of equivalences. For instance, the semantics consisting of the alphabet, traces, and stable failures stops from being a congruence when the choice operator is employed. This problem can be solved simply by adding one bit to the semantics, known as the *initial stability bit*. It tells whether the initial state is stable, that is, whether $\neg(\hat{s} \,{-\tau\!\rightarrow})$.

The congruence property is sensitive to the set of operators in use. This is one reason why there are so many semantic models in process algebras.

Information on stable failures can be taken into account in algorithms by attaching to each relevant state a set of minimal *acceptance sets*. An acceptance

set is the complement, with respect to the alphabet, of a set that the state refuses. Acceptance sets carry the same information as refused sets but tend to be smaller. All stable failures represented by a non-minimal acceptance set are also represented by their smaller acceptance sets, so storing non-minimal acceptance sets would be pointless.

5.3 On Building Operators from Other Operators

Often in computer science, an operator can be thought of as just an abbreviation of an expression written without using it, while another operator genuinely adds to the expressivity of the language. For instance, if the propositional operators $\wedge$ and $\neg$ are available, then $\vee$ is obtained as $\varphi \vee \eta \Leftrightarrow \neg(\neg\varphi \wedge \neg\eta)$, but if only $\wedge$ and $\vee$ are available, then $\neg$ cannot be constructed. In Sect. 4 we demonstrated that $||$, hiding, and relational renaming suffice to represent both all forms of communication and the use of local or shared variables. However, this does not imply that all reasonable operators or all reasonable ways of building systems could be built from them. This subsection is devoted to this issue.

First we have to discuss what do we mean by representing an operator as a function of other operators. We use the interrupt operator $\triangleright$ as an example. If f is built from other operators than $\triangleright$, and if $f(L_1, L_2)$ is isomorphic to $L_1 \triangleright L_2$ for every LTSs L_1 and L_2, then it is clear that $L_1 \triangleright L_2$ can be built from the other operators. Intuition might suggest that it is not possible, and we will soon see that it is indeed the case.

However, requiring isomorphism is usually unnecessarily strict. For instance, if we are only interested in the trace semantics, then it suffices that $f(L_1, L_2)$ has the same trace semantics as $L_1 \triangleright L_2$. Indeed, such an f can be constructed only using $||$ and relational renaming. For simplicity, we ignore the issue of successful termination, although taking it into account would not be difficult.

Let Σ_1 and Σ_2 be the alphabets of L_1 and L_2, respectively. For $1 \leq i \leq 2$, let Φ_i rename each $a \in \Sigma_i$ to (a, i). The alphabets of $L_1\Phi_1$ and $L_2\Phi_2$ are disjoint. Let Σ be their union. Let Φ rename each $(a, 1)$ and each $(a, 2)$ in Σ to a. Let L be the LTS who has two states $\hat{s}_L$ and s_L, whose alphabet is Σ, and whose transitions are $\{ (\hat{s}_L, (a, 1), \hat{s}_L) \mid a \in \Sigma_1 \} \cup \{ (\hat{s}_L, (a, 2), s_L) \mid a \in \Sigma_2 \} \cup \{ (s_L, (a, 2), s_L) \mid a \in \Sigma_2 \}$. Each visible transition of $(L_1\Phi_1 \,||\, L_2\Phi_2 \,||\, L)\Phi$ consists of a visible transition of L and either L_1 or L_2 (but not both). Clearly L always allows L_2 to execute visible transitions. On the other hand, L allows L_1 to execute visible transitions only as long as L_2 has not executed any.

We see that $(L_1\Phi_1 \,||\, L_2\Phi_2 \,||\, L)\Phi$ behaves otherwise like $L_1 \triangleright L_2$, except that it is not the starting of L_2 but the first visible transition of L_2 that stops L_1 from executing visible transitions, and nothing stops L_1 from executing invisible transitions. However, these differences do not affect the traces. Therefore, $\Sigma((L_1\Phi_1 \,||\, L_2\Phi_2 \,||\, L)\Phi) = \Sigma(L_1 \triangleright L_2)$ and $Tr((L_1\Phi_1 \,||\, L_2\Phi_2 \,||\, L)\Phi) = Tr(L_1 \triangleright L_2)$.

On the other hand, we will now show that if the equivalence preserves the alphabet and stable failures, then there is no function f such that $f(L_1, L_2)$ is equivalent to $L_1 \triangleright L_2$ and f can be built from only $||$, hiding, relational renaming, and action prefix. Such equivalences include isomorphism and bisimilarity. They

also include the CFFD-equivalence and the divergence-preserving variants of weak and branching bisimilarity mentioned later in this section.

Proposition 19. *Let $\simeq$ be an equivalence that preserves the alphabet and stable failures. There is no function f that can be built from only parallel composition, hiding, relational renaming, and action prefix, such that $L_1 \rhd L_2 \simeq f(L_1, L_2)$ for every LTSs L_1 and L_2. A similar result holds if also the choice operator may be used in building f and $\simeq$ also preserves initial stability.*

Proof. Assume that f exists. Let $\approx$ be defined by $L \approx L'$ if and only if $\Sigma(L) = \Sigma(L')$ and $\mathit{SFail}(L) = \mathit{SFail}(L')$. With the choice operator, $\approx$ also requires that the initial state of either none or both of L and L' is stable. By [28], $\approx$ is a congruence with respect to the mentioned operators. Therefore, if $L_1 \approx L_1'$ and $L_2 \approx L_2'$, then $L_1 \rhd L_2 \simeq f(L_1, L_2) \approx f(L_1', L_2') \simeq L_1' \rhd L_2'$, yielding $L_1 \rhd L_2 \approx L_1' \rhd L_2'$. This means that $\approx$ is a congruence with respect to $\rhd$, which is in contradiction with [28]. □

Proposition 18 lets us to state the above result as follows: the interrupt operator cannot be built from other common process-algebraic operators, if the semantics in use preserves the alphabet and deadlocks and is a congruence with respect to $||$. However, we saw that with the trace semantics it was possible. We conclude that whether or not an operator can be built from other operators depends on the semantics in use. We also conclude the following:

Despite the great expressive power of $||$, hiding, and relational renaming, they cannot model all useful ways of building systems.

5.4 CFFD-Semantics

Information on livelocks can be added to the semantics in the form of *divergence traces*. We say that state s *diverges* if and only if an infinite sequence of τ-events can be started at s. A divergence trace is a trace that ends in a diverging state. For instance, all traces of Fig. 8 left are divergence traces. The set of divergence traces is denoted with $\mathit{DivTr}(L)$.

When divergence traces are added, then also something else must be done to the semantics to maintain the congruence property. One possibility is to add the *infinite traces* $\mathit{InfTr}(L)$. They are the infinite sequences of visible actions that arise from infinite executions that start at the initial state. The resulting semantics is called *Chaos-free failures divergences semantics* or *CFFD-semantics* [40]. We will explain the odd name soon.

The definition of CFFD-semantics as usually presented in the literature lacks the trace component. This is because it can be derived from other components: $\mathit{Tr}(L) = \mathit{DivTr}(L) \cup \{\, \sigma \mid (\sigma, \emptyset) \in \mathit{SFail}(L) \,\}$. So it is implicitly present even if it is not explicitly mentioned. If the LTSs are finite, also infinite traces can be left out for a similar reason. On the other hand, the initial stability bit must be added if the choice or interrupt operator is used. This is why variants of the CFFD-semantics have been presented in the literature.

An alternative way of solving the congruence problem caused by divergence traces is to only consider *minimal* divergence traces and ignore everything after them. A divergence trace is minimal if and only if none of its proper prefixes is a divergence trace. This is called *catastrophic divergence*, and a process that diverges initially is sometimes called *chaos*.

This solution arises naturally from the mathematics used for giving a meaning to recursive process definitions of the kind in Sect. 4.6. It was chosen as the main semantics of the well-known theory of *Communicating Sequential Processes* (*CSP*) [13,23,25]. Unfortunately, it implies that, for instance, the LTS in Fig. 8 left is equivalent to chaos. Thus no information on its behaviour other than that it diverges initially is preserved. This is a big drawback in many applications. Therefore, when CFFD-semantics was invented, its name was chosen to emphasize the similarity to CSP and the absence of chaos. Also CSP researchers admit that it would be nice to see beyond divergence [24].

Figure 8 left has been produced using CFFD-semantics. So the information that it gives on the deadlocks and livelocks of the system is real. There are no deadlocks, but sometimes the system can be in a state where it only can execute $\mathsf{g}\langle 1\rangle$ or uc, and sometimes only $\mathsf{g}\langle 1\rangle$ is possible. For instance, we can reason from the figure that if the user refuses to take the money, BANK will eventually execute uc if it has not done that already. This is because according to Fig. 2, ATM cannot send the "done" message before the user has taken the money. The many livelocks in Fig. 8 arise from the possibility of the user trying again and again, repeatedly getting loss of connection. If a yes-answer gets through to ATM, then ATM must execute the $\mathsf{g}\langle 1\rangle$-transition to continue, which is seen in Fig. 8 as commitment to $\mathsf{g}\langle 1\rangle$ perhaps together with uc.

Readers of the figures produced with CFFD-semantics — the present author included — are sometimes confused by questions like the following. Absence of livelock implies that ATM is in its "yes"-branch. From there, all paths to livelocks go via the $\mathsf{g}\langle 1\rangle$-transition. There is no livelock in the centre state of Fig. 8 but there is in the bottom middle state, and the transition linking these two states is labelled by uc and not $\mathsf{g}\langle 1\rangle$. What went wrong in the reasoning?

The answer to this question is that CFFD-semantics does not preserve information about deadlocks and livelocks in other states of an execution than the last. Therefore, deadlocks and livelocks must only be analysed at the end states of executions, not during intermediate states. If the execution ends at the bottom middle state, then one must read deadlocks and livelocks only there, not in the centre state of the figure.

It would be possible to draw an LTS from which deadlock and livelock information could be read also in the middle of an execution. However, it would be bigger than the one in Fig. 8 left. There is a trade-off.

The more information is preserved, the bigger are the resulting LTSs.

This principle works both when choosing the set of visible actions and when choosing the semantics.

A variant of CFFD-semantics called *NDFD-* or *nondivergent failures divergences* semantics [14,33] is the weakest congruence that preserves all stuttering-insensitive properties expressible in classical linear temporal logic [19]. It does not preserve deadlocks, except when the congruence requirement forces it to do so. CFFD-semantics preserves the same and also deadlocks. This makes it useful for many but not all applications.

CFFD- and NDFD-semantics suffer from the same problem as process-algebraic methods in general: it is difficult to express so-called *fairness assumptions* that are used in temporal logics to guarantee progress. With fairness assumptions one could, for instance, remove livelocks in Fig. 8 left. Promising ideas towards solving this drawback were presented in [21,22], but, unfortunately, nobody has continued that research.

5.5 CFFD-Preorder

A *preorder* is a reflexive and transitive binary relation. A *precongruence* with respect to f is a preorder $\preceq$ such that if $L_1 \preceq L'_1, \ldots, L_n \preceq L'_n$, then $f(L_1, \ldots, L_n) \preceq f(L'_1, \ldots, L'_n)$. Every preorder induces an equivalence and every precongruence induces a congruence by $L \simeq L' \Leftrightarrow L \preceq L' \wedge L' \preceq L$. A precongruence that induces CFFD-equivalence is obtained by $L \preceq L' \Leftrightarrow \Sigma(L) = \Sigma(L') \wedge \mathit{SFail}(L) \subseteq \mathit{SFail}(L') \wedge \mathit{DivTr}(L) \subseteq \mathit{DivTr}(L') \wedge \mathit{InfTr}(L) \subseteq \mathit{InfTr}(L')$. We call it *CFFD-preorder*. The reason for requiring equality of alphabets instead of the subset relation is too technical to be discussed here [33].

Let $\preceq$ denote CFFD-preorder. If $L \preceq L'$, then whatever trace, stable failure, divergence trace, or infinite trace L can do, also L' can do, but not necessarily vice versa. In particular, if L' cannot do anything wrong — cannot execute a wrong visible action, cannot deadlock when it is not allowed to, and cannot livelock when it is not allowed to — then also L cannot do anything wrong. So correctness of L' implies the correctness of every L that satisfies $L \preceq L'$. Indeed, there is a theorem saying that if L' satisfies a stuttering-insensitive linear temporal logic formula and $L \preceq L'$, then also L satisfies the formula [33].

This implies that we need not know the components of a system precisely when verifying the correctness of the system. Instead, if we have many possible alternatives for a component, it suffices that we use those among them that are the biggest in CFFD-preorder. This is particularly important when also the users of the system are modelled. Sometimes the correctness of a system depends on the users to obey some rules. It is often easy to model the CFFD-biggest user that obeys the rules. If the system works correctly with it, then it works correctly with all users that obey the rules.

This also means that often verification does not consist of checking whether the system is equivalent to the specification but whether the system is at most the specification. Systems are often allowed to be better than their specifications. If we buy a fifo queue with capacity 3 and get a fifo queue with capacity 4 for the same price, we do not mind, although it is not equivalent to the specification.

Checking CFFD-equivalence of two LTSs is **PSPACE**-complete. Checking CFFD-preorder is **PSPACE**-complete in the size of the specification LTS but

polynomial time in the size of the system LTS. This is fortunate, because the system LTS is usually a parallel composition of components and thus much bigger than the specification LTS. Similar remarks apply to trace semantics, CSP, and linear temporal logic. In Sect. 5.6 we will see that all these semantics are called "linear time".

> Verification of linear-time properties is typically polynomial time in the size of the state space of the system and **PSPACE**-complete in the size of the state space of the specification.

To get a feeling of CFFD-preorder, let us discuss its extreme elements. For every alphabet Σ, there is a maximum element, that is, L' such that every L with the same alphabet satisfies $L \preceq L'$. It has an initial state s_1 and another state s_2, and the transitions $s_1 -\tau\rightarrow s_2$, $s_1 -\tau\rightarrow s_1$, and $s_1 -a\rightarrow s_1$ for every $a \in \Sigma$. Its traces are Σ^*, all traces are divergence traces, and (σ, Σ) is a stable failure for every trace σ. Also its set of infinite traces is maximal.

On the other hand, there is no minimum element. The LTS that has no transitions has no divergence traces, while the LTS that has a τ-transition from its initial state to itself but no other transitions has no stable failures. Thus a minimum element must have no divergence traces and no stable failures. However, ε is a trace of every LTS, so each LTS has the divergence trace ε or the stable failure $(\varepsilon, \emptyset)$. According to the above-mentioned theorem about CFFD-preorder and linear temporal logic, a minimum element would satisfy all satisfiable stuttering-insensitive formulas. It would thus satisfy all satisfiable specifications. It would be a single system that is good for everything! It is a sign of the healthiness of our theory that such a system does not exist.

With trace preorder, the LTS that has no transitions is a minimum element. Indeed, it satisfies all specifications that can be formulated in trace semantics. Trace semantics only preserves stuttering-insensitive safety properties. Safety properties require that the system must never do anything wrong. The LTS that has no transitions does not ever do anything wrong, because it does not ever do anything. We see that a specification formalism is not complete unless it can specify that the system must do something. Trace preorder cannot do that, but CFFD-preorder can, by disallowing divergence traces and stable failures appropriately.

Although CFFD-preorder has no minimum element, it has minimal elements. It is useful to know that if σ is a trace, then it is a divergence trace or $(\sigma, \emptyset)$ is a stable failure or both. If $(\sigma, \emptyset)$ is a stable failure, then, for every visible action a, σa is a trace or $(\sigma, \{a\})$ is a stable failure or both. Minimal elements are obtained by avoiding the option "both" and by restricting divergence traces to the minimal ones. We skip the proof (infinite traces cause some trouble).

Proposition 20. *An LTS L is CFFD-minimal if and only if for every $\sigma \in Tr(L)$ either*

- $(\sigma, \emptyset) \notin SFail(L)$ *and for every* $a \in \Sigma(L)$, $\sigma a \notin Tr(L)$*; or*
- $\sigma \notin DivTr(L)$ *and for every* $a \in \Sigma(L)$, $\sigma a \notin Tr(L)$ *or* $(\sigma, \{a\}) \notin SFail(L)$.

That is, if L livelocks immediately after some trace σ, then it cannot do anything else nor refuse anything after σ; and if it does not livelock immediately after σ, then, for each visible action a, it can execute a as the next visible action after either no or every way of executing σ. An important theme here is that if L can do or refuse something after one way of executing a trace, then it can do or refuse the same after every way of executing the same trace. In other words, *CFFD-minimal systems are deterministic.*

This statement is not a theorem but an intuitive statement, because we have not made it precise what deterministic means, but rely on intuition. The classical definition used in automata theory does not apply, because, for instance, it declares nondeterministic the LTS $s_1 \leftarrow a - \hat{s} - a \rightarrow s_2$. We will return to this issue in Sect. 5.7. There we can also tackle the opposite question, that is, are all deterministic systems CFFD-minimal.

Roughly speaking, the smaller a system is in CFFD-preorder, the more deterministic it is, and vice versa.

We did not model the user in the cash dispenser system. The unmodelled user corresponds to the LTS that has one state, a transition for each $a \in \Sigma$ from that state to itself, and nothing else. Proposition 20 implies that this user is not the most general reasonable user. It is CFFD-minimal and thus only represents itself in verification. The most general reasonable user of the cash dispenser system can be modelled as an LTS with three states and the transitions $\hat{s} - \mathsf{ci} \rightarrow s_1$, $\hat{s} - a \rightarrow \hat{s}$ for $a \in \Sigma \setminus \{\mathsf{ci}\}$, $\hat{s} - \tau \rightarrow s_2$, $s_1 - \mathsf{co} \rightarrow \hat{s}$, and $s_1 - a \rightarrow s_1$ for $a \in \Sigma \setminus \{\mathsf{co}\}$, where Σ consists of ci, co, nm, lc, and $\mathsf{w}\langle i \rangle$ and $\mathsf{g}\langle i \rangle$ for every $i \in \mathbb{N}$. This differs from the unmodelled user in that it can deadlock when the card is not in, modelling the possibility of the user going away and never again trying to withdraw money. The model involves the assumption that the user will not go away while the card is in.

For reasoning about the progress properties of the system from the point of view of the user, it is a good idea to make ci and co visible and the remaining actions invisible. With the unmodelled user, this produces the two-state LTS where ci and co alternate. With the model of the user described above, a deadlock is added to the initial state of the previous result. From these it is clear that the system does not livelock, and it deadlocks only when the user goes away while the card is not in.

The modelling of components and the interpretation of the resulting pictures when using CFFD-semantics was discussed in detail in [38].

A specification of a system or an assumption about its component must often be nondeterministic, to leave enough room for different valid implementations and users.

Often preorders are more appropriate than equivalences for comparing systems against specifications.

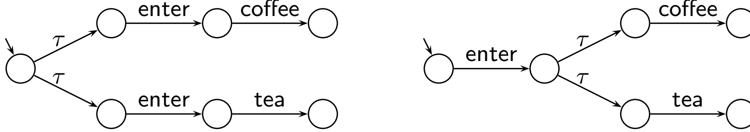


Fig. 11. Two CFFD-equivalent but not observation equivalent LTSs

5.6 Weak and Branching Bisimilarity

A famous abstract equivalence that is not based on sets of traces or failures is the *observation equivalence* of CCS, also known as *weak bisimilarity* [20]. Its definition resembles the definition of bisimilarity, but uses the $=\cdots\Rightarrow$-relation. Every visible action, possibly preceded and followed by invisible actions, must be simulated by a visible action, possibly preceded and followed by invisible actions. Also every (possibly empty) sequence of invisible actions must be simulated by a (possibly empty) sequence of invisible actions.

Definition 21. *Let $(S, \Sigma, \Delta, \hat{s})$ be an LTS. The relation "$\simeq$" $\subseteq S \times S$ is a* weak bisimulation*, if and only if for every $s_1 \in S$, $s_2 \in S$, $s \in S$, and $a \in \Sigma \cup \{\varepsilon\}$ such that $s_1 \simeq s_2$ the following hold:*

- *If $s_1 =a\Rightarrow s$, then there is $s' \in S$ such that $s_2 =a\Rightarrow s'$ and $s \simeq s'$.*
- *If $s_2 =a\Rightarrow s$, then there is $s' \in S$ such that $s_1 =a\Rightarrow s'$ and $s' \simeq s$.*

Two LTSs are observation equivalent *if and only if they have the same alphabet and their disjoint union has a weak bisimulation such that the initial states simulate each other.*

The choice and interrupt operators cause a congruence problem also to observation equivalence, so a variant known as *observation congruence* has been defined. Another variant is obtained by making the equivalence sensitive to livelocks, by requiring that if $s_1 \simeq s_2$, then either neither or both of s_1 and s_2 diverge. This equivalence is strictly stronger than CFFD-equivalence.

Observation equivalence is a *branching time* concept, while CFFD-equivalence is *linear time*. That is, individual executions and properties of their end states suffice for checking CFFD-equivalence, while observation equivalence requires a tree-like structure (or graph). Figure 11 left shows a professor who silently chooses between coffee and tea, and then enters a cafeteria and takes what she chose. The one on the right is otherwise similar, but makes the choice after entering the cafeteria (but without ensuring that both are available). They are CFFD-equivalent but not observation equivalent.

In Definition 21, the first and last state of the simulating execution must simulate the first and last state of the simulated execution, but the intermediate states need not simulate any states. In *branching bisimilarity* [41], also the intermediate states must simulate states along the simulated sequence. It is strictly stronger than observation equivalence. In its extension that takes divergences into account, it does not suffice that diverging states are simulated

by diverging states. Instead, each infinite sequence of invisible actions must be simulated by an infinite sequence of invisible actions. The resulting equivalence preserves [5] stuttering-insensitive computation tree logic [6] similarly to how CFFD-equivalence preserves classical stuttering-insensitive linear temporal logic.

5.7 Operational Determinism

"Deterministic" has an established definition in automata theory. A direct translation of the definition to LTSs is that an LTS $(S, \Sigma, \Delta, \hat{s})$ is deterministic if and only if it has no τ-transitions, and for every $a \in \Sigma$, $s \in S$, $s_1 \in S$, and $s_2 \in S$, if $s -a\rightarrow s_1$ and $s -a\rightarrow s_2$, then $s_1 = s_2$. This definition is not useful in process algebras, because it deems very few LTSs deterministic, and determinism is not preserved by bisimilarity. For instance, the LTSs $\hat{s} -a\rightarrow s_1$ and $s_1 \leftarrow a- \hat{s} -a\rightarrow s_2$ are bisimilar but only the former is deterministic.

Motivated by the discussion in Sect. 5.5, we could define that an LTS is deterministic if and only if it is CFFD-minimal. This is similar to the definition in CSP [23], except that there divergence is treated differently. However, we would like the definition not be tied to any particular semantics. Furthermore, we would like the definition to deem deterministic as many LTSs as possible, because we will soon present a proposition whose usefulness benefits from that. The following notion [10] is suitable.

Definition 22. *LTS $(S, \Sigma, \Delta, \hat{s})$ is* operationally deterministic*, if and only if for every $\sigma \in \Sigma^*$, $s_1 \in S$, and $s_2 \in S$, if $\hat{s} =\sigma\Rightarrow s_1$ and $\hat{s} =\sigma\Rightarrow s_2$, then the following hold:*

- *for every $a \in \Sigma$, if $s_1 =a\Rightarrow$, then $s_2 =a\Rightarrow$; and*
- *if s_1 diverges, then s_2 diverges.*

CFFD-minimal LTSs are precisely the LTSs that are operationally deterministic and satisfy the following condition: for every $\sigma \in DivTr(L)$ and $a \in \Sigma$, $\sigma a \notin Tr(L)$. That is, Proposition 20 requires that after executing a divergence trace, the LTS cannot do anything else than diverge; but Definition 22 does not require so.

Now we can state a proposition about operationally deterministic LTSs. Again, we skip the proof [10].

Proposition 23. *If LTSs L and L' are operationally deterministic, $\Sigma(L) = \Sigma(L')$, $Tr(L) = Tr(L')$, and $DivTr(L) = DivTr(L')$, then L and L' are branching bisimilar, divergence-preserving branching bisimilar, observation equivalent, divergence-preserving observation equivalent, CFFD-equivalent, NDFD-equivalent, and CSP-equivalent.*

This result has the practical application that if an LTS is operationally deterministic (and that can be tested very efficiently), then it can be processed with any algorithm that preserves the alphabet, traces, divergence traces, and operational

determinism, and the result is valid in each of the mentioned semantics. It has also the philosophical message that a multitude of semantics in process algebras collapses if systems are operationally deterministic. A similar result for semantics that ignore divergences, that does not assume that $\mathit{DivTr}(L) = \mathit{DivTr}(L')$, was developed in [7,42]. Unfortunately, to make the collapse also cover congruences with respect to the choice operator, something extra is needed. For instance, the requirement that $\hat{s}$ is stable may be added to the formulation of operational determinism.

That there are so many different semantics in process algebras is largely because operational nondeterminism is an important feature in concurrency. For instance, if all systems were operationally deterministic, the distinction between linear time and branching time would disappear.

5.8 Verification Techniques

In this subsection we mention some verification techniques that are related to the theory in this section. Detailed information can be found in the cited sources, and in many cases also in the tutorials [30,32,34].

A basic method is *compositional LTS construction*. It means putting some components of the system together, reducing their joint behaviour, putting the result together with the reduced behaviour of a neighbouring subsystem and so on, until a reduced behaviour of the system as a whole is obtained. The semantics that is used must be a congruence with respect to the operators used in building the system. The first explicit mentionings of this idea are perhaps in [27,18], but the idea is so obviously built into process-algebraic theories that it has certainly been known before that.

Reducing the behaviour means applying some algorithm to the LTS that produces an equivalent but (hopefully) smaller LTS. With bisimilarity-based equivalences, there is a unique smallest equivalent LTS, and it can be found in polynomial time [15]. With trace- and failure-based equivalences, smallest equivalent LTSs are not necessarily unique, and finding one is **PSPACE**-hard. The problem is a generalization of the problem of finding a minimal (not necessarily deterministic) finite automaton that accepts the same language as a given finite automaton.

Fortunately, it is not necessary to find a minimal LTS, it suffices that it is equivalent and small. So one can use heuristic algorithms that run in polynomial time. Furthermore, algorithms based on the well-known determinization and minimization algorithms of finite automata have been extended to the failure semantics world, and they have been reported to run reasonably well in practice, e.g., [4,23,39]. Please see [29] for more comments on the relative efficiency of verification using bisimulation-based vs. failure-based semantics.

Stubborn set methods save effort during the computation of parallel composition by leaving out orderings of events that have the same effect as other orderings that are not left out. This type of methods are also called "partial order". Stubborn set methods for the major process-algebraic semantics were presented in [31].

Independently of the semantics, compositional LTS construction suffers from the *spurious behaviour* problem. That is, the behaviour of a subsystem may be much bigger than the behaviour of the system as a whole. This is because systems often obey invariants that strongly restrict the possible combinations of local states of the components, but, in isolation, subsystems do not necessarily obey them. Consider a fifo queue of capacity k that can store two different messages. In isolation it has $1 + 2 + 2^2 + \cdots + 2^k = 2^{k+1} - 1$ states. When the same queue is in the well-known alternating bit protocol [2], there can be at most one place where successive messages are different. This reduces the number of possible states to k^2+k+1 (one empty queue, $2k$ with only one type of messages, and $2 \cdot \frac{1}{2}k(k-1)$ with two message types).

The spurious behaviour problem was pointed out and a solution for observation equivalence was presented in [9]. A general solution that applies to many semantics was presented in [16]. A key idea in these solutions is an *interface specification*, also known as *interface process*. It represents an assumption about the behaviour of the subsystem when it is within the system as a whole. For instance, one may represent the assumption that there is at most one place in the queue where successive messages are different. If the assumption is incorrect, then that is detected at the end of compositional LTS construction. Independently of whether it is correct, the interface process reduces the LTS of the subsystem. Interface processes require the addition of the notion of "undefined" to the semantics, but [16] shows how it can be done with very little need to rewrite tools.

If the result of compositional LTS construction is small, it can be analysed visually, like we did for Fig. 6 and 8. If it is small or big, one can compare it to a reference LTS with an equivalence or preorder checking algorithm. Preorder checking has the advantage that it can be done *on-the-fly*, that is, simultaneously with the computation of the behaviour of the system. This is a big advantage, because incorrect systems tend to have lots of spurious behaviour that the corresponding correct systems do not have. With on-the-fly verification, the computation may be terminated when the first counter-example has been found, so that most of the spurious behaviour will not be computed. Preorder checking is the major verification method with the *FDR* (Failures-Divergences Refinement) tool [23]. A CFFD-preorder version of this idea was presented in [12].

Some systems contain many identical components. Sometimes the behaviour of a subsystem with $n + 1$ components turns out equivalent to the behaviour with n components. (The probability of this happening can be increased with interface processes.) Then a simple induction argument yields that the behaviour of the system is the same for all numbers of the replicated components starting from n. Particularly intriguing applications of this idea were presented in [37], such as proving that the behaviour of a protocol is independent of the (finite) maximum number of times that it may re-transmit a message before giving up. The idea can also be used with precongruences: if $L \preceq L'$ and $f(L', K) \preceq L'$, then $f(L, K) \preceq f(L', K) \preceq L'$, yielding $f(f(L, K), K) \preceq f(L', K) \preceq L'$ and $f(\cdots f(f(f(L, K), K), K) \cdots, K) \preceq L'$ for any number of K-components. Outside process algebras, the idea has been presented in [43], among others.

In [26] it was pointed out that the notion of determinism is related to computer security. Consider a system with a trusted user and an untrusted user. The untrusted user must get no information about the behaviour of the trusted user. This holds, if the untrusted user's view to the system is deterministic. We already pointed out that there is a fast algorithm for checking determinism.

6 Conclusions

Compositionality at the structural level is routine in computer science and software engineering. There are modules, classes, and other kinds of units, and they can be nested. There are hierarchical Petri nets. The situation with compositionality of concurrent systems at the semantic level is confusing. On one hand, the idea is natural and it seems that it attracts many researchers. On the other hand, many widely valid basic facts have been found by process algebra research well before the year 2000, but seem little known.

The present author believes that one, but not the only, reason why process-algebraic results have failed to break through is that process-algebraic languages are cryptic and lead to cryptic fixed-point theories of semantics.

In this tutorial we have tried to make it clear that the semantic models are not tied to the cryptic languages, but apply to concurrent systems in general. We replaced recursion-based definitions of individual processes by state machines. For composing the system from its components, a small and natural set of operators was employed, and it was shown that in the end there are just synchronization patterns where some components do not participate, each one who participates does that by executing a modeller-chosen visible action, and the result has a modeller-chosen (not necessarily visible) action. Synchronous communication may seem unnatural and restricted, but we pointed out that it is the raw material from which all kinds of communication could be constructed. On the other hand, our formalism does not cover all reasonable ways of building systems, such as on-the-fly creation and abortion of processes.

The absence of event parameters from the definition of LTSs does not make the semantic theory incapable of processing them. It is just that the semantic theory is insensitive to event parameters, so it is easiest and most general to treat actions as arbitrary symbols. The user may assume any internal structure for actions (as long as τ remains invisible). For instance, when modelling transition fusion of coloured Petri nets, it may be useful to assume that actions contain tuples of data values.

Another reason for the lack of use of process-algebraic semantic models is that as such, compositional LTS construction often fails because of the spurious behaviour problem. People may have tried the basic form of compositional LTS construction, got disappointed, and rejected process algebras.

Again, this issue is not specific to process algebras but inherent in the compositional construction of behaviours. Interface processes help a lot, but they require making and modelling guesses about the behaviours of subsystems. So

their use is not fully automatic, reducing their attraction. Furthermore, it may be that the wide applicability of interface processes is not widely known. In any case, more case studies are needed to find out if interface processes are a sufficient solution.

It seems to the present author that current verification methods can process nontrivial systems, so the methods are useful, but they can process systems of industrial size only occasionally, so the methods do not meet the needs and are thus not used a lot. This holds for both process-algebraic compositional methods and verification methods in general.

In this tutorial we concentrated on the CFFD semantics. The reason is that often either it or a closely related semantics is very good for a task. If livelocks are not interesting but deadlocks are, throw divergences and infinite traces away but keep traces and stable failures. If deadlocks are not interesting but livelocks are, throw stable failures away but keep the rest. The main semantics of CSP can be used if it does not matter that there divergence is catastrophic. If branching-time properties are needed, then CFFD and CSP cannot be used, but some variant of weak or branching bisimilarity may be suitable.

We pointed out that use of a variable is essentially the same thing as parallel composition with it, and unfolding a variable can be postponed until the components of the system have been put together. They can often be put together in a stepwise manner in many different orderings. These open up possibilities for new research, to develop verification methods that apply to systems with variables but do not suffer from the effect that unfolding has to the size of the state space. Such methods must be capable of combining data manipulation steps from successive transitions, to liberate τ-transitions from data manipulation and thus make it possible to remove them in reductions. Also management of variable names is necessary, because data moves from one component to another, so names of local variables within components are not helpful in projected views. This problem was solved manually when drawing Fig. 6.

Because of networked systems and multi-core processors, today there is more need than ever to teach students basic facts about concurrency. In particular, it is important to make them realize how things may go wrong. Perhaps projected views such as in Fig. 6 and 8 can be used for that purpose.

Acknowledgements. The comments by the anonymous reviewers helped to improve this tutorial.

References

1. Arnold, A.: Finite Transition Systems. Prentice-Hall, Englewood Cliffs (1994)
2. Bartlett, K.A., Scantlebury, R.A., Wilkinson, P.T.: A Note on Reliable Full-Duplex Transmission over Half-Duplex Links. Communications of the ACM 12(5), 260–261 (1969)
3. Bolognesi, T., Brinksma, E.: Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems 14(1), 25–59 (1987)

4. Cleaveland, R., Hennessy, M.: Testing Equivalence as a Bisimulation Equivalence. Formal Aspects of Computing 5(1), 1–20 (1993)
5. De Nicola, R., Vaandrager, F.: Three Logics for Branching Bisimulation. Journal of the ACM 42(2), 458–487 (1995)
6. Emerson, E.A.: Temporal and Modal Logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, pp. 995–1072. The MIT Press/Elsevier (1990)
7. Engelfriet, J.: Determinacy $\longrightarrow$ (Observation Equivalence = Trace Equivalence). Theoretical Computer Science 36(1), 21–25 (1985)
8. Fernandez, J.-C.: An Implementation of an Efficient Algorithm for Bisimulation Equivalence. Science of Computer Programming 13(2-3), 219–236 (1989/1990)
9. Graf, S., Steffen, B., Lüttgen, G.: Compositional Minimisation of Finite State Systems Using Interface Specifications. Formal Aspects of Computing 8(5), 607–616 (1996)
10. Hansen, H., Valmari, A.: Operational Determinism and Fast Algorithms. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 188–202. Springer, Heidelberg (2006)
11. Hansen, H., Virtanen, H., Valmari, A.: Merging State-based and Action-based Verification. In: Lilius, J., Balarin, F., Machado, R.J. (ed.) 3rd International Conference on Application of Concurrency to System Design (ACSD 2003), pp. 150–156. IEEE Computer Society (2003)
12. Helovuo, J., Valmari, A.: Checking for CFFD-Preorder with Tester Processes. In: Graf, S., Schwartzbach, M. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 283–298. Springer, Heidelberg (2000)
13. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
14. Kaivola, R., Valmari, A.: The Weakest Compositional Semantic Equivalence Preserving Nexttime-less Linear Temporal Logic. In: Cleaveland, R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 207–221. Springer, Heidelberg (1992)
15. Kanellakis, P.C., Smolka, S.A.: CCS Expressions, Finite State Processes, and Three Problems of Equivalence. Information and Computation 86(1), 43–68 (1990)
16. Kangas, A., Valmari, A.: Verification with the Undefined: A New Look. In: Arts, T., Fokkink, W. (eds.) Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2003). Electronic Notes in Theoretical Computer Science, vol. 80, pp. 124–139 (2003)
17. Karsisto, K.: A New Parallel Composition Operator for Verification Tools. Dr.Tech. Thesis, Tampere University of Technology Publications 420, Tampere, Finland (2003)
18. Madelaine, E., Vergamini, D.: AUTO: A Verification Tool for Distributed Systems Using Reduction of Finite Automata Networks. In: Vuong, S.T. (ed.) Formal Description Techniques II (FORTE 1989), pp. 61–66. North-Holland (1990)
19. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems, vol. I: Specification. Springer (1992)
20. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
21. Puhakka, A.: Weakest Congruences, Fairness and Compositional Process-Algebraic Verification. Dr.Tech. Thesis, Tampere University of Technology Publications 468, Tampere, Finland (2004)

22. Puhakka, A., Valmari, A.: Liveness and Fairness in Process-Algebraic Verification. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 202–217. Springer, Heidelberg (2001)
23. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1998)
24. Roscoe, A.W.: Seeing Beyond Divergence. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) CSP25. LNCS, vol. 3525, pp. 15–35. Springer, Heidelberg (2005)
25. Roscoe, A.W.: Understanding Concurrent Systems. Springer (2010)
26. Roscoe, A.W., Woodcock, J.C.P., Wulf, L.: Non-interference through Determinism. Journal of Computer Security 4(1), 27–54 (1996)
27. Sabnani, K.K., Lapone, A.M., Uyar, M.Ü.: An Algorithmic Procedure for Checking Safety Properties of Protocols. IEEE Transactions on Communications 37(9), 940–948 (1989)
28. Valmari, A.: The Weakest Deadlock-Preserving Congruence. Information Processing Letters 53(6), 341–346 (1995)
29. Valmari, A.: Failure-Based Equivalences Are Faster Than Many Believe. In: Desel, J. (ed.) Structures in Concurrency Theory 1995. Workshops in Computing, pp. 326–340. Springer (1995)
30. Valmari, A.: Compositionality in State Space Verification Methods. In: Billington, J., Reisig, W. (eds.) ICATPN 1996. LNCS, vol. 1091, pp. 29–56. Springer, Heidelberg (1996)
31. Valmari, A.: Stubborn Set Methods for Process Algebras. In: Peled, D.A., Pratt, V.R., Holzmann, G.J. (eds.) Partial Order Methods in Verification: DIMACS Workshop. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 29, pp. 213–231. American Mathematical Society (1997)
32. Valmari, A.: The State Explosion Problem. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998)
33. Valmari, A.: A Chaos-Free Failures Divergences Semantics with Applications to Verification. In: Davies, J., Roscoe, B., Woodcock, J. (eds.) Millennial Perspectives in Computer Science, Proceedings of the 1999 Oxford–Microsoft Symposium in Honour of Sir Tony Hoare, pp. 365–382. Palgrave (2000)
34. Valmari, A.: Composition and Abstraction. In: Cassez, F., Jard, C., Rozoy, B., Ryan, M.D. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 58–98. Springer, Heidelberg (2001)
35. Valmari, A.: Simple Bisimilarity Minimization in $O(m \log n)$ Time. Fundamenta Informaticae 105(3), 319–339 (2010)
36. Valmari, A., Kervinen, A.: Alphabet-Based Synchronisation is Exponentially Cheaper. In: Brim, L., Jančar, P., Křetínský, M., Kučera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 161–176. Springer, Heidelberg (2002)
37. Valmari, A., Kokkarinen, I.: Unbounded Verification Results by Finite-State Compositional Techniques: 10^{any} States and Beyond. In: Lavagno, L., Reisig, W. (eds.) International Conference on Application of Concurrency to System Design (ACSD 1998), pp. 75–85. IEEE Computer Society (1998)
38. Valmari, A., Setälä, M.: Visual Verification of Safety and Liveness. In: Gaudel, M.-C., Woodcock, J. (eds.) FME 1996. LNCS, vol. 1051, pp. 228–247. Springer, Heidelberg (1996)
39. Valmari, A., Tienari, M.: An Improved Failures Equivalence for Finite-State Systems with a Reduction Algorithm. In: Jonsson, B., Parrow, J., Pehrson, B. (eds.) Protocol Specification, Testing and Verification XI, pp. 3–18. North-Holland (1991)

40. Valmari, A., Tienari, M.: Compositional Failure-Based Semantic Models for Basic LOTOS. Formal Aspects of Computing 7(4), 440–468 (1995)
41. van Glabbeek, R.J., Weijland, W.P.: Branching Time and Abstraction in Bisimulation Semantics. Journal of the ACM 43(3), 555–600 (1996)
42. Voorhoeve, M., Mauw, S.: Impossible Futures and Determinism. Information Processing Letters 80(1), 51–58 (2001)
43. Wolper, P., Lovinfosse, V.: Verifying Properties of Large Sets of Processes with Network Invariants. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 68–80. Springer, Heidelberg (1990)