Conjunction on Processes: Full-Abstraction via Ready-Tree Semantics ⋆

Gerald Lüttgen [1]

Department of Computer Science, University of York, York YO10 5DD, UK

Walter Vogler [2]

Institut für Informatik, Universität Augsburg, D-86135 Augsburg, Germany

Abstract

A key problem in mixing operational (e.g., process-algebraic) and declarative (e.g., logical) styles of specification is how to deal with inconsistencies arising when composing processes under conjunction. This article introduces a conjunction operator on labelled transition systems capturing the basic intuition of "*a and b = false*", and considers a naive preorder that demands that an inconsistent specification can only be refined by an inconsistent implementation.

The main body of the article is concerned with characterising the largest precongruence contained in the naive preorder. This characterisation will be based on what we call ready-tree semantics, which is a variant of path-based possible-worlds semantics. We prove that the induced ready-tree preorder is compositional and fully-abstract, and that the conjunction operator indeed reflects conjunction.

The article's results provide a foundation for, and an important step towards a unified framework that allows one to freely mix operators from process algebras and linear-time temporal logics.

Key words: Labelled transition system, conjunction, consistency preorder, ready-tree semantics, ready-tree preorder, full abstraction.

⋆ An extended abstract appeared in L. Aceto and A. Ingólfsdóttir, eds., *Intl. Conf. on Foundations of Software Science and Computation Structures* (FOSSACS 2006), vol. 3921 of Lecture Notes in Computer Science, pp. 261–276, Vienna, Austria, 2006. Springer-Verlag.

[1] Research support was partially provided by the NSF under grant CCR-9988489.

[2] Corresponding author. Email: walter.vogler@informatik.uni-augsburg.de; phone: +49 821 598-2120; fax: +49 821 598-2175

1 Introduction

Process algebra [1] and *temporal logic* [2] are two popular approaches to formally specifying and reasoning about reactive systems. The process-algebraic paradigm is founded on notions of *refinement*, where one typically formulates a system specification and its implementation in the same notation and then proves using *compositional reasoning* that the latter refines the former. The underlying semantics is often given operationally, and refinement relations are formalised as precongruences. In contrast, the temporal-logic paradigm is based on the use of temporal logics to formulate specifications abstractly, with implementations being denoted in an operational notation. One then verifies a system by establishing that it is a model of its specification.

Recently, two papers have been published aimed at marrying process algebras and temporal logics [3,4]. While the first paper introduces a semantic framework based on Büchi automata, the second paper considers labelled transition systems augmented with an "unimplementability predicate". This predicate captures *inconsistencies* arising when composing processes conjunctively; e.g., the composition $a \wedge b$ is contradictory since a run of a process cannot begin with both actions a and b. Note that one cannot simply interpret conjunction as synchronous composition and ignore inconsistencies. Otherwise, $a \wedge b$ would be deemed equivalent to the deadlock process $\mathbf{0}$. Hence, $\mathbf{0}$ would implement $a \wedge b$, although it neither implements a or b in any deadlock-sensitive implementation relation. The frameworks in [3,4] are equipped with a refinement preorder based on De Nicola and Hennessy's must-testing preorder [5]. However, the results obtained in [3,4] are unsatisfactory: the refinement preorder in [3] is not a precongruence, while the $\wedge$-operator in [4] is not *conjunction* with respect to the studied precongruence $\sqsubseteq$, i.e., it does not satisfy the law $p \wedge q \sqsubseteq r$ if and only if $p \sqsubseteq r$ and $q \sqsubseteq r$.

This article solves the deficiencies of [3,4] within a simple setting of labelled transition systems in which a state represents either an external (non-deterministic) or internal (disjunctive) choice. Moreover, states that are vacuously *true* or *false* are tagged accordingly. The tagging of *false* states, or inconsistent states, is given by an inductive *inconsistency predicate* that is defined very similar but subtly different to the unimplementability predicate of [4]. We then equip our setting with two operators: the conjunction operator $\wedge$ is in essence a synchronous composition on observable actions and an interleaving product on the unobservable action τ, but additionally captures inconsistencies; the disjunction operator $\vee$ simply resembles the process-algebraic operator of internal choice.

Our variant of labelled transition systems gives rise to a naive refinement preorder $\sqsubseteq_F$ requiring that an inconsistent specification cannot be refined except

by an inconsistent implementation. We characterise the *consistency preorder*, i.e., the largest precongruence contained in $\sqsubseteq_F$ when conjunctively closing under all contexts. To do so, we adapt van Glabbeek's *path-based possible-worlds semantics* [6] which in turn is motivated by the possible-worlds semantics of Veglioni and De Nicola [7]. We call the adapted semantics *ready-tree semantics* which is – at least when disallowing divergent behaviour – finer than both must-testing semantics [5] and ready-trace semantics [8], but coarser than ready simulation [9]. The resulting *ready-tree preorder* $\precsim$ is not only compositional for $\wedge$ and $\vee$ and fully-abstract with respect to $\sqsubseteq_F$, but also possesses several other desired properties. In particular, we prove that $\wedge$ ($\vee$) is indeed conjunction (disjunction) relative to $\precsim$, and that $\wedge$ and $\vee$ satisfy the expected boolean laws, such as the distributivity laws.

Our results are a significant first step towards the goal of developing a uniform calculus in which one can freely mix process-algebraic and temporal-logic operators. This will give engineers powerful tools to model system components at different levels of abstraction and to impose logical constraints on the execution behaviour of components. The proposed ready-tree preorder will allow engineers to step-wise and component-wise refine systems by trading off logical content for operational content.

Organisation

The next section presents our setting of labelled transition systems augmented with *true* and *false* predicates, together with a conjunction and a disjunction operator. Section 3 defines ready-tree semantics, addresses expressiveness issues of several ready-tree variants and introduces the ready-tree preorder. Our compositionality and full-abstraction results are proved in Section 4. The relation of our ready-tree preorder to established preorders is made precise in Section 5. Our framework is then extended by a parallel composition operator in Section 6, in which it is also applied to the structured specification and refinement-based design of *mode logics* of flight guidance systems. Finally, Section 7 discusses our results in light of related work, while Section 8 presents our conclusions and suggests directions for future research.

2 Labelled Transition Systems & Conjunction

This section first introduces our process-algebraic setting and particularly conjunctive composition informally, discusses semantic choices and their implications, and finally gives a formal account of our framework.

Our setting models processes as labelled transition systems, which may be composed conjunctively and disjunctively. As usual in process algebra, transition labels are actions taken from some alphabet $\mathcal{A} = \{a, b, \ldots\}$. When an action a is offered by the environment and the process under consideration is in a state having one or more outgoing a-transitions, the process must choose and perform one of them. If there is no outgoing a-transition, then the process stays in its state, at least in classical process-algebraic frameworks where the composition between a process and its environment is modelled using some parallel operator. However, in a conjunctive setting we wish to mark the composed state between process and environment as inconsistent, if the environment offers an action that the process cannot perform, or vice versa. Hence, taking ordinary synchronous composition as operator for conjunction is insufficient.

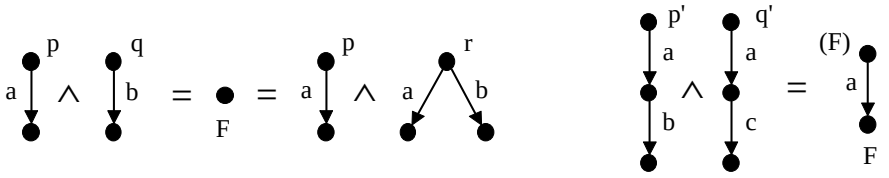

Fig. 1. Basic intuition behind conjunctive composition.

We illustrate this intuition behind our conjunction operator $\wedge$ and its implications by the example labelled transition systems of Fig. 1. First, consider the processes p, q and r. Process p and q specify that exactly action a and respectively action b is offered initially. Similarly, process r specifies that a and b are offered initially. From this perspective, $p \wedge q$ as well as $p \wedge r$ are *inconsistent* and should be tagged as such. Formally, our labelled transition systems will be augmented by an *inconsistency predicate* F, so that $p \wedge q$, $p \wedge r \in F$ in our example. We also refer to inconsistent states as *false*-states.

Now consider the conjunction $p' \wedge q'$ shown on the right in Fig. 1. Since both conjuncts require action a to be performed, $p' \wedge q'$ should have an a-transition. From the preceding discussion, this transition should lead to a *false*-state. No implementable process can meet these requirements of being able to perform a and being inconsistent afterwards. Thus, our inconsistency predicate will propagate backwards to the conjunction itself, as indicated in Fig. 1.

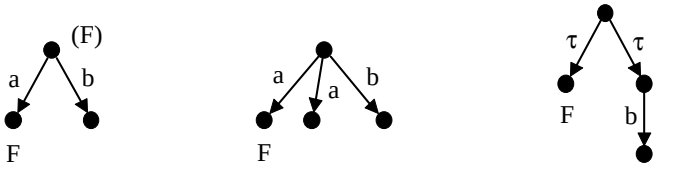

Fig. 2. Backward propagation of inconsistencies.

Fig. 2 shows more intricate examples of backward propagation. The inconsistency of the target state of the a-transition of the process on the left propagates backwards to its source state. This is the case although the source state is able to offer a transition leading to a consistent state. However, that transition can only be taken if the environment offers action b. The process is forced into the inconsistency when the environment offers action a.

The situation is different for the process in the middle, which has an additional a-transition leading to a consistent state. Here, the process is consistent, as it can choose to execute this new a-transition and thus avoid to enter a *false*-state. In fact, this choice can be viewed as a disjunction between the two a-branches. As an aside, note that in [4] the design decision was to consider a process already as inconsistent if some a-derivative is. While there might be an intuitive justification for that, it led to a setting where the implied conjunction operator did not reflect conjunction for the studied refinement preorder, i.e., where Thm. 21(1) did not hold.

Disjunction can be made explicit by using the classical *internal-choice* operator. This operator may as usual be expressed by employing the special, unobservable action $\tau \notin \mathcal{A}$ as shown on the right in Fig. 2. Hence, we may identify the internal-choice operator with the disjunction operator $\vee$ desired in our setting. Moreover, a disjunction $p \vee q$ is inconsistent if both p and q are *false*-states. In particular, the process on the right in Fig. 2 will represent *false*$\vee q$ in our approach, with q from Fig. 1, which clearly should be consistent.

2.2 Formalisation

For notational convenience we denote $\mathcal{A} \cup \{\tau\}$ by $\mathcal{A}_\tau$ and use $\alpha, \beta, \ldots$ as representatives of $\mathcal{A}_\tau$. We start off by defining our notion of *labelled transition system* (LTS). The LTSs considered here are augmented with a *false*-predicate F on states, as discussed above, and dually with a *true*-predicate T. A state in F represents inconsistent, empty behaviour, while a state in T represents completely underspecified, arbitrary behaviour.

Formally, an LTS is a quadruple $\langle P, \longrightarrow, T, F \rangle$, where P is the set of *processes* (states), $\longrightarrow \subseteq P \times \mathcal{A}_\tau \times P$ is the *transition relation*, and $T \subseteq P$ and $F \subseteq P$ are the *true-predicate* and *false-predicate*, respectively. We write $p \xrightarrow{\alpha} p'$ instead of $\langle p, \alpha, p' \rangle \in \longrightarrow$, $p \xrightarrow{\alpha}$ instead of $\exists p' \in P.\, p \xrightarrow{\alpha} p'$, and $p \longrightarrow$ instead of $\exists p' \in P,\, \alpha \in \mathcal{A}_\tau .\, p \xrightarrow{\alpha} p'$. When $p \xrightarrow{\alpha} p'$, we say that process p can perform an α-step to p', and we call p' an α-derivative of p. We also require an LTS to satisfy the following *τ-purity* condition: $p \xrightarrow{\tau}$ implies $\nexists a \in \mathcal{A}.\, p \xrightarrow{a}$, for all $p \in P$. Hence, each process represents either an external or internal (disjunctive) choice between its outgoing transitions. This restriction turns

out to be technically convenient, and we leave exploring the consequences of lifting it for future work.

The LTSs of interest to us need to satisfy four further properties, as stated in the following formal definition, where $\mathcal{I}(p)$ stands for the set $\{\alpha \in \mathcal{A}_\tau \mid p \xrightarrow{\alpha}\}$ of initial actions of process p, to which we also refer as *ready set*.

Definition 1 (Logic LTS) *An LTS* $\langle P, \longrightarrow, T, F\rangle$ *is a* logic LTS *if it satisfies the following conditions:*

(1) $T \cap F = \emptyset$
(2) $T \subseteq \{p \mid p \not\longrightarrow\}$
(3) $F \subseteq P$ *such that* $p \in F$ *if* $\exists \alpha \in \mathcal{I}(p)\, \forall p' \in P.\, p \xrightarrow{\alpha} p' \implies p' \in F$
(4) p *cannot stabilise (see below)* $\implies p \in F$

Naturally, we require that a process cannot be tagged *true* and *false* at the same time. As a *true*-process specifies arbitrary, full behaviour, any behaviour made explicit by outgoing transitions is already included implicitly; hence, any outgoing transitions may simply be cut off. The third condition formalises the backwards propagation of inconsistencies as discussed in the motivation section above.

The fourth condition relates to *divergence*, i.e., infinite sequences of τ-transitions. In many semantic frameworks, e.g. [10], divergence is considered catastrophic, while in our setting catastrophic behaviour is inconsistent behaviour. We view divergence only as catastrophic if a process cannot *stabilise*, i.e., if it cannot get out of an infinite, internal computation. While this is intuitive, there is also a technical reason to which we will come back shortly.

To formalise our notion of stabilisation, we first introduce a weak transition relation $\Longrightarrow_F \subseteq P \times (\mathcal{A}_\tau \cup \{\varepsilon\}) \times P$ (ε denoting the empty sequence), which is defined by:

(1) $p \overset{\varepsilon}{\Longrightarrow}_F p'$ if $p \equiv p' \notin F$, where $\equiv$ denotes syntactic equality;
(2) $p \overset{\varepsilon}{\Longrightarrow}_F p'$ if $p \notin F$ and $p \xrightarrow{\tau} p'' \overset{\varepsilon}{\Longrightarrow}_F p'$ for some p'';
(3) $p \overset{a}{\Longrightarrow}_F p'$ if $p \notin F$ and $p \xrightarrow{a} p'' \overset{\varepsilon}{\Longrightarrow}_F p'$ for some p''.

Our definition of a weak transition is slightly unusual: a weak transition cannot pass through *false*-states since these cannot occur in computations, and it does not abstract from τ-transitions preceding a visible transition. However, we only will use weak visible transitions from *stable* states, i.e., states with no outgoing τ-transition. Finally, we can now formalise stabilisation: a process p *can stabilise* if $p \overset{\varepsilon}{\Longrightarrow}_F p'$ for some stable p'.

Note that both Conds. (3) and (4) are inductively defined conditions. We refer to them as *fixed point conditions of* F *for LTS* . For convenience, we will often

write LTS instead of Logic LTS in the sequel. Moreover, whenever we mention a process p without stating a respective LTS explicitly, we assume implicitly that such an LTS $\langle P, \longrightarrow, T, F\rangle$ is given. We let tt (ff) stand for the *true* (*false*) process, which is the only process of an LTS with $tt \in T$ ($ff \in F$).

2.3 Operators

Our conjunction operator is essentially a synchronous composition for visible transitions and an asynchronous composition for τ-transitions. However, we need to take care of the T- and F-predicates.

Definition 2 (Conjunction Operator) *The conjunction of two Logic LTSs* $\langle P, \longrightarrow_P, T_P, F_P\rangle$, $\langle Q, \longrightarrow_Q, T_Q, F_Q\rangle$ *is the LTS* $\langle P \wedge Q, \longrightarrow_{P\wedge Q}, T_{P\wedge Q}, F_{P\wedge Q}\rangle$ *defined by:*

- $P \wedge Q =_{df} \{p \wedge q \,|\, p \in P,\, q \in Q\}$
- $\longrightarrow_{P\wedge Q}$ *is determined by the following operational rules:*

$$\begin{array}{rcl}
p \xrightarrow{\tau}_P p' & \Longrightarrow & p \wedge q \xrightarrow{\tau}_{P\wedge Q} p' \wedge q \\
q \xrightarrow{\tau}_Q q' & \Longrightarrow & p \wedge q \xrightarrow{\tau}_{P\wedge Q} p \wedge q' \\
p \xrightarrow{a}_P p',\ q \xrightarrow{a}_Q q' & \Longrightarrow & p \wedge q \xrightarrow{a}_{P\wedge Q} p' \wedge q' \\
q \in T_Q,\ p \xrightarrow{\alpha}_P p' & \Longrightarrow & p \wedge q \xrightarrow{\alpha}_{P\wedge Q} p' \wedge q \\
p \in T_P,\ q \xrightarrow{\alpha}_Q q' & \Longrightarrow & p \wedge q \xrightarrow{\alpha}_{P\wedge Q} p \wedge q'
\end{array}$$

- $p \wedge q \in T_{P\wedge Q}$ *if and only if* $p \in T_P$ *and* $q \in T_Q$
- $F_{P\wedge Q}$ *is the least subset of* $P \wedge Q$ *such that* $p \wedge q \in F_{P\wedge Q}$ *if at least one of the following conditions holds:*

(1) $p \in F_P$ *or* $q \in F_Q$
(2) $p \notin T_P$ *and* $q \notin T_Q$ *and* $p \wedge q \not\xrightarrow{\tau}_{P\wedge Q}$ *and* $\mathcal{I}(p) \neq \mathcal{I}(q)$
(3) $\exists \alpha \in \mathcal{I}(p \wedge q)\, \forall p' \wedge q'.\, p \wedge q \xrightarrow{\alpha}_{P\wedge Q} p' \wedge q' \implies p' \wedge q' \in F_{P\wedge Q}$
(4) $p \wedge q$ *cannot stabilise*

Note that the treatment of *true*-processes when defining $\longrightarrow_{P\wedge Q}$ and $T_{P\wedge Q}$ reflects our intuition that these processes allow arbitrary behaviour. We are left with explaining Conds. (1)-(4). Firstly, a conjunction is inconsistent if any conjunct is. Conds. (2) and (3) reflect our intuition of inconsistency and, respectively, backward propagation stated in the motivation section above. Cond. (4) is added to enforce Def. 1(4). We refer to Conds. (3) and (4) as *fixed point conditions of* F *for* $\wedge$.

It is easy to check that conjunction is well-defined, i.e., that the conjunctive composition of two Logic LTSs satisfies the four conditions of Def. 1. For Def. 1(1) in particular, note that $p \wedge q \in T_{P\wedge Q}$ does not satisfy any of the four

conditions for $F_{P \wedge Q}$.

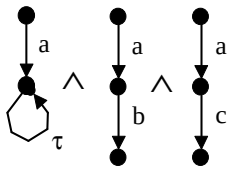

Fig. 3. Counter-example demonstrating non-associativity.

We may now demonstrate why we have treated non-escapable divergence as catastrophic in our setting. This is because, otherwise, our conjunction operator would not be associative as demonstrated by the example depicted in Fig. 3. If the conjunction is computed from the left, the result is the first conjunct. Computed from the right, the result is the same but with both processes being in F. Hence, in the first case, the divergence hides the inconsistency. Since this is not really plausible and associativity of conjunction is clearly desirable, we need some restriction for divergence; it turns out that restricting divergence to escapable divergence, i.e., potential stabilisation, is sufficient for our purposes.

Definition 3 (Disjunction Operator) *The disjunction of Logic LTSs* $\langle P, \longrightarrow_P, T_P, F_P \rangle$ *and* $\langle Q, \longrightarrow_Q, T_Q, F_Q \rangle$ *satisfying (w.l.o.g.)* $P \cap Q = \emptyset$*, is the Logic LTS* $\langle P \vee Q, \longrightarrow_{P \vee Q}, T_{P \vee Q}, F_{P \vee Q} \rangle$ *defined by:*

- $P \vee Q =_{df} \{p \vee q \,|\, p \in P,\, q \in Q\} \cup P \cup Q$
- $\longrightarrow_{P \vee Q}$ *is determined by the following operational rules:*

$$\begin{array}{rcl} \mathit{always} & \Longrightarrow & p \vee q \xrightarrow{\tau}_{P \vee Q} p \\ \mathit{always} & \Longrightarrow & p \vee q \xrightarrow{\tau}_{P \vee Q} q \\ p \xrightarrow{\alpha}_P p' & \Longrightarrow & p \xrightarrow{\alpha}_{P \vee Q} p' \\ q \xrightarrow{\alpha}_Q q' & \Longrightarrow & q \xrightarrow{\alpha}_{P \vee Q} q' \end{array}$$

- $T_{P \vee Q} = T_P \cup T_Q$*; in particular,* $p \vee q \notin T_{P \vee Q}$ *always*
- $F_{P \vee Q} = F_P \cup F_Q \cup \{p \vee q \,|\, p \in F_P \,, q \in F_Q\}$

The definition of disjunction, which reflects internal choice, is quite straightforward and well-defined. Only the definition of $T_{P \vee Q}$ for $p \vee q$ is unusual, as one would expect to simply have $p \vee q \in T$ whenever p or q is in T. However, then Cond. (2) of Def. 1 would be violated. Our alternative definition respects this condition and is semantically equivalent. In the sequel we leave out indices of relations and predicates whenever the context is clear.

2.4 Refinement Preorder

As the basis for our semantical considerations we now define a naive refinement preorder stating that an inconsistent specification cannot be implemented except by an inconsistent implementation.

Definition 4 (Naive Consistency Preorder) *The* naive consistency preorder $\sqsubseteq_F$ *on processes is defined by* $p \sqsubseteq_F q$ *if* $p \in F \implies q \in F$.

One of the main objectives of this article is to identify the corresponding fully-abstract preorder with respect to conjunction and disjunction, which is contained in $\sqsubseteq_F$. Our approach follows the testing idea of De Nicola and Hennessy [5], for which we define a testing relation $\sqsubseteq$ as usual. Note that a process and an observer need to be composed not simply synchronously but conjunctively. This is because we want the observer to be sensitive to inconsistencies, so that $p \sqsubseteq q$ if each "conjunctive observer" that sees an inconsistency in p also sees one in q.

Definition 5 (Consistency Testing Preorder) *The* consistency testing preorder $\sqsubseteq$ *on processes is defined as the conjunctive closure of the naive consistency preorder under all processes (observers), i.e.,* $p \sqsubseteq q$ *if* $\forall o.\ p \wedge o \sqsubseteq_F q \wedge o$.

To characterise the fully-abstract precongruence contained in $\sqsubseteq_F$ we will introduce *ready-tree semantics* which is a variant of van Glabbeek's path-based possible-worlds semantics [6], and an associated preorder, the *ready-tree preorder*. This preorder is compositional for conjunction and disjunction and characterises $\sqsubseteq$.

2.5 Example

As an illustration for our approach, consider process *spec* in Fig. 4. For $\mathcal{A} = \{a, b, c\}$, *spec* specifies that action c can only occur after action a. In the light of the above discussions, an implementation of this intuition should offer initially either just a, or a and b, or just b, so that *spec* is an internal choice between three states. Moreover, after an action a, nothing more is specified; after an action b, the same is required as initially.

While our specification of this simple behaviour may look quite complex, we may imagine that process *spec* is generated automatically from a temporal-logic formula. Fig. 4 also shows process *impl* which repeats sequence *abc*, and *spec* $\wedge$ *impl*. It will turn out that *spec* $\sqsubseteq$ *impl*, as we will show in Section 4.

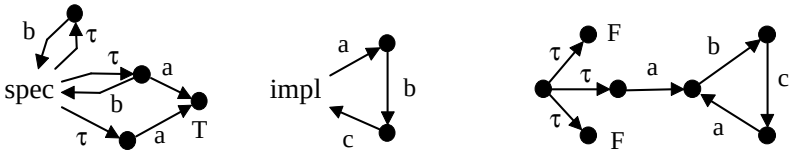

Fig. 4. Example processes.

3 Ready-Tree Semantics

A first guess for achieving a compositional semantics reflecting consistency testing is to use a kind of *ready-trace semantics* [8]. Such a semantics would refine trace semantics by checking the initial action set of every stable state along each trace. However, this is not sufficient when dealing with inconsistencies, since inconsistencies propagate backwards along traces as explained in Section 2. It turns out that a set of tree-like observations is needed, which leads to a denotational-style semantics which we call *ready-tree semantics*.

3.1 Observation Trees & Ready Trees

A tree-like observation can itself be seen as a deterministic LTS with empty F-predicate, reflecting that observers are internally consistent.

Definition 6 (Observation Tree) *An* observation tree *is a LTS* $\langle V, \rightarrow, T, \emptyset \rangle$ *satisfying the following properties:*

(1) $\langle V, \longrightarrow \rangle$ *is a non-empty tree whose root is referred to as* v_0
(2) $\forall v \in V.\ v$ *stable*
(3) $\forall v \in V, a \in \mathcal{I}(v)\ \exists_1 v' \in V.\ v \xrightarrow{a} v'$

We often denote such an observation tree by its root v_0*.*

Next we define the observations of a process p, called *ready trees*. Note that p can only be observed at its stable states.

Definition 7 (Ready Tree) *An observation tree* v_0 *is a* ready tree of p *if there is a labelling* $h : V \longrightarrow P$ *satisfying the following conditions:*

(1) $\forall v \in V.\ h(v)$ *stable and* $h(v) \notin F$
(2) $p \stackrel{\varepsilon}{\Longrightarrow}_F h(v_0)$
(3) $\forall v \in V, a \in \mathcal{A}.\ v \xrightarrow{a} v'$ *implies (a)* $h(v') = h(v) \in T$ *or*
(b) $h(v) \stackrel{a}{\Longrightarrow}_F h(v')$
(4) $\forall v \in V.\ (v \notin T$ *and* $h(v) \notin T)$ *implies* $\mathcal{I}(v) = \mathcal{I}(h(v))$

Intuitively, nodes v in a ready tree represent stable states $h(v)$ of p (cf. Cond. (1), first part) and transitions represent computations containing exactly one observable action (cf. Cond. (3)(b)). Since computations do not contain *false*-states, no represented state is in F (cf. Cond. (1), second part). Since p might not be stable, the root v_0 of a ready tree represents a stable state reachable from p by some internal computation (cf. Cond. (2)). If the state $h(v)$ represented by node v is in T, the subtree of v is arbitrary since $h(v)$ is considered to be completely underspecified (cf. Conds. (3)(a) and (4)). In case $h(v) \notin T$, one distinguishes two cases: (i) if $v \notin T$, then v and $h(v)$ must have the same initial actions, i.e., the same *ready set*; (ii) if $v \in T$, the observation stops at this node and nothing is required in Conds. (3) and (4).

In the following, we write $\mathrm{RT}(p)$ for the set of all ready trees of p, $\mathrm{fRT}(p)$ for the set of all ready trees of p that have finite depth (*finite-depth* ready trees), and $\mathrm{cRT}(p)$ for the set of ready trees $\langle V, \longrightarrow, T, \emptyset \rangle$ where $T = \emptyset$ (*complete* ready trees). Note that a complete ready tree is called *complete* as it never stops its task of observing; hence, complete ready trees are often infinite in practise. Moreover, *false*-states may be characterised as follows:

Lemma 8 *$RT(p) = \emptyset$ if and only if $p \in F$.*

PROOF. Direction "$\Longleftarrow$" follows immediately from Def. 7(2) and the definition of $\stackrel{\varepsilon}{\Longrightarrow}_F$. For Direction " $\Longrightarrow$ " we know by $p \notin F$ and Def. 1(4) of the existence of some p' such that $p \stackrel{\varepsilon}{\Longrightarrow}_F p' \stackrel{\mathcal{I}}{\not\longrightarrow}$. Hence, $tt \in \mathrm{RT}(p)$ by $h(tt) =_{\mathrm{df}} p'$. $\square$

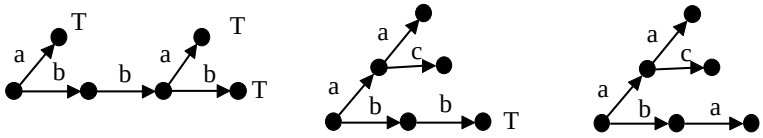

Fig. 5. Some ready trees of *spec*.

We illustrate our concept of ready trees by returning to our example of Fig. 4. Some of the ready trees of process *spec* are shown in Fig. 5. In the first ready tree, the observation stops after the third b. In the second tree, we see that we can observe an arbitrary tree after a, since the respective state of *spec* is in T. An arbitrary tree can also consist of just the root, as shown for the right-most a in the third tree; this tree is also complete. Process *impl* in Fig. 4 has only one complete ready tree which is an infinite path repeating sequence abc; this is also a ready tree of *spec*.

Our ready-tree semantics suggests the following refinement preorder:

Definition 9 (Ready-Tree Preorder) *The* ready-tree preorder $\sqsubseteq_{\lesssim}$ *on processes is defined as reverse ready-tree inclusion, i.e.,* $p \sqsubseteq_{\lesssim} q$ *if* $RT(q) \subseteq RT(p)$.

This preorder will turn out to be the desired fully-abstract preorder contained in the naive consistency preorder.

We first show that $\sqsubseteq_{\lesssim}$ could just as well be formulated on the basis of complete ready trees and, for finitely branching LTS, of finite-depth ready trees. A crucial notion for our results is the following:

Definition 10 (Ready-Tree Prefix) *Ready tree* v_0 *is a* prefix *of ready tree* w_0*, written* $v_0 \leq w_0$*, if there exists an injective mapping* $\rho : V \hookrightarrow W$ *such that:*

(1) $\rho(v_0) = w_0$
(2) $v \xrightarrow{a} v' \implies \rho(v) \xrightarrow{a} \rho(v')$
(3) $\rho(v) \xrightarrow{a} w' \implies v \in T$ *or* $(\exists v'.\ v \xrightarrow{a} v'$ *and* $\rho(v') = w')$
(4) $\rho(v) \in T \implies v \in T$

Intuitively, one observation is a prefix of another if it stops observing earlier. Recall that a *true*-node indicates that observation stops (cf. Cond. (3)). Intuitively, we obtain a prefix of w_0 by cutting all transitions from some nodes (and adding the latter to T), while cutting just some transitions of a node is not allowed. It is easy to see that our definition of $\mathrm{RT}(p)$ is closed under prefix:

Lemma 11 $(v_0 \leq w_0$ *and* $w_0 \in RT(p))$ *implies* $v_0 \in RT(p)$.

PROOF. Let $w_0 \in \mathrm{RT}(p)$ due to h and $v_0 \leq w_0$ with injection ρ. We define $h' : V \longrightarrow P$ such that $v \longmapsto h(\rho(v))$ and check that v_0 is a ready tree of p:

(1) $h'(v) = h(\rho(v))$ is stable and not in F by Def. 7(1) for w_0.
(2) $p \stackrel{\varepsilon}{\Longrightarrow}_F h(w_0) = h(\rho(v_0) = h'(v_0)$ by Def. 7(2) for w_0.
(3) $v \xrightarrow{a} v' \implies \rho(v) \xrightarrow{a} \rho(v') \implies h(\rho(v')) = h(\rho(v)) \in T$ or $h(\rho(v)) \stackrel{a}{\Longrightarrow}_F h(\rho(v')) \implies h'(v') = h'(v) \in T$ or $h'(v) \stackrel{a}{\Longrightarrow}_F h'(v')$.
(4) Assume $v \notin T$ and $h'(v) \notin T$. Then, $\rho(v) \notin T$ by Def. 10(4) as well as $h(\rho(v)) \notin T$ by the definition of h'. This implies $\mathcal{I}(\rho(v)) = \mathcal{I}(h(\rho(v))) = \mathcal{I}(h'(v))$. Furthermore, $\mathcal{I}(v) = \mathcal{I}(\rho(v))$ by Defs. 10(2) and (3) and since $v \notin T$. Hence, $\mathcal{I}(v) = \mathcal{I}(h'(v))$. $\square$

Note that, since we do not want to distinguish isomorphic observation trees, we may always assume, without loss of generality, that the embedding ρ in Def. 10 is the identity, i.e., that the node set V of the prefix is a subset of W.

Lemma 12 $\{v_0 \mid \exists w_0 \in cRT(p).\ v_0 \leq w_0\} = RT(p)$.

PROOF. Inclusion "$\subseteq$" is an application of Lemma 11; note that $\mathrm{cRT}(p) \subseteq \mathrm{RT}(p)$ by definition.

For proving the reverse inclusion "$\supseteq$", let $v_0 \in \mathrm{RT}(p)$ due to h. We construct a suitable w_0 such that the respective injection is the identity, by successively extending the T-nodes of v_0. Let v_0 be the 0-extension of v_0. Given the k-extension of v_0 we construct the $(k{+}1)$-extension as follows:

For each $v \in T$ with $h(v) \in T$, remove v from T. For each $v \in T$ with $h(v) \notin T$, and every $a \in \mathcal{I}(h(v))$, choose some p' with $h(v) \stackrel{a}{\Longrightarrow}_F p' \notin F$ and p' stable. Such a p' exists since $h(v) \notin F$ (due to Def. 7(1)) implies, by the first fixed point condition of F for LTS (Def. 1(3)), the existence of a $p'' \notin F$ with $h(v) \stackrel{a}{\longrightarrow} p''$. Moreover, by the second fixed point condition of F for LTS (Def. 1(4)), p'' can stabilise, i.e., there is a stable $p' \notin F$ such that $p'' \stackrel{\varepsilon}{\Longrightarrow}_F p'$. Now, choose a fresh node v_a and add $v \stackrel{a}{\longrightarrow} v_a$ into the tree, with $h(v_a) = p'$ and $v_a \in T$. Remove v from T.

Note that the $(k{+}1)$-extension is indeed a ready tree for p by construction. Finally, let w_0 be the component-wise union of all k-extensions with T set to the empty set. This yields a complete ready tree, i.e., $w_0 \in \mathrm{cRT}(p)$; note in particular that our construction ensures that $h(v) \notin T \implies \mathcal{I}(v) = \mathcal{I}(h(v))$. $\square$

As a consequence of Lemma 12 we obtain the following corollary:

Corollary 13

(1) $RT(p)$ is uniquely determined by $cRT(p)$, and vice versa.
(2) $RT(p) \subseteq RT(q) \iff cRT(p) \subseteq cRT(q)$
(3) $fRT(p) = \{v_0$ of finite depth $\mid \exists w_0 \in cRT(p).\, v_0 \leq w_0\}$

Before stating the next lemma we introduce the following definitions that allow us to approximate ready trees:

Definition 14 (k-Ready Tree) *A k-*tree *$\langle V, \longrightarrow, T, \emptyset\rangle$, where $k \in \mathbb{N}_0$, is an observation tree where all nodes have depth at most k, and T is the set of all nodes of depth k. A k-*ready tree *of p is a ready tree of p that is also a k-tree. Moreover, k–$RT(p) =_{df} \{v_0 \in RT(p) \mid v_0$ is a k-tree$\}$.*

Intuitively, k-trees represent observations of k steps.

Definition 15 (Limit) *Let $\vec{v}$ be an infinite sequence $(v_k)_{k\in\mathbb{N}}$ where $v_k \in k$–RT(p) and $v_k \leq v_{k+1}$, with the identity as injection, for all $k \in \mathbb{N}$. Then, $\lim \vec{v}$ is the component-wise union of all v_k with T set to empty; $\lim \vec{v}$ is called a* limit *of p.*

Observe that a node of some v_k in such a sequence is not in the *true*-predicate of v_{k+1}, whence nodes in T are successively pushed out. In the limit, we may thus set T to empty. Moreover, if $v_k = v_{k+1} = v_{k+2} = \ldots$ for some k, then the limit is v_k; this happens exactly when v_k is complete. We base the notion of finite branching on the weak transition relation $\stackrel{\alpha}{\Longrightarrow}_F$.

Definition 16 (Finite Branching) *Process p is finite branching if, for all p' reachable from p, there are only finitely many $\langle \alpha, p'' \rangle$ with $p' \stackrel{\varepsilon}{\Longrightarrow}_F \stackrel{\alpha}{\Longrightarrow}_F p''$.*

For finite-branching processes p, cRT(p) is characterised by the limits of p.

Lemma 17 *If p is finite branching, cRT(p) equals the set of all limits of p.*

PROOF. For proving inclusion "$\subseteq$", let $w_0 \in \mathrm{cRT}(p)$; again we refer to the tree's root as w_0, too, and denote the tree's node set by W. We define a sequence $\vec{v} = (v_k)_{k\in\mathbb{N}}$ as follows: v_k consists of all nodes of w_0 of depth at most k, and the arcs between them. Moreover, T is the set of all nodes at depth k. Hence, v_k is a k-tree, and $v_k \leq w_0$ with the identity as injection. By Lemma 11, each v_k is in RT(p). Obviously, $v_k \leq v_{k+1}$ for all $k \in \mathbb{N}$ and $w_0 = \lim(\vec{v})$.

For proving inclusion "$\supseteq$", let $\vec{v} = (v_k)_{k\in\mathbb{N}}$ with $w_0 \in \lim(\vec{v})$. Hence, for each v_k we have at least one h_k such that $v_k \in \mathrm{RT}(p)$ due to h_k. To show $w_0 \in \mathrm{cRT}(p)$ we have to find a labelling $g : W \longrightarrow P$ so that the definition of cRT is satisfied. This h will be assembled from the h_k's by an application of König's Lemma.

We construct a graph with vertices $\langle v_k, h \rangle$ such that $v_k \in k$–RT(p) due to h. Note again that there may be several such h. The edges of our graph are given by $\langle v_k, h \rangle \longrightarrow \langle v_{k+1}, h' \rangle$ if $h = h'_{|V_k}$. Since p is finite branching we have only finitely many $\langle v_k, h \rangle$ for each k. Adding a root vertex r that is connected to all $\langle v_0, h \rangle$, we therefore obtain an infinite, finitely branching tree.

According to König's Lemma, there exists an infinite path $\langle v_0, h_0 \rangle \rightarrow \langle v_1, h_1 \rangle \rightarrow \cdots$. We now set $g =_{\mathrm{df}} \bigcup_{k\in\mathbb{N}} h_k$. That g satisfies the conditions of the definition of cRT(p) is obvious for Conds. (1-3); for Cond. (4), observe that each node w of w_0 has some depth k, hence it is in v_{k+1} and not in T_{k+1}, and we can use that v_{k+1} satisfies Cond. (4). $\square$

Note that the premise "p is finite branching" is only needed for direction "$\supseteq$" in the above lemma. We may now obtain the following corollary of Cor. 13(3) and of Lemma 17, which is the key to proving compositionality and full abstraction of our ready-tree preorder in the next section.

Corollary 18

(1) $cRT(p) \subseteq cRT(q) \implies fRT(p) \subseteq fRT(q)$, *always.*
(2) $cRT(p) \subseteq cRT(q) \impliedby fRT(p) \subseteq fRT(q)$, *if p is finite branching.*

We conclude this section by pointing out that any process is ready-tree-equivalent to a process that is either inconsistent itself, or does not have any inconsistent state. If one normalises two processes by omitting inconsistent states and then calculates their conjunction, one obtains an equivalent process as first calculating the conjunction and subsequently normalising the result. This gives us a first indication that the above definition of conjunction is adequate.

4 Compositionality & Full Abstraction

This section presents our full-abstraction result of the ready-tree preorder $\sqsubseteq\!\!\!\!\sim$ with respect to the consistency testing preorder $\sqsubseteq$, and proves that $\wedge$ and $\vee$ are indeed conjunction and, respectively, disjunction for $\sqsubseteq\!\!\!\!\sim$. We first show that $\wedge$ and $\vee$ correspond to intersection and union on the semantic level, respectively. While the correspondence for $\vee$ holds for ready trees in general, the correspondence for $\wedge$ only holds for *complete* ready trees.

Theorem 19 (Set-Theoretic Interpretation of $\wedge$ and $\vee$)

(1) $cRT(p \wedge q) = cRT(p) \cap cRT(q)$
(2) $RT(p \vee q) = RT(p) \cup RT(q)$

PROOF. We first establish statement (1) of Thm. 19. For proving direction "$\subseteq$", take $v_0 \in \mathrm{cRT}(p \wedge q)$ due to h. We define $h'(v) =_{\mathrm{df}} p'$ if $h(v) = p' \wedge q'$ for some q', for all $v \in V$, and check the conditions of the definition of cRT:

(1) Since $p' \wedge q'$ is stable (not in F, resp.), also p' is stable (not in F, resp.) by our definition of $\wedge$ (Def. (2)).
(2) $p \wedge q \stackrel{\varepsilon}{\Longrightarrow_F} p' \wedge q'$ implies $p \stackrel{\varepsilon}{\Longrightarrow_F} p'$.
(3) Let $v \stackrel{a}{\longrightarrow} v'$. If $h(v') = h(v) = p' \wedge q' \in T$, then $h'(v) = h'(v') = p' \in T$, by Def. (2). If $h(v) = p' \wedge q' \stackrel{a}{\Longrightarrow_F} p'' \wedge q'' = h(v')$, then either $h'(v) = p' = p'' = h'(v') \in T$, or $h'(v) = p' \stackrel{a}{\Longrightarrow_F} p'' = h'(v')$; note that we can avoid

F-processes along $p' \stackrel{a}{\Longrightarrow}_F p''$ since we can do so along $p' \wedge q' \stackrel{a}{\Longrightarrow}_F p'' \wedge q''$, and that $h'(v) = p'$ is stable as noted in (1).

(4) Let $h'(v) = p' \notin T$ and $v \notin T$ (which is in fact always the case in complete ready trees). Then, $h(v) = p' \wedge q' \notin T$ by Def. 2. Hence, $\mathcal{I}(v) = \mathcal{I}(h(v)) = \mathcal{I}(p' \wedge q')$ by Def 7(4). Recalling that p' and q' are stable and that $p' \notin T$, the operational rules of Def. 2 show that
- either $q' \in T$ and $\mathcal{I}(p' \wedge q') = \mathcal{I}(p') = \mathcal{I}(h'(v))$,
- or $q' \notin T$ and $\mathcal{I}(p') = \mathcal{I}(q')$ by Def. 2(2). Observe $p' \wedge q' \notin F$ by Def. 7(1). Thus, again, $\mathcal{I}(p' \wedge q') = \mathcal{I}(p') = \mathcal{I}(h'(v))$.

.

Note that this direction of the theorem is also valid for RT in place of cRT.

For proving direction "$\supseteq$", take $v_0 \in \mathrm{cRT}(p) \cap \mathrm{cRT}(q)$ due to h_1 and h_2, respectively. Define $h(v) =_{\mathrm{df}} h_1(v) \wedge h_2(v)$. We check the four conditions of the definition of cRT, starting with Cond. (4):

Let $v \notin T$ and $h(v) \notin T$. Without loss of generality, $h_1(v) \notin T$ according to our definition of $\wedge$ (Def. (2)). If $h_2(v) \in T$, we have $\mathcal{I}(h(v)) = \mathcal{I}(h_1(v)) = \mathcal{I}(v)$ by Cond. (4) for h_1. If $h_2(v) \notin T$, then $\mathcal{I}(v) = \mathcal{I}(h_1(v)) = \mathcal{I}(h_2(v))$ by Cond. (4) for h_1 and h_2; hence, $\mathcal{I}(v) = \mathcal{I}(h(v))$ according to the definition of $p \wedge q$.

Conds. (1)-(3), are proved together. Note that, since $h_1(v)$ and $h_2(v)$ are stable, we have that $h(v)$ is stable, too. We will prove simultaneously that a number of processes are not in F. To do so, we will collect a number of processes in a list $\mathcal{F}$ and argue that the complement $\overline{\mathcal{F}}$ meets the conditions for F in $\wedge$ (Def. 2). Then we know that the least set $F_{p \wedge q}$ satisfying these conditions is contained in $\overline{\mathcal{F}}$, whence no process on our list is in F.

We now simply show that the processes on the list do not satisfy any of the four conditions, using for the fixed point conditions (Defs. 2(3) and (4)) that no process in $\mathcal{F}$ is in $\overline{\mathcal{F}}$.

Our list $\mathcal{F}$ firstly contains all processes $p' \wedge q'$, so that p' (q') is a process along the derivation $p \stackrel{\varepsilon}{\Longrightarrow}_F h_1(v_0)$ ($q \stackrel{\varepsilon}{\Longrightarrow}_F h_2(v_0)$) according to Def. 7(2). Analogously, we treat p' on the subderivation $p'' \stackrel{\varepsilon}{\Longrightarrow}_F h_1(v')$ contained in $h_1(v) \stackrel{a}{\Longrightarrow}_F h_1(v')$ and q' on the subderivation $q'' \stackrel{\varepsilon}{\Longrightarrow}_F h_2(v')$ contained in $h_2(v) \stackrel{a}{\Longrightarrow}_F h_2(v')$ according to Def. 7(3), if both derivations exist. Finally, if $h_1(v) \stackrel{a}{\Longrightarrow}_F h_1(v')$ ($h_2(v) \stackrel{a}{\Longrightarrow}_F h_2(v')$) exists and $h_2(v') = h_2(v) \in T$ ($h_1(v') = h_1(v) \in T$), we combine each such p' (q') with $h_2(v)$ ($h_1(v)$).

We next show that $\overline{\mathcal{F}}$ is consistent with our constraints on F (Defs. 2(1)-(4)):

(1) If $p' \in F$ or $q' \in F$, then $p' \wedge q'$ is not on the list, i.e., $p' \wedge q'$ is in $\overline{\mathcal{F}}$. In other words, if $p' \wedge q'$ is on the list, then $p' \notin F$ and $q' \notin F$ such that the first constraint on F is satisfied.

(2) Assume $p' \wedge q'$ is on the list, and $p' \notin T$ and $q' \notin T$ and $p' \wedge q'$ stable. The last condition implies $p' \equiv h_1(v)$ and $q' \equiv h_2(v)$ for some v. Since $v \notin T$ by completeness of v_0, we get $\mathcal{I}(h_1(v)) = \mathcal{I}(v) = \mathcal{I}(h_2(v))$ by Def. 7(4).

(3) Assume $p' \wedge q'$ is on the list $\mathcal{F}$. If $p' \wedge q' \xrightarrow{\tau}$, then (without loss of generality) $p' \xrightarrow{\tau} p''$ for some p'' on the same derivation as p'. Hence, $p' \wedge q' \xrightarrow{\tau} p'' \wedge q'$ which is also on our list $\mathcal{F}$.

If $p' \wedge q' \not\xrightarrow{\mathcal{I}}$, then $p' \equiv h_1(v)$ and $q' \equiv h_2(v)$ for some v. Let $a \in \mathcal{I}(h_1(v) \wedge h_2(v))$ and distinguish the following cases:

- $h_1(v) \in T$, i.e., $h_1(v) \equiv tt$: By Lemma 23, $h_1(v) \wedge h_2(v) \cong h_2(v)$. We must have $a \in \mathcal{I}(h_2(v))$, whence $h_2(v) \notin T$. Since $v \notin T$ by completeness of v_0, we have $\mathcal{I}(v) = \mathcal{I}(h_2(v))$ by Def. 7(4). Thus, $v \xrightarrow{a} v'$ for some v', and $h_2(v) \xrightarrow{a} q'' \stackrel{\varepsilon}{\Longrightarrow}_F h_2(v')$ by Def. 7(3). This implies $h_1(v) \wedge h_2(v) \xrightarrow{a} h_1(v) \wedge q''$ which is on our list $\mathcal{F}$.
- $h_2(v) \in T$, i.e., $h_2(v) \equiv tt$: This case is analogous to the first one.
- $h_1(v) \notin T$ and $h_2 \notin T$: We must have $a \in \mathcal{I}(h_1(v))$ and $a \in \mathcal{I}(h_2(v))$. As above, $v \xrightarrow{a} v'$ for some v'; we conclude $h_1(v) \wedge h_2(v) \xrightarrow{a} p'' \wedge q''$ which is on our list $\mathcal{F}$.

(4) Suppose $p' \wedge q'$ is on our list $\mathcal{F}$. Then, either $p' \wedge q' \not\xrightarrow{\mathcal{I}}$, or $p' \wedge q'$ lies along the way to $h_1(v') \wedge h_2(v') \not\xrightarrow{\mathcal{I}}$, using only processes on list $\mathcal{F}$.

This concludes the proof of Cond. (1). The validity of Cond. (2) is now immediate: by the above, $p \wedge q \stackrel{\varepsilon}{\Longrightarrow}_F h_1(v_0) \wedge h_2(v_0)$ since $p \stackrel{\varepsilon}{\Longrightarrow}_F h_1(v_0)$ and $q \stackrel{\varepsilon}{\Longrightarrow}_F h_2(v_0)$. To show the validity of Cond. (3), we consider $v \xrightarrow{a} v'$ and distinguish four cases, as suggested by Def. 7(3). We only show one case here as the others are equally easy: if $h_1(v) \xrightarrow{a} p' \stackrel{\varepsilon}{\Longrightarrow}_F h_1(v')$ and $h_2(v) \xrightarrow{a} q' \stackrel{\varepsilon}{\Longrightarrow}_F h_2(v')$, then $h_1(v) \wedge h_2(v) \xrightarrow{a} p' \wedge q' \stackrel{\varepsilon}{\Longrightarrow}_F h_1(v') \wedge h_2(v')$ using only processes not in F. This finishes the proof of statement (1) of Thm. 19.

The proof of statement (2) of Thm. 19 is much easier. For inclusion "$\supseteq$", let $v_0 \in \mathrm{RT}(p) \cup \mathrm{RT}(q)$, whence (without loss of generality) $v_0 \in \mathrm{RT}(p)$ due to h. Now, it is straightforward to check that $v_0 \in \mathrm{RT}(p \vee q)$ due to h. We only note that Cond (2) of Def. 7 follows from $p \vee q \xrightarrow{\tau} p \stackrel{\varepsilon}{\Longrightarrow}_F h(v_0)$, due to $p \notin F$ by Lemma 8. For the reverse inclusion "$\subseteq$", let $v_0 \in \mathrm{RT}(p \vee q)$ due to h. By Def. 7(2), $p \vee q \stackrel{\varepsilon}{\Longrightarrow}_F h(v_0)$ and (without loss of generality) $p \vee q \xrightarrow{\tau} p \stackrel{\varepsilon}{\Longrightarrow}_F h(v_0)$. Obviously, $v_0 \in \mathrm{RT}(p)$ by h. Note that, by Def. 7(2), all $h(v)$ are reachable from $h(v_0)$, whence h maps into P. □

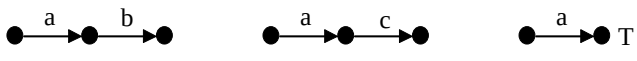

Fig. 6. Necessity of considering complete ready trees for conjunction.

Fig. 6 illustrates that Thm. 19(1) is invalid when considering all ready trees instead of complete read trees. The two processes displayed on the left and in the middle have the ready tree displayed on the right in common. However,

the conjunction of the two processes is false and has no ready trees. Intuitively, the shown common ready tree formalises an observation that finished too early to encounter the inconsistency.

Given Thm. 19, compositionality of our conjunction and disjunction operators for $\sqsubseteq\!\!\!\sim$ is now an immediate consequence.

Theorem 20 (Compositionality)

(1) $p \sqsubseteq\!\!\!\sim q \implies p \wedge r \sqsubseteq\!\!\!\sim q \wedge r$
(2) $p \sqsubseteq\!\!\!\sim q \implies p \vee r \sqsubseteq\!\!\!\sim q \vee r$

PROOF. The compositionality of $\wedge$ follows from the following implication chain: $\mathrm{RT}(p) \subseteq \mathrm{RT}(q) \implies$ (by Cor. 13(2)) $\mathrm{cRT}(p) \subseteq \mathrm{cRT}(q) \implies \mathrm{cRT}(p) \cap \mathrm{cRT}(r) \subseteq \mathrm{cRT}(q) \cap \mathrm{cRT}(r) \implies$ (by Thm. 19(1)) $\mathrm{cRT}(p \wedge r) \subseteq \mathrm{cRT}(q \wedge r) \implies$ (by Cor. 13(2)) $\mathrm{RT}(p \wedge r) \subseteq \mathrm{RT}(q \wedge r)$. The compositionality of $\vee$ can be proved analogously by referring to Thm. 19(2) instead of Thm. 19(1). □

Thm. 19 also allows us to prove that $\wedge$ and $\vee$ really behave as conjunction and disjunction with respect to our refinement relation.

Theorem 21 ($\wedge$ is And & $\vee$ is Or)

(1) $p \wedge q \sqsubseteq\!\!\!\sim r \iff p \sqsubseteq\!\!\!\sim r$ *and* $q \sqsubseteq\!\!\!\sim r$
(2) $r \sqsubseteq\!\!\!\sim p \vee q \iff r \sqsubseteq\!\!\!\sim p$ *and* $r \sqsubseteq\!\!\!\sim q$

PROOF. Part (1) follows from the following equivalences: $\mathrm{RT}(p \wedge q) \subseteq \mathrm{RT}(r) \iff$ (by Cor. 13(2)) $\mathrm{cRT}(p \wedge q) \subseteq \mathrm{cRT}(r) \iff$ (by Thm. 19(1)) $\mathrm{cRT}(p) \cap \mathrm{cRT}(q) \subseteq \mathrm{cRT}(r) \iff \mathrm{cRT}(p) \subseteq \mathrm{cRT}(r)$ and $\mathrm{cRT}(q) \subseteq \mathrm{cRT}(r) \iff$ (by Cor. 13(2)) $\mathrm{RT}(p) \subseteq \mathrm{RT}(r)$ and $\mathrm{RT}(q) \subseteq \mathrm{RT}(r)$. Again, for Part (2) we get a similar but simpler proof by referring to Thm. 19(2) instead of Thm. 19(1). □

Part (2) of the above theorem also implies a property demanded by system designers [11]: if q_1 is an implementation of p_1 and q_2 is an implementation of p_2, then the specification $p_1 \vee p_2$ can be implemented by either q_1 or q_2. To justify this within our framework, we first formalise the premise as $p_1 \sqsubseteq\!\!\!\sim q_1$ and $p_2 \sqsubseteq\!\!\!\sim q_2$. By compositionality, Thm. 20(2), we obtain $p_1 \vee p_2 \sqsubseteq\!\!\!\sim q_1 \vee q_2$. But this implies the desired statement $p_1 \vee p_2 \sqsubseteq\!\!\!\sim q_1$ and $p_1 \vee p_2 \sqsubseteq\!\!\!\sim q_2$, by Thm. 21(2).

In order to see that ready trees are indeed fully-abstract with respect to our naive consistency preorder, it now suffices to prove that $\sqsubseteq\!\!\!\sim$ coincides with our

consistency testing preorder. This means that $\sqsubseteq\!\!\!\sim$ is *the* adequate preorder in our setting of Logic LTSs with conjunction and disjunction.

Theorem 22 (Full Abstraction) $\sqsubseteq = \sqsubseteq\!\!\!\sim$

For the proof of this theorem, the following technical lemma will be convenient.

Lemma 23 *The LTSs of $tt \wedge p$ and p are isomorphic, written $tt \wedge p \cong p$.*

PROOF. It is easy to check that $tt \wedge p' \longmapsto p'$ is an isomorphism. Note that with $tt \wedge p' \in F$ iff $p' \in F$, the fixed point conditions of F are satisfied. □

We may now prove Thm. 22.

PROOF. We first prove the easier direction "$\supseteq$". If $p \sqsubseteq\!\!\!\sim q$ then, for all o, we have $p \wedge o \sqsubseteq\!\!\!\sim q \wedge o$ by Thm. 20. If $p \wedge o \in F$, then $\mathrm{RT}(p \wedge o) = \emptyset$ by Lemma 8. Hence, $\mathrm{RT}(q \wedge o) = \emptyset$, i.e., $q \wedge o \in F$ by Lemma 8 again. Thus, $p \sqsubseteq q$.

For proving the reverse inclusion, let $p \sqsubseteq q$ and $v_0 \in \mathrm{RT}(q)$ due to h. Note that v_0 is a process itself. We show that $q \wedge v_0 \notin F$. To do so, we use a fixed point argument similar to the one in the proof of Thm. 19. Here, our list firstly includes processes $q' \wedge v_0$, with q' on the derivation $q \stackrel{\varepsilon}{\Longrightarrow}_F h(v_0)$ (cf. Def. 7(3)), as well as processes $q' \wedge v'$, with q' along the derivations $q'' \stackrel{\varepsilon}{\Longrightarrow}_F h(v')$ that emerge due to $h(v) \stackrel{a}{\longrightarrow} q'' \stackrel{\varepsilon}{\Longrightarrow}_F h(v')$ in Def. 7(3). Furthermore, we include all processes $h(v') \wedge v'$ on our list, whenever $h(v') = h(v) \in T$ in Def. 7(3). Finally, we include all $q' \wedge v'$ such that $q' \notin F$, $v' \in T$ and q' is a process in the same LTS as q.

We now check that the complement of this list is a fixed point, i.e., satisfies Conds. (1)-(4) of Def. 2. Let $q' \wedge v'$ be on our list:

(1) According to the definition of observation trees (Def. 6), $v' \notin F$. Moreover, $q' \notin F$ by the definition of $\stackrel{\varepsilon}{\Longrightarrow}_F$, or $q' \in T$ or 'immediate', depending on why $q' \wedge v'$ has been included in the list.
(2) If $q' \notin T$ and $v' \notin T$ and $q' \wedge v' \stackrel{\tau}{\not\longrightarrow}$, then $q' \equiv h(v')$, as these are the only stable processes on our list. Hence, we may apply Def. 7(4) for v' to obtain $\mathcal{I}(q') = \mathcal{I}(v')$.
(3) Let $\alpha \in \mathcal{I}(q' \wedge v')$. If $\alpha = \tau$, then $q' \stackrel{\tau}{\longrightarrow} q''$ on the respective derivation according to the definition of our list. Then, $q' \wedge v' \stackrel{\tau}{\longrightarrow} q'' \wedge v'$ which is on our list as well. If $\alpha \neq \tau$, then q' is stable, whence $q' \equiv h(v')$ again or $v' \in T$. We proceed by a case distinction:

- $\underline{q' \in T}$: Hence, $v' \xrightarrow{\alpha} v''$ for some v'' and $v' \notin T$. Since $q' \not\stackrel{\alpha}{\Longrightarrow}_F$, we are in the case of Def. 7(3a), i.e., $h(v'') = h(v') = q'$. Moreover, $q' \wedge v' \xrightarrow{\alpha} q' \wedge v''$ which is on our list.
- $\underline{v' \in T}$: Hence, $q' \xrightarrow{\alpha} q''$ for some $q'' \notin F$. Then, $q' \wedge v' \xrightarrow{\alpha} q'' \wedge v'$ which is on our list.
- $\underline{q' \notin T \text{ and } v' \notin T}$: Hence, $v' \xrightarrow{\alpha} v''$ for some v''. Since $h(v') = q' \notin T$ we know, by Def. 7(3), that $q' \xrightarrow{\alpha} q'' \stackrel{\varepsilon}{\Longrightarrow}_F h(v'')$. Thus, $q' \wedge v' \xrightarrow{\alpha} q'' \wedge v''$ which is on our list.

(4) If $q' \wedge v'$ is on the list due to $q' \stackrel{\varepsilon}{\Longrightarrow}_F h(v')$, then $q' \wedge v' \stackrel{\varepsilon}{\Longrightarrow}_F h(v') \wedge v'$ along processes that are on our list, and $h(v') \wedge v'$ is stable since $h(v')$ is stable and v' is trivially stable.

If $q' \wedge v'$ is on the list due to $v' \in T$, then q' can stabilise. Thus, $q' \wedge v'$ can stabilise in an isomorphic way using only processes on the list.

Thus, we have established $q \wedge v_0 \notin F$. This implies by $p \sqsubseteq q$ that $p \wedge v_0 \notin F$. To show $v_0 \in \mathrm{RT}(p)$, we will construct a respective labelling g according to depth. Since $p \wedge v_0 \notin F$, process $p \wedge v_0$ can stabilise by Def. 1(4) with $p \wedge v_0 \stackrel{\varepsilon}{\Longrightarrow}_F p' \wedge v_0$ for some stable p' with $p \stackrel{\varepsilon}{\Longrightarrow}_F p'$. We define $g(v_0) =_{\mathrm{df}} p'$, so that Cond. (2) of Def. 7 is satisfied, and Cond. (1) of Def. 7 holds for v_0. Note that $g(v_0) \wedge v_0 \equiv p' \wedge v_0 \notin F$ by the definition of $\stackrel{\varepsilon}{\Longrightarrow}_F$.

Assume that g is defined up to depth k such that Conds. (1), (3) and (4) of Def. 7 hold for all v with depth less than k, and that Cond. (1) is satisfied for depth k as well. Moreover, assume that $g(v) \wedge v \notin F$ whenever $g(v)$ is defined. These assumptions are our induction hypothesis, which we have just checked for $k = 0$. For each v at depth k we proceed as follows. If $v \in T$, then Conds. (3) and (4) are vacuously true; since v has no children at depth $k+1$, we are done. Thus, let $v \notin T$ and distinguish the following cases:

- $\underline{g(v) \in T}$: For all v' with $v \xrightarrow{a} v'$, define $g(v') =_{\mathrm{df}} g(v)$. Thus, Conds. (3) and (4) are satisfied for v, and $g(v')$ is stable and not in F. Since $g(v) \in T$ we have, by a consequence of Lemma 23, that $g(v) \wedge v$ and v are isomorphic. As $v \xrightarrow{a} v' \notin F$, this implies $g(v') \wedge v' \equiv g(v) \wedge v' \notin F$.
- $\underline{g(v) \notin T}$: We first show Cond. (4) whose premise is now true. As $g(v) \wedge v \notin F$, the facts $v \notin T$, $g(v) \notin T$ and $g(v) \wedge v$ stable imply $\mathcal{I}(v) = \mathcal{I}(g(v))$.

  Next, we show Cond. (3), i.e., how to extend g to all v' with $v \xrightarrow{a} v'$ such that $g(v) \stackrel{a}{\Longrightarrow}_F g(v')$. Consider some $v \xrightarrow{a} v'$. Since $\mathcal{I}(v) = \mathcal{I}(g(v))$ we know $a \in \mathcal{I}(g(v) \wedge v)$ and, since $g(v) \wedge v \notin F$, there is some p' such that $g(v) \wedge v \xrightarrow{a} p' \wedge v' \notin F$. But $p' \wedge v' \notin F$ implies that $p' \wedge v'$ can stabilise according to Def. 1(4), with some derivation $p' \wedge v' \stackrel{\varepsilon}{\Longrightarrow}_F p'' \wedge v' \not\xrightarrow{\tau}$. In particular, p'' is stable and not in F. In addition, $g(v) \xrightarrow{a} p' \stackrel{\varepsilon}{\Longrightarrow}_F p''$. Note that all processes along the derivation $p' \stackrel{\varepsilon}{\Longrightarrow}_F p''$ are not in F since, otherwise, some process along $p' \wedge v' \stackrel{\varepsilon}{\Longrightarrow}_F p'' \wedge v'$ would be in F. Finally, we now define $g(v') =_{\mathrm{df}} p''$.

  When applying this construction to all v' with $v \xrightarrow{a} v'$, Cond. (3) is

satisfied for v. Furthermore, Cond. (1) is satisfied for all v', and $g(v') \wedge v' \notin F$.

Treating all v at level k as above, we extend g to depth $k+1$ such that the induction hypothesis now also holds for $k+1$. With this induction, we can thus define g for each v in the observation tree. Hence, $v_0 \in \mathrm{RT}(p)$ due to g. $\square$

The following proposition states the validity of several boolean properties desired of conjunction and disjunction operators. Here, $=$ denotes the kernel of our consistency testing preorder (ready-tree preorder).

Proposition 24 (Properties of $\wedge$ and $\vee$)

Commutativity:	$p \wedge q = q \wedge p$	$p \vee q = q \vee p$
Associativity:	$(p \wedge q) \wedge r = p \wedge (q \wedge r)$	$(p \vee q) \vee r = p \vee (q \vee r)$
Idempotence:	$p \wedge p = p$	$p \vee p = p$
False:	$p \wedge \mathit{ff} = \mathit{ff}$	$p \vee \mathit{ff} = p$
True:	$p \wedge \mathit{tt} = p$	$p \vee \mathit{tt} = \mathit{tt}$
Distributivity:	$p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r)$

PROOF. All the above properties are straightforward since $\wedge$ ($\vee$) on processes corresponds to $\cap$ ($\cup$) on complete ready trees by Thm. 19, and since complete ready trees and ready trees induce the same preorder by Cor. 13(2). Moreover, $\mathrm{cRT}(\mathit{ff})$ is the empty set of complete observation trees, while $\mathrm{cRT}(\mathit{tt})$ is the set of all complete observation trees. $\square$

We may also obtain the expected results for relating $\wedge$ and $\vee$ to $\sqsubseteq$.

Proposition 25 (Relating $\wedge$, $\vee$ to $\sqsubseteq$)

(1) $p \wedge q = q \iff p \sqsubseteq q$
(2) $p \vee q = p \iff p \sqsubseteq q$

PROOF. Both of these statements follow from Thm. 19 and Cor. 13(2). $\square$

We conclude this section by briefly returning to the illustrative processes *spec* and *impl* of Fig. 4. We have already remarked that the only complete ready tree of the latter is also one of the former. Hence, by Thm. 22, *impl* is indeed a refinement of *spec* according to our ready-tree preorder. Considering the conjunction of these processes, also shown in Fig. 4, it might be easier to see this using Prop. 25(1).

5 Comparing Ready-Tree Semantics to Other Semantics

In the following, we compare ready-tree semantics to four other semantics, namely *possible-worlds semantics*, *ready-trace semantics*, *failures semantics* and *ready simulation* [6,8]. Since our treatment of divergence is different from the one of failures semantics, we restrict our discussion to τ-free processes in Sections 5.2 and 5.3.

5.1 Possible-Worlds Semantics

Our ready-tree semantics is in essence the *path-based possible-worlds semantics* of van Glabbeek [6]. This semantics is inspired by the possible-worlds semantics that was introduced by Veglioni and De Nicola in [7]. Their idea was to consider a specification that offers a choice between different behaviours as "standing for a set of models, where each model represents one of the possible behaviours specified" [7]. However, their semantics had several technical shortcomings which were pointed out and addressed by van Glabbeek in his handbook article [6]. Van Glabbeek refers to Veglioni and De Nicola's original semantics as *state-based* possible-worlds semantics and has coined the corrected version *path-based* possible-worlds semantics.

Despite the strong similarities, there are differences between van Glabbeek's path-based possible-worlds semantics and our ready-tree semantics. Firstly, van Glabbeek's framework does not consider τ-transitions and thus does not address divergence. Secondly, our framework of Logic LTS does not only include τ-transitions but also a true- and a false-predicate. In ready trees, the true-predicate has the effect of terminating observation; this concept does not exist in possible-worlds semantics. Thirdly, van Glabbeek's semantics uses deterministic cyclic labelled transition systems, in addition to what we call ready trees. However, doing so has no effect on the semantics' expressive power.

Note that within our setting of τ*-pure* LTSs, the theory of ready-tree semantics is – when putting the issue of divergence aside – almost fully determined by the sub-theory of τ-free LTSs. The reason can be expressed in process algebra by saying that the kernel of $\precsim$, which we call *ready-tree equivalence*, satisfies the law

$$a.\sum_{i\in I}\tau.b_i.p_i = \sum_{i\in I} a.b_i.p_i$$

This law allows any τ-pure process to be rewritten as a ready-tree-equivalent process that features no τ-moves at all, except for the case that the process allows an initial τ-move. Even in that case one could get rid of τ-moves, albeit at the price of having multiple initial states for modelling initial nondeterminism.

5.2 Ready-Trace Semantics

A *ready trace* [12] of a process is a sequence of actions that it can perform and where, at the beginning of the trace, between any two actions and at the end, the ready set of the process reached at the respective stage is inserted. Such a ready trace can be understood as a particular type of ready tree that consists only of a single path and includes additional transitions representing the ready sets. These additional transitions ensure that each state on the path has, for each action in its ready set, exactly one transition that either belongs to the path or ends in a *true*-state.

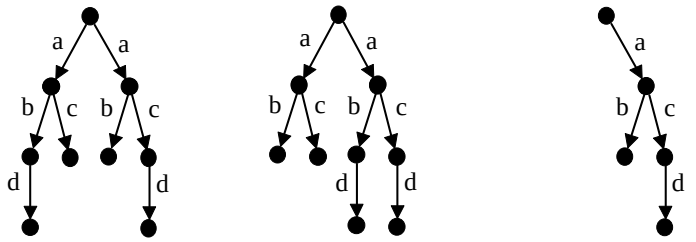

Fig. 7. Ready-tree semantics is strictly finer than ready-trace semantics.

For example, the first ready tree in Fig. 5 in Section 3 represents the ready trace $\{a, b\}b\{b\}b\{a, b\}$. Consequently, the ready traces of a process can be read off from its ready trees, and ready-tree inclusion implies ready-trace inclusion. The reverse implication does not hold as demonstrated by the two left-most processes in Fig. 7. These possess the same ready traces; however, the observation tree on the right-hand side is a ready tree of the first, but not of the second process.

5.3 Failures Semantics

The *failures semantics* of a process is the set of its refusal pairs. Such a pair consists of a trace followed by a refusal set, i.e., a set of actions that the process reached by the trace cannot perform. Such a refusal pair can be read off from the respective ready trace by deleting all its ready sets and adding a set of actions having an empty intersection with the last ready set on the trace. Thus, ready-tree semantics is finer than failures semantics.

5.4 Ready Simulation

Intuitively, a process q *ready-simulates* some process p if there exists a simulation relation from p to q such that related states have identical ready sets [9,13].

Definition 26 (Ready Simulation on Logic LTS)
Let $\langle P, \longrightarrow_P, T_P, F_P\rangle$ *and* $\langle Q, \longrightarrow_Q, T_Q, F_Q\rangle$ *be two Logic LTS. A relation* $\mathcal{R} \subseteq P \times Q$ *is a* ready simulation relation*, if the following conditions hold, for any* $\langle p, q\rangle \in \mathcal{R}$ *and* $a \in \mathcal{A}$*:*

(1) $q \in T_Q$ *implies* $p \in T_P$*;*
(2) $q \xrightarrow{\tau} q'$ *and* $p \notin T$ *implies* $\exists p'.\, p \stackrel{\varepsilon}{\Longrightarrow}_F p'$ *and* $\langle p', q'\rangle \in \mathcal{R}$*;*
(3) $q \xrightarrow{a} q'$ *and* $p \notin T$ *implies* $\exists p'.\, p \stackrel{a}{\Longrightarrow}_F p'$ *and* $\langle p', q'\rangle \in \mathcal{R}$*;*
(4) q *stable and* $p \notin T$ *implies* p *stable and* $\mathcal{I}(p) = \mathcal{I}(q)$*.*

We say that p ready simulates q*, in symbols* $p \preceq_r q$*, if there exists a ready simulation relation* $\mathcal{R}$ *and some* p' *with* $p \stackrel{\varepsilon}{\Longrightarrow}_F p'$ *such that* $\langle p', q\rangle \in \mathcal{R}$*.*

Item (4) also ensures that p in Item (3) is stable, and that hence its weak transition is of the special form we require in this paper. In the definition of $p \preceq_r q$, p may perform a weak transition first; this allows an unstable p to ready simulate a stable q, and it corresponds to the special root-condition in Definition 7(2).

It turns out that ready simulation includes the ready-tree preorder. This is a valuable result for applications of our refinement theory, as will become evident in Section 6.2.

Theorem 27 $\preceq_r \subseteq \sqsubseteq_{\approx}$.

This result has been shown in [6] for τ-free labelled transition systems and can be adapted to our framework of Logic LTS. The key observation is that, when $p \preceq_r q$ and tracing a ready tree of p, ready simulation translates this ready tree to the same ready tree for q.

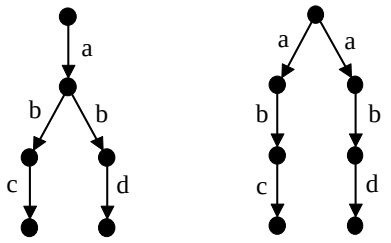

Fig. 8. The ready-tree preorder is strictly coarser than ready simulation.

Fig. 8 shows that the ready-tree preorder is indeed *strictly* coarser than ready simulation. Both processes displayed have the same ready trees, all of which are paths. However, the second process cannot even simulate the first one.

6 Specification, Design & Refinement – An Example

As explained earlier, our motivation for studying conjunction on processes is to provide a basis for combining operational and logical styles of specification. Moreover, any such *heterogeneous* specification language should be equipped with a compositional refinement preorder that allows one to trade off operational contents for logical contents.

This section demonstrates that Logic LTS, together with the ready-tree preorder, provide the foundation for realising our vision. We first show how our setting, which so far only incorporates logical operators, namely conjunction and disjunction, may be augmented with a parallel composition operator. We then apply this extended framework to a non-trivial example which is concerned with the specification and design of a simple *mode logic* [14]. Mode logics are key components of modern digital control systems, such as flight guidance systems installed in aircraft. We detail how our heterogeneous specification can be refined step-by-step, first to an abstract design and then to a detailed, fully operational design.

6.1 Parallel Composition

We start off by defining a simple parallel composition operator on Logic LTS, namely the *fully synchronous product* $\|$. The only slight difficulty is in defining the transitions of a process that is composed with a *true*-process, such as tt.

Intuitively, a true-process can autonomously decide which set of actions to offer initially. Within a fully synchronous product with some process p, this means that, e.g., $p \,\|\, tt$ may behave according to *any* subset of initial actions of p. In particular, tt is not a neutral element for parallel composition, as is to be expected. In the following, it is convenient to write $\mathcal{I\!I}(p)$ for the set $\{a \in \mathcal{A} \,|\, p \stackrel{\varepsilon}{\Longrightarrow}_F \stackrel{a}{\longrightarrow}\}$. Note that $\mathcal{I\!I}(p)$ is finite if p is finite-branching. We also assume that, without loss of generality, all LTSs except observation trees contain tt as a process in their T-set. Finally, the process tt_A, for $A \subseteq \mathcal{A}$ with $A \neq \emptyset$, denotes a true-process that has autonomously chosen to offer the actions in A next. Hence, it can engage in any a-transition with $a \in A$ and thereafter behave like tt again.

Definition 28 (Synchronous Parallel Composition) *The synchronous parallel composition of Logic LTSs* $\langle P, \longrightarrow_P, T_P, F_P\rangle$, $\langle Q, \longrightarrow_Q, T_Q, F_Q\rangle$ *is the LTS* $\langle P \,\|\, Q, \longrightarrow_{P\,\|\,Q}, T_{P\,\|\,Q}, F_{P\,\|\,Q}\rangle$ *defined by:*

- $P \,\|\, Q =_{df} \{p \,\|\, q \mid p \in P, q \in Q\} \cup \{tt \,\|\, q,\, tt_A \,\|\, q \mid \emptyset \neq A \subseteq \mathcal{I\!I}(q)\,, q \in Q\} \cup \{p \,\|\, tt,\, p \,\|\, tt_A \mid A \subseteq \mathcal{I\!I}(p)\,, p \in P\}$

- $\longrightarrow_{P \parallel Q}$ *is determined by the following operational rules:*

$$
\begin{array}{rcll}
p \xrightarrow{\tau}_P p' & \Longrightarrow & p \parallel q \xrightarrow{\tau}_{P \parallel Q} p' \parallel q & \\
q \xrightarrow{\tau}_Q q' & \Longrightarrow & p \parallel q \xrightarrow{\tau}_{P \parallel Q} p \parallel q' & \\
p \xrightarrow{a}_P p',\ q \xrightarrow{a}_Q q' & \Longrightarrow & p \parallel q \xrightarrow{a}_{P \parallel Q} p' \parallel q' & \\
p \in T_P,\ \emptyset \neq A \subseteq \mathcal{I}(q) & \Longrightarrow & p \parallel q \xrightarrow{\tau}_{P \parallel Q} tt_A \parallel q & (*) \\
q \in T_Q,\ \emptyset \neq A \subseteq \mathcal{I}(p) & \Longrightarrow & p \parallel q \xrightarrow{\tau}_{P \parallel Q} p \parallel tt_A & (**)
\end{array}
$$

- $p \parallel q \in T_{P \parallel Q}$ *if and only if* $p \in T_P$ *and* $q \in T_Q$
- $p \parallel q \in F_{P \parallel Q}$ *if and only if* $p \in F_P$ *or* $q \in F_Q$

Note that the parallel composition of two Logic LTS is indeed a Logic LTS. Indeed, our definition of parallel composition is almost identical to the one for conjunction and differs only in two aspects.

Firstly, and most importantly, it differs in the treatment of inconsistencies as inconsistent processes are only inherited from the argument LTS. In other words, when composing LTSs in parallel no new inconsistencies arise, whereas a conjunctive composition may add new inconsistencies. As a simple example, let us re-consider the processes p and q of Fig. 1, which specify that exactly action a and respectively action b is offered initially. Both $p \parallel q$ and $p \wedge q$ have no outgoing transitions. However, their conjunctive composition reveals an inconsistency, i.e., $p \wedge q \in F$, while their parallel composition is consistent, i.e., $p \parallel q \notin F$.

Secondly, the treatment of *true*-processes in the definition of the transition relation $\longrightarrow_{P \parallel Q}$ is special. This is because true-processes do not behave as neutral elements with respect to parallel composition, as explained earlier. For example, operational rule $(**)$ states that, when composing process p with some $q \in T_Q$, process q decides autonomously on the set of initial actions A to be offered next. This explains the τ-transition of $p \parallel q$ to $p \parallel tt_A$. In the next step, $p \parallel tt_A$ can only engage in an action in $\mathcal{I}(p) \cap A$. Note that it is therefore sufficient to consider in our operational rule only those A that satisfy $\emptyset \neq A \subseteq \mathcal{I}(p)$. Summarising the behaviour of a parallel composition of p with some $q \in T_Q$, process q nondeterministically chooses, for each execution step, which transitions of p (if any) to cut off.

In order to convey the exact behaviour of parallel composition on the semantic level of ready trees, we first define a notion of parallel composition on observation trees.

Definition 29 *Let* $\langle V_1, \longrightarrow_1, T_1, \emptyset \rangle$ *and* $\langle V_2, \longrightarrow_2, T_2, \emptyset \rangle$ *be observation trees with roots* v_0^1 *and* v_0^2*, respectively. The parallel composition of these trees, written* $v_0^1 \Qoppa\!\!\parallel\!\!\Qoppa v_0^2$*, is the observation tree* $\langle V, \longrightarrow, T, \emptyset \rangle$*, where*

- $V =_{df} \{v_1 \pitchfork v_2 \mid v_1 \in V_1\ , v_2 \in V_2\}$, *restricted to vertices reachable from* $v_0^1 \pitchfork v_0^2$*;*
- $v_1 \pitchfork v_2 \stackrel{a}{\longrightarrow} v_1' \pitchfork v_2'$ *if* $v_1 \stackrel{a}{\longrightarrow}_1 v_1'$ *and* $v_2 \stackrel{a}{\longrightarrow} v_2'$*;*
- $v_1 \pitchfork v_2 \in T$ *if* $v_1 \in T_1$ *or* $v_2 \in T_2$*.*

It is straightforward to check that the parallel composition of two observation trees is indeed an observation tree. We lift the definition of $\pitchfork$ elementwise to sets $\mathcal{T}_1, \mathcal{T}_2$ of observation trees, i.e., $\mathcal{T}_1 \pitchfork \mathcal{T}_2 =_{\text{df}} \{v_0^1 \pitchfork v_0^2 \mid v_0^1 \in \mathcal{T}_1,\, v_0^2 \in \mathcal{T}_2\}$. The following proposition relates parallel composition on LTSs to parallel composition on ready trees.

Proposition 30 *$RT(p \parallel q) = RT(p) \pitchfork RT(q)$.*

PROOF. We start off by proving direction "$\subseteq$". Let v_0 with $h : V \longrightarrow P \parallel Q$ be a ready tree of $p \parallel q$. Moreover, let us initially assume that $T_P = T_Q = \emptyset$. We construct two observation trees v_0^1 and v_0^2 with $h_1 : V_1 \longrightarrow P$ and $h_2 : V_2 \longrightarrow Q$. The trees v_0^1 and v_0^2 contain V as well as all of v_0's edges and its T-set. In addition, we set $h_1(v) =_{\text{df}} p'$ and $h_2(v) =_{\text{df}} q'$, whenever $h(v) = p' \parallel q'$. Hence, Conds. (1)-(3) of Def. 7 are satisfied. Furthermore, at this state, tree v_0 is the parallel composition of the two trees constructed so far.

We now turn to Cond. (4) of Def. 7 and take some $v \notin T$ such that $h(v) \notin T$. Let $h(v) = p' \parallel q'$. Then, $\mathcal{I}(v) = \mathcal{I}(p') \cap \mathcal{I}(q')$. For each $a \in \mathcal{I}(p') \setminus \mathcal{I}(v)$, we add to the tree v_0^1 a new vertex v' and an edge $v \stackrel{a}{\longrightarrow} v'$, and we put v' into T_1. Further, we choose a stable p'' with $p' \stackrel{a}{\Longrightarrow}_F p''$ and set $h_1(v') =_{\text{df}} p''$. Note that this addition of vertex and edge does not change the composition of the two constructed trees since $a \notin \mathcal{I}(q')$ – even if one repeats our extension construction for all new vertices in both trees. Moreover, the new vertex satisfies Conds. (1), (3) and (4) of Def. 7.

We repeat this construction for each v and a, and analogously for the second tree. Now all vertices in the constructed trees satisfy Cond. (4) of Def. 7. As already observed above, the parallel composition of the two trees is indeed the studied tree of $p \parallel q$.

If T_P or T_Q contain *true*-states, then one has to slightly change the construction in the following way. Assume that transition $v \stackrel{a}{\longrightarrow} v'$ is the first on some path from v_0^1 where the corresponding computation $h(v) \stackrel{a}{\Longrightarrow}_F h(v')$ uses operational rule $(*)$, and let this computation have the form $h(v) = p' \parallel q' \stackrel{a}{\Longrightarrow}_F p'' \parallel q'' \stackrel{\tau}{\longrightarrow} tt_A \parallel q'' \stackrel{\varepsilon}{\Longrightarrow}_F tt_A \parallel q''' = h(v')$, for some p'', q''. We set $h_1(v') =_{\text{df}} p'' \in T_P$ and also for the vertices v'' of the branch of v'. Note that, in this branch, Conds. (1), (3) and (4) of Def. 7 are satisfied without further additions. We can make similar alterations if the operational rule $(**)$, or both operational rules $(*)$ and $(**)$, have been used, and analogously for $p \parallel q \stackrel{\varepsilon}{\Longrightarrow}_F h(v_0)$.

For proving the reverse inclusion "$\supseteq$", we take two ready trees v_0^1 and v_0^2 of p and q with labellings $h_1 : V_1 \longrightarrow P$ and $h_2 : V_2 \longrightarrow Q$, respectively. Identifying the vertices with the action sequences that lead to them, we can consider $v_0^1 \pitchfork v_0^2$ as the intersection $\langle V, \longrightarrow, T, \emptyset \rangle$ of these trees.

Let us initially assume that no processes in T_P and T_Q play a role for the trees v_0^1 and v_0^2, respectively. We define $h(v) =_{\mathrm{df}} h_1(v) \,\|\, h_2(v)$ and check Def. 7. Verifying Conds. (1)-(3) is easy. For Cond. (4), we take $v \notin T$ (and $h(v) \notin T$ by assumption). The set $\mathcal{I}(v)$ is the intersection of $\mathcal{I}(v) = \mathcal{I}(h_1(v))$ in v_0^1 and $\mathcal{I}(v) = \mathcal{I}(h_2(v))$ in v_0^2, while $\mathcal{I}(h(v)) = \mathcal{I}(h_1(v)) \cap \mathcal{I}(h_2(v))$ by Def. 28.

Now, consider the case that $v \xrightarrow{a} v'$ in V, $h_1(v) \notin T_P$ and $h_1(v) \stackrel{a}{\Longrightarrow}_F h_1(v') \in T_P$ according to Cond. (3b) of Def. 7, while $h_2(v') \notin T_Q$. Let $A' =_{\mathrm{df}} \mathcal{I}(v')$ in v_0^1 and $A =_{\mathrm{df}} A' \cap \mathcal{I}(h_2(v'))$. Then, $h_1(v) \,\|\, h_2(v) \stackrel{a}{\Longrightarrow}_F h_1(v') \,\|\, h_2(v') \xrightarrow{\tau} tt_A \,\|\, h_2(v') = h(v')$. This satisfies Conds. (1) and (3) in Def. 7. For Cond. (4), note that $\mathcal{I}(h(v')) = A = A' \cap \mathcal{I}(h_2(v'))$, where $\mathcal{I}(v') = A'$ in v_0^1 and $\mathcal{I}(v') = \mathcal{I}(h_2(v'))$ in v_0^2.

The case of $v \xrightarrow{a} v'$ in V with $h_1(v) = h_1(v') \in T_P$ is similar: $h(v) = tt_A \,\|\, h_2(v) \xrightarrow{a} tt \,\|\, q' \stackrel{\varepsilon}{\Longrightarrow}_F tt \,\|\, h_2(v') \xrightarrow{\tau} tt_B \,\|\, h_2(v') = h(v')$ for suitable A, B and q'. Analogously, we treat the case $h_2(v') \in T_Q$ and the case $h_1(v') \in T_P$ and $h_2(v') \in T_Q$. $\Box$

Prop. 30 is the key for proving that operator $\|$ on LTS is compositional.

Theorem 31 (Compositionality) $p \lesssim q \implies p \,\|\, r \lesssim q \,\|\, r$, *for any* r.

PROOF. $p \lesssim q \iff$ (by Def. 9) $\mathrm{RT}(q) \subseteq \mathrm{RT}(p) \iff$ (by the definition of $\pitchfork$ on ready tree sets) $\mathrm{RT}(q) \pitchfork \mathrm{RT}(r) \subseteq \mathrm{RT}(p) \pitchfork \mathrm{RT}(r) \iff$ (by Prop. 30) $\mathrm{RT}(q \,\|\, r) \subseteq \mathrm{RT}(p \,\|\, r) \iff$ (by Def. 9) $p \,\|\, r \lesssim q \,\|\, r$. $\Box$

6.2 Specifying & Designing a Mode Logic

Our example concerns the design of mode logics which can, e.g., be found in flight guidance systems. We assume a rather simple mode logic that only controls the horizontal and vertical axes of a small aircraft.

The mode of each of the two axes can be either `ON` or `OFF`. If its status is `ON`, this is signalled via event $\mathtt{on}_1$ and $\mathtt{on}_2$, respectively. Note that we will consistently index states and events of the horizontal mode by 1 and of the vertical mode by 2. Each mode i, for $i \in \{1, 2\}$, may toggle between states $\mathtt{ON}_i$ and $\mathtt{OFF}_i$ upon the switch event $\mathtt{sw}_i$.

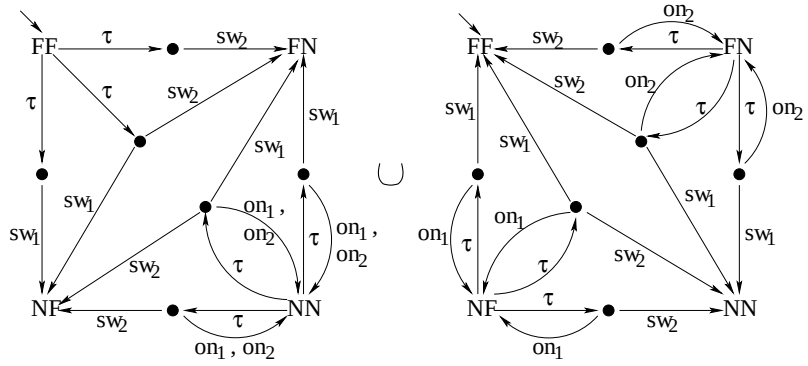

Fig. 9. Monolithic specification of the mode logic.

A typical, monolithic specification SPEC of our mode logic might look like the one sketched in Fig. 9. Here the states of the horizontal and vertical mode are abbreviated by two letters, where, e.g., FN encodes that the horizontal mode is in state OFF and the vertical mode is in state ON. Moreover, the monolithic specification has so many transitions that, in order to ensure readability, we needed to draw its LTS in two diagrams that need to be superposed in the obvious way. In each of the major four states FF, FN, NF and NN, at least one of the two modes offers its switch event. This is each encoded by an internal choice (disjunction) via three τ-transitions to an intermediate state where either $\mathtt{sw}_1$, or $\mathtt{sw}_2$, or both are offered.

The specification SPEC of Fig. 9 is so difficult to read that no system designer would take this as a starting point for design. Instead, designers know the architecture of mode logics very well, which has not changed in decades; hence, they naturally prefer to specify new mode logics in a component-based fashion.

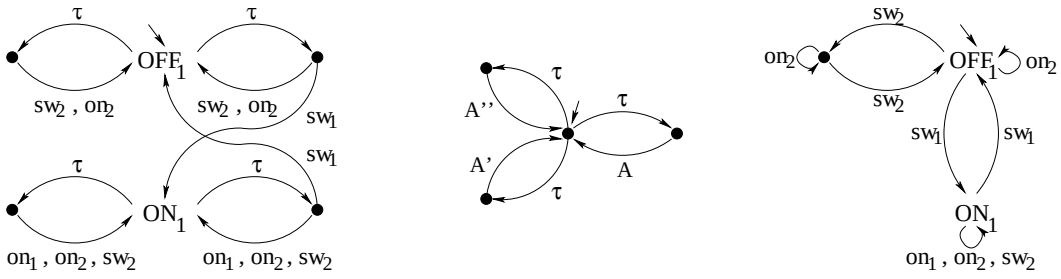

Fig. 10. Abstract design (left), constraint (middle) and concrete design (right).

Accordingly, a designer starts off by developing an abstract design of each mode in isolation. A possible design $\mathtt{SPEC}_1^a$ of the horizontal mode is depicted on the left in Fig. 10. In each principal state, $\mathtt{OFF}_1$ and $\mathtt{ON}_1$, the mode can internally decide either to offer event $\mathtt{sw}_1$ (via the τ-transitions pointing right), or not to offer this event (via the τ-transitions pointing left). The rationale is that, under certain conditions which we will discuss later, a mode should not allow certain switch events. The abstract design $\mathtt{SPEC}_2^a$ of the vertical mode is analogous to the horizontal mode, but with indices 1 and 2 interchanged.

One may now consider the synchronous parallel composition $\mathtt{SPEC}_1^a \parallel \mathtt{SPEC}_2^a$ as the mode logic's abstract design. However, this structured design is not a refinement of the monolithic specification $\mathtt{SPEC}$: $\mathtt{SPEC} \sqsubseteq\!\!\!\!\sim \mathtt{SPEC}_1^a \parallel \mathtt{SPEC}_2^a$ does *not* hold since $\mathtt{SPEC}_1^a \parallel \mathtt{SPEC}_2^a$ can deadlock whereas $\mathtt{SPEC}$ cannot. This is because both $\mathtt{SPEC}_1^a$ and $\mathtt{SPEC}_2^a$ can autonomously decide to move away from their initial $\mathtt{OFF}$ states via their left τ-transitions, thereby reaching a composed state with no outgoing transition. In order to fix this problem, the designer decides not to modify the abstract design of either $\mathtt{SPEC}_1^a$ or $\mathtt{SPEC}_2^a$, but simply to conjunctively add a declarative constraint $\mathtt{C}_{\mathtt{dead}}$.

Further thought reveals that what is demanded is actually not deadlock freedom but something stronger, namely that at least one of the events $\mathtt{sw}_1$ and $\mathtt{sw}_2$ must be enabled at any state of the abstract design. This constraint is expressed via the LTS $\mathtt{C}_{\mathtt{dead}}$ which is sketched in the middle of Fig. 10. This LTS has a τ-branch for each $A \subseteq \{\mathtt{on}_1, \mathtt{on}_2, \mathtt{sw}_1, \mathtt{sw}_2\}$ such that $\mathrm{sw}_1 \in A$ or $\mathtt{sw}_2 \in A$. Each τ-branch returns to its initial state via a bundle of transitions, one for each action $a \in A$. In Fig. 10, this bundle of transitions is simply depicted as a single transition labelled A. One may think of this LTS being automatically generated from a temporal logic formula that states "*always* ($\mathtt{sw}_1$ or $\mathtt{sw}_2$)".

This leads to the overall abstract design

$$(\mathtt{SPEC}_1^a \parallel \mathtt{SPEC}_2^a) \wedge \mathtt{C}_{\mathtt{dead}} \ .$$

Conjoining $\mathtt{C}_{\mathtt{dead}}$ has the effect of marking all those states of $\mathtt{SPEC}_1^a \parallel \mathtt{SPEC}_2^a$ as inconsistent that have not at least one of the events $\mathtt{sw}_1$ and $\mathtt{sw}_2$ enabled. Thus, any paths that only lead to deadlock are implicitly removed. It is straightforward to check that $\mathtt{SPEC} \sqsubseteq\!\!\!\!\sim (\mathtt{SPEC}_1^a \parallel \mathtt{SPEC}_2^a) \wedge \mathtt{C}_{\mathtt{dead}}$ holds. Indeed, the LTS of $(\mathtt{SPEC}_1^a \parallel \mathtt{SPEC}_2^a) \wedge \mathtt{C}_{\mathtt{dead}}$ coincides with the one of $\mathtt{SPEC}$, except that each single τ-transition of $\mathtt{SPEC}$ is refined by multiple, but confluent τ-transitions.

The mode logic our designer wishes to built is special in that it must ensure that both modes can never be simultaneously in their $\mathtt{ON}$ state. Again, rather than making messy changes to the monolithic specification or the abstract design, the designer decides simply to conjoin a second constraint $\mathtt{C}_{\mathtt{on}}$. This constraint is defined analogously to constraint $\mathtt{C}_{\mathtt{dead}}$ but the sets $A \subseteq \{\mathtt{on}_1, \mathtt{on}_2, \mathtt{sw}_1, \mathtt{sw}_2\}$ are chosen such that $\{\mathtt{on}_1, \mathtt{on}_2\} \not\subseteq A$. This modifies the abstract design to

$$((\mathtt{SPEC}_1^a \parallel \mathtt{SPEC}_2^a) \wedge \mathtt{C}_{\mathtt{dead}}) \wedge \mathtt{C}_{\mathtt{on}} \ .$$

and, due to Thm. 20(1), we have that this refines $\mathtt{SPEC} \wedge \mathtt{C}_{\mathtt{on}}$ with respect to our ready-tree preorder.

The designer now wishes to step-wise refine this abstract design into a more concrete design which may then be handed over to the implementation team. In particular, the constraints $\mathtt{C}_{\mathtt{dead}}$ and $\mathtt{C}_{\mathtt{on}}$ shall be removed by refining the

abstract designs $\mathtt{SPEC}_1^a$ and $\mathtt{SPEC}_2^a$ to more concrete designs $\mathtt{SPEC}_1^c$ and $\mathtt{SPEC}_2^c$, respectively, which entail the two constraints.

The designer recognises that both constraints $\mathtt{C}_{\mathtt{on}}$ and $\mathtt{C}_{\mathtt{dead}}$ may be eliminated without the help of new, auxiliary actions. The abstract design of the horizontal mode, for example, may be made more concrete as depicted by the LTS on the right in Fig. 10, to which we refer as $\mathtt{SPEC}_1^c$. In its $\mathtt{OFF}_1$ state, the mode simply toggles on event $\mathtt{sw}_2$ via the auxiliary state to the left. More precisely, whenever the vertical mode is switched on, i.e., i.e., after an odd number of $\mathtt{sw}_2$ events, the horizontal mode disables the $\mathtt{sw}_1$ event. When the vertical mode is switched off, the horizontal mode enables $\mathtt{sw}_1$ again. Moreover, when the horizontal mode is in state $\mathtt{ON}_1$, then the arrival of a $\mathtt{sw}_2$ event does not have any effect.

The concrete design $\mathtt{SPEC}_1^c$ indeed refines its abstract design $\mathtt{SPEC}_1^a$ because of $\mathtt{SPEC}_1^a \sqsubseteq \mathtt{SPEC}_1^c$. This is a consequence of the fact that $\mathtt{SPEC}_1^a$ *ready simulates* $\mathtt{SPEC}_1^c$, which can easily be checked by referring to Def. 26, and that ready simulation is included in our ready-tree preorder according to Thm. 27. As a consequence, we obtain

$$\begin{aligned}\mathtt{SPEC} \wedge \mathtt{C}_{\mathtt{on}} &\sqsubseteq ((\mathtt{SPEC}_1^a \parallel \mathtt{SPEC}_2^a) \wedge \mathtt{C}_{\mathtt{dead}}) \wedge \mathtt{C}_{\mathtt{on}} \\ &\sqsubseteq ((\mathtt{SPEC}_1^c \parallel \mathtt{SPEC}_2^a) \wedge \mathtt{C}_{\mathtt{dead}}) \wedge \mathtt{C}_{\mathtt{on}} \\ &\sqsubseteq ((\mathtt{SPEC}_1^c \parallel \mathtt{SPEC}_2^c) \wedge \mathtt{C}_{\mathtt{dead}}) \wedge \mathtt{C}_{\mathtt{on}}\end{aligned}$$

by first refining $\mathtt{SPEC}_1^a$ and then, analogously, $\mathtt{SPEC}_2^a$. Here, $\mathtt{SPEC}_2^c$ is defined as $\mathtt{SPEC}_1^c$ but with indices 1 and 2 interchanged.

As intended, $\mathtt{SPEC}_1^c \parallel \mathtt{SPEC}_2^c$ already satisfies both constraints $\mathtt{C}_{\mathtt{dead}}$ and $\mathtt{C}_{\mathtt{on}}$, since $\mathtt{C}_{\mathtt{dead}} \sqsubseteq \mathtt{SPEC}_1^c \parallel \mathtt{SPEC}_2^c$ and $\mathtt{C}_{\mathtt{on}} \sqsubseteq \mathtt{SPEC}_1^c \parallel \mathtt{SPEC}_2^c$. Again, this can best be seen via a ready simulation relation. Using Prop. 25(1) and Prop. 24 (idempotence and associativity) repeatedly, we may conclude our sequence of refinement steps and obtain the mode logic's concrete design as follows:

$$\begin{aligned}&\sqsubseteq (\mathtt{SPEC}_1^c \parallel \mathtt{SPEC}_2^c) \wedge (\mathtt{SPEC}_1^c \parallel \mathtt{SPEC}_2^c) \wedge \mathtt{C}_{\mathtt{on}} \\ &\sqsubseteq (\mathtt{SPEC}_1^c \parallel \mathtt{SPEC}_2^c) \wedge \mathtt{C}_{\mathtt{on}} \\ &\sqsubseteq (\mathtt{SPEC}_1^c \parallel \mathtt{SPEC}_2^c) \wedge (\mathtt{SPEC}_1^c \parallel \mathtt{SPEC}_2^c) \\ &\sqsubseteq (\mathtt{SPEC}_1^c \parallel \mathtt{SPEC}_2^c)\ .\end{aligned}$$

Hence, our advocated approach supports the step-wise and component-wise refinement of abstract, mixed operational and declarative specifications to concrete, purely operational designs.

7 Related Work

Traditionally, process-algebraic and temporal-logic formalisms are not mixed but co-exist side by side [15,16]. Indeed, the process-algebra school often uses synchronous composition and internal choice to model conjunction and disjunction, respectively. The compositionality of classic process-algebraic refinement preorders, such as failures semantics [10] and must-testing [5], enables component-based reasoning. However, inconsistencies in specifications are not captured so that, e.g., the conjunctive composition of a and b is identified with deadlock rather than *ff*. In contrast, the temporal-logic school distinguishes between deadlock and *ff* but does not support component-based refinement.

Much research on mixing operational and logical styles of specification avoids dealing with inconsistencies by translating one style into the other. On the one hand, operational content may be translated into logic formulas, as is implicitly done in Lamport's TLA [17] or in the work of Graf and Sifakis [18]. In these approaches, logical implication serves as refinement relation [19]. On the other hand, logical content may be translated into operational content. This is the case in automata-theoretic work, such as Kurshan's work on ω-automata [20], which includes synchronous and asynchronous composition operators and uses maximal trace inclusion as refinement relation. However, both logical implication and trace inclusion are insensitive to deadlock and thus do not support component-based reasoning.

The idea to develop a specification language that combines logics and process algebra is already advocated by Bouajjani, Graf and Sifakis in [21]. In this paper, a μ-calculus-like logic is given, and a subset of its formulas is extended to a set of processes, e.g., with an operator of parallel composition. One main result is an adequacy theorem which states that two processes are bisimilar exactly if they satisfy the same set of formulas. No behavioural notion of refinement is investigated. In this approach, logical formulas and processes overlap, but a logical operator like conjunction is only defined on the former, while a process operator like parallel composition is only defined on the latter. Such an overlap can also be found in the work of Hennessy and Plotkin [22], where a very simple process algebra with a disjunction operator is studied. Operationally, the respective processes can be understood as acyclic finite LTSs (without true- and false-predicate) in our setting and the disjunction corresponds to our disjunction; in contrast to our aims, deadlock is ignored and the behaviour of a process consists simply of its traces. A main result is that the disjunction operator behaves like disjunction w.r.t. the satisfaction for some modal logic: if p or q satisfy a formula, then so does $p \vee q$.

Holmström follows a related approach in [23], where he extends the modal μ-calculus by operators of the process algebra CCS [24] and equips the mixed

language with an incomplete set of proof rules. The specification $\phi|\psi$, for example, is "refined" by process $p|q$ if p refines ϕ and q refines ψ. In contrast to our work, refinement simply means logical satisfaction and is not related to a process-algebraic refinement relation; e.g. the conjunction of specifications is simply defined as intersection, in the style of logical satisfaction, and does not account for inconsistencies in the sense of our approach. In particular, each process seen as specification is only satisfied by itself; as a consequence, the conjunction of different processes is always inconsistent.

A seminal approach to compositional refinement relations in a mixed setting was proposed by Olderog in [25], where process-algebraic constructs are combined with trace formulas expressed in a predicate logic. In this approach, trace formulas can serve as processes, but not vice versa. Thus, freely mixing operational and logical styles is not supported and, in particular, conjunction cannot be applied to processes. For his setting, Olderog develops a denotational semantics that is a slight variation of standard failures semantics. Remarkably, an inconsistent formula is given a semantics that is not an element of the appropriate domain, as is stated on pp. 172-173 of [25].

Recently, a more general approach to combining process-algebraic and temporal-logic approaches was proposed in two papers by Cleaveland and Lüttgen [3,4], which adopt a variant of De Nicola and Hennessy's must-testing preorder [5] as refinement preorder. However, Cleaveland and Lüttgen have not successfully solved the challenge of defining a semantics that is both deadlock-sensitive and compositional, and in which the conjunction operator and the refinement relation are compatible in the sense of Prop. 25(1). Our work solves this problem in the basic setting of Logic LTS. Key for the solution is our new understanding of inconsistency, which is reflected by the fact that we consider processes a and $a + b$ as inconsistent, whereas they were treated as consistent in [4]. Observe that also in failure semantics and must-testing, a and $a + b$ are inconsistent in the sense that they do not have a common implementation.

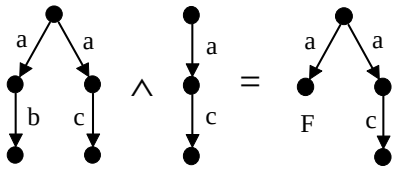

Fig. 11. Backward propagation of inconsistency.

In addition, our backward propagation of inconsistency (cf. Def. 1(3)) is in line with traditional semantics, as is illustrated in Fig. 11. The first conjunct specifies the second conjunct with respect to failures semantics and must-testing, whence their conjunction, also shown in Fig. 11, should be consistent. In fact, the conjunction equals the second process in our ready-tree semantics.

Another, very different setting aimed at increasing the expressive power of

process-algebraic specifications was presented by Larsen and Thomsen [26,27]. Their setting employs *modal transition systems* which distinguish between required and allowed transitions, and leads to a natural notion of refinement which is based on turning allowed transitions into required transitions. In this model, it is also possible to define logical conjunction [28].

Finally, it must be noted that the term *consistency* as used here has little in common with the same term in [29]. In that paper, two specifications are called consistent if they have at least one implementation in common. In our setting, this is trivially the case since ff implements any specification, as $p \precsim ff$ for any p. However, one may rephrase the question posed in [29] for our setting and ask under which circumstances two specifications would permit a common, *consistent* implementation. According to Thm. 21(1) as well as Def. 4 and Thm. 22, a necessary condition is for the conjunction of the two specifications to be consistent. This condition is also sufficient: due to Prop. 24 (idempotence and associativity) and 25, we have $p \wedge p \wedge q = p \wedge q$ and $p \sqsubseteq p \wedge q$; analogously, $q \sqsubseteq p \wedge q$ holds.

8 Conclusions & Future Work

This article proposed a notion of conjunction on processes. Our framework was one of τ-pure Logic LTSs, with distinguished *true*- and *false*-states. Key for defining the conjunction operator was the careful, inductive formalisation of an inconsistency predicate. The implied ready-tree semantics is, in essence, van Glabbeek's path-based possible-worlds semantics, extended from τ-free labelled transition systems to our framework. Moreover, the resulting ready-tree preorder is fully-abstract with respect to a naive preorder that allows inconsistent specifications to be refined only by inconsistent implementations. It is also compositional for conjunction, disjunction – corresponding to internal choice – and parallel composition with full synchronisation.

Consequently, this article solves the problems of defining conjunction which are reported in closely related work [3,4], albeit in a simpler setting that only considers process-algebraic operators on a small scale. Standard laws of boolean algebra hold as expected, due to the fact that conjunction and disjunction on LTSs correspond to intersection and union on ready trees, respectively. It is the simplicity of our setting that brought the subtleties of defining a fully-abstract semantics in the presence of conjunction to light, and which offered a way forward in addressing the challenge of defining "logical" process calculi, i.e., process calculi that allow one to freely mix process-algebraic and temporal-logic operators [4].

Future work shall extend our results to richer frameworks. Firstly, we plan

to lift our requirement of τ-purity on LTS. This requires some care, as interpreting disjunction as internal choice in non-τ-pure settings may lead to counter-intuitive behaviour, as is noted in [11].

Secondly, we plan to add further standard process-algebraic operators to our setting, such as asynchronous parallel composition, hiding and recursion. In particular hiding is likely to prove challenging due to its transformation of observable infinite behaviour into divergent behaviour.

Thirdly, our framework shall be semantically extended from LTS to Büchi LTS [3] so that one may express liveness and fairness properties, and syntactically to linear-time temporal-logic formulas [4]. We also wish to explore tool support along the lines described in [30].

Acknowledgements

We thank Rance Cleaveland for many fruitful discussions and particularly for suggesting the use of an inconsistency predicate. We are also grateful to the anonymous referees, particular for pointing out the similarities of ready-tree semantics to possible-worlds semantics.

References

[1] J. Bergstra, A. Ponse, S. Smolka, Handbook of Process Algebra, Elsevier Science, 2001.

[2] A. Pnueli, The temporal logic of programs, in: FOCS '77, IEEE Computer Society Press, 1977, pp. 46–57.

[3] R. Cleaveland, G. Lüttgen, A semantic theory for heterogeneous system design, in: FSTTCS 2000, Vol. 1974 of LNCS, Springer-Verlag, 2000, pp. 312–324.

[4] R. Cleaveland, G. Lüttgen, A logical process calculus, in: EXPRESS 2002, Vol. 68,2 of ENTCS, Elsevier Science, 2002.

[5] R. De Nicola, M. Hennessy, Testing equivalences for processes, TCS 34 (1983) 83–133.

[6] R. van Glabbeek, The linear time – branching time spectrum I, in: Handbook of Process Algebra, Elsevier Science, 2001, Ch. 1, pp. 3–99.

[7] S. Veglioni, R. De Nicola, Possible worlds for process algebras, in: CONCUR '98, Vol. 1466 of LNCS, Springer-Verlag, 1998, pp. 179–193.

[8] R. van Glabbeek, The linear time – branching time spectrum II, in: CONCUR '93, Vol. 715 of LNCS, Springer-Verlag, 1993, pp. 66–81.

[9] B. Bloom, S. Istrail, A. Meyer, Bisimulation can't be traced, J. ACM 42 (1) (1995) 232–268.

[10] S. Brookes, C. Hoare, A. Roscoe, A theory of communicating sequential processes, J. ACM 31 (3) (1984) 560–599.

[11] M. Steen, H. Bowman, J. Derrick, E. Boiten, Disjunction of LOTOS specifications, in: FORTE/PSTV '97, Vol. 107 of IFIP Conf. Proc., Chapman & Hall, 1998, pp. 177–192.

[12] J. Baeten, J. Bergstra, J. Klop, Ready-trace semantics for concrete process algebra with the priority operator, Computer J. 30 (6) (1987) 498–506.

[13] K. Larsen, A. Skou, Bisimulation through probabilistic testing, Inform. & Comp. 94 (1) (1991) 1–28.

[14] S. Miller, Specifying the model logic of a flight guidance system in CoRE and SCR, in: FMSP '98, ACM Press, 1998, pp. 44–53.

[15] G. Boudol, K. Larsen, Graphical versus logical specifications, TCS 106 (1) (1992) 3–20.

[16] M. Dam, Process-algebraic interpretations of positive linear and relevant logics, J. Log. Comput. 4 (6) (1994) 939–973.

[17] L. Lamport, The temporal logic of actions, TOPLAS 16 (3) (1994) 872–923.

[18] S. Graf, J. Sifakis, A logic for the description of non-deterministic programs and their properties, Inform. & Control 68 (1–3) (1986) 254–270.

[19] M. Abadi, G. Plotkin, A logical view of composition, TCS 114 (1) (1993) 3–30.

[20] R. Kurshan, Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach, Princeton Univ. Press, 1994.

[21] A. Bouajjani, S. Graf, J. Sifakis, A logic for the description of behaviours and properties of concurrent systems, in: REX Workshop, Vol. 354 of LNCS, Springer-Verlag, 1988, pp. 398–410.

[22] M. Hennessy, G. Plotkin, Finite conjunctive nondeterminism, in: Concurrency and Nets: Advances in Petri nets, Springer-Verlag, 1987, pp. 233–244.

[23] S. Holmström, A refinement calculus for specifications in Hennessy-Milner logic with recursion, FAC 1 (3) (1989) 242–272.

[24] R. Milner, Communication and Concurrency, Prentice Hall, 1989.

[25] E. Olderog, Nets, Terms and Formulas, Cambridge Tracts in Theoretical Computer Science 23, Cambridge Univ. Press, 1991.

[26] K. Larsen, B. Thomsen, A modal process logic, in: LICS '88, IEEE Computer Society Press, 1988, pp. 203–210.

[27] K. Larsen, Modal specifications, in: Automatic Verification Methods for Finite State Systems, Vol. 407 of LNCS, Springer-Verlag, 1989, pp. 232–246.

[28] K. Larsen, B. Steffen, C. Weise, A constraint oriented proof methodology based on modal transition systems, in: TACAS '95, Vol. 1019 of LNCS, Springer-Verlag, 1995, pp. 17–40.

[29] M. Steen, J. Derrick, E. Boiten, H. Bowman, Consistency of partial process specifications, in: AMAST '98, Vol. 1548 of LNCS, Springer-Verlag, 1999, pp. 248–262.

[30] R. Cleaveland, O. Sokolsky, Equivalence and preorder checking for finite-state systems, in: Handbook of Process Algebra, Elsevier Science, 2001, pp. 391–424.