To appear in the Handbook of Process Algebra; Jan Bergstra, Alban Ponse and Scott Smolka (eds.), Elsevier, 2000

Action Refinement

Roberto Gorrieri
Dipartimento di Scienze dell'Informazione, Università di Bologna
Mura Anteo Zamboni 7, I–40127 Bologna
gorrieri@cs.unibo.it

Arend Rensink
Department of Computer Science, University of Twente
Postbus 217, NL–7500 AE Enschede
rensink@cs.utwente.nl

Abstract

In this chapter, we give a comprehensive overview of the research results in the field of action refinement during the past 12 years. The different approaches that have been followed are outlined in detail and contrasted to each other in a uniform framework. We use two running examples to discuss their effects, benefits and disadvantages. The chapter contains results only; appropriate references are given to the original papers containing the proofs.

Contents

1 Introduction

1.1 What is action refinement about?

A widely accepted approach to specify the behaviour of concurrent systems relies on state/transition abstract machines, such as labelled transition systems: an activity, supposed to be atomic at a certain abstraction level, can be represented by a transition, the label of which is the name of the activity itself. Once these *atomic actions* are defined, one technique to control the complexity of a concurrent system specification is by means of (horizontal) modularity: a complex system can be described as composed of smaller subsystems. Indeed, this is the main achievement of process algebras: the specification is given as a term whose subterms denotes subcomponents; the specification, as well as its analysis, can be done component-wise, focussing on few details at a time.

However, from a software engineering viewpoint, the resulting theory may in many cases still be unsuitable, as the abstraction level is fixed once and for all by the given set of atomic actions. In the development of software components, it may be required to compare systems that belong to conceptually different abstraction levels (where the change of the level is usually accompanied by a change in the sets of actions they perform) in order to verify if they realise essentially the same functionality. Once the sets of actions at the different abstraction levels are defined, a technique (orthogonal to the previous one) for controlling the complexity of concurrent system specifications is by means of *vertical* modularity: a complex system can be first described succinctly as a simple, abstract specification and then refined stepwise to the actual, complex implementation; the specification, as well as the analysis on it, can be done level by level, focussing each time on the relevant details introduced by passing from the previous level to the current one. This well-known approach is sometimes referred to as *hierarchical specification methodology.* It has been successfully developed for sequential systems, yielding, for instance, a technique known as *top-down systems design*, where a high-level "instruction" is macro-expanded to a lower level "module" until the implementation level is reached (see, e.g., [141]).

In the context of process algebra, this refinement strategy amounts to introducing a mechanism for transforming high-level primitives/actions into lower level processes (i.e., processes built with lower level actions).

Example 1.1 *As a running (toy) example in this chapter, we consider a data base which can be* queried *using an operation qry and* updated *using an operation upd. Both are atomic, i.e., once invoked their effect is as if they finish immediately, and no concurrently invoked action can interfere with them. The latter operation is then transformed into a transaction consisting of two phases, req in which the update is requested and cnf in which it is confirmed. The question addressed in this chapter is what the behaviour of the data base on the resulting lower level of abstraction should be.*

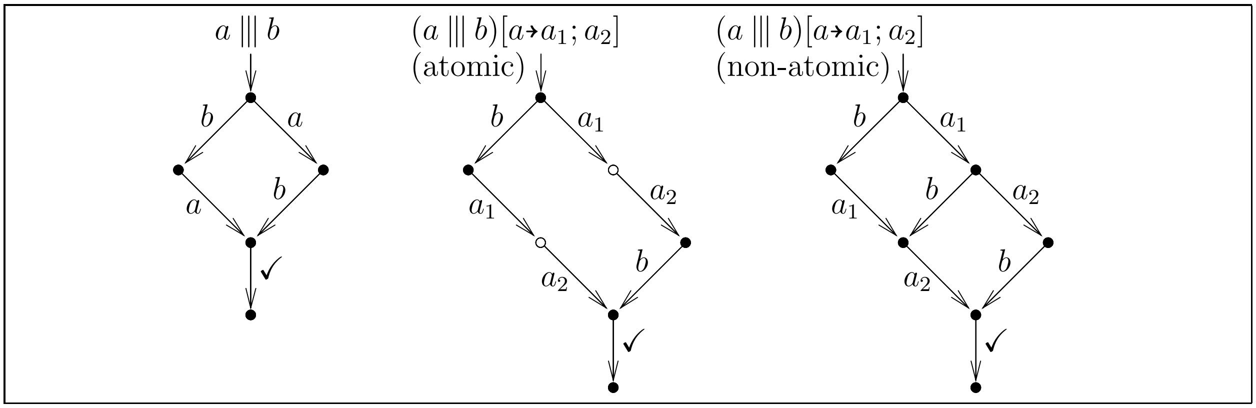
There are several ways to go about studying this issue, depending on some choices that can be taken. These are discussed in the following subsections.

1.2 Refinement operator vs. hierarchy of descriptions

In traditional programming languages, there is an operator that supports a hierarchical specification methodology: the declaration (and call) of a procedure, given by (some syntactical variant of) "`let` $a = u$ `in` t". This specifies that the abstract name a is declared to equal its body u in the scope t. So, whenever a is encountered during the execution of t, u is executed in its place. Similarly, one way to support vertical modularity in process algebra is by introducing an explicit operator, called action refinement and written $t[a \to u]$, which plays a role similar to that of procedure call: it is nothing but a declaration, introducing the name a for its body u in the scope t. The discussion about the possible meanings of the refinement operator is postponed to Sections 1.3 and 1.4; here we simply recall that the main problem faced by the advocates of this approach, the so-called *congruence problem*, is to find an observational equivalence which respects the refinement operator. A non-exhaustive list of papers following this approach in process algebra is [8, 9, 13, 19, 31, 37, 38, 48, 50, 69, 89, 110, 121, 126] and in semantic models [22, 42, 52, 62, 63, 93, 130, 131, 132].

Most of this chapter is devoted to a study of the operator for action refinement within process algebra. However, also another approach to support vertical modularity is discussed in this chapter: a hierarchy of descriptions, equipped with a suitable implementation relation establishing an ordering among them. Typically, a concurrent system, described at several levels of detail, can be seen as a collection of different albeit related systems. Each of these systems may be described in a particular (process algebraic) language. Therefore, in order to relate the various systems, it is necessary that we are able to correctly relate the different languages. The implementation of a language into another language may be often seen as the definition of the primitives of the former as derived operators of the latter. Of course, if we assume that all the systems are described in the same language (as we do in this chapter, as described in Section 1.7), the task is easier. Anyway, some work is needed: a suitable partition of the action set by abstraction levels, a *refinement function* associating lower level processes to high level actions, and an *implementation relation* that states when a low level process implements a high level process, according to the refinement function. This "vertical" implementation relation is not to be confused with existing "horizontal" implementation relations, such as trace or testing pre-orders, which rather reflect the idea that a given system implements another on the *same* abstraction level, by being closer to an actual implementation, for instance more deterministic.

Although action refinement as an operator or through a hierarchy of descriptions are solutions to the same problem, a comparison is not easy. On the one hand, a single language with a mechanism for hierarchy among its operators is a quite appealing approach, as it permits to define the horizontal and the vertical modularities in a uniform way. Indeed, this is in the line of the development of sequential languages: the definition of control abstraction mechanisms, such as procedures and functions, or of data abstraction mechanisms, such as abstract data types, should be considered a standard way of "internalising", in the usual horizontal modularity, concepts that are typical of the vertical one. On the other hand, this approach has also some disadvantages: no clear separation of the abstraction levels in the specification (free combination of horizontal and vertical operations) and no clear distinction of what actions are of what level (confusion may be risky, as we will discuss in Section 1.4 and, much more extensively, in Section 6). Another major difference between the two approaches is the following. According to the former approach, given a specification S and a refinement function, there is *only one* possible implementation I; hence, there is no need to develop a notion of *correctness* of the implementation: the implementation I is what one obtains by applying the operator of refinement to S. On the contrary, the latter approach may admit several different implementations for a given specification, namely all those that are correct according to the vertical implementation relation. See Section 1.7 for a further discussion. For the time being, we concentrate on the interpretation of action refinement as an operator.



$a \mathbin{|||} b$
$(a \mathbin{|||} b)[a \to a_1; a_2]$
(atomic)
$(a \mathbin{|||} b)[a \to a_1; a_2]$
(non-atomic)
b
a
a_1
a_2
$\checkmark$

Figure 1: Atomic and non-atomic refinement of $a \mathbin{|||} b$

1.3 Atomic vs. non-atomic action refinement

The basic approaches to action refinement can be divided into two main groups. On the one hand, there is *atomic refinement* [13, 19, 48, 72, 75, 97], where one takes the point of view that actions are atomic and their refinements should in some sense preserve this atomicity. On the other hand, there is a more liberal notion of refinement — called *non-atomic* — according to which atomicity is always relative to the current level of abstraction, and may in a sense be destroyed by refinement.

To better explain this issue, let us consider one simple example: $a \mathbin{|||} b$, representing the parallel composition of a and b, and the refinement of a by $a_1; a_2$. Figure 1 shows the labelled transition systems for $a \mathbin{|||} b$ and $(a \mathbin{|||} b)[a \to a_1; a_2]$ when refinement is atomic and non-atomic, respectively. In this figure, black dots represent abstract observable states (i.e., states that are relevant for observation and where every atomic action is completed) and white dots denote concrete invisible states (i.e., states that represent intermediate execution steps at a lower level of abstraction and that cannot be seen by an external observer). It is easy to see the difference: if the refinement is atomic, there is no observable state in between the execution of a_1 and a_2 (all-or-nothing); moreover, action b cannot be executed in between their execution (non-interruptible).

The atomic approach seems well-suited in some cases. For instance, when implementing one language into another one, one needs to implement the primitives of the former as compound programs of the latter. In this case, keeping atomicity in the implementation may be a vital feature for correctness. One example of this is [77], where Milner's CCS is mapped to a finer grained calculus; each transition of CCS is implemented as a suitable transaction (sequence of transitions executed atomically) of the latter calculus. Another example supporting atomic refinement is when mutual exclusion on a shared resource is necessary to prevent faulty behaviour. For instance, when refining an abstract write operation a on a shared variable x as a complex process u, we would like to allow further, possibly parallel, reading or writing operations on x only after the completion of u. Further arguments in favour of atomic refinement may be found in [19].

On the other hand, also the non atomic approach has its own adherents; see [9, 31, 50, 63, 89, 93, 110, 121, 133]. Actually, this approach is on the whole more popular than the former. For instance, in the example of Figure 1, if b is an action completely independent of a it seems unreasonable to impose the restriction that b stays idle while executing the sequence $a_1\, a_2$. In our opinion, the choice between atomic and non-atomic refinement should be driven by the application at hand.

1.4 Syntactic vs. semantic action refinement

There are essentially two interpretations of action refinement, which we call *syntactic* and *semantic*. In the syntactic approach, the treatment of action refinement closely resembles the copy rule for procedure call (i.e., inlining the body for the calling action) in sequential programming. For instance, [110] exploits a *static* copy rule (syntactic replacement before execution); instead [8] follows the *dynamic* copy rule: as soon as a is to occur while executing t, the first action of u is performed reaching, say, a state u', after which that occurrence of a in t is syntactically replaced by u'. In either case, the semantics of $t[a \rightarrow u]$ is, *by definition*, the semantics of the term $t\{u/a\}$. Among other things, this implies that the process algebra is to be equipped with an operation of sequential composition (rather than the more standard action prefix) as studied, for instance, in the context of ACP [11], since otherwise it would not be closed under the necessary syntactic substitution.

In the *semantic* interpretation, a substitution operation is defined in the semantic domain used to interpret terms. Then the semantics of $t[a \rightarrow u]$ can be defined using this operator. For example, when using event structures as semantic domains, an event structure $\mathcal{E} = \llbracket u \rrbracket$, representing the semantics of u, would be substituted for every a-labelled event d in the event structure $\llbracket t \rrbracket$. The refinement operation preserves the semantic embedding of events: e.g., if d is in conflict with an event e, then all the events of $\mathcal{E}$ will be in conflict with e, and similarly for the order relation. Investigations of such refinement operators can be found in e.g. [17, 31, 32, 41, 42, 50, 93, 60, 63, 64, 68, 121, 131, 133] over the semantic domains of Prime, Free, Flow and Stable Event Structures, Configuration Structures, Families of Posets, Synchronisation Trees, Causal Trees, ST Trees and Petri Nets. The advocates of the semantic substitution approach (to which the authors of this chapter belong) claim that the starting point is introducing a notion of semantic refinement as pure substitution in a semantic model, which, usually, is not very difficult. In contrast, the hard part is finding the operational definition for the syntactic operator of refinement that correctly implements semantic substitution, i.e., a concurrent counterpart of the copy rule in the sequential case.

These two approaches are inherently different; simple examples showing this are given in Section 6. Essentially, syntactically substituting u for a in t produces a confusion of the abstraction levels that is not possible with semantic substitution; such a confusion may originate new communications between processes that were not possible at the higher level, and, conversely, may destroy at the lower level some communications established at the higher level. Conceptually, this is an undesirable situation which, in general, prevents the definition of an algebraic theory for action refinement. Technically, syntactic refinement corresponds to a *homomorphism* between algebras whose signature is given by the language (since such a homomorphism is essentially generated by a mapping from actions to subterms, which is required to distribute over all the operators); on the other hand, semantic refinement defines an operation on the semantic model which is *compositional*. As the two approaches do not coincide, we cannot expect to be able, in general, to define compositional homomorphisms. In Section 6, we compare the two approaches with the aim to identify under which restrictions they yield the same result. That is, we report about conditions under which the following diagram commutes:

$$\begin{tikzcd} t[a \rightarrow u] \arrow[r, "\text{syntactic ref.}"'] \arrow[d, "\text{semantic ref.}"'] & t\{u/a\} \arrow[d] & \textit{syntax} \\ \llbracket t \rrbracket[a \rightarrow \llbracket u \rrbracket] \arrow[r] & \llbracket t\{u/a\} \rrbracket & \textit{semantics} \end{tikzcd} \tag{1}$$

Not only does the result give a clearer understanding of the theory of action refinement, but also it is interesting for applications of action refinement to know when semantic refinement can be implemented by the conceptually simpler syntactic substitution.

It should be mentioned that a dual approach to the one described above is followed in [94]: they take syntactic refinement as a starting point and adapt the language (using an operator for *self-synchronisation*) so that semantic refinement coincides with it.

Concerning atomic refinement, one can observe that this is naturally definable as a form of semantic refinement on trees labelled on sequences of actions, but can be also represented via syntactic substitution, provided that the process replacing the action is atomised (see Sections 3 and 6 for more details).

1.5 Interleaving vs. true concurrency

As mentioned above, when action refinement is an operator of the language, a natural key problem is that of finding a notion of equivalence that is a congruence for such operator. Formally, given a candidate equivalence notion $\simeq$, we want to find the coarsest relation $\equiv$, contained in $\simeq$, that is a congruence for all the operators of the language; to be precise:

- whenever $u_1 \equiv u_2$, then $t[a \to u_1] \equiv t[a \to u_2]$;
- whenever $t_1 \equiv t_2$, then $t_1[a \to u] \equiv t_2[a \to u]$.

The first half of the congruence problem turns out to be easy: the main requirement is that one makes a clear distinction between deadlock (where a system can do nothing at all) and termination (where a system can do nothing except terminate, i.e., relinquish control); see also [61]. This distinction is easily made if one models termination as a special action (in this paper denoted $\checkmark$). The reason why a congruence has to make this distinction may be understood by considering an example. Let $t = a; b$, $u_1 = c$ —denoting the execution of c leading to successful termination— and $u_2 = c; \mathbf{0}$ —denoting the execution of c leading to deadlock. u_1 and u_2 are equivalent when ignoring termination; however, $t[a \to u_1]$ can perform b after c, while $t[a \to u_2]$ cannot.

A solution to the second half of the congruence problem, on the other hand, may either be easy (in cases where one can stay within interleaving semantics) or rather difficult (in cases where one is forced to move to truly concurrent semantics), depending on the assumptions and the algebraic properties one wants to impose on the operator.

- One can define semantic action refinement as an operator on transition systems (e.g., see Section 2.3 below) that is well-defined up to strong bisimilarity. (For the case where internal moves are abstracted, the situation is slightly less straightforward: the construction is not well-defined up to the standard (rooted) weak bisimilarity, instead one has to resort to (rooted) *delay* or *branching* bisimilarity. A detailed discussion can be found in Section 2.4 below.) At any rate, it should be clear that there is no intrinsic reason why an interleaving relation cannot be a congruence for action refinement.

- However, the operator referred to above fails to satisfy a very intuitive and important property: namely, it does not distribute over parallel composition. If one wants this property to hold, the easiest way is to adopt *atomic* refinement instead, as argued in [48] (see above and Section 3); both strong and weak bisimilarity are congruences for this operator. The price is that one has either to distinguish concrete and abstract states or to use action sequences rather than single actions as transition labels.

- Atomic refinement is not always appropriate. If one requires a non-atomic refinement operator that distributes over parallel composition, this can still be defined on standard transition systems, provided that refinement is disallowed for all actions that decide choices, as well as all actions that occur concurrently with themselves; see [38]. Once more, strong bisimilarity is a congruence for the resulting operator. As a consequence of the limitations on refinable actions, this operator no longer distributes over *choice*. We do not discuss this operator any further.

- If one wants action refinement to distribute over parallel composition *and* to be non-atomic *and* to be applicable to all actions, irregardless of their position in a term, *then* it becomes necessary to use a model that is more expressive than standard transition systems. Among the earliest observations of this fact are [115, 32]. Since it has received widespread attention in the literature, we will discuss this issue in more detail.

Let us consider $a \mathbin{|||} b$ and $a; b + b; a$, which are equivalent in interleaving semantics: the former represents the concurrent execution of the actions a and b, the latter their execution in either order. When refining a to $a_1; a_2$ (and distributing the refinement over parallel and sequential composition), the resulting processes become equivalent to, respectively, $(a_1; a_2) \mathbin{|||} b$ and $a_1; a_2; b + b; a_1; a_2$; these are no longer equivalent in interleaving semantics, as only the former can execute the sequence $a_1\, b\, a_2$. It follows that the required congruence $\equiv$ cannot equate $a \mathbin{|||} b$ and $a; b + b; a$.

A solution to this problem, which has received a large amount of attention in the literature on action refinement, is to move to so-called *truly concurrent* semantics, i.e., semantic models that contain more information about the concurrency of the system's activities than the standard interleaving semantics. For instance, *event-based models* (inspired by Winskel's *event structures*, see [139, 140]) have been investigated for this purpose in [42, 50, 63, 64, 121]. In Section 4 we present an example of an event-based truly concurrent (operational and denotational) semantics for a language with parallel composition, synchronisation and refinement.

Isomorphism of event-based models gives rise to a congruence; however, it has been argued that this relation is now too strong (rather than too weak as in the case of the interleaving relations), in that it makes more distinctions than strictly necessary. This can be repaired by interpreting the event-based model up to a weaker relation than isomorphism, such as for instance *history-preserving bisimilarity* (see [60, 116, 46]); or, alternatively, by considering less distinguishing models such as *causal trees* (see [39, 42]). It turns out that the *minimal* amount of information one must add (giving rise to the *coarsest* congruences contained in existing interleaving relations) is to distinguish the (related) beginnings and endings of all actions. This is called the ST-principle, after the name chosen by Van Glabbeek and Vaandrager in [65], where it appeared for the first time. In Section 5 we give an overview of truly concurrent observational criteria and the congruence properties to which they give rise; another, very systematic and detailed summary can be found in [64].

As pointed out by Meyer [105, 106] and Vogler [134], the issue of finding congruences with respect to action refinement is also relevant to another, quite different area, namely that of completeness in the presence of process variables. This, too, is briefly discussed in Section 5.

It should be noted that the issue of finding the fully abstract model for action refinement can also be avoided altogether, by interpreting terms as *functions* on denotations rather than basic denotations. Namely, assume *Act* is the set of actions, *Lang* the language under consideration and $\mathcal{M}^{\mathit{flat}}$ is the space of denotational models for the flat (i.e., refinement-free) language fragment $\mathit{Lang}^{\mathit{flat}}$, with denotational constructions $\overline{op}$ for all operators op of $\mathit{Lang}^{\mathit{flat}}$. Based on these, one can define a new denotational model $\mathcal{M} = (\mathit{Act} \to \mathcal{M}^{\mathit{flat}}) \to \mathcal{M}^{\mathit{flat}}$: i.e., objects of $\mathcal{M}$ are *functions* that yield a “flat” model when provided with an arbitrary mapping $f\colon \mathit{Act} \to \mathcal{M}^{\mathit{flat}}$ that “pre-evaluates” (in fact, refines) all action occurrences.

Denotational constructions $\overline{op}$ on $\mathcal{M}$ are obtained by pointwise extension from $\mathcal{M}^{\mathit{flat}}$:

$$\overline{op}(M_1, \ldots, M_n) = \lambda f. \overline{op}(M_1(f), \ldots, M_n(f))$$

for all $M_i \in \mathcal{M}$ $(i = 1, \ldots, n)$, except if op is actually a constant action a (interpreted as a nullary operator), in which case

$$\overline{op} = \lambda f. f(a) \ .$$

$\mathcal{M}$ allows a straightforward definition of refinement, as an operator $\_[a \to \_]\colon (\mathcal{M} \times \mathcal{M}) \to \mathcal{M}$:

$$M_1[a \to M_2] = \lambda f. M_1(f \pm (a \mapsto M_2(f)))$$

where $f \pm (a \mapsto M)$ with $M \in \mathcal{M}^{\mathit{flat}}$ denotes the function $\mathit{Act} \to \mathcal{M}^{\mathit{flat}}$ mapping a to M and all $b \neq a$ to $f(b)$. This immediately gives rise to a corresponding semantic function $[\![\_]\!]\colon \mathit{Lang} \to \mathcal{M}$.

This approach is followed in [83]. An advantage is that, since $\mathcal{M}^{\mathit{flat}}$ is itself not required to be compositional for refinement, it can still consist of basic interleaving models, such as labelled transition systems or (in [83]) traces. On the other hand, this interpretation of terms as functions is intensional: it does not give rise to a concrete representation of behaviour. This is in sharp contrast to the spirit of the rest of the paper. We will not attempt a further comparison.

1.6 Strict vs. relaxed forms of refinement

Non-atomic refinement is more flexible than atomic refinement, because it allows the concurrent execution of a process u refining an action a with the actions in t independent of a. For instance, in $(a \mathbin{|||} b)[a \to a_1; a_2]$, b can be executed in between a_1 and a_2 only if refinement is non-atomic (see Figure 1).

Nonetheless, there are other desirable forms of flexibility that non-atomic refinement, as defined above, is unable to offer. For instance, consider $t = (a; b)[a \to a_1; a_2]$ where actually only a_1 is a necessary precondition (i.e., a cause) for the occurrence of b. According to the definition of non-atomic refinement, t is equivalent to $a_1; a_2; b$, which fails to show that b may be executed independently of a_2. Unfortunately, in the semantic domains usually considered, it happens that the causal context of actions is tightly preserved, hence enforcing causality in the refined system also when not strictly necessary. We could say that traditional refinement is too *strict*: it forces all abstract causalities to be inherited in the implementation.

A possible solution, proposed by Janssen, Poel and Zwiers [91] and Wehrheim [136, 126] is to introduce a certain degree of *relaxation* of the causal ordering during refinement. Technically, this is achieved by means of a *dependency relation* over the universe of actions, combined with a weak form of sequential composition that may allow the execution of actions of the second component if they are independent of those occurring in the first component. In the example above, one may declare that only a_1 and b are dependent, hence in $a_1 \cdot a_2 \cdot b$ (where $\cdot$ is weak sequential composition), b can be performed before a_2.

A possible relaxation of another kind concerns choice rather than sequential composition. Traditional action refinement requires that alternative actions, once refined, give rise to strictly alternative processes. A different view is taken in [51], where refinement is defined over event structures and the conflict relation is not respected tightly by refinement, e.g., conflicting events may be refined to processes that are not completely mutually exclusive. The intuitive motivation for this is that, in a competition, many actions are anyway performed by both processes before the decision of which will be served is taken.

Yet another form of relaxation concerning choice requires that not all options specified by a refinement function must indeed be offered by the refined system. For instance, if we refine a to $a'; b + a'; c$, the abstract system $a; d$ is implemented by the concrete system $(a'; b + a'; c); d$. An interesting alternative is to take the decision about which option to implement during the refinement step, hence allowing $a'; b; d$ or $a'; c; d$ as an implementation, or to turn the nondeterministic choice into a deterministic one, hence allowing $a'; (b + c); d$ as an implementation. This kind of design step is in line with the sort of transformation allowed by standard implementation relations (such as trace or failure inclusion). We do not know of any paper dealing with such relaxed forms of action refinement (see [124] for a discussion on the problems raised by such an approach in presence of communication).

1.7 Vertical implementation

Another way to obtain a more relaxed notion of action refinement, already briefly mentioned in Section 1.2, is by abandoning the notion of an operator and regarding action refinement as an implementation relation instead. There is a long tradition in defining process refinement theories based on the idea that a process I is an implementation of another process S if I is more directly executable, in particular more deterministic according to the chosen semantics. Examples can be found in, for example, [27, 43, 108]; see also [12] for a collection of papers in this line. As these theories do not take changes in the level of abstraction on which S and I are described into account, we call such implementation relations *horizontal*. On the other hand, as also pointed above, almost no theory has been developed to compare systems that realise essentially the same functionality but belong to conceptually different abstraction levels. For this purpose, we have introduced the concept of *vertical* implementation. Some sensible criteria that any vertical implementation relation should satisfy are listed below:

1. It is parametric w.r.t. a *refinement* function r that maps abstract actions of the specification to concrete processes and thus fixes the implementation of the basic building blocks of the abstract system.

2. It is flexible enough (*i*) to offer several possible implementations for any given specification and (*ii*) not to require that the ordering of abstract actions is tightly preserved at the level of their implementing processes, i.e., refinement need not to be strict (as discussed in Section 1.6, above).

3. It is a generalisation of existing *horizontal* implementation relations; i.e., if the refinement function is the identity, then the vertical implementation should collapse to the horizontal implementation. So, the theory of horizontal and vertical implementations can be integrated uniformly.

As we have seen in the previous sections, the classic work on action refinement in process algebra, where it is interpreted as a substitution-like operation on the syntactic or semantic domain, satisfies few of these requirements. In particular, the consequence of the classic approach is that there is *only one* possible implementation for a given specification; in other words, the action refinement function is used as a *prescriptive tool* to specify the only way abstract actions are to be implemented. However, there is no deep motivation for this functional point of view (only one implementation) in favour of a relational one (more than one implementation). For instance, when considering $(a \mathbin{|||} b)[a \to a_1; a_2]$, there is no real reason for not accepting also $a_1; a_2; b + b; a_1; a_2$ as a possible, more sequential implementation; similarly, for $(a; b + b; a)[a \rightarrow a_1; a_2]$ the more parallel implementation $a_1; a_2 \mid\mid\mid b$ could be admitted.

The concept of action refinement through vertical implementation has a striking consequence. The congruence problem (discussed at length in Section 1.5 above) simply disappears: since a specification may admit non-equivalent implementations, *a fortiori* two equivalent specifications need not to have equivalent implementations. Hence, there is no longer a need to move to truly concurrent semantics. This has the advantage of allowing to reuse most existing techniques developed for interleaving semantics. In particular, it is a natural requirement that vertical implementation may collapse to some horizontal relation, by hiding all the actions that were refined, reminiscent of the interface refinement principle discussed in [26]. This makes it possible to mix vertical refinement with established methods for horizontal implementation.

Some of the basic ideas behind this approach were proposed first (in a restrictive setting) in [72] and later (independently) in [118, 119]. See Section 8, based on [123], for more details.

1.8 Overview of the Chapter

We can classify the choices we have made in the material of this chapter according to the discussion above.

Operator vs. hierarchy. In all the next sections, except the last one (Section 8), we consider some process algebra enriched with an operator for action refinement.

Atomic vs. non-atomic. In all the sections that deal with the operator of action refinement but one, we stick to non-atomic refinement. Atomic refinement is dealt with in Section 3, where we consider a process algebra with an operator for parallel composition but without communication, and also (briefly) in Section 7 in the context of action dependencies.

Syntactic vs. semantic. Throughout the chapter we use semantic refinement, as the operation of action refinement is always defined on a semantic domain (trees or event structures), with the exception of Section 6, where we report sufficient conditions to guarantee that the two approaches are the same.

Interleaving vs. true concurrency. We discuss the simplest cases (sequential systems in Section 2 and atomic refinement in Section 3) using interleaving semantics; then, when parallel composition comes into play and refinement is non-atomic, we move to truly concurrent semantics (Sections 4 and 5).

Strict vs. relaxed. All the sections on the operation deal with strict refinement, except Section 7, where we discuss other approaches using a dependency relation to ensure some form of relaxation of the causality relation.

Vertical implementation is covered in Section 8.

The chapter is organised as follows: Section 2 studies non-atomic refinement for the class of sequential systems. We introduce many concepts that will be used throughout the paper, such as well-formedness conditions on admissible terms, allowable refinements and some standard interleaving behavioural equivalences. Section 3 deals with a larger language with parallel composition (but without communication) under the assumption that refinement is atomic. The operational semantics is not standard, making use of concrete invisible states; the denotational semantics uses trees labelled on sequences. Section 4 deals with non-atomic refinement for a full-fledged process algebra. The denotational semantics is given in terms of stable event structures and the operational semantics is very concrete: process terms and transitions are tagged by event annotations. Then, observational semantics and congruence issues for the full language are discussed in Section 5. In Section 6 we present some conditions ensuring that syntactic and semantic action refinement coincide. Relaxed forms of refinement are recalled in Section 7, while Section 8 reports on the issue of vertical implementation relations.

As much as possible, we have presented all the results of this chapter in a single format, by using a common underlying process algebraic language in which the different approaches have been formulated. Predictably, however, various theories come with their own specific concepts, operators and limitations, making a fully integrated presentation impossible. To enable the reader to find his way among the different language fragments and well-formedness conditions used in the various sections, Table 16 (Page 85) gives an overview.

Note that in this chapter, we have limited ourselves to a process algebraic understanding of action refinement. Thus, we have not included a comparison with methods addressing the same concerns in other fields of theoretical computer science, such as logic-based, state-based or stream-based refinement; e.g., [10, 28, 29, 54, 96, 98].

1.9 Whither action refinement?

After enjoying a broad interest in the years 1989–1995 and thereabouts (as evidenced by the sheer number of papers quoted above, as well as the PhD theses [2, 33, 36, 53, 56, 71, 90, 92, 118, 128, 137]), the subject of action refinement has left the central stage of research in process algebra. Yet we feel that the basic principle underlying action refinement, concerning changes in the level of abstraction and the grain of atomicity at which a system is described, is not less relevant now than it was a decade ago. Indeed, the same principle is very fundamental in, for instance, object-based systems, where the method interface of an object on the one hand and its implementation on the other are prime candidates for a description in terms of action refinement.

If we analyse the kind of work that has received the most attention during the aforementioned period, it can be seen that the main issue that has been studied, almost to the exclusion of everything else, is the congruence question for the refinement operator. This has greatly enhanced the insight in the relative advantages of various truly concurrent semantics, some of which are now also being used in other contexts (e.g., ST-semantics for stochastic process algebras, see [23, 25]); and similar for variations on weak (interleaving) bisimulation (see, for instance, [68]). Nevertheless, with the benefit of hindsight, the concentration on this one subject can also be seen to have had some disadvantages: on the one hand, the refinement operator itself has not found much practical use, whereas on the other, this singular focus has prevented the development of alternative approaches, such as the use of action refinement as a correctness criterion rather than an operator. This is what we are currently trying to remedy with the concept of vertical implementation.

Having realised this change in perspective, it becomes clear that there is an enormous amount of work yet to be done. In particular, no attention at all has yet been paid to the extremely important question of how to integrate data refinement and action refinement. Similarly, the only aspect of atomicity addressed so far is the (non-)interference of atomic actions; at least as important is their all-or-nothing nature, in particular the possibility to *abort* an atomic action that (on a concrete level) has already started. Remarkably, in the context of process algebra the latter issue has not been addressed by *any* theory of action refinement we know. Furthermore, although we have begun to explore some of the possible variations on vertical bisimulation, similar studies should be initiated for vertical testing, since the lack of atomicity of test primitives is causing hitherto unexplored problems. Finally, the most challenging task is to apply the emerging theory to realistic applications, be it in the form of case studies of any size or by integrating it with the design phase of some software engineering lifecycle.

Acknowledgements. We wish to thank Mario Bravetti, Pierpaolo Degano, Thomas Firley, Walter Vogler and Heike Wehrheim for their comments on preliminary versions of this chapter.

2 Sequential Systems

We start our technical presentation by addressing the simplest case: a process algebra without parallelism and communication. In this case, we can easily accommodate the new construct of action refinement within interleaving semantics.

2.1 The sequential language

We assume the existence of a universe *Act* of visible actions, ranged over by a, b, $\ldots$, and an invisible action $\tau \notin \mathit{Act}$; we write $\mathit{Act}_\tau = \mathit{Act} \cup \{\tau\}$, ranged over by α. Furthermore, we assume a set *Var* of process variables, ranged over by x, y, z. The language of *sequential processes* (or *agents*), denoted Lang^{seq} and ranged over by t, u, v, is the set of *well-formed* terms generated by the following grammar:

$$\mathsf{T} ::= \mathbf{0} \mid \mathbf{1} \mid \alpha \mid \mathsf{V} + \mathsf{V} \mid \mathsf{T};\mathsf{V} \mid \mathsf{T}/A \mid \mathsf{T}[a \to \mathsf{V}] \mid x \mid \mu x.\mathsf{V} \ ,$$

where $a \in \mathit{Act}$, $\alpha \in \mathit{Act}_\tau$, $A \subseteq \mathit{Act}$ and $x \in \mathit{Var}$ are arbitrary. In this grammar, T is an arbitrary term whereas V stands for a so-called *virgin operand*, on which we impose the condition that it may not contain an *auxiliary operator* —where the constant $\mathbf{1}$ is the only auxiliary operator of Lang^{seq}. These matters are discussed below in more detail.

We first go into the intuitive meaning of the operators, at the same time informally introducing the concepts of well-formedness and guardedness.

- $\mathbf{0}$ is a deadlocked process that cannot proceed.
- $\mathbf{1}$ is a terminated process, that is, a process that immediately terminates with a transition labelled $\checkmark \notin \mathit{Act}_\tau$, expressed by $\mathbf{1} \xrightarrow{\checkmark} \mathbf{0}$.

  $\mathbf{1}$ is an *auxiliary operator*, in the sense that we expect the "user" of the language to write down terms not containing $\mathbf{1}$. Rather, it will be needed to express the semantics of other operators (notably, sequential composition). The fact that an operator is auxiliary influences the well-formedness of terms; see below.
- α can execute action α, and then terminates, i.e., $\alpha \xrightarrow{\alpha} \mathbf{1}$ ($\xrightarrow{\checkmark} \mathbf{0}$).
- $t + u$ indicates a CCS-like choice between the behaviours described by the sub-terms t and u. The choice is decided by the first action that occurs from either sub-term, after which the other sub-term is discarded.

  We call the operands t and u *virgin*, because (due to the semantics of choice) it is clear that they are untouched, in the sense of not having participated in any transition —otherwise the choice would have been resolved. An important general well-formedness condition on terms is that *virgin operands may not contain auxiliary operators.* Specifically, in this case, we require that t and u contain no occurrence of $\mathbf{1}$. (The technical consequence of this condition is that neither operand can be terminated, and hence no $\checkmark$-transition can resolve a choice. This circumvents some intricate technical problems, for instance in the definition of a denotational event-based model; see, e.g., [118, Example 3.7]. A different solution with essentially the same consequence was chosen in [7, 49], where a choice may terminate only if *both* operands can do so.)

- $t; u$ is the sequential composition of t and u, i.e., t proceeds until it terminates properly, after which u takes over; if t does not terminate properly, u is not enabled. This semantics is in the line of ACP [11], and differs from the one in, e.g., [109], where u starts after t is deadlocked. We have chosen the former as it is the only one distinguishing correctly between deadlock and termination.

  Again, u is a *virgin operand*: once it participates in a transition, t is discarded from the term.

  The two main motivations for using sequential composition instead of CCS action prefixing are as follows. On the one hand, the operational (sometimes also the denotational) semantics for action refinement can be naturally defined only by means of a sequential composition operator, as we will see later on; on the other hand, syntactic substitution, which is the way action refinement is implemented in many papers, is naturally defined by means of such an operator.

- t/A behaves as t, except that the actions in A are *hidden*, i.e., turned into the internal action τ that cannot be observed by any external observer.

- $t[a \to u]$ is a process t where a is refined to u. The operand u is *virgin*. Since, by well-formedness, virgin operands may not contain auxiliary operators, it is certain that the refinement of an action is an agent that is not terminated; this prevents the so-called *forgetful refinement* (the implementation of an action by $\mathbf{1}$), which is not only technically difficult, as discussed, e.g., in [31, 118], but also counter-intuitive.

  Apart from forbidding forgetful refinements, we are still rather generous in allowing some types of refining agents that are rather questionable, such as deadlocked or never-ending ones. Intuitively, it seems natural to require that an action, which by itself cannot deadlock, is implemented by a process that cannot deadlock either, since otherwise refinement would introduce deadlocks. However, imposing such a semantic restriction is an unnecessary burden; on the one hand, there is no technical problem in managing deadlocked refining agents; on the other hand, it is not easy to characterise syntactically a large class of deadlock-free processes (while, semantically, the problem is even undecidable).

  Similar arguments hold for infinite refinement: intuitively, an action is certainly accomplished in a finite amount of time; therefore its refinement should eventually terminate, as required e.g. in [48, 90]. A typical term satisfying this requirement under a suitable fairness assumption is $\mu x.a; x + b$. Again, however, allowing arbitrary recursion in refining agents does not complicate matters, whereas restricting it to some special cases would.

- $x \in \mathit{Var}$ is a process variable, presumably bound by some encompassing recursive operator (see next item), or to be replaced by substitution. The variables of t that are not bound are called *free*: we write $\mathit{fv}(t)$ for the free variables of a term t and $\mathit{fv}(t, u)$ for $\mathit{fv}(t) \cup \mathit{fv}(u)$. A term t is called *closed* if $\mathit{fv}(t) = \emptyset$. The free variables can be *instantiated* by substitution: $t\{u/x\}$ denotes the substitution within the term t of every free occurrence of x by the term u. The free variables and their instantiation are formally defined in Table 1.

- $\mu x.t$ with $x \in \mathit{Var}$ is a recursive term. It can be understood through its unfolding, $t\{\mu x.t/x\}$. The variable x is considered to be *bound* in $\mu x.t$, meaning that it cannot be affected by

Table 1: Free variables and syntactic substitution (where *op* is an arbitrary operator)

t	$fv(t)$	$t\{u/x\}$
$op(t_1,\ldots,t_n)$	$\bigcup_{1\leq i\leq n} fv(t_i)$	$op(t_1\{u/x\},\ldots,t_n\{v/x\})$
y	$\{y\}$	$\begin{cases} u & \text{if } y = x \\ y & \text{otherwise} \end{cases}$
$\mu y.t_1$	$fv(t_1) \setminus \{y\}$	$\mu z.(t_1\{z/y\}\{u/x\})$ where $z \notin \{x\} \cup fv(t,u)$

substitution. Therefore, the identity of bound variables is considered irrelevant; in fact, we apply the standard technique of identifying all terms up to renaming of the bound variables, meaning that if y is a fresh variable not occurring free in t, then $\mu x.t$ and $\mu y.t\{y/x\}$ are identified in all contexts. As a further well-formedness condition, we require that all recursion variables are *guarded*, in the sense defined below.

In $\mu x.t$, the recursion body t is considered to be a *virgin operand*, and hence by well-formedness may not contain the auxiliary operator **1**.

We will use $Lang^{seq,fin}$ to denote the recursion-free fragment of $Lang^{seq}$.

Guardedness. The notion of guardedness is used to simplify the proof of correspondence of various kinds of operational and denotational semantics developed in this chapter for different fragments of *Lang*. As usual, the idea is that the semantic model of a recursive term corresponds to a fixpoint of the denotational function generated by its body, and this fixpoint is unique if the variable is guarded in the term; see for instance [108]. Therefore, if one defines the denotational semantics as the fixpoint, and on the other hand, proves that the operational semantics also gives rise to a fixpoint, then the two must coincide.

Definition 2.1 *We first define what it means for a term to be a* guard.

- **1** *and* x *are* not *guards;*
- **0***,* α *and* $t + u$ *are guards;*
- $t[a \rightarrow u]$ *is a guard if* t *is a guard;*
- *For all other operators* op*,* $op(t_1,\ldots,t_n)$ *is a guard if one of the* t_i *is a guard.*

Next, we define what it means for a variable to be guarded in a term.

- x *is guarded in* y *(*$\in$ *Proc) if* $x \neq y$*;*
- x *is guarded in* $t;u$ *if* x *is guarded in* t *and either* t *is a guard or* x *is guarded in* u*;*
- x *is guarded in* $\mu y.t$ *if either* $x = y$ *or* x *is guarded in* t*;*
- *For all other operators* op*,* x *is guarded in* $op(t_1,\ldots,t_n)$ *if* x *is guarded in all* t_i*.*

(Note that the last clause "for all operators *op*" includes the case where *op* is a constant, i.e., **0**, **1** or α.) A typical example of guarded recursion is $\mu x.(\mathbf{1};a);x$, and of non-guarded recursion, $\mu x.x \parallel\!| a$.

Table 2: Transition rules for Lang^{seq}.

$$\frac{}{\mathbf{1} \xrightarrow{\checkmark} \mathbf{0}} \qquad \frac{}{\alpha \xrightarrow{\alpha} \mathbf{1}} \qquad \frac{t \xrightarrow{\alpha} t'}{t+u \xrightarrow{\alpha} t'} \qquad \frac{u \xrightarrow{\alpha} u'}{t+u \xrightarrow{\alpha} u'}$$

$$\frac{t \xrightarrow{\alpha} t'}{t;u \xrightarrow{\alpha} t';u} \qquad \frac{t \xrightarrow{\checkmark} t' \quad u \xrightarrow{\alpha} u'}{t;u \xrightarrow{\alpha} u'} \qquad \frac{t \xrightarrow{\lambda} t' \quad \lambda \notin A}{t/A \xrightarrow{\lambda} t'/A} \qquad \frac{t \xrightarrow{a} t' \quad a \in A}{t/A \xrightarrow{\tau} t'/A}$$

$$\frac{t \xrightarrow{a} t' \quad u \xrightarrow{\alpha} u'}{t[a \rightarrow u] \xrightarrow{\alpha} u';(t'[a \rightarrow u])} \qquad \frac{t \xrightarrow{\lambda} t' \quad \lambda \neq a}{t[a \rightarrow u] \xrightarrow{\lambda} t'[a \rightarrow u]} \qquad \frac{t\{\mu x.t/x\} \xrightarrow{\alpha} t'}{\mu x.t \xrightarrow{\alpha} t'}$$

Well-formedness. To summarise the well-formedness conditions on Lang^{seq}:

- A virgin operand may contain no auxiliary operators. (The virgin operands are the ones denoted V in the grammar of Lang^{seq}; the only auxiliary operator in Lang^{seq} is $\mathbf{1}$.)
- Recursion is allowed on guarded variables only.

2.2 Operational semantics

A labelled transition system (lts, for short) is a tuple $\langle \mathit{Lab}, N, \longrightarrow\rangle$, where Lab is a set of labels (ranged over by λ), N a set of nodes (ranged over by n) and $\longrightarrow \subseteq N \times \mathit{Lab} \times N$ a (labelled) transition relation. $(n, \lambda, n') \in \longrightarrow$ is more often denoted $n \xrightarrow{\lambda} n'$. Sometimes we use transition systems with initial states, $\langle \mathit{Lab}, N, \longrightarrow, \iota\rangle$, where $\iota \in N$. Unless explicitly stated otherwise, Lab will equal $\mathit{Act}_{\tau\checkmark} = \mathit{Act}_\tau \cup \{\checkmark\}$ in this chapter; we usually omit it. N will often correspond to the terms of a language —for instance, Lang^{seq}. Furthermore, the transition systems we use in this chapter all satisfy the following special properties regarding $\checkmark$-labelled transitions:

Termination is deterministic. If $n \xrightarrow{\checkmark} n'$ and $n \xrightarrow{\lambda} n''$, then $\lambda = \checkmark$ and $n'' = n'$.

Termination is final. If $n \xrightarrow{\checkmark} n'$, then there is no $\lambda \in \mathit{Lab}$ such that $n' \xrightarrow{\lambda}$.

Finally, some more terminology regarding transition systems:

- $n \in N$ is called *terminated* if $n \xrightarrow{\checkmark}$;
- $n \in N$ is called *deadlocked* if there is no $\lambda \in \mathit{Lab}$ such that $n \xrightarrow{\lambda}$;
- $n \in N$ is called *potentially deadlocking* if either n is deadlocked or there is a $n \xrightarrow{\lambda} n'$ with $\lambda \neq \checkmark$ such that n' is potentially deadlocking.

One of the main uses of lts's in this chapter is to provide operational semantics. The definition of the transition relation is according to Plotkin's SOS approach [114], meaning that it is the least relation generated by a set of axioms and rules concerning the transition predicate. For Lang^{seq}, the operational rules are given in Table 2. Most operational rules are standard; the only unusual ones are those for refinement. If the action to be refined is executed by t, then the first action of the refinement u is executed instead, followed by the whole residual u'; only when u' is terminated, the computation will proceed with t' subject to the refinement.

Note that, due to the well-formedness of terms (especially the condition that **1** does not occur in the operands of choice), termination is indeed deterministic and final.

The essential consequence of guardedness can be expressed in terms of its operational semantics if we apply the operational rules also on open terms, as follows:

Proposition 2.2 *Assume x to be guarded in t.*

1. $t\{u/x\} \xrightarrow{\lambda} t'$ *if and only if* $t \xrightarrow{\lambda} t''$ *for some* t'' *with* $t' = t''\{u/x\}$.
2. $\mu x.t \xrightarrow{\alpha} t'$ *if and only if* $t \xrightarrow{\alpha} t''$ *for some* t'' *with* $t' = t''\{\mu x.t/x\}$.

2.3 Denotational Semantics

We now present a denotational semantics for Lang^{seq}, using edge-labelled trees as a model. An edge-labelled tree $\mathcal{T}$ is an lts with initial state $\langle N, \longrightarrow, \iota\rangle$, which is connected, acyclic, has no $\longrightarrow$-predecessor for ι and precisely one $\longrightarrow$-predecessor for all other nodes. The trees that are used to refine actions are always non-terminated. The class of all trees is denoted $\mathbf{T}$.

A tree can be easily obtained from an lts with initial state through the *unfolding* operation. The nodes of the unfolding correspond to *paths* through $\mathcal{T}$, starting from the initial state and including all intermediate nodes and labels. Formally, given the lts $\mathcal{T} = \langle N, \longrightarrow, \iota\rangle$, its unfolding, $\mathit{Unf}\,\mathcal{T}$ is the tree $\langle N', \longrightarrow', \iota\rangle$, where:

- $N' = \{n_1\lambda_1 n_2\lambda_2 \ldots \lambda_{k-1}n_k \in (N\ \mathit{Lab})^* N \mid n_1 = \iota, \forall 1 \leq i \leq k{:}\, n_i \xrightarrow{\lambda_i} n_{i+1}\}$
- $\longrightarrow' = \{(\vec{n}\, n_1, \lambda, \vec{n}\,\lambda\, n_1') \mid n_1 \xrightarrow{\lambda} n_1'\}$.

We are now ready to define the operations on trees.

Deadlock. $\mathcal{T}_\bot = \langle\{\mathbf{0}\}, \emptyset, \mathbf{0}\rangle$.

Termination. $\mathcal{T}_\checkmark = \langle\{\mathbf{1}, \mathbf{0}\}, \{(\mathbf{1}, \checkmark, \mathbf{0})\}, \mathbf{1}\rangle$.

Single action. $\mathcal{T}_\alpha = \langle\{\alpha, \mathbf{1}, \mathbf{0}\}, \{(\alpha, \alpha, \mathbf{1}), (\mathbf{1}, \checkmark, \mathbf{0})\}, \alpha\rangle$.

Sequential composition. $\mathcal{T}_1;\mathcal{T}_2 = \mathit{Unf}\,\langle N, \longrightarrow, \iota\rangle$, where

- $N = N_1 \cup \{(n, n') \mid n \xrightarrow{\checkmark}_1, n' \in N_2\}$;
- $\longrightarrow\ = \{(n, \alpha, n') \mid n \xrightarrow{\alpha}_1 n' \not\xrightarrow{\checkmark}_1\}$
  $\cup \{(n, \alpha, (n', \iota_2)) \mid n \xrightarrow{\alpha}_1 n' \xrightarrow{\checkmark}_1\ \}$
  $\cup \{((n, n'), \lambda, (n, n'')) \mid n \xrightarrow{\checkmark}_1, n' \xrightarrow{\lambda}_2 n''\}$;
- $\iota = \iota_1$ if $\iota_1 \not\xrightarrow{\checkmark}_1$, otherwise $\iota = (\iota_1, \iota_2)$ (i.e., when $\mathcal{T}_1$ is isomorphic to $\mathcal{T}_\checkmark$).

Choice. If $\iota_i \not\xrightarrow{\checkmark}_i$ for $i = 1, 2$ and $N_1 \cap N_2 = \emptyset$, then $\mathcal{T}_1 + \mathcal{T}_2 = \mathit{Unf}\,\langle N, \longrightarrow, \iota\rangle$, where

- $N = N_1 \cup N_2 \cup \{(\iota_1, \iota_2)\}$;
- $\longrightarrow\ = \longrightarrow_1 \cup \longrightarrow_2 \cup \{((\iota_1, \iota_2), \alpha, n) \mid \iota_1 \xrightarrow{\alpha}_1 n\} \cup \{((\iota_1, \iota_2), \alpha, n) \mid \iota_2 \xrightarrow{\alpha}_2 n\}$;
- $\iota = (\iota_1, \iota_2)$.

Hiding. $\mathcal{T}/A = \langle N, \longrightarrow', \iota\rangle$, where

- $\longrightarrow' = \{(n, \lambda, n') \mid n \xrightarrow{\lambda} n', \lambda \notin A\} \cup \{(n, \tau, n') \mid n \xrightarrow{\alpha} n', \alpha \in A\}$.

Refinement. If $\iota_2 \not\xrightarrow{\checkmark}_2$, then $\mathcal{T}_1[a \to \mathcal{T}_2] = \mathit{Unf}\,\langle N, \longrightarrow, \iota\rangle$, where

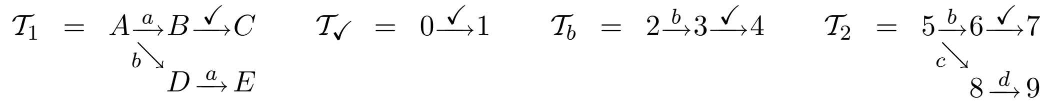
- $N = \{(n_1, n_2) \mid n_1 \in N_1, n_2 \in N_{R(n_1)}\}$;
- $\to = \{((n_1, n_2), \mu, (n_1', n_2')) \mid n_1 \xrightarrow{\lambda}_1 n_1', n_2 \xrightarrow{\checkmark}_{R(n_1)}, \iota_{R(n_1')} \xrightarrow{\mu}_{R(n_1')} n_2'\}$
  $\cup \{((n_1, n_2), \alpha, (n_1, n_2')) \mid n_2 \xrightarrow{\alpha}_{R(n_1)} n_2'\}$;
- $\iota = (\iota_1, \iota_{R(\iota_1)})$.

and $R\colon N_1 \to \mathbf{T}$ is defined as follows:

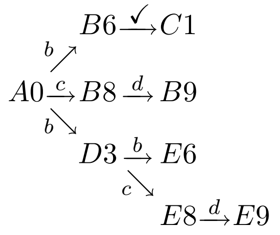
$$R\colon\; n \mapsto \begin{cases} \mathcal{T}_{\checkmark} & \text{if } n = \iota_1 \\ \mathcal{T}_{\lambda} & \text{if } n' \xrightarrow{\lambda} n \text{ for } \lambda \neq a \\ \mathcal{T}_2 & \text{if } n' \xrightarrow{a} n \end{cases}$$

Some comments on the operations above are mandatory. In the operation of sequential composition, the second argument is "reproduced" in as many copies as the number of the nodes that are terminated. The choice operation is nothing but a coalesced sum of trees. Note that these operations are both defined with the help of the *unfold* operation on trees; this simplifies the presentation. The refinement operation is rather unusual, but it holds also when the two trees are infinite as well as when the second tree is potentially deadlocking. A function R is defined, associating to each node n of $\mathcal{T}_1$ a tree depending on the label λ of its incoming transition (which is uniquely determined since $\mathcal{T}_1$ is a tree): $R(n)$ equals $\mathcal{T}_\lambda$ if λ is anything but the action to be refined, $\mathcal{T}_2$ if λ is the action to be refined, and $\mathcal{T}_{\checkmark}$ if there is no incoming transition (n is the initial state). Then each edge $n \xrightarrow{\lambda}_1 n'$ of $\mathcal{T}_1$ is replaced by the tree $R(n')$.

Example 2.3 *Consider the following trees (where the node identities are made explicit):*



Note that $\mathcal{T}_1$ is a model of $a + b; a; \mathbf{0}$ and $\mathcal{T}_2$ is a model of $b + c; d; \mathbf{0}$. The refinement $\mathcal{T}_1[a \to \mathcal{T}_2]$ is then given by the tree



The first thing to make sure of is that the above constructions indeed yield trees, satisfying all the conditions listed above. Since we have defined them through unfolding, for all the operators the proof is straightforward.

Proposition 2.4 *Each of the operators above applied to trees, when defined, again yields a tree.*

In order to deal with recursion, we use an approximation ordering over the class $\mathbf{T}$, which is essentially the one used in [138]. We assume that $\mathcal{T}_i = \langle N_i, \to_i, \iota_i \rangle$ for $i = 1, 2$.

$$\mathcal{T}_1 \sqsubseteq \mathcal{T}_2 \quad :\Leftrightarrow \quad N_1 \subseteq N_2,\ \to_1 = \to_2 \cap (N_1 \times \mathit{Lab} \times N_1),\ \iota_1 = \iota_2$$

(where the symbol ":$\Leftrightarrow$" stands for "is defined to hold if and only if".) Then the following property states that this gives rise to an appropriate semantic domain, namely a (bottom-less) complete partial order, containing a least upper bound for all $\sqsubseteq$-chains. The proof is omitted as it is a variation of the one in [138, Theorem 3.4].

Table 3: Denotational tree semantics; $\overline{op}$ is the semantic counterpart of op

$$
\begin{array}{rcll}
[\![\mathbf{0}]\!] & = & \mathcal{T}_\bot & \\
[\![\mathbf{1}]\!] & = & \mathcal{T}_\checkmark & \\
[\![\alpha]\!] & = & \mathcal{T}_\alpha & \\
[\![op(t_1,\ldots,t_n)]\!] & = & \overline{op}([\![t_1]\!],\ldots,[\![t_n]\!]) & (op \text{ any non-nullary operator}) \\
[\![\mu x.t]\!] & = & \bigcup_{i\in\mathbb{N}}[\![\mu^i x.t]\!] &
\end{array}
$$

Proposition 2.5 *$\langle \mathbf{T}, \sqsubseteq\rangle$ is a complete partial order with minimal elements $\langle\{n\},\emptyset,n\rangle$ for all n and least upper bounds $\bigsqcup_i \mathcal{T}_i = \langle\bigcup_i N_i, \bigcup_i \rightarrow_i, \iota\rangle$ for all $\sqsubseteq$-directed sets $\{\mathcal{T}_i\}_i \subseteq \mathbf{T}$ (with $\iota_i = \iota$ for all i).*

The absence of a bottom element is not deleterious, since the minimal elements do just as well as the starting point of approximation; in particular, note that $\mathcal{T}_\bot$ is a minimal element of $\mathbf{T}$. The next observation is that all the operations defined above are $\sqsubseteq$-continuous functions on $\mathbf{T}$. Indeed, as termination and deadlock are dealt with properly, also the operations of sequential composition (notably on its first argument) and refinement are continuous. Hence, recursion can be computed as the limit of the chain of its approximations. We define approximations of recursive terms, $\mu^i x.t$ for all $i \in \mathbb{N}$, in the standard way (see [138]):

$$
\begin{array}{rcl}
\mu^0 x.t & = & \mathbf{0} \\
\mu^{i+i} x.t & = & t\{\mu^i x.t/x\}
\end{array}
$$

The denotational tree semantics of $Lang^{seq}$ is then given in Table 3. In order to express in what sense the operational and denotational semantics coincide, we first have to go into the issue of semantic relations among transition systems.

2.4 Behavioural semantics and congruences

In an interleaving operational semantics such as the above, a widely accepted equivalence relation on processes is *strong bisimilarity*; see [108]. Here we recall its definition.

Definition 2.6 *Let $\mathcal{T}$ be a transition system.*

- *A* bisimulation *over $\mathcal{T}$ is a symmetric relation $\rho \subseteq N \times N$ such that for all $n_1 \; \rho \; n_2$ and $n_1 \xrightarrow{\lambda} n_1'$, there exists n_2' such that $n_2 \xrightarrow{\lambda} n_2'$ and $n_1' \; \rho \; n_2'$;*
- Strong bisimilarity *over $\mathcal{T}$, denoted $\sim$, is the largest bisimulation over $\mathcal{T}$.*

Moreover, we can also compare states from different transition systems by taking the (disjoint) union of the transition systems and then applying the above definition. A standard proof (cf. [11, 108]) shows that bisimulation is a congruence over $Lang^{seq}$. Alternatively, one can use the "SOS format" theory (e.g., [18, 44, 81]; see [6] for an overview) for this purpose (the rules in Table 2 are in fact in the De Simone format of [44]).

Proposition 2.7 *$\sim$ is a congruence over $Lang^{seq}$.*

Since our operational rules are in the De Simone format, [4] guarantees there is a complete axiomatisation for $Lang^{seq,fin}$, notably also for the operator of action refinement. It turns out that the set of axioms is nothing but the usual set for ACP (see [11]), together with some axioms stating that refinement distributes over the other operators (see Section 6).

Furthermore, an inductive argument very similar to the one used to prove Proposition 2.7 establishes the compatibility of the operational and denotational semantics.

Proposition 2.8 *$t \sim [\![t]\!]$ for all $t \in Lang^{seq}$.*

Proof sketch. By induction on the structure of t. In fact, for those tree constructions that make use of the unfolding operator (choice, sequential composition and refinement) the induction step can already be proved on the transition system obtained *before* unfolding. Since (it is easily shown that) $Unf\,\mathcal{T} \sim \mathcal{T}$ for every lts $\mathcal{T}$, this suffices.

As an example, we will sketch the case for refinement. Let $\mathcal{T}_i = [\![t_i]\!]$ for $i = 1, 2$, and assume $t_i \sim \mathcal{T}_i$. Now $t_1[a \rightarrow t_2]$ is proved by the fact (of which we omit the proof) that the following relation is a bisimulation:

$$\rho = \{(t_1[a \rightarrow t_2], (\iota_1, \iota_{\mathcal{T}_\checkmark}))\} \cup \{(u_2; u_1[a \rightarrow t_2], (n_1, n_2)) \mid u_1 \sim n_1, u_2 \sim n_2\} \ .$$

For the case of recursion, the proof relies on well-formedness, in particular guardedness of recursion: using Proposition 2.2 and also Proposition 2.7, it can be proved that $t\{u/x\} \sim u$ implies $u \sim \mu x.t$ for every guarded recursive term $\mu x.t$. Since by construction, $[\![\mu x.t]\!]$ is a fixpoint of the semantic function that maps all $[\![u]\!]$ to $[\![t\{u/x\}]\!]$, we are done. □

τ-abstracting equivalences. The action τ represents an internal activity that should be considered invisible to some extent. In the literature, the most widely adopted τ-insensitive equivalence is (rooted) *weak bisimilarity*; see, e.g., [108]. To recall the definition, let $\Rightarrow \;=\; \xrightarrow{\tau}^*$.

Definition 2.9 *Let $\mathcal{T}$ be a labelled transition system.*

- *A* weak bisimulation *over $\mathcal{T}$ is a symmetric relation $\rho \subseteq N \times N$ such that for all $n_1 \, \rho \, n_2$ and $n_1 \xrightarrow{\lambda} n_1'$, one of the following holds:*
  - *$\lambda = \tau$ and $n_1' \, \rho \, n_2$;*
  - *there exists n_2' such that $n_2 \Rightarrow\xrightarrow{\lambda}\Rightarrow n_2'$ and $n_1' \, \rho \, n_2'$.*
- *A* root *of a binary relation $\rho \subseteq N \times N$ is a relation $\hat\rho \subseteq \rho$ such that for all $n_1 \, \hat\rho \, n_2$*
  - *if $n_1 \xrightarrow{\tau} n_1'$, then there exists n_2' such that $n_2 \Rightarrow\xrightarrow{\tau}\Rightarrow n_2'$ and $n_1' \, \rho \, n_2'$;*
  - *if $n_2 \xrightarrow{\tau} n_2'$, then there exists n_1' such that $n_1 \Rightarrow\xrightarrow{\tau}\Rightarrow n_1'$ and $n_1' \, \rho \, n_2'$.*
- Weak bisimilarity *over $\mathcal{T}$, denoted $\approx_{\mathsf{w}}$, is the largest weak bisimulation over $\mathcal{T}$, and* rooted weak bisimilarity, *denoted $\simeq_{\mathsf{w}}$, is the largest root of $\approx_{\mathsf{w}}$.*

Unfortunately, (weak and) rooted bisimilarity are not congruences for action refinement, as illustrated by the following example, originally due to [67] (subsumed by [68]), which essentially shows that the third τ-law of [87] does not hold in presence of refinement.

Example 2.10 *Consider $t = a; (b+\tau)$ and $t' = a; (b+\tau)+a$. It is not difficult to observe that $t \simeq_{\mathsf{w}} t'$ (in fact, this is an instance of the third τ-law of [108]). Unfortunately, $t[a \rightarrow a_1; a_2] \not\simeq_{\mathsf{w}} t'[a \rightarrow a_1; a_2]$ because $t'[a \rightarrow a_1; a_2] \xrightarrow{a_1} a_2; (\mathbf{1}[a \rightarrow a_1; a_2])$, hence reaching a state where b is no longer possible, while no bisimilar state can be reached from $t[a \rightarrow a_1; a_2]$ with a transition labelled a_1.*

This example shows that $\simeq_{\mathsf{w}}$ is not preserved by refinement, because the branching structure of processes is not preserved enough by rooted weak bisimilarity. For this reason, Van Glabbeek and Weijland proposed in [67] a finer equivalence, called *branching* bisimilarity, which is a congruence for action refinement. However, the natural question is to single out the *coarsest* congruence for action refinement inside (rooted) weak bisimilarity. The answer is not branching bisimilarity, but rather (rooted) *delay* bisimilarity, based on an equivalence in [107] and provided with this name in [68]. We recall the definitions and the precise results.

Definition 2.11 *Let $\mathcal{T}$ be a labelled transition system.*

- *A* delay [branching] bisimulation *over $\mathcal{T}$ is a symmetric relation $\rho \subseteq N \times N$ such that for all $n_1 \, \rho \, n_2$ and $n_1 \xrightarrow{\lambda} n_1'$, one of the following conditions holds:*
  - *$\lambda = \tau$ and $n_1' \, \rho \, n_2$;*
  - *there exist n_2', n_2'' such that $n_2 \Rightarrow n_2'' \xrightarrow{\lambda} n_2'$ such that $n_1' \, \rho \, n_2'$ [and $n_1 \, \rho \, n_2''$].*
- *A* delay [branching] root *of a binary relation $\rho \subseteq N \times N$ is a relation $\hat{\rho} \subseteq \rho$ such that for all $n_1 \, \hat{\rho} \, n_2$*
  - *if $n_1 \xrightarrow{\tau} n_1'$, then there exist n_2', n_2'' such that $n_2 \Rightarrow n_2'' \xrightarrow{\tau} n_2'$ and $n_1' \, \rho \, n_2'$ [and $n_1 \, \rho \, n_2''$];*
  - *if $n_2 \xrightarrow{\tau} n_2'$, then there exist n_1', n_1'' such that $n_1 \Rightarrow n_1'' \xrightarrow{\tau} n_1'$ and $n_1' \, \rho \, n_2'$ [and $n_1'' \, \rho \, n_2$].*
- Delay [branching] bisimilarity *over $\mathcal{T}$, denoted $\approx_{\mathsf{d}}$ [$\approx_{\mathsf{b}}$], is the largest delay [branching] bisimulation over $\mathcal{T}$, and* rooted delay [branching] bisimilarity*, denoted $\simeq_{\mathsf{d}}$ [$\simeq_{\mathsf{b}}$], is the largest delay [branching] root of $\approx_{\mathsf{d}}$ [$\approx_{\mathsf{b}}$].*

Proposition 2.12

1. *$\simeq_{\mathsf{d}}$ is the coarsest congruence over $Lang^{seq}$ contained in $\simeq_{\mathsf{w}}$;*
2. *$\simeq_{\mathsf{b}}$ is a congruence over $Lang^{seq}$ contained in $\simeq_{\mathsf{d}}$.*

The proof of congruence of Clause 1 is originally due to [52], while the proof that it is the *coarsest* congruence is due to [34]. Walker in [135] proved that the specific axioms of delay bisimilarity are the first and the second τ-laws of rooted weak bisimilarity. Finally, the proof of Clause 2 can be found in [41, 68], whereas the transitivity of $\simeq_{\mathsf{b}}$ is separately addressed in [15].

2.5 Application: A very simple data base

We now give a very simple example of the use of action refinement. In this and similar examples further on, for the sake of readability we use an alternative representation of recursion: instead of writing recursion operators within terms, we write process invocations, where the processes are defined elsewhere as part of the terms' global context. Thus, instead of $\mu x.t$ we may invoke a process X, provided $X := t\{X/x\}$ is a process definition. See also Milner [108]. Moreover, every semantics we discuss in this chapter equates the terms $\mathbf{1}; t$ and t for arbitrary terms t; for that reason, it is always safe to treat the two terms as equal, and in examples we will usually do so.

Consider a distributed data base that can be queried and updated. We first deal with the case where the state of the data base is completely abstracted away from. The behaviour of the system is given by $Data^1_S$, defined by

$$Data^1_S \; := \; (qry + upd); Data^1_S \; .$$

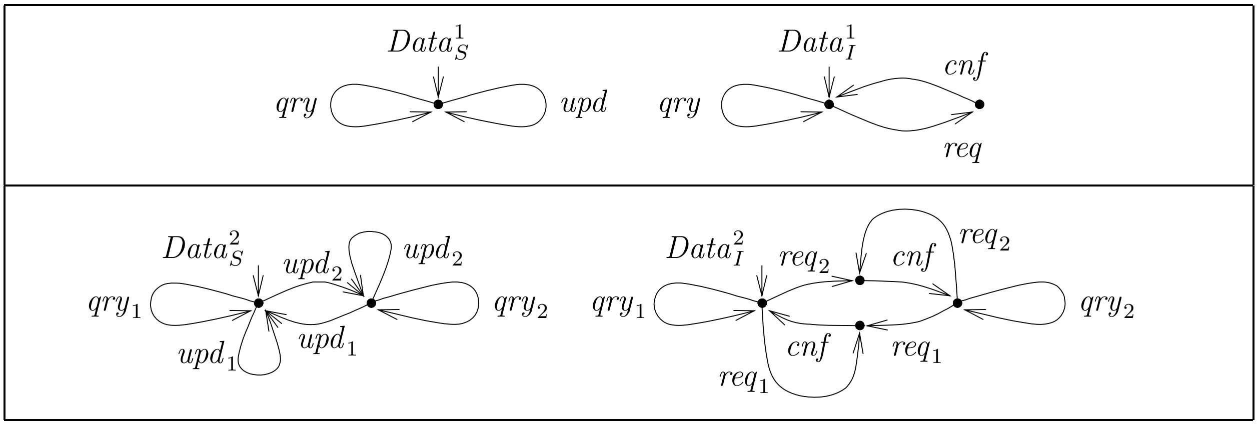


Figure 2: Specification and refinement of a data base with 1 state (above) and 2 states (below)

The operational semantics of $Data_S^1$ is depicted in Figure 2. Now suppose that updating is refined so that it consists of two separate stages, in which the update is *requested* and *confirmed*, respectively. In our setting, this can be expressed by refining action *upd* to *req*; *cnf*, thus obtaining the process $Data_I^1 := Data_S^1[\mathit{upd} \to \mathit{req}; \mathit{cnf}]$, also depicted in Figure 2. Note that in $Data_I^1$, the query operation cannot be performed in between the two stages of the update operation.

If we make this slightly more realistic by taking into consideration that the state of the data base can take different values, say from 1 up to n (with initial state 1), we arrive at the following specification:

$$\begin{array}{rcl} Data_S^n & := & State_1 \\ State_i & := & qry_i; State_i + \sum_{k=1}^n upd_k; State_k \quad (\text{for } i = 1, \ldots, n) \ . \end{array}$$

Hence $Data_S^n$ specifies that after an update action, where a value is written, any number of consecutive queries can be performed, each of which reads the value just written. The behaviour of $Data_S^n$ for $n = 2$ is again depicted in Figure 2. Refining the actions upd_i to $req_i; cnf$ results in

$$Data_I^2 := Data_S^2[upd_1 \to req_1; cnf][upd_2 \to req_2; cnf] \ .$$

Like in the case of $Data_I^1$, between request and confirmation querying the data base is disabled. If it is desired that querying be enabled at that point (for instance because the confirmation action does not change the data base state, so it is safe to read it), this requires a more flexible notion of action refinement; see Sections 7 and 8.

3 Atomic Refinement

The basic idea underlying atomic action refinement is the following. An abstract specification describes a system in terms of executions of basic actions, that —by their nature— are intrinsically atomic; hence, when a specification is made more detailed via action refinement, the atomicity of an abstract action should be preserved by the concrete process implementing that abstract action. The atomic execution of a process means that it enjoys the following two properties:

All-or-nothing: the concrete process is either executed completely, or not at all; this implies that the process is not observable during its execution, but only before and after.

Non-interruptible: no other process can interrupt its execution; this implies that the atomic process is never interleaved with others.

Observe that action refinement in Section 2 is not atomic: the problem described in Example 2.10 simply disappears if we assume that $a_1; a_2$ is executed atomically, as the intermediate state where only a_1 has been performed is not observable. (Hence, as we will see, rooted weak bisimilarity is a congruence for atomic refinement.) There are many real problems where the assumption of atomicity is vital, e.g., when mutual exclusion on a shared resource is necessary to prevent faulty behaviour. For more discussion see also Section 1.3.

In order to understand atomic action refinement, it is useful to enhance the language with a mechanism for making the execution of processes atomic. For this purpose, we add an atomiser construct. The operational model has to be extended accordingly: it is necessary to divide the set of states into the *abstract* (or observable) ones, in which no atomic process is running, and the *concrete* (or unobservable) ones, which correspond to the intermediate states of some atomic process. With these extensions, we can easily accommodate atomic action refinement within interleaving semantics. Papers following this approach include [13, 19, 72, 75, 97]. Moreover, [48] takes an intermediate position where action refinement is non-interruptible but not all-or-nothing; that is, the concrete states are observable. Other papers dealing with some (weaker) form of atomicity in process algebra are [16, 111].

In the above discussion, we have implicitly assumed that it is possible to put processes in parallel; otherwise, the notion of interruption would not be meaningful. To make the assumption valid, we enlarge the process algebra of the previous section with an operator for parallel composition; however, for the sake of simplicity, we ignore communication for now. The problems due to communication are treated extensively in Section 4, and are not essentially different for atomic and non-atomic refinement.

3.1 Parallel composition and atomiser

The language we consider in this section, denoted $\mathit{Lang}^{\mathit{atom}}$, extends $\mathit{Lang}^{\mathit{seq}}$ of the previous section (Page 13) with several new operators (underlined):

$$\mathsf{T} ::= \mathbf{0} \mid \mathbf{1} \mid \alpha \mid \mathsf{V}+\mathsf{V} \mid \mathsf{T};\mathsf{V} \mid \underline{\mathsf{T} \mathbin{|||} \mathsf{T} \mid \langle\mathsf{V}\rangle \mid *\mathsf{T}} \mid \mathsf{T}/A \mid \mathsf{T}[a \rightarrow \mathsf{V}] \mid x \mid \mu x.\mathsf{V}\ .$$

See also Table 16 (Page 85) for a complete overview of the different languages used in this chapter. The definitions of free variables and syntactic substitution (Table 1) and of guardedness (Definition 2.1) extend directly to the new operators. Their intuitive meaning and associated well-formedness conditions are as follows:

- $t \mathbin{|||} u$ is the parallel composition of the behaviours described by t and u, where all the actions can be done by either sub-term in isolation (no communication), except for the termination action $\checkmark$ on which t and u have to synchronise. We impose as a well-formedness condition that either t or u (or both) are *abstract* (where the notion of an abstract state is defined below).
- $\langle t\rangle$ behaves like t, except that the intermediate states of the execution are not observable. In particular, if t has no path ending in a $\checkmark$-transition, $\langle t\rangle$ does not offer any observable behaviour. The operand of $\langle t\rangle$ is *virgin*, and hence may contain no auxiliary operator (either $\mathbf{1}$ or $*$).

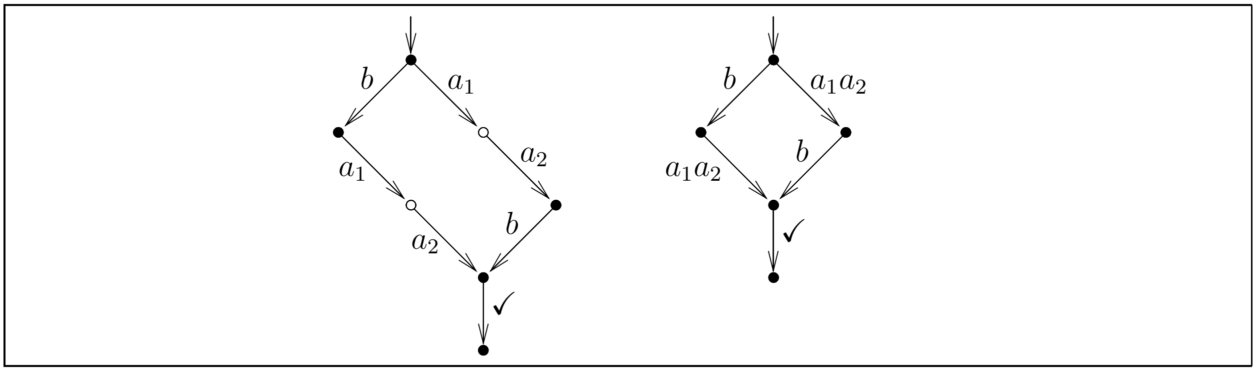


Figure 3: Two representations for $\langle a_1; a_2\rangle \mathbin{|||} b$

- $*t$ is an auxiliary operator expressing the residual of some atomised process; its meaning differs from that of t only in that the former is considered to be a *concrete* term (unless it is terminated), while the latter is an *abstract* term (provided it contains no more occurrences of $*$). Being an auxiliary operator, due to well-formedness $*$ may not occur in any virgin operand.

The operational semantics for $\langle t\rangle$ can be given at two different description levels. For illustration, in Figure 3 we depict two alternative semantics for $\langle a_1; a_2\rangle \mathbin{|||} b$. One possible semantics describes the intermediate states of the execution; this requires to distinguish observable states and unobservable ones (depicted as white circles) in the labelled transition system, as only the former states should be considered when defining behavioural semantics. This line has been followed in, e.g., [75] and also in [76, 72] where an axiomatisation of the low level operator used for the atomiser (called *strong prefixing*) is given. The alternative semantics is more abstract, as the intermediate states are not described in the operational model, at the price of labelling transitions with action sequences. This line has been proposed in, e.g., [72]. Here we follow the former approach in the operational semantics (although the atomiser construct presented here is more general than in the papers cited above) and the second approach in the denotational semantics, where we use trees labelled on sequences.

Abstract terms. In the well-formedness condition imposed above on parallel composition as well as in the operational rules for parallel composition introduced below, we use the concept of an *abstract* term. The set of abstract terms is defined inductively by the following rules:

- $\mathbf{0}$, $\mathbf{1}$, any variable $x \in \mathit{Var}$ and any action $\alpha \in \mathit{Act}_{\checkmark}$ are abstract;
- if $t \xrightarrow{\checkmark}$, then $*t$ is abstract;
- if t and u are abstract, then $t + u$, $t; u$, t/A, $t \mathbin{|||} u$, $\langle t\rangle$, $t[a \rightsquigarrow u]$ and $\mu x.t$ are abstract.

All the terms in Lang^{atom} that are not abstract are called *concrete*. As we will see, if $*t$ is abstract, which is the case if and only if $t \xrightarrow{\checkmark}$, then both $*t$ and t are bisimilar to $\mathbf{1}$. It follows that the abstract states are roughly (namely, up to bisimilarity) given by the $*$-less fragment of Lang^{atom}. On the other hand, concrete states are terms with some non-terminated subterm in the scope of a $*$-operator.

Well-formedness. To summarise the well-formedness conditions on Lang^{atom}:

Table 4: Transition rules.

$$\frac{t \xrightarrow{\alpha} t' \quad u \text{ abstract}}{t \,|||\, u \xrightarrow{\alpha} t' \,|||\, u} \qquad \frac{u \xrightarrow{\alpha} u' \quad t \text{ abstract}}{t \,|||\, u \xrightarrow{\alpha} t \,|||\, u'} \qquad \frac{t \xrightarrow{\checkmark} t' \quad u \xrightarrow{\checkmark} u'}{t \,|||\, u \xrightarrow{\checkmark} t' \,|||\, u'}$$

$$\frac{t \xrightarrow{\alpha} t'}{\langle t \rangle \xrightarrow{\alpha} *t'} \qquad \frac{t \xrightarrow{\lambda} t'}{*t \xrightarrow{\lambda} *t'}$$

$$\frac{t \xrightarrow{\lambda} t' \quad \lambda \neq a}{t[a \rightarrow u] \xrightarrow{\lambda} t'[a \rightarrow u]} \qquad \frac{t \xrightarrow{a} t' \quad u \xrightarrow{\alpha} u'}{t[a \rightarrow u] \xrightarrow{\alpha} (*u'); t'[a \rightarrow u]}$$

- No virgin operand (V in the grammar) contains an auxiliary operator (**1** or $*$);
- At least one of the operands of parallel composition must be abstract;
- Recursion is allowed on guarded variables only.

Operational semantics. The labelled transition system is $\langle Lang^{atom}, \longrightarrow \rangle$, where $\longrightarrow$ is the transition relation defined by the rules in Table 4 (only the rules for the new operators and the two rules for refinement are reported). Let us comment on the rules. The rules for parallel composition give priority to the concrete component, if any: if only one of the two components is abstract, this has to remain idle. Observe that there is no way to reach a state where both components are concrete, starting from an initial abstract state (in fact, such a state would not be well-formed). Moreover, if both components are abstract, the rules allow both to proceed. The rule for the atomiser is simple: $\langle t \rangle$ does what t does, but the reached state is concrete (if not properly terminated). A concrete state $*t$ does what t does; the only difference is that the reached state is still concrete (if not properly terminated). The rules for sequential composition are responsible for the $*$ clean-up, when reaching an abstract state. The rule for refinement shows that the residual u' is not only to be non-interruptible (as in the corresponding rule of the previous section), but that the execution is to be all-or-nothing (by making the intermediate states concrete).

Example 3.1 *Consider $t = \langle a \rangle; b$ and $u = \langle a; b \rangle$. It is easy to see that $t \xrightarrow{a} (*\mathbf{1}); b \xrightarrow{b} \mathbf{1}$ where all the states are abstract, while $u \xrightarrow{a} *(\mathbf{1}; b) \xrightarrow{b} *\mathbf{1}$, where the intermediate state $*(\mathbf{1}; b)$ is concrete. So, $t \,|||\, c$ will allow the execution of c in between a and b, while $u \,|||\, c$ will forbid this behaviour.*

3.2 Denotational Semantics

We present a denotational semantics for the $*$-less fragment of $Lang^{atom}$, using trees (see Section 2.3 for the precise definition of a tree) labelled on $Act_\tau^+ \cup \{\checkmark\}$ as a model. We use w to range over Act_τ^+.

The definitions for the basic operations are the same as in Section 2.3, except that sequential composition and choice have to be adapted to sequences. Here we report only the definitions for the two new operations (parallel composition and the atomiser) and for refinement. (Note that $*$ is not modelled denotationally.)

Parallel. If $N_1 \cap N_2 = \emptyset$, then $\mathcal{T}_1 \,|||\, \mathcal{T}_2 = Unf\langle N, \longrightarrow, \iota \rangle$ where:

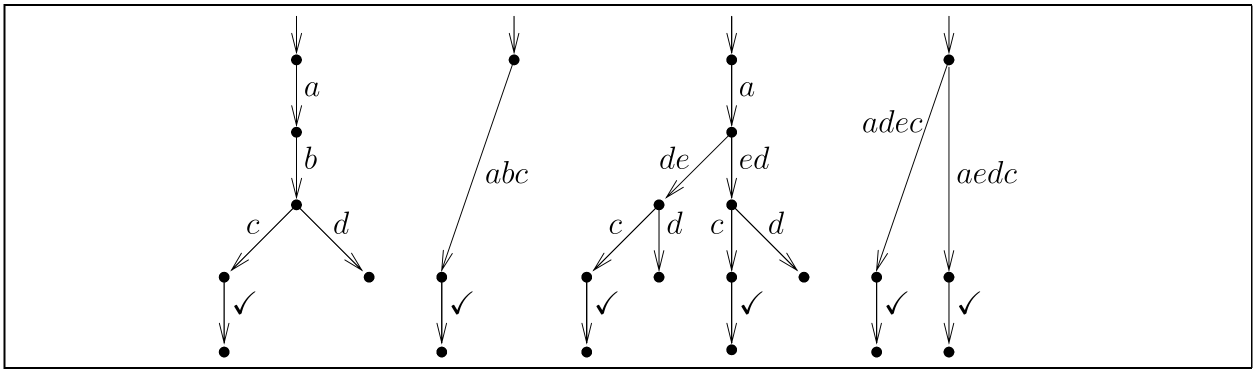


Figure 4: Atomic trees denoting $t = a;b;(c+d;\mathbf{0})$, $\langle t\rangle$, $t[b \to d \mathbin{|||} e]$ and $\langle t\rangle[b \to d \mathbin{|||} e]$.

- $N = N_1 \times N_2$;
- $\longrightarrow\, = \{((n_1,n_2),w,(n'_1,n_2)) \mid n_1 \xrightarrow{w}_1 n'_1\} \cup \{((n_1,n_2),w,(n_1,n'_2)) \mid n_2 \xrightarrow{w}_2 n'_2\}$;
- $\iota = (\iota_1, \iota_2)$.

Atomiser. $\langle \mathcal{T} \rangle = \mathit{Unf}\langle N, \longrightarrow', \iota\rangle$ where:

- $\longrightarrow'\, = \{(\iota, w_1 \cdots w_k, n) \mid k \geq 1, \iota \xrightarrow{w_1} \cdots \xrightarrow{w_k} n \xrightarrow{\checkmark}\,\} \cup \{(n, \checkmark, n') \mid n \xrightarrow{\checkmark} n'\}$.

Action Refinement. If $\iota_2 \not\xrightarrow{\checkmark}_2$, then $\mathcal{T}_1[a \to \mathcal{T}_2] = \mathit{Unf}\langle N_1, \longrightarrow, \iota_1\rangle$ where:

- $\longrightarrow\, = \{(n, w', n') \mid n \xrightarrow{w}_1 n', w' \in R(w)\} \cup \{(n, \checkmark, n') \mid n \xrightarrow{\checkmark} n'\}$

in which $R(w)$ is the set of expansions of w, defined as follows:

$$\begin{array}{rrcl} R\colon & \varepsilon & \mapsto & \{\varepsilon\} \\ & \alpha w & \mapsto & \{\alpha\} \cdot R(w) \quad \text{if } a \neq \alpha \\ & a w & \mapsto & \{w' \mid \iota_2 \xrightarrow{w'\checkmark}_2\,\} \cdot R(w) \end{array}$$

and $\cdot$ is the concatenation operation on sets of strings.

The operation of parallel composition is nothing but the unfolding of the lts obtained by making the (asynchronous) product of the two trees. The operation of atomising a tree yields a tree that is of depth two (counting ✓). This indeed ensures that the resulting behaviour is atomic, as it is performed in one single step (followed by ✓). Moreover, all the paths in $\mathcal{T}$ that do not reach a terminated node are simply omitted; if all paths are so, then $\langle \mathcal{T} \rangle$ is isomorphic to $\mathcal{T}_\perp$ (the tree modelling the deadlock constant $\mathbf{0}$). Finally, the refinement operation is a sort of relabelling: an arc labelled w is replaced by possibly many arcs, connecting the same two end nodes, each one with a label obtained by macro-expansion of w where every occurrence of a in w is replaced by one of the possible terminated sequences of $\mathcal{T}$ (i.e., the steps of $\langle \mathcal{T} \rangle$); subsequently, the resulting transition system is unfolded. The denotational semantics for the $*$-less fragment of $\mathit{Lang}^{\mathit{atom}}$ is obtained as a direct extension of Table 3 to the new operators. An example is given in Figure 4.

3.3 Congruences and axiomatisations

As done in the previous section, strong bisimulation equivalence is used to compare the two semantics. However, the operational semantics is quite different, as states may be also concrete. So,

Table 5: Some typical axioms for $\langle \_ \rangle$, sound modulo $\sim$ (above the line) and $\simeq_{\mathsf{w}}$ (all).

$\langle t; \mathbf{0} \rangle$	$=$	$\mathbf{0}$
$\langle \alpha \rangle$	$=$	α
$\langle \langle t_1 \rangle ; \langle t_2 \rangle \rangle$	$=$	$\langle t_1 ; t_2 \rangle$
$\langle t_1 ; (t_2 + t_3) \rangle$	$=$	$\langle t_1 ; t_2 \rangle + \langle t_1 ; t_3 \rangle$
$\langle \tau ; t \rangle$	$=$	$\langle t \rangle$

we first need to define which long steps are to be used in the definition of bisimulation for the operational semantics.

To this aim, from the transition relation defined operationally over Lang^{atom} (Tables 2 and 4), we derive a "long step" transition relation over the *abstract* terms in Lang^{atom}, denoted $\longrightarrow$ as well but labelled by $\mathit{Act}_\tau^+ \cup \{\checkmark\}$. Long step transitions relate a pair of abstract states as follows: $t \xrightarrow{w} t'$ iff $t \xrightarrow{\alpha_1} t_1 \xrightarrow{\alpha_2} t_2 \cdots \xrightarrow{\alpha_k} t'$, where all t_i are concrete ($i = 1, \ldots, k-1$) and $w = \alpha_1 \cdots \alpha_k$. Bisimulation equivalence is thus defined by setting $\mathit{Lab} = \mathit{Act}_\tau^+ \cup \{\checkmark\}$ in Definition 2.6; it relates only those abstract states that are able to match their long steps. Under this interpretation, the following holds:

Proposition 3.2

1. *$t \sim [\![t]\!]$ for all $*$-less $t \in \mathit{Lang}^{atom}$;*
2. *$\sim$ is a congruence over the $*$-less fragment of Lang^{atom}.*

In order to prove the first clause, for each long step of the operational semantics one finds a corresponding single step in the denotational semantics, and vice versa; the correspondence is set by an inductive argument on the term structure (taking into account the guardedness of recursive terms), much like for Proposition 2.8.

As strong bisimilarity is a congruence, it is natural to look for axioms for the atomiser and refinement operators. Some typical axioms for the atomiser are reported in Table 5; the axiomatic treatment of refinement is deferred to Section 6. Another axiom system for a language with atomiser is presented in [97]; since the assumptions there are quite different, it is no surprise that also the axioms are quite different.

Passing to τ-insensitive equivalence, we need to abstract the observable behaviour of a one-step sequence w, by removing all the occurrences of τ in it. Let $w \setminus \tau$ denote the resulting sequence; e.g., $a\tau b \setminus \tau = ab \setminus \tau = ab\tau \setminus \tau = ab$ and $\tau\tau\tau \setminus \tau = \varepsilon$. Then we define the relation $\Longrightarrow$ as follows:

$$t \Rightarrow u \quad :\Leftrightarrow \quad t \xrightarrow{w_1} t_1 \cdots \xrightarrow{w_n} u$$

where $w_i \setminus \tau = \varepsilon$ for $1 \leq i \leq n$. Rooted weak bisimilarity (see Definition 2.9), in our setting of transition labelled on sequences, is defined modulo the removal of τs; that is, u has to simulate a transition $t \xrightarrow{w} t'$ in one of the following ways:

- either $w \setminus \tau = \varepsilon$ and (t', u) is in the relation;
- or $u \Longrightarrow\xrightarrow{w'}\Longrightarrow u'$ with $w' \setminus \tau = w \setminus \tau$ and (t', u') is in the relation.

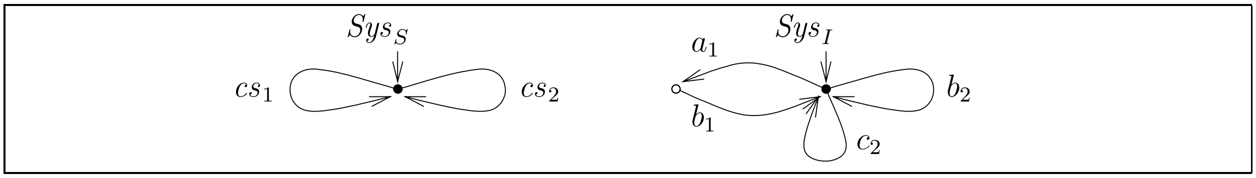


Figure 5: Critical sections and their refinement: $Sys_I = Sys_S[cs_1 \rightarrow a_1; b_1][cs_2 \rightarrow b_2 + c_2]$

We observe that $\simeq_{\mathsf{w}}$ is a congruence for atomic refinement. In fact, the counterexample reported in Example 2.10 simply disappears if refinement is atomic.

Example 3.3 *Consider again* $t = a; (b + \tau)$ *and* $t' = a; (b + \tau) + a$ *such that* $t \simeq_{\mathsf{w}} t'$. *According to the rules for atomic refinement and rooted weak bisimilarity,* $t[a \rightarrow a_1; a_2] \simeq_{\mathsf{w}} \langle a_1; a_2 \rangle; (b + \tau) \simeq_{\mathsf{w}} \langle a_1; a_2 \rangle; (b + \tau) + \langle a_1; a_2 \rangle \simeq_{\mathsf{w}} t'[a \rightarrow a_1; a_2]$.

Summarising, we have the following result:

Proposition 3.4 $\simeq_{\mathsf{w}}$ *is a congruence over the* $*$*-less fragment of* $Lang^{atom}$.

Table 5 contains a typical axiom for $Lang^{atom}$ up to $\simeq_{\mathsf{w}}$.

3.4 Application: Critical sections

Consider two processes repeatedly entering a critical section. Abstractly, the activities in the critical section are modelled as a single action, cs_i for $i = 1, 2$, and the system is specified by $Sys_S = Proc_1 \mid\mid\mid Proc_2$ where the processes $Proc_i$ are given by

$$Proc_i := cs_i; Proc_i \ .$$

Now suppose cs_1 is refined into $a_1; b_1$ and cs_2 into $b_2 + c_2$; that is, the abstract system is refined into $Sys_I = Sys_S[cs_1 \rightarrow a_1; b_1][cs_2 \rightarrow b_2 + c_2]$. Due to the nature of atomic refinement, the refined critical sections do not overlap: in particular, b_2 and c_2 cannot occur in between a_1 and b_1. The behaviour of the abstract and refined systems are given in Figure 5.

Unfortunately, the atomicity of the refinement can also affect actions for which it was not intended. This can be seen by considering extensions of the above processes that also have a non-critical section, ncs_i:

$$Proc'_i := ncs_i; cs_i; Proc_i \ .$$

If we refine the cs_i-actions in $Proc'_1 \mid\mid\mid Proc'_2$ in the same way as above, then it turns out that not only the critical section (cs_2) but also the non-critical section (ncs_2) of $Proc'_2$ is prevented from proceeding during the critical section of $Proc'_1$ (between a_1 and b_1). This may not be the intended behaviour. However, in order to avoid it, more information regarding the dependencies between actions is needed than the current framework provides: in particular, it must be made clear that, in contrast to cs_1 and cs_2, ncs_1 and cs_2 are independent and may safely overlap. In Section 7.2 we show how one can use action dependencies to solve this type of problem.

4 Non-atomic refinement: An event-based model

In this and the next section, we consider refinement in a fully general language, comparable with CCS, CSP or ACP. Compared to Section 3, we extend parallel composition with synchronisation,

Table 6: Transition rules for synchronisation.

$$\frac{t \xrightarrow{\alpha} t' \quad \alpha \notin A}{t \parallel_A u \xrightarrow{\lambda} t' \parallel_A u} \qquad \frac{u \xrightarrow{\alpha} u' \quad \alpha \notin A}{t \parallel_A u \xrightarrow{\lambda} t \parallel_A u'} \qquad \frac{t \xrightarrow{\lambda} t' \quad u \xrightarrow{\lambda} u' \quad \lambda \in A_{\checkmark}}{t \parallel_A u \xrightarrow{\lambda} t' \parallel_A u'}$$

resulting in terms $t \parallel_A u$ where $A \subseteq \mathit{Act}$ is the *synchronisation set*; $t \mathbin{|||} u$ is a special case where $A = \emptyset$. On the other hand, we drop the atomiser construct. The resulting language, *Lang*, is thus generated by the following grammar (where the new operator w.r.t. Lang^{seq} of Section 2 is underlined):

$$\mathsf{T} ::= \mathbf{0} \mid \mathbf{1} \mid \alpha \mid \mathsf{V} + \mathsf{V} \mid \mathsf{T};\mathsf{V} \mid \underline{\mathsf{T} \parallel_A \mathsf{T}} \mid \mathsf{T}/A \mid \mathsf{T}[a \to \mathsf{V}] \mid x \mid \mu x.\mathsf{V} \ .$$

See also Table 16 (Page 85) for a complete overview of the different languages used in this chapter. As before, well-formedness implies that the auxiliary operator $\mathbf{1}$ does not occur in virgin operands (V in the grammar) and that recursion is guarded. A more extensive discussion of *Lang* follows below. In this section, we develop an event-based operational and denotational semantics for *Lang*; in the next section, we review the relevant congruence questions studied over the last decade. Note that, except for the refinement operator, *Lang* essentially corresponds to a fragment of ACP [11]; in particular, the parallel composition (which is based on TCSP, see [113]) has a standard operational semantics given in Table 6.

It is clear from the discussion in the introduction that the standard lts semantics is not discriminating enough to provide a compositional model for non-atomic refinement in the general setting of *Lang*. To amend this, we add information in the form of *events*. Events serve to give a closer identification of which occurrence of an action is being executed; as a consequence, the same occurrence can be recognised among different transitions or different runs.

For the purpose of formalisation, we assume the existence of a global universe of events, *Evt*, with $* \notin \mathit{Evt}$ ($*$ being a special event used to model synchronisation and refinement of event structures), which is closed under pairing, also with $*$; i.e., $(\mathit{Evt}_* \times \mathit{Evt}_*) \setminus (*,*) \subseteq \mathit{Evt}$ where $\mathit{Evt}_* = \mathit{Evt} \cup \{*\}$. The set of *singular* (i.e., *non-paired*) events is denoted $\mathit{Evt}^\bullet = \mathit{Evt} \setminus (\mathit{Evt}_* \times \mathit{Evt}_*)$; this set is assumed to be countably infinite. Evt_* is ranged over by d, e; subsets of Evt_* are denoted E, F, G.

In the operational semantics, events are used to enrich the transition labels: these are now taken from $\mathit{Act}^{\mathit{Evt}}_{\tau\checkmark} = (\mathit{Evt} \times \mathit{Act}_\tau) \cup \{\checkmark\}$ rather than just $\mathit{Act}_{\tau\checkmark}$; i.e., the non-termination transitions have associated event identities. This allows us to distinguish the independent execution of actions from their interleaving. Denotationally, we capture the behaviour of processes through *event structures.*

4.1 Event annotations and operational semantics

An immediate technical question is how event identities are generated and kept distinct for different occurrences within a given model, given the fact that the terms of *Lang* do not contain any information to this purpose. There are several possible answers:

- By interpreting models up to isomorphism, and implicitly or explicitly selecting isomorphic representatives to guarantee distinctness of events; see, e.g., Winskel [140].

- By generating event identities automatically from the term structure, as in [47, 21]; see also [78, 23] for an application of the same principle in a timed semantics for process algebras.

- By adding information about event identities explicitly to the terms, through their *annotation* at "compile time", i.e., before evaluating them; see Langerak [100].

The first method is less suitable for operational semantics, since there not all event identities are known beforehand; hence we do not follow this approach. For event-based semantics, the third method is mathematically less sophisticated but gives rise to an (in our opinion) simpler presentation. For that reason, instead of studying *Lang* directly, we consider the annotated language *Lang*(*Evt*) generated by the following grammar (new or adapted operators are underlined):

$$t ::= \mathbf{0} \mid \mathbf{1} \mid \underline{{}_e\alpha} \mid \mathsf{V}+\mathsf{V} \mid \mathsf{T};\mathsf{V} \mid \mathsf{T} \parallel_A \mathsf{T} \mid \mathsf{T}/A \mid \underline{\mathsf{T}[a{\to}\mathsf{V}, \vec{e}{\to}\vec{\mathsf{T}}] \mid {}_e x} \mid \underline{{}_e[\mathsf{T}]} \mid \mu x.\mathsf{V}$$

where $e \in \mathit{Evt}^\bullet$. As an additional well-formedness condition, we require that the event annotations are compatible. To make this precise, we introduce a set $E_t \subseteq \mathit{Evt}^\bullet$ of events syntactically occurring within t, which is only defined if the annotations are compatible.

- $\mathbf{0}$ and $\mathbf{1}$ are the deadlock, resp. termination constants as before. We define $E_\mathbf{0} = E_\mathbf{1} = \emptyset$.
- ${}_e\alpha$ denotes the action $\alpha \in \mathit{Act}_\tau$ annotated with the event $e \in \mathit{Evt}^\bullet$ used to model its occurrence. We define $E_{{}_e\alpha} = \{e\}$.
- $t + u$ denotes choice; we require $E_t \cap E_u = \emptyset$ and define $E_{t+u} = E_t \cup E_u$.
- $t; u$ denotes sequential composition; we require $E_t \cap E_u = \emptyset$ and define $E_{t;u} = E_t \cup E_u$.
- $t \parallel_A u$ denotes the parallel composition of t and u with synchronisation over A, meaning that actions in A (and also the termination action $\checkmark$) may only be executed by t and u simultaneously, and all others independently by either operand. We use $t \parallel\!| u = t \parallel_\emptyset u$ as a special case. As for the previous binary operators, we require $E_t \cap E_u = \emptyset$ and define $E_{t\parallel_A u} = E_t \cup E_u$.
- t/A denotes hiding; we define $E_{t/A} = E_t$.
- $t[a{\to}u, \vec{e}{\to}\vec{v}]$ denotes the refinement of the actions a occurring in t by u; the latter may not contain the termination constant $\mathbf{1}$. Moreover, $\vec{e}{\to}\vec{v}$ denotes a vector of *pending refinements* $e_1{\to}v_1, \ldots, e_n{\to}v_n$ (with $n = |\vec{e}| = |\vec{v}|$), where the e_i identify distinct occurrences of a in t (i.e., $e_i = e_j$ implies $i = j$) that are currently being executed, or actually refined; the v_i represent the corresponding non-terminated states reached during the execution of the refinement body u. We require $E_t \cap E_u = \emptyset$ and $E_{v_i} \subseteq E_u$ for all $1 \leq i \leq n$, and define $E_{t[a{\to}u,\vec{e}{\to}\vec{v}]} = E_t \cup E_u$.

  In the sequel, we write $\{\vec{e}\} = \{e_1, \ldots, e_n\}$ for the set of events in $\vec{e}$.
- ${}_e x$ denotes the process variable $x \in \mathit{Var}$, annotated with an event $e \in \mathit{Evt}^\bullet$. This annotation is used in order to prevent the re-occurrence of event identities in recursive unfoldings. We define $E_{{}_e x} = \{e\}$.
- ${}_e[t]$, with $e \in \mathit{Evt}^\bullet$, models the unfolding of the process invocation ${}_e x$ within the body of a recursive definition $\mu x.t$; its effect is the "relocation" of the events in t to the fresh range $\{e\} \times \mathit{Evt}$. We require t to be well-formed (in particular, E_t should be defined) and define $E_{{}_e[t]} = \{e\}$. The notion of syntactic substitution is adapted accordingly; see below.
- $\mu x.t$ denotes the recursive binding of the (guarded) process variable x in t. We let $E_{\mu x.t} = E_t$.

The definition of the free variables and substitution is extended to *Lang*(*Evt*) (see Table 1); the only interesting new case is ${}_e x$, for which $\mathit{fv}({}_e x) = \{x\}$ and ${}_e x\{t/x\} = {}_e[t]$. The latter preserves the event annotation of process variables, and shows the reason why we need terms of the form ${}_e[t]$ at all. For instance, the first three approximations of the annotated term $\mu x.{}_0 a; {}_1 x$ (according to the definition in Section 2.3) are given by $\mathbf{0}$, ${}_0 a; {}_1[\mathbf{0}]$ and ${}_0 a; {}_1[{}_0 a; {}_1[\mathbf{0}]]$.

Well-formedness. To summarise the well-formedness conditions on $\mathit{Lang}(\mathit{Evt})$:

- No virgin operand (V in the grammar) contains an occurrence of an auxiliary operator ($\mathbf{1}$);
- Recursion is allowed on guarded variables only;
- The set E_t of event annotations is defined.

We then have the following property (which is straightforward to prove):

Proposition 4.1 *For all well-formed $t, u \in \mathit{Lang}(\mathit{Evt})$, $t\{u/x\}$ is well-formed and $E_{t\{u/x\}} = E_t$.*

Annotating and stripping. Annotated terms are no more than an auxiliary device to give an event-based semantics for *Lang*. To make the connection between *Lang* and $\mathit{Lang}(\mathit{Evt})$ more explicit, consider a partial function $\mathit{strip}\colon \mathit{Lang}(\mathit{Evt}) \to \mathit{Lang}$ that is undefined on terms ${}_e[t]$ and on $t[a \to u, \vec{e} \to \vec{v}]$ for nonempty $\vec{e}$, and otherwise removes all event annotations. It should be clear that *strip* is surjective, meaning that there is an annotated term for any $t \in \mathit{Lang}$; however, it is far from injective, since there are many ways to annotate t in a well-formed manner. For instance, one particularly simple method is to assume $\mathbb{N} \subseteq \mathit{Evt}^\bullet$ and consecutively number the subterms to be annotated, from left to right; e.g., $(a; b \parallel_a c; a)[a \to \mu x.d; x]$ is annotated according to this method as $({}_0a; {}_1b \parallel_a {}_2c; {}_3a)[a \to \mu x.{}_4d; {}_5x]$. When defining the semantics, we will make sure that the chosen annotation of a term makes no difference; that is, if $\mathit{strip}(t) = \mathit{strip}(u)$ for $t, u \in \mathit{Lang}(\mathit{Evt})$ then t and u will be equivalent.

4.2 Operational event-based semantics

A major advantage of annotated terms is that their operational semantics is a straightforward extension of the standard operational semantics (given in Tables 2 and 4), except for the refinement operator, which now has to be treated in full generality. The presentation below is based on [121].

Event transition systems. As stated above, in order to model the behaviour of $\mathit{Lang}(\mathit{Evt})$ we take transition labels from $\mathit{Act}^{\mathit{Evt}}_{\tau\checkmark}$ $(= (\mathit{Evt} \times \mathit{Act}_\tau) \cup \{\checkmark\})$ rather than $\mathit{Act}_{\tau\checkmark}$. The idea is that the event identifier uniquely identifies the action occurrence; in other words, on the level of events the transition system is deterministic. Furthermore, if two events can be executed in either order (in a given state), we call them *independent* (in that state); the state reached is independent of the ordering, and moreover, any set of pairwise independent events will remain independent if one of them is executed. Formally:

Definition 4.2 *An event transition system is an $\mathit{Act}^{\mathit{Evt}}_{\tau\checkmark}$-labelled transition system such that for all $n \in N$*

- *If $n \xrightarrow{e,\alpha_1} n'_1$ and $n \xrightarrow{e,\alpha_2} n'_2$, then $\alpha_1 = \alpha_2$ and $n'_1 = n'_2$.*
- *e_1 and e_2 are called n-*independent *if $n \xrightarrow{e_1,\alpha_1}\xrightarrow{e_2,\alpha_2} n'$ and $n \xrightarrow{e_2,\alpha_2}\xrightarrow{e_1,\alpha_1} n''$; if e_1 and e_2 are n-independent, then $n' = n''$.*
- *If e_1, e_2, e_3 are pairwise n-independent and $n \xrightarrow{e_1,\alpha_1} n'$, then e_2 and e_3 are n'-independent.*

It follows that if a set of events F is pairwise n-independent for some node n, then starting in n, the events in F can be executed in arbitrary order, and the state reached is independent of the order of execution. We use $n \xrightarrow{F}$ to denote that F is pairwise n-independent. For instance, we will see that ${}_0a \,|||\, {}_1b \xrightarrow{\{(0,*),(*,1)\}}$, signalling the independence of a and b, but $t = {}_0a;{}_1b + {}_2b;{}_3a \not\xrightarrow{\{0,2\}}$ although $t \xrightarrow{0,a}$ and $t \xrightarrow{2,b}$, since here the actions a and b are not independent.

To compare the behaviour of event transition systems, we define an extension of strong bisimulation which allows events to be converted bijectively between the behaviours under comparison, in order to abstract away sufficiently from the precise occurrence identifiers.

Definition 4.3 *Let $\langle Act^{Evt}_{\tau\surd}, N, \to\rangle$ be an event transition system.*

- *For any bijective $\phi\colon Evt \to Evt$, a ϕ-simulation is a relation $\rho \subseteq N \times N$ such that for all $n_1 \,\rho\, n_2$:*
  - *if $n_1 \xrightarrow{e,\alpha} n_1'$ then $n_2 \xrightarrow{\phi(e),\alpha} n_2'$ with $n_1' \,\rho\, n_2'$.*
  - *if $n_1 \xrightarrow{\surd} n_1'$ then $n_2 \xrightarrow{\surd} n_2'$ with $n_1' \,\rho\, n_2'$.*
- *n_1 and n_2 are called* event bisimilar*, denoted $n_1 \sim_{\mathsf{ev}} n_2$, if there is a ϕ-simulation ρ (for some ϕ) such that ρ^{-1} is a ϕ^{-1}-simulation.*

Note that, in contrast to more sophisticated relations such as history-preserving bisimilarity (cf. [60]), in the above definition there is a single mapping ϕ establishing a static, one-to-one relation between the events of the transition systems under comparison. This makes for a very discriminating equivalence: for instance, for event transition systems generated by stable event structures, event bisimilarity coincides with isomorphism of the event structures' configurations (see the discussion after Proposition 4.15). Indeed, we will have ${}_1a + {}_2a \not\sim_{\mathsf{ev}} {}_3a$, even though both ${}_1a \sim_{\mathsf{ev}} {}_3a$ and ${}_2a \sim_{\mathsf{ev}} {}_3a$.

The event-based operational semantics of *Lang*(Evt) is given in Table 7. For the fragment without refinement (the part of the table above the line), the semantics is unoriginal; see [99] for a very similar set of rules.

Extended refinement functions. The definition of the operational semantics of refinement requires some care. During the execution of a refined action, the *remainder* of the refinement must be "stored" somewhere in the term. For that purpose, we have extended refinements from terms $t[a{\to}u]$ to terms $t[a{\to}u, e_1{\to}v_1, \ldots, e_k{\to}v_k]$ for arbitrary finite k; for all $1 \leq i \leq k$, e_i is an event of t under refinement, and v_i is the current remainder. The events in $\{\vec{e}\}$ are called *busy.* The remainder is kept only as long as it is not terminated; furthermore, during the same period, the event e_i is not actually executed, i.e., it is kept in the term t. The latter is necessary to make sure that events that causally follow e_i are not executed prematurely, before the refinement of e_i is terminated.

Note that the e_i may very well resolve choices within t. Since we do not let t execute the e_i as long as the corresponding v_i is not terminated, there is the danger that an event that actually conflicts with one of the busy events, say e_i, is executed as well, despite the fact that the choice has actually been resolved in favour of e_i. In order to prevent this from happening, only events of t that are independent of all the e_i may occur.

Some notation: for all $1 \leq i \leq k$, $(\vec{e}{\to}\vec{v}) \pm (e_i{\to}v')$ denotes the replacement of the residual $e_i{\to}v_i$ by $e_i{\to}v'$, and $(\vec{e}{\to}\vec{v}) \setminus e_i$ denotes the removal of the residual $e_i{\to}v_i$ from the vector.

The rules in Table 7 (below the line) can be understood as follows:

- The first rule concerns the execution of a non-refined action; the premise $t' \xrightarrow{\{\vec{e}\}}$ in this rule and others makes sure that the event d is not in conflict with any of events e_i currently under refinement. The event identity is changed from d to $(d, *)$, for uniformity with events that are properly refined.

Table 7: Operational event-based semantics; $\lambda \in Act^{Evt}_{\tau\checkmark}$ arbitrary

$$\frac{}{\mathbf{1} \xrightarrow{\checkmark} \mathbf{0}} \qquad \frac{}{{}_e\alpha \xrightarrow{e,\alpha} \mathbf{1}} \qquad \frac{t \xrightarrow{\alpha} t'}{t + u \xrightarrow{\alpha} t'} \qquad \frac{u \xrightarrow{\alpha} u'}{t + u \xrightarrow{\alpha} u'} \qquad \frac{t \xrightarrow{\lambda} t' \quad \lambda \neq \checkmark}{t;u \xrightarrow{\lambda} t';u} \qquad \frac{t \xrightarrow{\checkmark} t' \quad u \xrightarrow{\alpha} u'}{t;u \xrightarrow{\alpha} u'}$$

$$\frac{t \xrightarrow{e,\alpha} t' \quad \alpha \notin A}{t \parallel_A u \xrightarrow{(e,*),\alpha} t' \parallel_A u} \qquad \frac{u \xrightarrow{e,\alpha} u' \quad \alpha \notin A}{t \parallel_A u \xrightarrow{(*,e),\alpha} t \parallel_A u'} \qquad \frac{t \xrightarrow{d,\alpha} t' \quad u \xrightarrow{e,\alpha} u' \quad \alpha \in A}{t \parallel_A u \xrightarrow{(d,e),\alpha} t' \parallel_A u'}$$

$$\frac{t \xrightarrow{\checkmark} t' \quad u \xrightarrow{\checkmark} u'}{t \parallel_A u \xrightarrow{\checkmark} t' \parallel_A u'} \qquad \frac{t \xrightarrow{\lambda} t' \quad \lambda \notin Evt \times A}{t/A \xrightarrow{\lambda} t'/A} \qquad \frac{t \xrightarrow{e,a} t' \quad a \in A}{t/A \xrightarrow{e,\tau} t'/A}$$

$$\frac{t \xrightarrow{d,\alpha} t'}{{}_e[t] \xrightarrow{(e,d),\alpha} {}_e[t']} \qquad \frac{t \xrightarrow{\checkmark} t'}{{}_e[t] \xrightarrow{\checkmark} {}_e[t']} \qquad \frac{t\{\mu x.t/x\} \xrightarrow{\alpha} t'}{\mu x.t \xrightarrow{\alpha} t'}$$

$$\frac{t \xrightarrow{d,\alpha} t' \xrightarrow{\{\vec{e}\}} \quad \alpha \neq a}{t[a{\to}u,\, \vec{e}{\to}\vec{v}] \xrightarrow{(d,*),\alpha} t'[a{\to}u,\, \vec{e}{\to}\vec{v}]} \qquad \frac{t \xrightarrow{\checkmark} t'}{t[a{\to}u] \xrightarrow{\checkmark} t'[a{\to}u]}$$

$$\frac{t \xrightarrow{d,a} t' \xrightarrow{\{\vec{e}\}} \quad u \xrightarrow{e',\alpha} u' \not\xrightarrow{\checkmark}}{t[a{\to}u,\, \vec{e}{\to}\vec{v}] \xrightarrow{(d,e'),\alpha} t[a{\to}u,\, \vec{e}{\to}\vec{v},\, d{\to}u']} \qquad \frac{t \xrightarrow{d,a} t' \xrightarrow{\{\vec{e}\}} \quad u \xrightarrow{e',\alpha} u' \xrightarrow{\checkmark}}{t[a{\to}u,\, \vec{e}{\to}\vec{v}] \xrightarrow{(d,e'),\alpha} t'[a{\to}u,\, \vec{e}{\to}\vec{v}]}$$

$$\frac{v_i \xrightarrow{e',\alpha} v' \not\xrightarrow{\checkmark}}{t[a{\to}u,\, \vec{e}{\to}\vec{v}] \xrightarrow{(e_i,e'),\alpha} t[a{\to}u,\, (\vec{e}{\to}\vec{v}) \pm (e_i{\to}v')]} \qquad \frac{t \xrightarrow{e_i,a} t' \quad v_i \xrightarrow{e',\alpha} v' \xrightarrow{\checkmark}}{t[a{\to}u,\, \vec{e}{\to}\vec{v}] \xrightarrow{(e_i,e'),\alpha} t'[a{\to}u,\, (\vec{e}{\to}\vec{v}) \setminus e_i]}$$

- The second rule concerns termination; note that there can be no remaining refinements.
- The third and fourth rules concern the start of a new refinement instance; the latter deals with the case where the refinement immediately terminates again (i.e., it consisted of a single action only).
- The last two rules deal with the continuation of a busy event, which either remains busy (if the remainder is still not terminated) or disappears from the scene (if the remainder is terminated).

Example 4.4 *Consider the derivable transitions of the term $t = ({}_0b; {}_1a \parallel_a ({}_2c + {}_3a))[a{\to}{}_4d; {}_5d]$.*

- *The parallel composition gives rise to*

$$\begin{tikzcd}
{}_0b; {}_1a \parallel_a ({}_2c + {}_3a) \arrow[r, "{(0,*),b}"] \arrow[d, "{(*,2),c}"'] & \mathbf{1}; {}_1a \parallel_a ({}_2c + {}_3a) \arrow[r, "{(1,3),a}"] \arrow[d, "{(*,2),c}"'] & \mathbf{1} \parallel_a \mathbf{1} \arrow[r, "\checkmark"] & \mathbf{0} \parallel_a \mathbf{0} \\
{}_0b; {}_1a \parallel_a \mathbf{1} \arrow[r, "{(0,*),b}"] & \mathbf{1}; {}_1a \parallel_a \mathbf{1} & &
\end{tikzcd}$$

- *Taking the refinement into account, we get*

$$\begin{tikzcd}
({}_0b;{}_1a \parallel_a ({}_2c+{}_3a))[a\to{}_4d;{}_5d] \arrow[r, "((0,*),*),b"] \arrow[d, "((*,2),*),c"'] & (\mathbf{1};{}_1a \parallel_a ({}_2c+{}_3a))[a\to{}_4d;{}_5d] \arrow[r, "((1,3),4),d"] \arrow[d, "((*,2),*),c"'] & \cdots \\
({}_0b;{}_1a \parallel_a \mathbf{1})[a\to{}_4d;{}_5d] \arrow[r, "((0,*),*),b"] & (\mathbf{1};{}_1a \parallel_a \mathbf{1})[a\to{}_4d;{}_5d] &
\end{tikzcd}$$

where the upper right hand transition leads to

$$\begin{aligned}
(\mathbf{1};{}_1a \parallel_a ({}_2c+{}_3a))[a\to{}_4d;{}_5d] &\xrightarrow{((1,3),4),d} (\mathbf{1};{}_1a \parallel_a ({}_2c+{}_3a))[a\to{}_4d;{}_5d,(1,3)\to\mathbf{1};{}_5d] \\
&\xrightarrow{((1,3),5),d} (\mathbf{1} \parallel_a \mathbf{1})[a\to{}_4d;{}_5d] \xrightarrow{\checkmark} (\mathbf{0} \parallel_a \mathbf{0})[a\to{}_4d;{}_5d]
\end{aligned}$$

In particular, note that $(\mathbf{1};{}_1a \parallel_a ({}_2c+{}_3a))[a\to{}_4d;{}_5d,(1,3)\to\mathbf{1};{}_5d] \not\xrightarrow{((*,2),*),c}$.

The first property to be proved is that the operational semantics indeed gives rise to an event transition system as defined in Definition 4.2. Furthermore, it is important to show that the associated equivalence (in this case, event bisimilarity) is a congruence. The proof is a straightforward variation on the standard case (modified by the fact that event identities may be converted).

Proposition 4.5 $\sim_{\mathsf{ev}}$ *is a congruence over Lang*(Evt)*.*

Moreover, it can be shown that all annotations of a given term in *Lang* are event bisimilar; i.e., the following holds (proved by induction on term structure):

Proposition 4.6 *For all* $t,u \in \mathit{Lang}(\mathit{Evt})$, *if* $\mathit{strip}(t) = \mathit{strip}(u)$ *then* $t \sim_{\mathsf{ev}} u$.

4.3 Stable event structures

The denotational semantics for *Lang*(*Evt*) we present here is based on the *stable event structures* of Winskel [140], extended with a termination predicate as in [64].

Definition 4.7 *A stable event structure is a tuple* $\mathcal{E} = \langle E, \#, \vdash, \mathit{Ter}, \ell\rangle$, *where*

- $E \subseteq \mathit{Evt}_*$ *is a set of* events*;*
- $\# \subseteq E \times E$ *is an irreflexive and symmetric* conflict relation*. The reflexive closure of* $\#$ *is denoted* $\#^=$*. The set of finite, consistent (i.e., conflict-free) subsets of* E *will be denoted* $\mathit{Con} = \{F \subseteq_{\mathit{fin}} E \mid \nexists d,e \in F\colon d \# e\}$.
- $\vdash \subseteq \mathit{Con} \times E$ *is an* enabling relation*, which satisfies*

  Saturation: *If* $F \vdash e$ *and* $F \subseteq G \in \mathit{Con}$, *then* $G \vdash e$*;*

  Stability: *If* $F \vdash e$, $G \vdash e$ *and* $F \cup G \cup \{e\} \in \mathit{Con}$, *then* $F \cap G \vdash e$.

  The set of initial events *will be denoted* $\mathit{Ini} = \{e \in E \mid \emptyset \vdash e\}$.
- $\mathit{Ter} \subseteq \mathit{Con}$ *is a* termination predicate *on sets of events, which satisfies*

  Completeness: *If* $F \in \mathit{Ter}$ *and* $F \subseteq G \in \mathit{Con}$, *then* $F = G$.
- $\ell\colon E \to \mathit{Act}_\tau$ *is a* labelling function.

The class of stable event structures is denoted **ES**. In the sequel, we drop the qualifier “stable” and just talk about event structures. We sometimes write $E_{\mathcal{E}}$, $\#_{\mathcal{E}}$ etc. for the components of an event structure $\mathcal{E}$ and E_i, $\#_i$ etc. for the components of an event structure $\mathcal{E}_i$. As a further notational convention, we write $e_1, \ldots, e_n \vdash e$ for $\{e_1, \ldots, e_n\} \vdash e$ and $\vdash e$ for $\emptyset \vdash e$. Two event structures $\mathcal{E}_1, \mathcal{E}_2$ are said to be isomorphic, denoted $\mathcal{E}_1 \cong \mathcal{E}_2$, if there is a bijection $\phi\colon E_1 \to E_2$ (called an isomorphism) such that $d \#_1 e$ iff $\phi(d) \#_2 \phi_2(e)$, $F \vdash_1 e$ iff $\phi(F) \vdash_2 \phi(e)$, $F \in \mathit{Ter}_1$ iff $\phi(F) \in \mathit{Ter}_2$ and $\ell_1 = \ell_2 \circ \phi$.

The intuition behind stable event structures is that an action a may occur if an event $e \in E$ with $\ell(e) = a$ is enabled, meaning that $F \vdash e$ for the set F of events that have occurred previously. If two events are enabled simultaneously, i.e., $F \vdash d$ and $F \vdash e$, then either $d \# e$, meaning that after d or e has occured, the other is ruled out; or d and e are independent. A more formal definition of the meaning is given through the *configurations* of an event structure; see further below.

With respect to the standard definition in [140], the only modification is the extension with the termination predicate. For this, we have used a solution due to [64]: termination is modelled by a predicate on sets of events that may hold only for the *complete* consistent sets (i.e., those that are maximal w.r.t. $\subseteq$). Note that this completeness condition implies an analogy to the conditions for the enabling relation:

Saturation: If $F \in \mathit{Ter}$ and $F \subseteq G \in \mathit{Con}$, then $G \in \mathit{Ter}$;

Stability: If $F, G \in \mathit{Ter}$ and $F \cup G \in \mathit{Con}$, then $F \cap G \in \mathit{Ter}$.

In fact, $F \in \mathit{Ter}$ can also be interpreted as $F \vdash e_{\checkmark}$ for some special termination event $e_{\checkmark}$ signalling termination —where, however, we do not really model $e_{\checkmark}$ explicitly. As a consequence of the completeness of Ter, the following holds:

Lemma 4.8 *If $\mathcal{E} \in$ **ES** such that $\emptyset \in \mathit{Ter}$, then $\mathcal{E} = \langle \emptyset, \emptyset, \emptyset, \{\emptyset\}, \emptyset \rangle$.*

In order to deal with recursion, just as for the tree semantics in the previous sections we use an approximation ordering over the class of models, in this case **ES**. It is essentially the one used in [140].

$$\begin{array}{rl} \mathcal{E}_1 \sqsubseteq \mathcal{E}_2 :\Leftrightarrow & E_1 \subseteq E_2, \\ & \#_1 = \#_2 \cap (E_1 \times E_1), \\ & \vdash_1 = \vdash_2 \cap (\mathit{Con}_1 \times E_1), \\ & \mathit{Ter}_1 = \mathit{Ter}_2 \cap \mathit{Con}_1, \\ & \ell_1 = \ell_2 \restriction E_1 \end{array}$$

Moreover, for an arbitrary set $\{\mathcal{E}_i\}_{i \in I} \subseteq$ **ES**, the component-wise union of the $\mathcal{E}_i$ is written $\bigsqcup_{i \in I} \mathcal{E}_i$. We then recall the following property (see [140, Theorem 4.4]):

Proposition 4.9 *$\langle \mathbf{ES}, \sqsubseteq \rangle$ is a complete partial order, with bottom element $\langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$ and least upper bounds $\bigsqcup_i \mathcal{E}_i = \langle \bigcup_i E_i, \bigcup_i \#_i, \bigcup_i \vdash_i, \bigcup_i \mathit{Ter}_i, \bigcup_i \ell_i \rangle$ for all $\sqsubseteq$-directed sets $\{\mathcal{E}_i\}_i \subseteq$ **ES**.*

4.4 Denotational event-based semantics

We now define a number of partial operations on **ES**, corresponding to the operators of $\mathit{Lang}(\mathit{Evt})$. By relying on the annotation of terms, we will make sure that the operations are only applied where they are defined. For arbitrary $F \subseteq \mathit{Evt}_* \times \mathit{Evt}_*$, we use the following notation:

$$\begin{array}{rcl} \pi_i(F) & = & \{e_i \mid (e_1, e_2) \in F\} \quad (i = 1, 2) \\ F(d) & = & \{e \mid (d, e) \in F\} \end{array}$$

Thus π_i projects onto the i'th component, and $F(d)$ is a function-like use of F (albeit set-valued, since F is actually a relation). The definition of the constructions is inspired by [30, 64, 140], except for our treatment of termination. Note especially the construction for refinement, $\mathcal{E}[\mathcal{R}]$: this relies on a function $\mathcal{R}$ mapping all the events of $\mathcal{E}$ (and not just the a-labelled ones as in the operational semantics) to (non-terminated) event structures.

Definition 4.10 *We use the following constructions on event structures:*

Deadlock. $\mathcal{E}_\perp = \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$.

Termination. $\mathcal{E}_\checkmark = \langle \emptyset, \emptyset, \emptyset, \{\emptyset\}, \emptyset \rangle$.

Single action. $\mathcal{E}_{e,\alpha} = \langle \{e\}, \emptyset, \{(\emptyset, e), (\{e\}, e)\}, \{\{e\}\}, \{(e, \alpha)\} \rangle$ *for arbitrary* $e \in \mathit{Evt}_*$.

Sequential composition. *If* $E_1 \cap E_2 = \emptyset$, *then* $\mathcal{E}_1; \mathcal{E}_2 = \langle E, \#, \vdash, \mathit{Ter}, \ell \rangle$ *such that*

- $E = E_1 \cup E_2$;
- $\# = \#_1 \cup \#_2$;
- $\vdash = \{(F, e) \mid F \cap E_1 \vdash_1 e, F \cap E_2 \in \mathit{Con}_2\} \cup \{(F, e) \mid F \cap E_1 \in \mathit{Ter}_1, F \cap E_2 \vdash_2 e\}$;
- $\mathit{Ter} = \{F \subseteq E \mid F \cap E_1 \in \mathit{Ter}_1, F \cap E_2 \in \mathit{Ter}_2\}$;
- $\ell = \ell_1 \cup \ell_2$.

Choice. *If* $E_1 \cap E_2 = \emptyset$ *and* $\emptyset \notin \mathit{Ter}_1 \cup \mathit{Ter}_2$, *then* $\mathcal{E}_1 + \mathcal{E}_2 = \langle E, \#, \vdash, \mathit{Ter}, \ell \rangle$ *such that*

- $E = E_1 \cup E_2$;
- $\# = \#_1 \cup \#_2 \cup (E_1 \times E_2) \cup (E_2 \times E_1)$;
- $\vdash = \vdash_1 \cup \vdash_2 \cup (\mathit{Con}_1 \times \mathit{Ini}_2) \cup (\mathit{Con}_2 \times \mathit{Ini}_1)$;
- $\mathit{Ter} = \mathit{Ter}_1 \cup \mathit{Ter}_2$;
- $\ell = \ell_1 \cup \ell_2$.

Parallel composition. $\mathcal{E}_1 \parallel_A \mathcal{E}_2 = \langle E, \#, \vdash, \mathit{Ter}, \ell \rangle$ *such that*

- $E = (\bigcup_{a \in A} \ell_1^{-1}(a) \times \ell_2^{-1}(a)) \cup (\ell_1^{-1}(\mathit{Act}_\tau \setminus A) \times \{*\}) \cup (\{*\} \times \ell_2^{-1}(\mathit{Act}_\tau \setminus A))$;
- $\#^= = \{((d_1, d_2), (e_1, e_2)) \in E \times E \mid d_1 \#_1^= e_1 \neq * \vee d_2 \#_2^= e_2 \neq *\}$;
- $\vdash = \{(F, (e_1, e_2)) \in \mathit{Con} \times E \mid \forall i \in \{1, 2\}\colon e_i = * \vee \pi_i(F) \setminus * \vdash_i e_i\}$;
- $\mathit{Ter} = \{F \in \mathit{Con} \mid \forall i \in \{1, 2\}\colon \pi_i(F) \setminus * \in \mathit{Ter}_i\}$;
- $\ell = \{((e_1, e_2), \alpha) \in E \times \mathit{Act}_\tau \mid \alpha = \ell_1(e_1) \vee \alpha = \ell_2(e_2)\}$.

Hiding. $\mathcal{E}_1/A = \langle E_1, \#_1, \vdash_1, \mathit{Ter}_1, \ell \rangle$, *where* $\ell(e) = \tau$ *if* $e \in \ell_1^{-1}(A)$ *and* $\ell(e) = \ell_1(e)$ *otherwise.*

Refinement. *If* $\mathcal{R}\colon E_1 \to (\mathbf{ES} \setminus \mathcal{E}_\checkmark)$, *then* $\mathcal{E}_1[\mathcal{R}] = \langle E, \#, \vdash, \mathit{Ter}, \ell \rangle$ *such that*

- $E = \bigcup_{e \in E_1} (\{e\} \times E_{\mathcal{R}(e)})$;
- $\# = \{((d_1, d_2), (e_1, e_2)) \in E \times E \mid d_1 \#_1 e_1 \vee (d_1 = e_1 \wedge d_2 \#_{\mathcal{R}(d_1)} e_2)\}$;
- $\vdash = \{(F, (e_1, e_2)) \in \mathit{Con} \times E \mid \mathit{ready}(F) \vdash_1 e_1, F(e_1) \vdash_{\mathcal{R}(e_1)} e_2\}$;
- $\mathit{Ter} = \{F \in \mathit{Con} \mid \mathit{ready}(F) \in \mathit{Ter}_1\}$;
- $\ell = \{((e_1, e_2), \alpha) \in E \times \mathit{Act}_\tau \mid \alpha = \ell_{\mathcal{R}(e_1)}(e_2)\}$.

where for all $F \in \mathit{Con}$, $\mathit{ready}(F) = \{d \in \pi_1(F) \mid F(d) \in \mathit{Ter}_{\mathcal{R}(d)}\}$ *is the set of events from* $\mathcal{E}_1$ *whose refinement has reached a terminated configuration.*

Relocation. $e \times \mathcal{E}_1 = \langle E, \#, \vdash, \mathit{Ter}, \ell \rangle$ *such that*

- $E = \{e\} \times E_1$;
- $\# = \{((e, d_1), (e, e_1)) \mid d_1 \#_1 e_1\}$;
- $\vdash = \{(\{e\} \times F, (e, d)) \mid F \vdash_1 d\}$;
- $\mathit{Ter} = \{\{e\} \times F \mid F \in \mathit{Ter}_1\}$;
- $\ell = \ell_1 \circ \pi_2$.

Example 4.11 *Consider again the term $t = ({}_0b; {}_1a \parallel_a ({}_2c + {}_3a))[a \mapsto {}_4d; {}_5d]$ of Example 4.4.*

- *The subterm ${}_0b; {}_1a$ is modelled by the structure $\mathcal{E}_1 = \mathcal{E}_{0,b}; \mathcal{E}_{1,a}$ with $E_1 = \{0, 1\}$, no conflicts, enablings $\vdash_1 0$ and $0 \vdash_1 1$, termination predicate $\{\{0, 1\}\}$ and labelling $0 \mapsto b, 1 \mapsto a$. Analogous for ${}_4d; {}_5d$.*

- *The subterm ${}_2c + {}_3a$ is modelled by the structure $\mathcal{E}_2 = \mathcal{E}_{2,c} + \mathcal{E}_{3,a}$ with $E_2 = \{2, 3\}$, conflict $2 \#_2 3$, enablings $\vdash_2 2$ and $\vdash_2 3$, termination predicate $\{\{2\}, \{3\}\}$ and labelling $2 \mapsto c, 3 \mapsto a$.*

- *The parallel composition yields $\mathcal{E}_3 = \mathcal{E}_1 \parallel_a \mathcal{E}_2$ with $E_3 = \{(0, *), (*, 2), (1, 3)\}$, conflict $(*, 2) \#_3 (1, 3)$, enablings $\vdash_3 (0, *)$, $(0, *) \vdash_3 (1, 3)$ and $\vdash_3 (*, 2)$, termination predicate $\{\{(0, *), (1, 3)\}\}$ and labelling $(0, *) \mapsto b, (*, 2) \mapsto c, (a, 3) \mapsto a$.*

- *The refinement in t gives rise to $\mathcal{R}\colon E_3 \to$ **ES** with $\mathcal{R}(0, *) = \mathcal{E}_{(0,*),b}$, $\mathcal{R}(*, 2) = \mathcal{E}_{(*,2),c}$ and $\mathcal{E}_{(1,3)} = \mathcal{E}_{4,d}; \mathcal{E}_{5,d}$. t is then modelled by $\mathcal{E} = \mathcal{E}_3[\mathcal{R}]$ with $E = \{e_1, e_2, e_3, e_4\}$ where*

  $$e_1 = ((0, *), *) \quad e_2 = ((*, 2), *) \quad e_3 = ((1, 3), 4) \quad e_4 = ((1, 3), 5)$$

  and with conflicts $e_2 \# e_3$ and $e_2 \# e_4$, enablings $\vdash e_1$, $e_1 \vdash e_3$, $e_3 \vdash e_4$ and $\vdash e_2$, termination predicate $\{\{e_1, e_3, e_4\}\}$ and labelling $e_1 \mapsto b, e_2 \mapsto c, e_3 \mapsto d, e_4 \mapsto d$.

The first thing to make sure of is that the above constructions indeed yield event structures, and moreover, that an isomorphic argument gives rise to an isomorphic result.

Proposition 4.12 *Each of the operators in Definition 4.10 maps into* **ES** *and is well-defined up to isomorphism.*

For refinement, this was proved in [64]; for the other operators except sequential composition, see [30] (except for the termination predicate, whose construction, however, is very similar to that of enabling). For sequential composition, finally, the proof is straightforward.

The denotational event structure semantics of *Lang*(*Evt*) is given in Table 8. It should be noted that the disjointness requirements in the denotational constructions for choice and sequential composition are guaranteed to be satisfied due to the well-formedness of annotated terms. The most interesting definition is that for refinement: in principle, the semantic refinement function maps the events of the term being refined to the event structures obtained as the denotational semantics of the syntactic refinement function; however, the events that are in conflict with some busy event of the refinement are mapped to the deadlocked structure instead, to model the fact that, even if such events have not yet been removed from t (as we have seen in Example 4.4, choices in t are not resolved syntactically until the refinement has terminated), they nevertheless play no further role in the overall behaviour.

The following is the denotational counterpart to Proposition 4.6, namely that two annotations of the same basic term give rise to isomorphic event structures:

Proposition 4.13 *For all $t, u \in \mathit{Lang}(\mathit{Evt})$, if $\mathit{strip}(t) = \mathit{strip}(u)$ then $[\![t]\!] \cong [\![u]\!]$.*

Table 8: Denotational event structure semantics; $\overline{op}$ is the semantic counterpart of op

$$
\begin{array}{rcl}
[\![\mathbf{0}]\!] & = & \mathcal{E}_{\perp} \\
[\![\mathbf{1}]\!] & = & \mathcal{E}_{\checkmark} \\
[\![{}_{e}\alpha]\!] & = & \mathcal{E}_{e,\alpha} \\
[\![t[a \to u, \vec{e} \to \vec{v}]]\!] & = & [\![t]\!][\mathcal{R}] \text{ , where } \mathcal{R}(d) = \begin{cases} \mathcal{E}_{*,b} & \text{if } \ell_{[\![t]\!]}(d) = b \neq a \text{ and } \nexists e_i \in \{\vec{e}\} \colon d \; \#_{[\![t]\!]} \; e_i \\ [\![u]\!] & \text{if } \ell_{[\![t]\!]}(d) = a \text{ and } \nexists e_i \in \{\vec{e}\} \colon d \; \#^{=}_{[\![t]\!]} \; e_i \\ [\![v_i]\!] & \text{if } d = e_i \\ \mathcal{E}_{\perp} & \text{otherwise} \end{cases} \\
[\![{}_{e}[t]]\!] & = & e \times [\![t]\!] \\
[\![\mu x.t]\!] & = & \bigcup_{i \in \mathbb{N}} [\![\mu^i x.t]\!] \\
[\![op(t_1, \ldots, t_n)]\!] & = & \overline{op}([\![t_1]\!], \ldots, [\![t_n]\!]) \quad \text{for all other operators } op
\end{array}
$$

4.5 Compatibility of the semantics

In order to compare the event-based operational and denotational semantics, we must first decide on a relation up to which we wish them to be compatible. Isomorphism of the denotational semantics is certainly much stronger than (event) bisimilarity of the operational semantics; in fact, this was also already true for the sequential language in Section 2. We therefore choose event bisimilarity as the compatibility criterion. (Similar compatibility results of event-based operational and denotational semantics, albeit not including refinement, were given in [45, 102].

This means that we have to generate event transition systems from event structures. The most natural way to do this is to regard the event structure's *configurations* as states and define transitions from each configuration to all its direct $\subset$-successors. Formally:

Definition 4.14 *Let $\mathcal{E} \in$ **ES**.*

- *An* event trace *of $\mathcal{E}$ is a sequence $e_1 \cdots e_n \in E^*$ such that $\neg(e_i \#^{=} e_j)$ for all $1 \leq i < j \leq n$ (it is duplication- and conflict-free) and $e_1, \ldots, e_i \vdash e_{i+1}$ for all $1 \leq i < n$ (it is secured).*
- *A* configuration *of $\mathcal{E}$ is a set $F \subseteq E$ such that $F = \{e_1, \ldots, e_n\}$ for some event trace $e_1 \cdots e_n$.*
- *An event transition between configurations, $F \xrightarrow{e,\alpha} G$, holds iff $e \notin F$, $G = F \cup \{e\}$ and $\alpha = \ell(e)$. Furthermore, there is a transition $F \xrightarrow{\checkmark} \bullet$ for each $F \in Ter$, where $\bullet$ is a special state introduced only for this purpose.*

For arbitrary $\mathcal{E}$, we denote $\mathcal{T}(\mathcal{E}) = \langle \mathcal{C}(\mathcal{E}) \cup \{\bullet\}, \longrightarrow, \emptyset \rangle$ (where $\mathcal{C}(\mathcal{E})$ is the set of configurations of $\mathcal{E}$ and $\longrightarrow$ the transition relation defined above). Note that $\mathcal{T}(\mathcal{E})$ is indeed an event transition system in the sense of Definition 4.2. For instance, the event structure $\mathcal{E}$ developed in Example 4.11 gives rise to the following transition system $\mathcal{T}(\mathcal{E})$:

$$\begin{tikzcd}
\emptyset \arrow[r, "{e_1,b}"] \arrow[d, "{e_2,c}"'] & \{e_1\} \arrow[r, "{e_3,d}"] \arrow[d, "{e_2,c}"'] & \{e_1, e_3\} \arrow[r, "{e_4,d}"] & \{e_1, e_3, e_4\} \arrow[r, "\checkmark"] & \bullet \\
\{e_2\} \arrow[r, "{e_1,b}"] & \{e_1, e_2\} & & &
\end{tikzcd}$$

It is easy to see that this is event bisimilar to the transition system generated by the operational semantics (Example 4.4). Since information is lost in the generation of configurations from event structures, one can expect event bisimilarity to be strictly weaker than event structure isomorphism. The following proposition states that it is indeed weaker.

Proposition 4.15 *If $\mathcal{E}_1 \cong \mathcal{E}_2$ then $\mathcal{T}(\mathcal{E}_1) \sim_{\mathsf{ev}} \mathcal{T}(\mathcal{E}_2)$.*

To see that the reverse implication does not hold, consider the event structure semantics of the terms ${}_1a; {}_2b; {}_3c$ and ${}_1a; ({}_2b \mathbin{|||} {}_3c) \parallel_{b,c} {}_4b; {}_5c$. In the first of these, the third, c-labelled event (with identity 3) is enabled by the b-labelled event 2 only; i.e., $2 \vdash 3$. In the second, on the other hand, the c-labelled event (here $(3,5)$) is enabled by the combination of a- and b-events; i.e., $(1,*), (2,4) \vdash (3,5)$. (In fact, event bisimilarity of event structures coincides with isomorphism of the *sets of configurations* of those event structures.)

We can now state the compatibility result of the event-based operational and denotational semantics of *Lang*.

Proposition 4.16 *For any $t \in Lang(Evt)$, $t \sim_{\mathsf{ev}} \mathcal{T}([\![t]\!])$.*

4.6 Application: A simple data base

To show the potential of non-atomic refinement of synchronising actions, we consider a variation on the data base example of Section 2.5. Namely, we assume that, in addition to updating and querying the data base, one may also make a backup copy of it. We abstract away from the state of the data base; extending it to the n-state version (see Section 2.5) makes no essential difference to the current example (except for making the transition system more complicated). The behaviour of the date base is specified by $Data_S$, defined by

$$\begin{aligned} Data_S &:= State \parallel_{upd} Backup \\ State &:= (qry + upd); State \\ Backup &:= (copy + upd); Backup \ . \end{aligned}$$

Now suppose that on a more concrete level, both the update and the copy operations are split into two phases: $upd \to req; cnf$ (as in Section 2.5) and $copy \to back; copy$ (where *back* is a request to make a backup copy and *copy* is re-used on the concrete level to stand for the actual operation of copying). Thus, the refined behaviour is specified by

$$Data_I := Data_S[upd \to req; cnf][copy \to back; copy]$$

The abstract and refined behaviour are shown in Figure 6. Note that *qry* and *copy* in $Data_S$ are independent (which is evident from the fact that these actions give rise to a diamond in the event transition system). As a consequence of this independence, in $Data_I$ *qry* may occur while the (refined) backup operation is in progress, i.e., between *back* and the concrete *copy*, whereas *req* is disabled at that time; on the other hand, *qry* is *not* enabled while the (refined) update operation is in progress, i.e., between *req* and *cnf*. In fact, $Data_I$ is (both interleaving and event) bisimilar to the process where the refinements are interpreted syntactically, given by $State^{flat} \parallel_{req,cnf} Backup^{flat}$ where

$$\begin{aligned} State^{flat} &:= (qry + req; cnf); State^{flat} \\ Backup^{flat} &:= (back; copy + req; cnf); Backup^{flat} \ . \end{aligned}$$

The relation between semantic and syntactic refinement is discussed extensively in Section 6. A final observation is that $Data_S$ is interleaving bisimilar (but not event bisimilar) to a process where all actions are specified sequentially, given by $Data_S^{seq}$ with the following definition:

$$Data_S^{seq} := (qry + copy + upd); Data_S^{seq} \ .$$

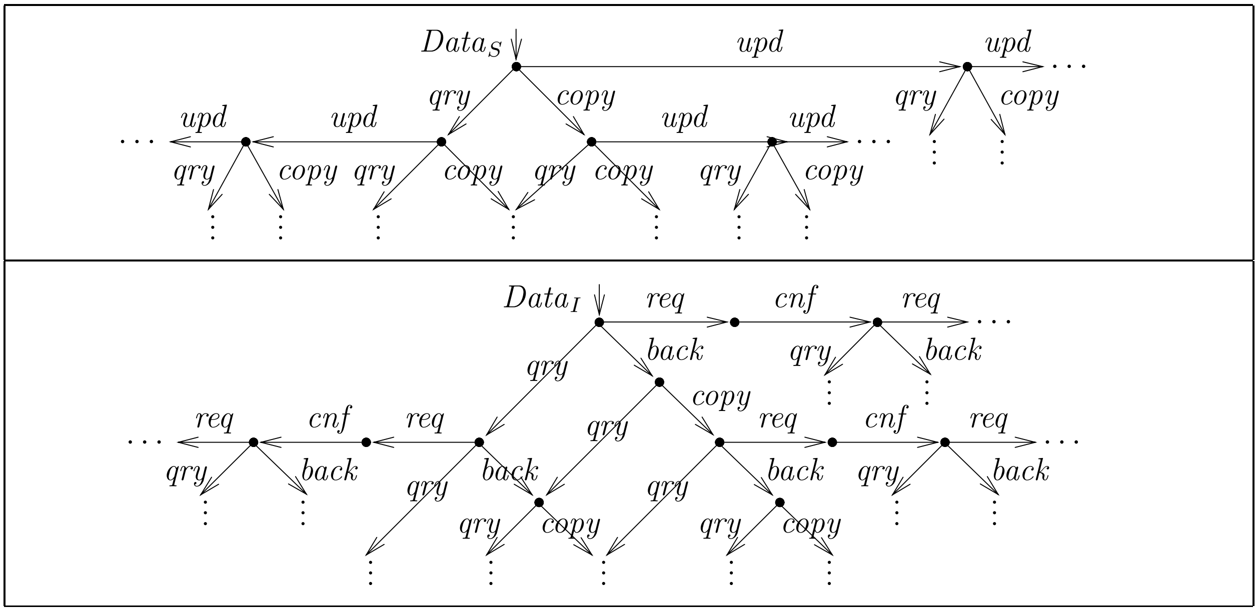


Figure 6: An initial fragment of the event-based operational semantics of $Data_S$ and $Data_I$ (event identities omitted)

If we refine $Data_S^{seq}$ in the same way as $Data_S$, the resulting behaviour is not interleaving bisimilar to $Data_I$: in the refinement of $Data_S^{seq}$, *qry* cannot occur between *back* and *copy*. Once again, this is evidence of the fact that interleaving bisimilarity is not a congruence for the refinement operator.

5 Non-atomic refinement: Other observational congruences

In the previous section, we have chosen a particular modelling principle (namely, action occurrences are events) and shown that this can be used to give semantics to a process algebra with action refinement; moreover, the corresponding notion of (event) bisimulation is a congruence. However, one may wonder if this congruence is the most suitable one. For instance, event bisimilarity distinguishes between the terms "$(a + b); c$" and "$a; c + b; c$"; for most purposes this distinction is not relevant.

A large part of the literature on action refinement is devoted to the quest for alternative congruences, in particular ones that are weaker (less distinguishing) than event bisimilarity. As a special instance, one can search for the *coarsest* congruence for action refinement contained within a given interleaving relation. There are two ways to go about this quest: to define equivalences over event structures that are weaker than isomorphism and investigate the induced relation on terms, or to develop alternative models altogether. An impressive overview based on the first of these methods has been drawn up by Van Glabbeek and Goltz in [64].

In this section, we review three possible bases for alternative models: *pomsets* (Section 5.1), *causal links* (Section 5.2) and *splitting* (Section 5.3). Pomsets give rise to a congruence on the level of traces but not when combined with bisimilarity. In contrast, causal links can be combined with bisimilarity into a congruence. Finally, with some care, splitting can be used to obtain coarsest congruences. In fact, in Section 5.4 we present an operational semantics directly based on the idea of splitting.

The material in this section is based on the language *Lang* introduced in the previous section

(Page 29); the semantics in Section 5.4 uses an extension of it, denoted $Lang^{ST}$.

5.1 Pomsets

The use of pomsets for the modelling of distributed system behaviour has been proposed first by Grabowski [79] (who, among other things, introduces the special class of so-called *N-free* or *series-parallel* pomsets, also investigated by Aceto in [1], as an auxiliary device to study Petri nets) and strongly advocated by Pratt [115] (who introduced the term "pomset"); other references are Jónsson [95] (who investigates pomsets from a more mathematical perspective), Gischer [55] (who works out Pratt's framework in more detail) and Rensink [122]. Furthermore, also *Mazurkiewicz traces* (see [103, 104]) are tightly connected to a certain class of pomsets.

Mathematically, a pomset is an isomorphism class of labelled partial orders. For the current purpose, the elements of the partial orders are events and their labels are action names. The basic definitions are listed below.

- A labelled partial order (lpo) is a tuple $p = \langle E, <, \ell \rangle$, where $E \subseteq \mathit{Evt}$ is a set of events causally ordered by the irreflexive and transitive relation $< \subseteq E \times E$, and $\ell \colon E \to \mathit{Act}_{\tau\checkmark}$ is a labelling function.
- If $p_i = \langle E_i, <_i, \ell_i \rangle$ is an lpo for $i = 1, 2$, then p_1 and p_2 are called isomorphic if there is a bijection $f \colon E_1 \to E_2$ such that $d <_1 e$ iff $f(d) <_2 f(e)$ for all $d, e \in E_1$ and $\ell_1 = \ell_2 \circ f$.
- The isomorphism class of a poset $\langle E, <, \ell \rangle$ is called a *pomset*. The class of all pomsets is denoted Pom.

By taking isomorphism classes of posets, the precise identities of events are abstracted away from *before* they can be used to pinpoint the exact moment at which choices are resolved. Consequently, a pomset-based model is more abstract than an event-based model and thus induces a weaker equivalence over Lang.

Pomset transitions from event transitions. In fact, from the event transition systems used for the operational semantics in the previous section (Definition 4.2), one can generate pomset-labelled transition systems, in the following fashion:

- Each sequence of event-action-pairs, say $(e_1, \alpha_1)\,(e_2, \alpha_2) \cdots (e_n, \alpha_n) \in (\mathit{Evt} \times \mathit{Act}_\tau)^*$, can be regarded as an lpo that happens to be totally ordered: namely $p = \langle E, <, \ell \rangle$ where $E = \{e_1, \ldots, e_n\}$, $e_i < e_j$ iff $i < j$ and $\ell(e_i) = \alpha_i$ for all $1 \leq i, j \leq n$.

- For each pair of states s, s' between which there is a path not containing a $\checkmark$-labelled transition, one can generate an lpo by considering *all* paths from s to s' and intersecting their ordering. That is, let $p_1, \ldots, p_n$ with $p_i = \langle E_i, <_i, \ell_i \rangle$ be the set of totally ordered lpo's generated from transition paths between s and s' in the fashion described above; then $E_i = E_j$ and $\ell_i = \ell_j$ for all $1 \leq i, j \leq n$ due to the properties of the transition system. This gives rise to another lpo $q = \langle E, <, \ell \rangle$ with $E = E_1$, $< = \bigcap_{1 \leq i \leq n} <_i$ and $\ell = \ell_1$; we let $s \xrightarrow{q} s'$.

- $\checkmark$-labelled transitions are left unchanged.

- Each lpo thus obtained can be turned into a pomset by taking its isomorphism class.

The idea is that in the event-based view, each sequence of transitions corresponds to the execution of a causally ordered set of labelled events; hence an lpo. Due to the construction of the transition system, there is an exact correspondence between the paths between the states and the linearisations of this lpo. The original lpo can therefore be reconstructed through the intersection of all these linearisations.

Pomset-labelled transition systems can be compared in a number of ways; for instance, one may look only at the pomset traces, or one may apply the natural notion of bisimilarity.

Pomset traces. The pomset traces of a term of *Lang* are given by the outgoing pomset transitions of the initial state. There is a distinguished class of *terminated* pomset traces, which are the ones leading to a state with an outgoing ✓-transition. We call two terms *pomset trace equivalent* if they give rise to equal sets of terminated and non-terminated pomsets.

As an alternative to this roundabout construction, one can directly define an algebra of (sets of) pomsets; in fact, this is the subject of most of the pomset papers cited above. For the synchronisation-free fragment of $\mathit{Lang}^{\mathit{fin}}$ (i.e., containing $\_ \parallel_A \_$ only for $A = \emptyset$, but including refinement), such an algebra is for instance defined in [110]. The extension to recursion is straightforward; the extension to synchronisation, although technically awkward, is also unproblematic —see for instance [116]. (The awkwardness is due to that fact that the synchronisation of two pomsets is in general not a single-valued operation: for instance, the synchronisation of $\boxed{\begin{matrix} a & & \\ a & \to & b \end{matrix}}$ and $\boxed{a \to a}$ on a yields $\boxed{\begin{matrix} & & a \\ & \nearrow & \\ a & \to & b \end{matrix}}$ and $\boxed{a \to a \to b}$ as two possible outcomes.)

We do not go into detail here concerning the definition of the operators, except to remark that the refinement operator on sets of pomsets crucially depends on the ability to distinguish the terminated pomsets from the non-terminated ones (see also Section 1.5). For instance, if one ignores this distinction then $a;\mathbf{0}$ and a give rise to the same sets of pomsets, and yet $(b;c)[b \to a;\mathbf{0}]$ and $(b;c)[b \to a]$ should not be pomset trace equivalent. Since in the restricted setting of [110], the terminated pomsets coincide with the maximal ones, the distinction is easy there.

It follows from the existence of an algebraic operator on the semantic model that pomset trace equivalence is a congruence for refinement. This fact was first stated in [32]. As a consequence of the construction of pomsets from event transition systems and the existence of a full *Lang*-algebra over sets of pomsets, this can be extended to the following result.

Theorem 5.1 *Pomset trace equivalence is a congruence over Lang that is coarser than* $\sim_{\mathsf{ev}}$.

Pomset bisimilarity. The above construction of pomsets from event transition systems actually gives rise to a much richer model than just pomset traces: namely, a transition system labelled on $\mathit{Pom} \cup \{\checkmark\}$. Having a transition system, we can once more apply the principles of bisimulation. We call two terms *pomset bisimilar* if their associated pomset-labelled transition systems are strongly bisimilar (see also Boudol and Castellani [20]). Surprisingly (as first observed by Van Glabbeek and Goltz in [60]), pomset bisimilarity is *not* a congruence for refinement. A simple counter-example is given by the pomset bisimilar terms $t = a;(b+c) + a \mathbin{|||} b$ and $u = t + a;b$, whose refinements $t[a \to a_1;a_2]$ and $u[a \to a_1;a_2]$ are *not* pomset bisimilar: after the execution of a_1 there is a behaviour that only $u[a \to a_1;a_2]$ may show, namely a state where a_2 and b are executable only sequentially and c is not executable at all. For more details see [64]; an extensive discussion can also be found in [133].

As an alternative to the indirect construction of pomset transition systems from event transition systems, [20] presents an operational semantics directly on pomset transition systems, albeit only for the synchronisation- and refinement-free fragment of $\mathit{Lang}^{\mathit{fin}}$. The extension to synchronisation and recursion is once more straightforward; however, from the fact that pomset bisimilarity is not a congruence for refinement, it follows that an operational semantics for refinement will be difficult to find.

5.2 Causal links

A different modelling principle is obtained if one takes not the events themselves but the *causality* between them as the essential notion. Operationally, this can be done by adding a set of *causal links* to each transition that point back to those previous transitions on which the current one depends. This idea was first worked out in the *causal trees* of Darondeau and Degano [39]. They use a particular simple way to implement the causal links: namely, the links are given as positive natural numbers corresponding to the number of transitions one must count back from the current one. Note that for this to work, it is essential that each state have a unique predecessor, i.e., that the model be a tree. In our setting, there is an additional class of $\checkmark$-labelled transitions. Thus, a causal tree is a tree-shaped transition system labelled on $(\mathbf{2}^{\mathbb{N}^{>0}} \times \mathit{Act}_\tau) \cup \{\checkmark\}$ (where $\mathbb{N}^{>0}$ denotes the set of positive natural numbers).

Causal transitions from event transitions. The event transition systems of the previous section (Definition 4.2) give rise to causal trees in the following way. Let $\mathcal{T}$ be an arbitrary event transition system.

- The states of the causal tree derived from $\mathcal{T}$ are vectors of events $\vec{e} = e_1 \cdots e_k$ (with $e_i \in \mathit{Evt}$ for $1 \leq i < k$ and $e_k \in \mathit{Evt} \cup \{\checkmark\}$) corresponding to traces of t. To be precise, $\vec{e}$ is a state iff $\iota \xrightarrow{e_1,\alpha_1} \cdots \xrightarrow{e_{k-1},\alpha_{n-1}} n'$ for some n' and either $n' \xrightarrow{e_k,\alpha_k} n''$ (if $e_k \in \mathit{Evt}$) or $n' \xrightarrow{\checkmark} n''$ (if $e_k = \checkmark$) for some n''. Note that if n'' exists, then it is uniquely determined by $\vec{e}$; we denote $n'' = \iota_{\vec{e}}$.
- The outgoing non-termination transitions of $\vec{e}$ are defined by induction on $|\vec{e}|$. Each event transition $\iota_{\vec{e}} \xrightarrow{e',\alpha} n'$ gives rise to a causal transition $\vec{e} \xrightarrow{K,\alpha} \vec{e}\,e'$, where the set K of causal links is determined in one of the following two ways.
  - If $\iota_{e_1 \cdots e_{k-1}} \xrightarrow{e',\alpha}$ (i.e., e' was already enabled in the direct e_k-predecessor of $\iota_{\vec{e}}$) then (due to the inductive definition) we already had derived the causal transition
  $$e_1 \cdots e_{k-1} \xrightarrow{K',\alpha} e_1 \cdots e_{k-1}e'$$
  for some set $K' \subseteq \mathbb{N}^{>0}$. We let $K = K' + 1$ $(= \{i + 1 \mid i \in K'\})$.
  - If $\iota_{e_1 \cdots e_{k-1}} \not\xrightarrow{e',\alpha}$, then we had already derived the causal transition
  $$e_1 \cdots e_{k-1} \xrightarrow{K',\alpha} \vec{e}$$
  for some $K' \subseteq \mathbb{N}^{>0}$. We let $K = (K' + 1) \cup \{1\}$.
- Termination transitions are given by $\vec{e} \xrightarrow{\checkmark} \vec{e}\checkmark$ whenever $\iota_{\vec{e}} \xrightarrow{\checkmark}$.

Having obtained causal trees from terms of *Lang*, one can again study induced relations over *Lang*, generated by different interpretations of causal trees. For instance, we call two terms *causally bisimilar* if their respective causal trees are bisimilar. It was first observed in [42] that causal bisimilarity is a congruence for action refinement. This can again be generalised to the following result:

Proposition 5.2 *Causal bisimilarity is a congruence over Lang, which is weaker than* $\sim_{\mathsf{ev}}$.

A similar property holds for causal traces (the linear-time counterpart of causal trees), which in fact constitute an alternative representation for pomset traces, studied in [120].

Instead of recovering the causal links from a more concrete semantic model, as we have done here, it is possible to define a *Lang*-algebra of causal trees (see [40, 42]) or to use causal trees as an operational model. With respect to the latter, see [50] for a structural operational semantics for (essentially) $\mathit{Lang}^{\mathit{fin}}$, which can easily be extended to deal with recursion. Since the semantics is given in SOS style, that paper also reports a set of axioms for the calculus.

As a final remark, we recall that there are several alternative (indeed historically preceding) characterisations of causal bisimilarity. Rabinovich and Trakhtenbrot first introduced the relation in [116] as *behaviour structure bisimilarity.* In [60], the same relation is characterised as *history preserving bisimilarity* over event structures. Finally, the *mixed ordering bisimilarity* of [46] also gives rise to the same equivalence. See [3, 64] for further details concerning these correspondences.

5.3 Splitting actions

Having found (in causal bisimilarity) a bisimulation-based equivalence that is a congruence with respect to action refinement, a natural question is if this is the *coarsest* of its kind, i.e., the coarsest congruence for action refinement contained in (standard) interleaving bisimilarity. An analogous question can be posed for pomset trace or causal trace equivalence. It turns out that the answer to either question is "no": in order to obtain coarsest congruences, one needs to add information to the interleaving model in a way that is different from either pomsets or causal links.

Example 5.3 *There exist processes with distinct causal structures that yet cannot be distinguished up to bisimilarity by any refinement context. For instance, consider the bisimilar (but not causally bisimilar) terms* $a \mathbin{|||} b; \mathbf{0}$ *(where* a *and* b *are causally independent) and* $(a; b \mathbin{|||} b) \parallel_b b$ *(where there is alternative way to perform* b*, causally following* a*). It is not difficult to convince oneself that there is no way to make them non-bisimilar (in the interleaving sense) by refining their actions.*

So, what is the coarsest congruence within strong bisimilarity? Intuitively, by refining an action we are implicitly assuming that such an action is no longer non-interruptible, as the execution of its refinement can be interleaved with the execution of other concurrent processes. In the example above, after refining a into $a_1; a_2$, the existence of the sequence $a_1\, b\, a_2$ shows the interruptability of the refined a. Hence, a natural question is whether the actions could not *a priori* be described as split into phases: for instance, a^+ for the beginning and a^- for the ending of an arbitrary action a. If one interprets the resulting transition systems up to strong bisimilarity, $a \mathbin{|||} b; \mathbf{0}$ and $a; b; \mathbf{0} + b; a; \mathbf{0}$ (for example) are distinguished whereas $a \mathbin{|||} b; \mathbf{0}$ and $(a; b \mathbin{|||} b) \parallel_b b$ are not (see Figure 7).

This is the basic idea behind the so-called *split* semantics, probably introduced for the first time in [84] (although the principle is already mentioned by Hoare in [88]). Actions can be split in any number of phases, say n, giving rise to a family of split_n-equivalences for every interleaving equivalence. In fact, splitting into n phases can be seen as a restricted form of action refinement where the refinement body can only consist of sequences of length n.

Split bisimilarity. For (essentially) the synchronisation-free fragment of *Lang* (or alternatively a language with CCS-like communication but without restriction), Aceto and Hennessy [8] established that strong split_2-bisimilarity is the coarsest congruence for action refinement contained in strong bisimilarity. Actually, in this language, split_n-bisimilarity coincides with split_{n+1}-bisimilarity for any $n \geq 2$.

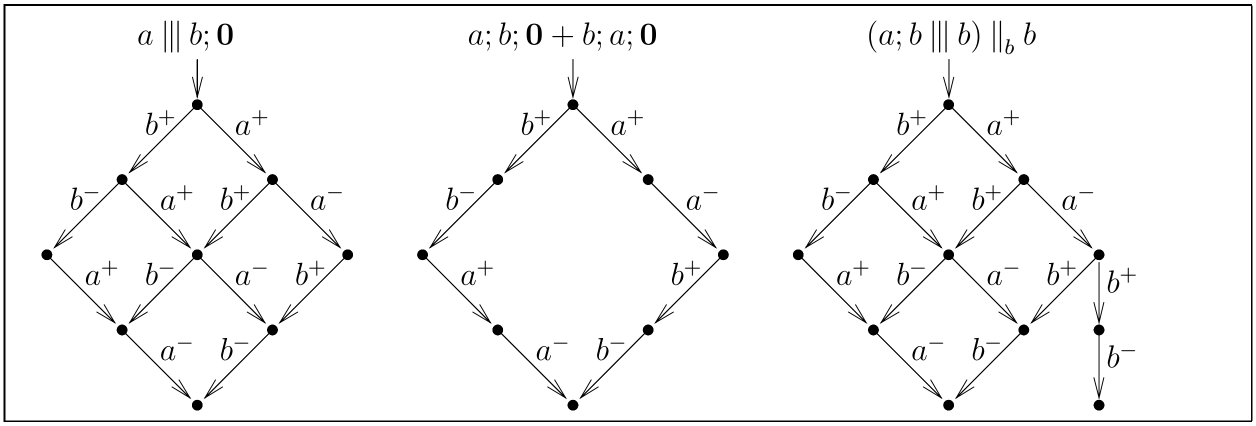

$a \mid\mid\mid b; \mathbf{0}$
$a; b; \mathbf{0} + b; a; \mathbf{0}$
$(a; b \mid\mid\mid b) \parallel_b b$
a^+
b^+
a^-
b^-

Figure 7: Three $split_2$ transition systems.

However, as soon as synchronisation is included, $split_2$-bisimilarity is no longer a congruence: splitting actions in more and more phases yields more and more discriminating equivalences. Van Glabbeek and Vaandrager [66] present an example, parametric in n, to show that the *split*-bisimilarities form a strict chain. The example is based on confusion on identity of auto-concurrent actions (i.e., multiple occurrences of the same action, concurrently executed) that $split_n$ semantics may bring to light when there are more than $n-1$ such actions. A simplified example that clarifies the difference between $split_2$ and $split_3$ is the so-called *owl example*, originally proposed by Van Glabbeek in 1991.

Example 5.4 *Consider $t = (t_1 \parallel_S t_2)[\phi]$, where $S = \{c', c'', d, d', e, e'\}$ and ϕ relabels c' and c'' into c, d' into d and e' into e (leaving all the other labels fixed).*

$$\begin{array}{rcl} t_1 & = & a; ((c; (c'; e' \mid\mid\mid d' \ + \ e) \ + \ c''; d' \mid\mid\mid e') \mid\mid\mid d; \mathbf{0}) \\ t_2 & = & b; ((c; (c''; d' \mid\mid\mid e' \ + \ d) \ + \ c'; e' \mid\mid\mid d') \mid\mid\mid e; \mathbf{0}) \end{array}$$

Symmetrically, we define $u = (u_1 \parallel_S u_2)[\phi]$, where

$$\begin{array}{rcl} u_1 & = & a; ((c; (c'; d' \mid\mid\mid e' \ + \ d) \ + \ c''; e' \mid\mid\mid d') \mid\mid\mid e; \mathbf{0}) \\ u_2 & = & b; ((c; (c''; e' \mid\mid\mid d' \ + \ e) \ + \ c'; d' \mid\mid\mid e') \mid\mid\mid d; \mathbf{0}) \end{array}$$

The two interleaving transition systems are shown in Figure 8. The labels of the inner transitions are omitted for improving readability: they can be assigned by looking at the label of the parallel outer transitions. It is not difficult to see that the two transition systems are interleaving bisimilar. For instance, the sequence $b\,a\,c$ leads to bisimilar states in both systems: the "wing" of t is simulated by the "body" of u, and vice versa.

The $split_2$ transition systems for t and u can be guessed by considering that all the "diamonds" in the graph are divided into four sub-diamonds. The two resulting graphs are bisimilar (so t and u are $split_2$ bisimilar) by following an argument similar to the above. The crucial point is reached after the sequence $a_1\,a_2\,c_1\,b_1\,b_2\,c_1$. In that state (which is the same in both graphs), the c_2 executed by t is matched by the 'symmetric' c_2 in the graph for u. This means that whenever t completes the c causally dependent of, let say, a then u completes the c caused by b.

The same game cannot be played when splitting actions into three phases; indeed, it is always possible to recognise observationally *in which of the two directions we are moving, i.e., which of the*

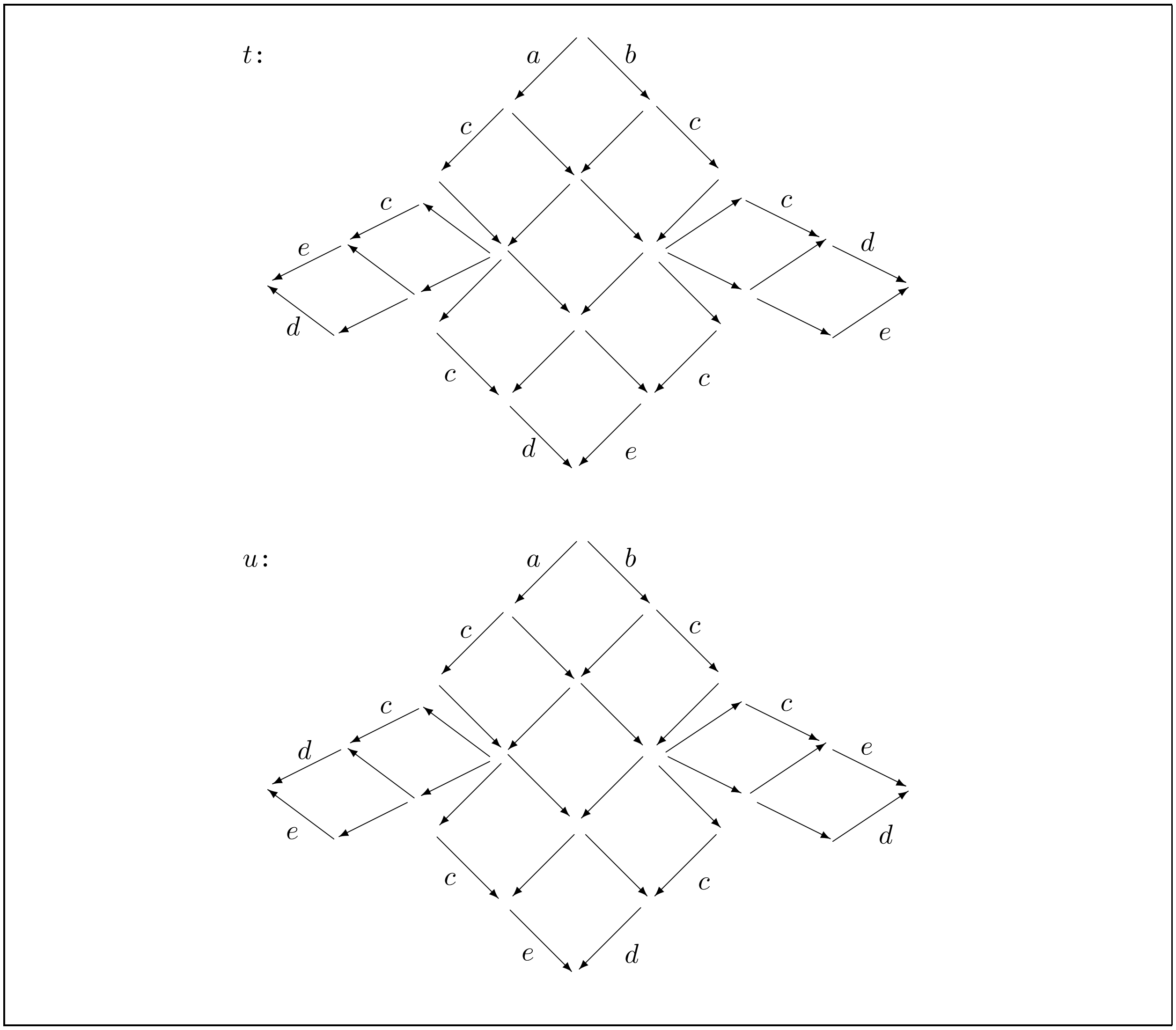


Figure 8: The "owl" example.

two c's we are going to complete. This informal consideration is illustrated by the sequence

$$a\, c_1\, b\, c_2\, c_1\, c_3\, e$$

which can be executed by t but not by u (unsplit actions stand for the consecutive execution of their three phases).

Two natural questions then arise:

- What is the limit of this chain of $split_n$-bisimulation equivalences?
- How does the chain limit relate to the coarsest congruence problem?

It turns out (see Gorrieri and Laneve [73, 74]) that the limit of strong $split_n$-bisimilarity is an equivalence known in the literature as strong ST-bisimilarity, originally due to Van Glabbeek and Vaandrager [65]. The ST-principle improves over *split* by including a mechanism for reconstructing the individuality of each action occurrence. Typically, this is done by associating to each action one unique identifier that is used when it is necessary to know the right beginning for the currently executed ending. By the limit theorem of [74], it follows that strong ST-bisimilarity is the coarsest congruence for splitting of actions; and in fact, the result extends to general refinement (see [57, 133] for a more general congruence result, [30] for a coarsest congruence result for any form of refinement in the model of stable event structures and the survey paper [64] for another proof of coarsest congruence on several event-based models). Hence, strong ST-bisimilarity is the required coarsest congruence contained in strong bisimilarity.

It has also been proved in [9] that rooted weak ST-bisimilarity (called refine equivalence in that paper) is the coarsest congruence for (syntactic) action refinement contained in rooted weak bisimilarity.

Split traces. If one starts the investigation by considering trace semantics [88] instead of bisimulation, then the answers to the two questions above are slightly different. In the full language *Lang*, $split_n$-trace equivalence again gives rise to a strict chain of equivalences. However, as shown in [101], the limit of the chain in this case is not ST-trace equivalence.

Example 5.5 *Consider $t = a; b \mathbin{|||} b; c; \mathbf{0}$ and $u = (a; b; c \mathbin{|||} b; c) \parallel_c c$. The only pomset trace of t is $\boxed{\begin{array}{l} a{\to}b \\ b{\to}c \end{array}}$, whereas u additionally has $\boxed{\begin{array}{l} a{\to}b{\to}c \\ b \end{array}}$. However, t and u cannot be distinguished on the basis of their $split_n$-traces. To see this, first note that the only possible difference can be that u has a trace that t cannot match; in particular, a trace that is a linearisation of $\boxed{\begin{array}{l} a{\to}b{\to}c \\ b \end{array}}$. Thus, such a distinguishing trace must reflect the fact that c is caused by the upper b (the one following a) rather than the lower b. However, no matter in how many phases we split b, this distinction cannot be made: in particular, any $split_n$-trace $b_1 \cdots b_i\, a_1 \cdots a_n\, b_1 \cdots b_n\, c_1$ of u can be matched by t because the $b_{i+1} \cdots b_n$-subsequence might belong to the first b-occurrence as well as the second.*

On the other hand, t and u are not ST-trace equivalent. For instance, although both processes can perform the $split_2$-trace $b_1\, a_1\, a_2\, b_1\, b_2\, c_1$, in t the b_2 must be the end of the first b_1 whereas in u it may also be the end of the second b_1.

The limit of $split_n$-trace semantics has been characterised in [134] and called *swap*-equivalence. It turns out that *swap*-equivalence is not a congruence for general action refinement. For instance, although (as we argued) t and u in Example 5.5 are *swap*-equivalent, refining b into $v = b_1; b_2 + b_3$ shows up their difference, since $u[b{\to}v]$ has trace $b_1\, a\, b_3\, c$ which $t[b{\to}v]$ cannot match. Vogler has proved in [131, 132] that the coarsest congruence for action refinement contained in trace equivalence is ST-trace equivalence.

Further split semantics. In the linear time – branching time spectrum [59], in between bisimilarity and trace equivalence there are many other equivalences, among which failure [27] (or testing [43]) equivalence is the most relevant. The issues above have been investigated for this case as well: Vogler in [131, 132] proves that ST-failure semantics is a congruence for action refinement, and in [134] conjectures that it is also the coarsest congruence.

Most of the work mentioned above has been developed on semantic models rather than process algebra. In addition to the work already cited above, papers dealing with operational *split*- and ST-semantics include [31, 37]. Characterisations of ST-failure (or testing) semantics in process algebra are reported in [5] on a simple process algebra, and in [85] on CCS.

5.4 ST-semantics

We go somewhat deeper into the ST-semantics of terms. Below, we present a conversion from event transition systems into ST-transition systems as well as a (compatible) operational ST-semantics for *Lang* —including, of course, the refinement operator.

ST-transition systems. It is clear from the above discussion that the main idea in ST-semantics is to distinguish the start and end of actions, in such a way that there is an unambiguous association between the two. There are several different ways to encode this association.

- The most straightforward method is to associate with the start of the action a unique fresh identifier, much like the event annotation, and annotate the end of the action with the same identifier. The disadvantage of this method is that the choice of identifier is free, and therefore it cannot be guaranteed that the same identifier is used in bisimilar states; one therefore needs a more elaborate, indexed version of bisimulation like we already had for event bisimilarity (Definition 4.3); see, for instance, [57, 9, 23].

- The above method can be improved by somehow fixing the choice of the fresh identifier so that it depends uniquely on the set of identifiers currently in use for "outstanding" action occurrences, i.e., actions that have started but not finished. This can be done for instance by imposing a total ordering on the set of all identifiers. One can then safely assume that the choice of identifier is always the same in bisimilar states. For an example, see [24].

- Alternatively, one can assume a tree-like structure and let the end of the action point back to the start through a natural number indicating the number of transitions one has to count back to find the associated start. This idea was used, e.g., in [73, 74, 31]. The disadvantage (which in fact also holds for causal transition systems as defined in Section 5.2) is that the resulting transition systems are necessarily infinite state for all recursive behaviours.

- Here we use yet another principle, developed more recently in [24], which is in some sense a mixture of the above: the end actions indeed carry counters, but instead of transitions it is the number of outstanding occurrences (of the same action) that is counted. As a consequence, ordinary bisimulation can be used to compare ST-transition systems. Moreover, the operational semantics not only generates finite state transition systems for the refinement-free fragment of *Lang* whenever the standard interleaving semantics does (which is also the case for the semantics of [23]), but also provides finite state models for action refinement.

As labels in the ST-semantics, for all $a \in Act$ we use a^+ to denote the start of a fresh occurrence of a, and a_i^- with $i \in \mathbb{N}^{>0}$ to denote the end of the i'th occurrence of a that was still outstanding (counting backwards, that is, the most recent occurrence is numbered 1). We denote $Act^{ST} = \{a^+ \mid a \in Act\} \cup \{a_i^- \mid a \in Act, i \in \mathbb{N}^{>0}\}$ and $Act^{ST}_{\tau\checkmark} = Act^{ST} \cup \{\tau, \checkmark\}$.

Definition 5.6 *An ST-transition system is an $Act^{ST}_{\tau\checkmark}$-labelled transition system, such that for all states $n \in N$ and action $a \in Act$ there is a number $O(n,a)$ of outstanding occurrences where*

- $O(\iota, a) = 0$ *for all* $a \in Act$;
- *If* $n \xrightarrow{a^+} n'$ *then* $O(n', a) = O(n, a) + 1$ *and* $O(n', b) = O(n, b)$ *for all* $a \neq b$;
- *If* $n \xrightarrow{a_i^-} n'$ *then* $i \leq O(n, a)$, $O(n', a) = O(n, a) - 1$ *and* $O(n', b) = O(n, b)$ *for all* $a \neq b$;
- *If* $n \xrightarrow{\tau} n'$ *then* $O(n', a) = O(n, a)$ *for all* $a \in Act$;

- $n \xrightarrow{a_i^-}$ *for all* $a \in \mathit{Act}$ *and* $1 \leq i \leq O(n, a)$.

(Note that this uniquely determines $O(n, a)$ for all reachable states.) As stated above, to compare ST-transition systems we will use ordinary bisimulation.

ST-transitions from event transitions. The event transition systems used in the previous section (Definition 4.2) give rise to ST-transition systems in the following way. Let $\mathcal{T}$ be an arbitrary event transition system.

- The states of the ST-transition system derived from $\mathcal{T}$ are tuples $(n, \vec{e})$ for some node $n \in N$ and finite vector of events $\vec{e} = e_1 \cdots e_k$ such that $n \xrightarrow{e_1,\alpha_1} \cdots \xrightarrow{e_k,\alpha_k} n'$ for some n' and $n \xrightarrow{e_i,\alpha_i}$ for all $1 \leq i \leq k$. (In other words, the events in $\vec{e}$ are all enabled in n and can also be executed serially; this means that they are pairwise independent.) Due to the properties of event transition systems, the node n' is uniquely determined by n and $\vec{e}$; we denote $n' = n_{\vec{e}}$. It follows that also $n' = n_{\vec{d}}$ for an arbitrary permutation $\vec{d}$ of $\vec{e}$.

- The outgoing transitions of $(n, \vec{e})$ are the following (where for all $1 \leq i \leq k$, α_i is the action associated with e_i, and $\vec{e} \setminus i$ denotes the removal of the i'th element of $\vec{e}$):

  - If $n_{\vec{e}} \xrightarrow{e',a}$ such that e' is independent, in n, of all e_i, then $(n,\, \vec{e}) \xrightarrow{a^+} (n,\, \vec{e}\,e')$;
  - If $n_{\vec{e}} \xrightarrow{e',\tau}$ such that e' is independent, in n, of all e_i, then $(n,\, \vec{e}) \xrightarrow{\tau} (n,\, \vec{e}\,e')$;
  - For all $1 \leq i \leq k$, if $\alpha_i \in \mathit{Act}$ then let $a = \alpha_i$ and $m = |\{i \leq j \leq k \mid \alpha_j = a\}|$; then $(n,\, \vec{e}) \xrightarrow{a_m^-} (n_{e_i},\, \vec{e} \setminus i)$.
  - For all $1 \leq i \leq k$, if $\alpha_i = \tau$ then $(n,\, \vec{e}) \xrightarrow{\tau} (n_{e_i},\, \vec{e} \setminus i)$.
  - If $\vec{e} = \varepsilon$ and $n \xrightarrow{\checkmark} n'$, then $(n,\, \vec{e}) \xrightarrow{\checkmark} (n',\, \varepsilon)$.

The number of outstanding a-occurrences of an ST-transition system constructed in this way equals the number of a's in the $\vec{e}$: that is, $O((n, \vec{e}), a) = |\{1 \leq j \leq k \mid \alpha_j = a\}|$ for all states $(n, \vec{e})$. It is not difficult to check that this construction indeed gives rise to an ST-transition system in the sense of Definition 5.6.

Operational ST-semantics. In addition to the indirect construction of ST-transition systems through the event-based operational semantics, presented above, we will also give a direct ST-operational semantics for *Lang*. To be precise, we use a language Lang^{ST} with the following grammar (changes with respect to *Lang* are underlined; see also Table 16 (Page 85) for a complete overview of the different languages used in this chapter).

$$\mathsf{T} ::= \mathbf{0} \mid \mathbf{1} \mid \alpha \mid \underline{\alpha^-} \mid \mathsf{V} + \mathsf{V} \mid \mathsf{T}; \mathsf{V} \mid \underline{\mathsf{T} \parallel_{A,M} \mathsf{T}} \mid \mathsf{T}/A \mid \underline{\mathsf{T}[a \to \mathsf{V}, \vec{\mathsf{T}}]_M} \mid x \mid \mu x.\mathsf{V}\ .$$

We briefly discuss the new operators.

- α^- is an auxiliary operator denoting the intermediate state reached by α after it has started; depending on whether α is visible or not, it performs the corresponding end action or another τ-action. (That is to say, τ's are split as well, mainly to preserve some intuitive axioms up to bisimilarity.) As usual with auxiliary operators, α^- is not allowed in virgin operands (V in the grammar).

- $t \parallel_{A,M} u$ denotes parallel composition extended with a mapping $M \colon \mathit{Act} \to \{0, 1\}^*$ to be discussed below.

- $t[a \to u, \vec{v}]_M$ denotes refinement, extended with a vector $\vec{v}$ consisting of *pending* (or *busy*) refinements, entirely analogous to the busy refinements used in the previous section, except that here we do not need the identities of the events being refined. (In fact, those identities are encoded in the position of the pending refinement within the vector $\vec{v}$.) We use $\vec{v} \setminus i$ to denote the removal of the i'th element of the vector, and $\vec{v} \pm (i, v')$ to denote the replacement of the i'th element by v'. Again, M is a mapping, this time from *Act* to $\{0, \ldots, |\vec{v}|\}^*$, whose purpose is explained below.

In the last two operators, M may be omitted if it maps all $a \in \mathit{Act}$ to the empty string; thus, *Lang* is embedded in Lang^{ST}. The well-formedness conditions on Lang^{ST} thus come down to the usual:

- No virgin operand (V in the grammar) contains an auxiliary operator ($\mathbf{1}$ or α^-);
- Recursion is allowed on guarded variables only.

The challenge in the definition of the operational semantics of Lang^{ST} is to adjust the counters in the a_i^--labels of the operands' transitions in such a way that they always point to the correct start action. For the flat, sequential fragment of *Lang* this presents no particular problem, since there the order in which new actions are started and old ones are finished is directly derived from the operands. In the case of synchronisation and refinement, however, we are faced with the problem that there is more than one operand that is *active*, i.e., may have outstanding action occurrences: in the case of parallel composition, both operands are active, and in the case of refinement, both the term being refined and all pending refinements are active. This means that the numbering of the outstanding action occurrences in the combined behaviour changes with respect to the numbering in the individual (active) operands. For instance, in a term $a \,|||\, a$, the left hand side may do a^+ and be ready to do a_1^-, after which the right hand side may also do a^+ and be ready to do a_1^-; however, in the combined behaviour the a_1^--actions should be distinguished since they refer to different start actions. We use a reverse numbering of outstanding actions in which the last action started is numbered 1, etc. This means that in the above example, the left hand side's a_1^--transition should be renumbered, in the combined behaviour, to a_2^-.

To do this kind of renumbering systematically, we append a mapping M to the synchronisation and refinement operators, which to every action a associates a string w of *active operand positions*: the i'th element of w equals p precisely if the i'th outstanding a-occurrence in the combined behaviour originated from the active operand in position p. For instance, in the behaviour $a \,|||\, a$ considered above, after an a^+-transition of the left hand side (operand 0) the associated string is "0" whereupon after the a^+-transition of the right hand side (operand 1), the associated string becomes "1 0" (note that the "1" is inserted *in front* of the string; this is in order to implement the reverse numbering of outstanding actions, mentioned above).

As is apparent from this example, we use natural numbers to identify active operand positions, starting with 0. Thus, the active operand positions of $t \parallel_A u$ are $\{0, 1\}$, and those of $t[a \to u, \vec{v}]$ are $\{0, \ldots, |\vec{v}|\}$. The association string for each action is given by a string w of active operand positions. We use $w[i]$ to denote the i'th element of w, $w \setminus i$ to denote the removal of the i'th element from w (where $1 \leq i \leq |w|$ in both cases) and $p \cdot w$ to denote the insertion of the operand position $p \in \mathbb{N}$ in front of w. Furthermore, we use mappings $M \colon \mathit{Act} \to \mathbb{N}^*$ (with only finitely many non-empty images) to associate such strings to actions, and use the following operations to adjust such mappings:

$$[a, p] \cdot M : \quad b \quad \mapsto \quad \begin{cases} p \cdot M(a) & \text{if } b = a \\ M(b) & \text{otherwise} \end{cases}$$

Table 9: Operational ST-semantics for simple actions and synchronisation

$$\frac{}{a \xrightarrow{a^+} a^-} \qquad \frac{}{a^- \xrightarrow{a_1^-} \mathbf{1}} \qquad \frac{}{\tau \xrightarrow{\tau} \tau^-} \qquad \frac{}{\tau^- \xrightarrow{\tau} \mathbf{1}}$$

$$\frac{t \xrightarrow{a^+} t' \quad a \notin A}{t \parallel_{A,M} u \xrightarrow{a^+} t' \parallel_{A,[a,0]\cdot M} u} \qquad \frac{u \xrightarrow{a^+} u' \quad a \notin A}{t \parallel_{A,M} u \xrightarrow{a^+} t \parallel_{A,[a,1]\cdot M} u'}$$

$$\frac{t \xrightarrow{a_i^-} t' \quad a \notin A \quad m = \hat{M}(0, a_i^-)}{t \parallel_{A,M} u \xrightarrow{a_m^-} t' \parallel_{A,M\setminus(a,m)} u} \qquad \frac{u \xrightarrow{a_i^-} u' \quad a \notin A \quad m = \hat{M}(1, a_i^-)}{t \parallel_{A,M} u \xrightarrow{a_m^-} t \parallel_{A,M\setminus(a,m)} u'}$$

$$\frac{t \xrightarrow{\tau} t'}{t \parallel_{A,M} u \xrightarrow{\tau} t' \parallel_{A,M} u} \qquad \frac{u \xrightarrow{\tau} u'}{t \parallel_{A,M} u \xrightarrow{\tau} t \parallel_{A,M} u'}$$

$$\frac{t \xrightarrow{\lambda} t' \quad u \xrightarrow{\lambda} u' \quad \mathit{type}(\lambda) \in A \cup \{\checkmark\}}{t \parallel_{A,M} u \xrightarrow{\lambda} t' \parallel_{A,M} u'}$$

$$M \setminus (a,i) : \quad b \quad \mapsto \quad \begin{cases} M(a) \setminus i & \text{if } b = a \\ M(b) & \text{otherwise.} \end{cases}$$

The purpose of M is to determine the renumbering of action occurrences. We use $\hat{M}(p, a_i^-)$ to denote the number that the i'th outstanding a-action of the p'th operand gets according to M. To determine this number, one must count the p's in $M(a)$: the index of the i'th occurrence of p in $M(a)$ is the number we are looking for. Formally: $\hat{M}(p, a_i^-) = m$ such that $M(a)[m] = p$ and $|\{j \leq m \mid M(a)[j] = p\}| = i$. For instance, if $M(a) = 2\,1\,0\,2$, then $\hat{M}(2, a_1^-) = 1$ and $\hat{M}(2, a_2^-) = 4$.

The operational rules for termination, choice, hiding and recursion are as before (see Table 2) and omitted here. The operational rules for simple actions and synchronisation are presented in Table 9. The meaning of the operational rules for "$t \parallel_{A,M} u$" is the following. When t performs a^+ ($a \notin A$), $M(a)$ is extended to reflect the renumbering of this new outstanding occurrence; on the other hand, when t (operand 0) performs a_i^-, the occurrence i is renumbered according to $M(a)$ and the occurrence is subsequently removed from $M(a)$. This removal automatically adjusts the renumbering scheme so that the remaining outstanding occurrences are always numbered consecutively, as required in Definition 5.6.

The number of rules for synchronisation has grown from 3 to 7; this is because we have to distinguish start actions, end actions and τ-actions. For the case of refinement we even have 10 rules, given in Table 10. There are four groups of rules concerning terms of the form $t[a \to u, \vec{v}]_M$ (compare also the event-based rules in Table 7):

- The first group deals with non-a-transitions of t; these are analogous to the non-synchronising transitions of parallel composition.

- The second group deals with a^+-transitions of t. This starts a new refinement; the initial

Table 10: Operational ST-semantics for the refinement operator

$$\frac{t \xrightarrow{b^+} t' \quad b \neq a}{t[a\to u,\ \vec{v}]_M \xrightarrow{b^+} t'[a\to u,\ \vec{v}]_{[b,0]\cdot M}} \qquad \frac{t \xrightarrow{b_i^-} t' \quad b \neq a \quad m = \hat{M}(0, a_i^-)}{t[a\to u,\ \vec{v}]_M \xrightarrow{b_m^-} t'[a\to u,\ \vec{v}]_{M\setminus(b,m)}}$$

$$\frac{t \xrightarrow{\lambda} t' \quad \lambda \in \{\tau, \checkmark\}}{t[a\to u,\ \vec{v}]_M \xrightarrow{\lambda} t'[a\to u,\ \vec{v}]_M}$$

$$\frac{t \xrightarrow{a^+} t' \quad u \xrightarrow{b^+} u'}{t[a\to u,\ \vec{v}]_M \xrightarrow{b^+} t'[a\to u,\ \vec{v}\,u']_{[b,|\vec{v}|+1]\cdot M}} \qquad \frac{t \xrightarrow{a^+} t' \quad u \xrightarrow{\tau} u'}{t[a\to u,\ \vec{v}]_M \xrightarrow{\tau} t'[a\to u,\ \vec{v}\,u']_M}$$

$$\frac{v_p \xrightarrow{b^+} v'}{t[a\to u,\ \vec{v}]_M \xrightarrow{b^+} t[a\to u,\ \vec{v} \pm (p, v')]_{[b,p]\cdot M}}$$

$$\frac{v_p \xrightarrow{b_i^-} v' \not\xrightarrow{\checkmark} \quad m = \hat{M}(p, b_i^-)}{t[a\to u,\ \vec{v}]_M \xrightarrow{b_m^-} t[a\to u,\ \vec{v} \pm (p, v')]_{M\setminus(b,m)}} \qquad \frac{v_p \xrightarrow{\tau} v' \not\xrightarrow{\checkmark}}{t[a\to u,\ \vec{v}]_M \xrightarrow{\tau} t[a\to u,\ \vec{v} \pm (p, v')]_M}$$

$$\frac{t \xrightarrow{a_p^-} t' \quad v_p \xrightarrow{b_i^-} v_p' \xrightarrow{\checkmark} \quad m = \hat{M}(p, b_i^-)}{t[a\to u,\ \vec{v}]_M \xrightarrow{b_m^-} t'[a\to u,\ \vec{v} \setminus p]_{M\setminus(b,m)}} \qquad \frac{t \xrightarrow{a_p^-} t' \quad v_p \xrightarrow{\tau} v_p' \xrightarrow{\checkmark}}{t[a\to u,\ \vec{v}]_M \xrightarrow{\tau} t'[a\to u,\ \vec{v} \setminus p]_M}$$

action of u is either a start action or an internal action. The list of pending refinements is extended, creating a new active operand position.

- The third group deals with transitions of a pending refinement v_p in $\vec{v}$ (where the index p corresponds to the active operand positions) that do not lead to a terminated state. These are also comparable to the non-synchronising transitions of parallel composition.
- The fourth group deals with transitions of a pending refinement v_p that do lead to a terminated state. Such a transition can only be labelled by an end action or an internal action. The corresponding end action of t can now also occur, and the remainder of the pending refinement can be discarded.

A careful investigation of the operational semantics reveals that it yields an ST-transition system in the sense of Definition 5.6. Moreover, (standard) bisimilarity, applied to the ST-semantics, is a congruence over Lang^{ST}. For an independent proof of this property as well as a complete axiomatisation of ST-bisimilarity over a large class of recursive processes see [24]. Furthermore, the semantics satisfies the finiteness properties claimed above. Let $[\![t]\!]^{op}_{ST}$ denote the operational ST-semantics of t.

Theorem 5.7 (see [24]) *Let $t, u \in \mathit{Lang}$ be arbitrary.*

(i) If t is a refinement-free term with finite state interleaving semantics, then $[\![t]\!]^{op}_{ST}$ is finite state.

(ii) If $[\![t]\!]^{op}_{ST}$ and $[\![u]\!]^{op}_{ST}$ are finite state, then $[\![t[a\rightsquigarrow u]]\!]^{op}_{ST}$ is finite state.

As a final result of this section, we have that the indirect and direct ST-semantics of *Lang* coincide. To be precise: if we denote the event-based operational semantics of a term $t \in \mathit{Lang}(\mathit{Evt})$ by $[\![t]\!]^{op}_{ev}$ and the conversion of an event transition system $\mathcal{T}$ into an ST-transition system by $ST(\mathcal{T})$, the following can be proved by induction on the structure of t:

Theorem 5.8 *For arbitrary (well-formed) $t \in \mathit{Lang}(\mathit{Evt})$, $ST([\![t]\!]^{op}_{ev}) \sim [\![\mathit{strip}(t)]\!]^{op}_{ST}$.*

Proof sketch. A state $(n, \vec{e})$ from $ST([\![t]\!]^{op}_{ev})$ corresponds to a term of $\mathit{Lang}(\mathit{Evt})$ (n is a derivative of t) plus a sequence $\vec{e}$ of initial events of n that are considered to have started but not yet finished. The events in $\vec{e}$ are thus independent in the sense of Definition 4.2. $(n, \vec{e})$ can therefore

- start a new action, say α, precisely if $n \xrightarrow{d,\alpha} n'$ for some d and n' such that $n' \xrightarrow{\{\vec{e}\}}$
- finish any of the actions corresponding to events in $\{\vec{e}\}$; the index is derived from the ordering in $\vec{e}$
- terminate if n is terminated and $\vec{e}$ is empty.

The correspondence with derivatives of $\mathit{strip}(t)$ in the ST-semantics is intricate especially for synchronisation and refinement terms: the mapping M used in the ST-semantics is obtained by unraveling the vector $\vec{e}$ in order to determine the source of each event (and vice versa to reconstruct the events from the mapping M). □

5.5 Application: A simple data base

To show the advantages of the above ST-semantics, both in providing a congruence for refinement and in yielding finite state systems, we return to the example of Section 4.6. Figure 9 shows the

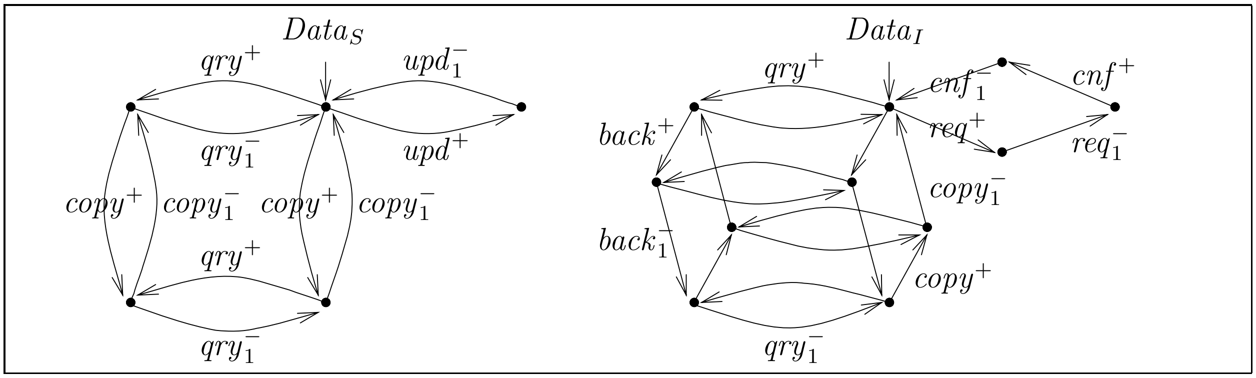


Figure 9: Operational ST-semantics of $Data_S$ and $Data_I$

ST-semantics of the abstract system $Data_S := State \parallel_{upd} Backup$ and the refinement $Data_I := Data_S[upd \to req; cnf][copy \to back; copy]$, where

$$\begin{aligned} State &:= (qry + upd); State \\ Backup &:= (copy + upd); Backup \ . \end{aligned}$$

The independence of the *qry*- and *copy*-actions, noted before in Section 4.6, is apparent from the ST-transition system by the fact that *qry* may be started during *copy*, i.e., between $copy^+$ and $copy_1^-$ (and vice versa); on the other hand, *qry* may not be started during *upd*. As a consequence, in $Data_I$ *qry* may interrupt the refinement of *copy* at any time, but may not interrupt the refinement of *upd*.

5.6 ω-completeness

As pointed out by Meyer in [105, 106], the issue of coarsest congruences is relevant to another, quite different subject as well, namely that of *ω-completeness* of equational theories. To explain this, we first recall the syntactic interpretation of action refinement discussed in Section 1.4 above, and note that the syntactic substitution of an action by a term is quite similar to the syntactic substitution of a process variable by a term. Also recall that (in the standard interpretation) two open terms (i.e., with free process variables) are considered equivalent iff their instantiations are equivalent; in other words, if the equivalence cannot be invalidated by some substitution of terms for process variables. But this in turn corresponds precisely (given the similarity between action substitution and variable substitution just pointed out) to the second congruence property listed at the beginning of this sub-section.

An equational theory is called ω-complete if any valid equation between open terms is derivable. We have seen just now that an equation between open terms is valid iff all instantiations are valid; this in turn is the case (interpreting variables as actions) iff the terms are equivalent under all action refinements, i.e., iff they are equivalent in the coarsest congruence. Summarising, we have:

> *In a setting where syntactic and semantic action refinement coincide, a complete axiomatisation of the coarsest refinement congruence contained in $\simeq$ gives rise to an ω-complete axiomatisation of $\simeq$, when interpreting all actions as variables.*

A similar idea is pursued in [134]: if we consider expressions built with variables (which stand for languages) and the operations of concatenation, choice (union) and parallel composition (shuffle), then substitution of languages for variables can be seen as action refinement; validity of equations of such expressions can be checked using ST trace semantics. Analogously, if variables take values on words, validity of equations can be checked using *swap*-equivalence.

6 Semantic versus Syntactic Substitution

We have stated in the introduction that there are two fundamentally different ways to regard action refinement: as an operator in the signature, or as a homomorphism (in fact, an automorphism) of the algebra establishing a transformation of terms. Since the semantic effect of the action refinement operator $t[a \rightsquigarrow u]$ is understood as the substitution, in some sense, of the action a by the behaviour of u in the *semantic* model of t, whereas a homomorphism effectively replaces a by u *syntactically* in t, the two interpretations are actually quite close; yet they do not always coincide. In the previous sections, we have considered the semantic interpretation of action refinement; below, we study the syntactic interpretation and investigate the difference between the two. The material of this section is mainly based on [69].

Note that if the two interpretations coincide, we automatically have an axiomatisation for the action refinement operator: namely, it distributes over all other operators. In fact, this distribution property is a strong point in favour of the syntactic interpretation. On the other hand, the precise notion of syntactic substitution has to be considered carefully, as it may depend on the operators available in the language and on the kind of refinement we are interested in (atomic or non-atomic).

In the above discussion, we first relied on the algebraic notion of an automorphism and then stated that this corresponds to the syntactic substitution of actions by terms. The connection between the concepts of automorphism and syntactic substitution is established by regarding the term algebra generated by the signature as the algebra over which the automorphism is defined. However, there are several respects in which this algebraic concept does not quite cover the notion of syntactic substitution we need.

- It does not tell us how to deal with recursion, which is not an operator in the usual algebraic sense but rather a fixpoint constructor, i.e., a higher-order operator. As it turns out, the straightforward syntactic substitution within the body of the recursion (which does not affect the process variables bound by recursion) gives rise to a satisfactory solution.

- It does not take operators into account that *bind* actions, such as hiding and (to some degree) synchronisation. Strictly speaking, an automorphism f on the term algebra would transform each term t/A into $op(f(t))$, where op is the f-image of the unary operator $\_/A$; the natural choice of op is $\_/A'$ for some A'. As we will see below, in some cases this is not an appropriate way to model action refinement.

Hence, although the intuition of an automorphism is initially helpful to understand action refinement algebraically, it is actually only an approximation. In the remainder of this section we will no longer appeal to this intuition.

6.1 Finite sequential systems

Let us first consider again the simple sequential language of Section 2, restricted, moreover, to closed, finite terms (i.e., containing no process variables or recursion operators). We call a term *flat* if it contains no occurrence of the refinement operator. In defining the syntactic substitution of a term for an action in $Lang^{seq,fin}$, we already encounter the problem with respect to the hiding operator, mentioned above.

Table 11: Sort and action substitution

t	$\mathcal{S}(t)$	$t\{u/a\}$
α	$\begin{cases} \emptyset & \text{if } \alpha = \tau \\ \{\alpha\} & \text{otherwise} \end{cases}$	$\begin{cases} u & \text{if } \alpha = a \\ \alpha & \text{otherwise} \end{cases}$
t_1/A	$\mathcal{S}(t_1) \setminus A$	$t_1\{u/a\}/A \quad \text{if } a \notin A \text{ and } \mathcal{S}(u) \cap A = \emptyset$
$t_1[b \rightsquigarrow t_2]$	$\begin{cases} \mathcal{S}(t_1) & \text{if } b \notin \mathcal{S}(t_1) \\ (\mathcal{S}(t_1) \setminus b) \cup \mathcal{S}(t_2) & \text{otherwise} \end{cases}$	$t_1\{u/a\}[b \rightsquigarrow t_2\{u/a\}] \quad \text{if } a \neq b \text{ and } b \notin \mathcal{S}(u)$
$op(t_1, \ldots, t_n)$	$\mathcal{S}(t_1) \cup \cdots \cup \mathcal{S}(t_n)$	$op(t_1\{u/a\}, \ldots, t_n\{u/a\}) \quad \text{(other operators)}$
x	$\mathcal{S}_x$	x
$\mu x.t_1$	$\mathcal{S}_x$	$\mu x.(t_1\{u/a\})$

Example 6.1 *Suppose we want to substitute the action a by the term $b;c$ within t.*

1. *If $t = a/\{a\}$ (specifying an invisible action), then a naive substitution gives rise to $(b;c)/\{a\}$. Hence, after refinement we suddenly obtain visible behaviour where before there was none.*
2. *A slightly more sophisticated refinement of $t = a/\{a\}$ would yield $(b;c)/\{b,c\}$ (i.e., the set of hidden actions is adapted as well). This is still unexpected, since the refined system has two invisible steps instead of just one (and thus the refinement is not well-defined up to strong bisimilarity). Worse yet, using this principle $(a;b)/\{a\}$ would give rise to $(b;c;b)/\{b,c\}$, where the abstract b-action disappears even though it is not refined itself.*
3. *If $t = a/\{b\}$ (specifying the execution of a), then a naive substitution gives rise to $(b;c)/\{b\}$. This starts with an invisible action rather than the expected execution of b.*

Technically, these problems all have to do with the fact that, in our semantics, hiding acts like a *binder* of the hidden actions, just like restriction does in CCS (see [9]). Moreover, precisely the same problem occurs for action substitution within non-flat terms, since the refinement operator is a binder, too: in the term $t[a \rightsquigarrow u]$, all occurrences of a in t are bound by the refinement. By analogy with the syntactic substitution of free variables within the body of recursion, defined in Table 1, here we have to make sure that there is no confusion of "free" and "bound" actions during substitution. For that purpose, we first have to have a clear notion of the "free actions" of a term. In the terminology of Milner [108], these are given by the *sort* (called the *alphabet* in [11, 88]). The term sort is defined in Table 11. The sort is somewhat comparable to the *type* of a term in functional languages. It is an upper bound to the actions that the process will do during its lifetime. This is expressed by the following key proposition (in the type analogy corresponding to *subject reduction*):

Proposition 6.2 *If $t \xrightarrow{\alpha} t'$ then $\mathcal{S}(t') \cup \{\alpha \mid \alpha \neq \tau\} \subseteq \mathcal{S}(t)$.*

To guarantee the necessary absence of confusion, we take a rather drastic solution: the syntactic substitution of a term for an action will be undefined whenever a "bound action", i.e., a hidden action or an action being refined, either equals the action being replaced by substitution or is contained in the sort of the term substituted for it. The definition of action substitution is given in Table 11; see also [9].

The partiality of action substitution is subsequently smoothed out of existence as follows. Let $\equiv$ be the smallest congruence over $\mathit{Lang}^{\mathit{seq}}$ such that

$$\begin{array}{rcl} t/(A \uplus \{a\}) & \equiv & t\{b/a\}/(A \uplus \{b\}) \quad (b \notin \mathcal{S}(t)) \\ t[a \to u] & \equiv & t\{b/a\}[b \to u] \quad (b \notin \mathcal{S}(t)) \end{array}$$

for an arbitrary b such that the substitutions on the right hand side are defined. It is not difficult to prove that, given a large enough universe of actions Act, for all $t\{u/a\}$ there is a $t' \equiv t$ such that $t'\{u/a\}$ is defined; moreover, as a consequence of the fact that $\equiv$ is a congruence, if $t \equiv t'$ then $t\{u/a\} \equiv t'\{u/a\}$ whenever both are defined. In other words, action substitution is total and well-defined modulo $\equiv$. Since $\equiv$ is finer than $\sim$ and in fact finer than any semantic equivalence we consider in this chapter, it follows that we may always pick an appropriate $\equiv$-representative such that action substitution is defined. In the sequel, we implicitly assume that such a representative has been chosen, and proceed as if action substitution is total. (A similar assumption concerning free variables is made in [14] and elsewhere in the literature on functional programming.)[1]

We now set out to show that (with the definition of action substitution developed above), in $\mathit{Lang}^{\mathit{seq},\mathit{fin}}$ there is no difference between the semantic interpretation of the action refinement operator (see Table 2) and its interpretation as action substitution (see Table 11).

To formulate the result we are after, we define a *flattening* function over sequential terms, which removes all refinement operators from a given process expression by performing the corresponding syntactic substitution.

Definition 6.3 *The* flattening *of a term $t \in \mathit{Lang}^{\mathit{seq},\mathit{fin}}$, denoted $\mathit{flat}(t)$, is defined inductively on the structure of t as follows:*

$$\begin{array}{rcl} \mathit{flat}(\mathit{op}(t_1, \ldots, t_n)) & := & \mathit{op}(\mathit{flat}(t_1), \ldots, \mathit{flat}(t_n)) \\ \mathit{flat}(t[a \to u]) & := & \mathit{flat}(t)\{\mathit{flat}(u)/a\} \ . \end{array}$$

The coincidence of syntactic and semantic substitution is then concisely stated by the equation $t = \mathit{flat}(t)$. The first result of this kind is the following:

Theorem 6.4 *$t \sim \mathit{flat}(t)$ for any $t \in \mathit{Lang}^{\mathit{seq},\mathit{fin}}$.*

Proof sketch. The idea is to prove that refinement satisfies the defining equations of substitution (Table 11) except for that for refinement (and recursion); that is, to show that the following equations hold up to $\sim$:

$$\begin{array}{rcl} \alpha[a \to u] & = & \begin{cases} u & \text{if } \alpha = a \\ \alpha & \text{otherwise} \end{cases} \\ (t/A)[a \to u] & = & t[a \to u]/A \quad \text{if } a \notin A \text{ and } \mathcal{S}(u) \cap A = \emptyset \\ \mathit{op}(t_1, t_2)[a \to u] & = & \mathit{op}(t_1[a \to u], t_2[a \to u]) \quad \text{for other operators } \mathit{op} \ . \end{array}$$

(These equations in fact hold up to $\sim_{\mathsf{ev}}$.) Using these equations, it is straightforward to show that for all *flat* terms $t \in \mathit{Lang}^{\mathit{seq},\mathit{fin}}$ the following holds up to $\sim$ (resp. $\sim_{\mathsf{ev}}$):

$$t[a \to u] = t\{u/a\}$$

(by induction over the structure of t). The actual theorem then follows by the definition of *flat* and the fact that $\sim$ is a congruence (using a further induction over the structure of t). □

[1] An alternative solution would have been to build the necessary conversion of bound actions into the definition of action substitution, as we have done for variable substitution in Table 1. However, we find the present definition somewhat clearer.

6.2 Recursive sequential systems

In extending Theorem 6.4 the full sequential language Lang^{seq}, we encounter several problems. The major one of these is that it is in general not possible to flatten terms of Lang^{seq} in which refinement and recursion are arbitrarily mixed.

Example 6.5 *Consider the term* $t = \mu x.(a; x[a \rightarrow a; b])$. *This specifies a process having as partial runs all prefixes of the infinite string* $a\, b^0\, a\, b^1\, a\, b^2\, a\, b^3 \cdots$. *This set of partial runs is not context-free; it follows that the behaviour of* t *cannot be specified in the flat fragment of* Lang^{seq}, *not even modulo trace equivalence, let alone modulo any stronger semantics. In other words,* t *admits no flattening within* Lang^{seq}.

A similar situation occurs with $t = \mu x.a[a \rightarrow a; (b + x)]$, *where recursion and refinement are mixed in a different way but to much the same effect.*

In order to avoid problems of this kind, for the purpose of this section we restrict ourselves to instances of the refinement operator in which both operands of refinement are closed terms.[2]

Another, more technical issue is that the notions of sort and syntactic action substitution have yet to be defined for process variables and recursive terms.

- To define the sort of open terms, we impose the requirement that all process variables are implicitly sorted; i.e., there is a pre-defined sort $\mathcal{S}_x \subseteq \mathit{Act}$ for all $x \in \mathit{Var}$. Furthermore, $\mathcal{S}(\mu x.t)$ is set to $\mathcal{S}_x$. We then impose the following *well-sortedness* condition on recursive terms (in addition to the existing well-formedness conditions): $\mu x.t$ is well-sorted only if $\mathcal{S}(t) \subseteq \mathcal{S}_x$.
- With respect to action substitution, the natural solution is to define $(\mu x.t)\{u/a\}$ to equal $\mu x.t\{u/a\}$ and $x\{u/a\}$ to equal x. At first sight, this invalidates the correspondence between action refinement and action substitution, since it gives rise to $\mu x.(a; x\{b/a\}) = \mu x.a; x$ which clearly does not have the same behaviour as $\mu x.(a; x[a \rightarrow b])$. However, we are saved by the new requirement that the operands of refinement must be closed terms; this precisely rules out terms of the form $x[a \rightarrow b]$.

The resulting definitions of the sort and action substitution for process variables and recursive terms were already given in Table 11.

Well-formedness. To summarise the well-formedness conditions in this section:

- No virgin operand contains an occurrence of an auxiliary operator;
- The operands of refinement must be closed terms;
- Recursion is allowed on guarded variables only;
- Recursion must be well-sorted.

The "correctness criterion" for the sort of recursive terms is that Proposition 6.2 now also holds for all closed terms of Lang^{seq}. As for action substitution, to prove the corresponding extension of

[2] This is essentially the same restriction we used in [69], except that there we worked with process environments rather than recursion operators. In that paper, we justified the restriction by another well-formedness condition, namely that the sorts of t and u in $t[a \rightarrow u]$ have to be disjoint.

Theorem 6.4 we have to adapt the notion of flattening to process variables and recursive terms as well:

$$\begin{array}{rcl} \mathit{flat}(x) & = & x \\ \mathit{flat}(\mu x.t) & = & \mu x.\mathit{flat}(t) \ . \end{array}$$

We now come to the correspondence result for the full sequential language.

Theorem 6.6 *$t \sim \mathit{flat}(t)$ for any $t \in \mathit{Lang}^{\mathit{fin}}$.*

Proof sketch. We have seen that Theorem 6.4 essentially consisted of proving $t[a \rightarrow u] \sim t\{u/a\}$ by induction on the structure of t. Since t is bound to be closed, for the purpose of the current theorem we only have to extend the existing proof with the case for $t = \mu x.t_1$; i.e., we show

$$(\mu x.t_1)[a \rightarrow u] \sim \mu x.t_1[a \rightarrow u] \ .$$

The proof consists of showing that

$$\rho = \{(u_1\{\mu x.t_1/x\}[a \rightarrow u], u_1\{\mu x.t_1[a \rightarrow u]/x\}) \mid \mathit{fv}(u_1) \subseteq \{x\}\}$$

is a bisimulation. The proof obligation follows as a special case (taking $u_1 = x$). □

6.3 Atomic refinement

We now consider the language $\mathit{Lang}^{\mathit{atom}}$ with atomic refinement, discussed in Section 3 (Page 23). This entails taking two new operators into account: parallel composition (without synchronisation) and the atomiser. The definition of the sort and action substitution for these new operators follows from the "other operators"-clause of Table 11.

The definition of flattening in $\mathit{Lang}^{\mathit{atom}}$ is a variation on the one we had before: the difference is that the effect of *atomically* refining an action a by u is not the straight execution of u but rather the *atomic* execution of u, i.e., the execution of $\langle u \rangle$. This gives rise to a function $\mathit{flat}^{\mathit{atom}}$, whose definition for the flat fragment of $\mathit{Lang}^{\mathit{atom}}$ is analogous to the definition of *flat* (Definition 6.3) but for the refinement operator is given by

$$\mathit{flat}^{\mathit{atom}}(t[a \rightarrow u]) := \mathit{flat}^{\mathit{atom}}(t)\{\langle \mathit{flat}^{\mathit{atom}}(u) \rangle / a\} \ .$$

This is actually the key point distinguishing atomic and non-atomic refinement. We obtain the analogous result to Theorem 6.6; the proof is also analogous, and hence omitted.

Theorem 6.7 *$t \sim \mathit{flat}^{\mathit{atom}}(t)$ for any $t \in \mathit{Lang}^{\mathit{atom}}$*

6.4 Synchronisation

In the full language with non-atomic refinement, we encounter the operator for parallel composition with synchronisation, indexed by the set of synchronising actions. When we do action substitution in such a term, this explicit occurrence of action names gives rise to the same type of confusion as for hiding (see Example 6.1).

Example 6.8 *Consider the replacement of a by $b; c$ in the term $t \in \mathit{Lang}$.*

Table 12: Sort and action substitution for parallel composition with synchronisation

t	$\mathcal{S}(t)$	$t\{u/a\}$
$t_1 \parallel_A t_2$	$\mathcal{S}(t_1) \cup \mathcal{S}(t_2)$	$\begin{cases} t_1\{u/a\} \parallel_{(A\setminus\{a\})\cup\mathcal{S}(u)} t_2\{u/a\} & \text{if } a \in A \text{ and } \mathcal{S}(u) \cap \mathcal{S}(t) \subseteq A \\ t_1\{u/a\} \parallel_A t_2\{u/a\} & \text{if } a \notin A \text{ and } \mathcal{S}(u) \cap A = \emptyset \end{cases}$

1. *If $t = a\|_a a$, specifying a single execution of a, then a naive substitution gives rise to $t\{b;c/a\} = b;c \parallel_a b;c$, which can do two times $b;c$ in parallel and hence is not an intuitively correct refinement of t.*

2. *A slightly more sophisticated substitution in t above yields $b;c\|_{b,c}b;c$, which is more reasonable; however, according to the same principle, $c;a \parallel_a a$, which can perform c, gives rise to $c;b;c \parallel_{b,c} b;c$, which is deadlocked.*

3. *If $t = a;c\|_{b,c}c$, then the substitution gives rise to $t\{b;c/a\} = b;c;c\|_{b,c}c$; this is not a reasonable refinement since the former can do $a;c$ whereas the latter is deadlocked.*

However, in contrast to hiding, in the case of synchronisation the actions are not bound; rather, our operational semantics (Section 4) specifies that actions remain visible after synchronisation, and are therefore still available for further synchronisation or, indeed, refinement. For this reason, although action substitution for parallel composition needs a side condition that is comparable to the one for hiding in Table 11, we cannot argue (as we did before) that substitution is still totally defined modulo $\equiv$. Sort and action substitution for synchronisation are defined in Table 12. Note that when replacing a synchronising action, the synchronisation set is adapted accordingly; this corresponds to the "more sophisticated" solution pointed out in Example 6.8.2.

However, our problems are not yet over. Consider $t = t_1 \parallel_A t_2$. As long as we do not substitute actions in A, and moreover take care that A and $\mathcal{S}(u)$ are disjoint, the desired distributivity property $t[a{\to}u] = t_1[a{\to}u] \parallel_A t_2[a{\to}u]$ holds up to $\sim_{\mathsf{ev}}$; hence everything goes smoothly (see the proof sketch of Theorem 6.4). However, when a occurs in A, then $t\{u/a\}$ and $t[a{\to}u]$ may be different.

Example 6.9 *We show some t and u such that $t[a{\to}u] \not\sim t\{u/a\}$.*

- *Consider $t = a \parallel_a a$ and $u = b + b;c$. t always terminates after having performed a, and so $t[a{\to}u]$ terminates after either b or $b\,c$, but $t\{u/a\}$ does not always terminate:*

  $$t\{u/a\} = (b + b;c) \parallel_{b,c} (b + b;c) \xrightarrow{b} \mathbf{1} \parallel_{b,c} c$$

  where the right hand term is deadlocked.

- *Consider $t = (a \mathbin{|||} a;b) \parallel_a (a \mathbin{|||} a;b)$ and $u = c;d$. In $t[a{\to}u]$, there is a c-transition after which in every state reachable by $c\,d$, either no b or two consecutive b's are enabled:*

  $$t[a{\to}u] = ((_0a \mathbin{|||} {_1a}) \parallel_a (_2a \mathbin{|||} {_3a};{_4b}))[a{\to}{_5c};{_6d}] \xrightarrow{(e,5),c} ((_0a \mathbin{|||} {_1a}) \parallel_a (_2a \mathbin{|||} {_3a};{_4b}))[a{\to}{_5c};{_6d}, e{\to}{_6d}]$$

  where $e = ((0,*),(2,*))$. On the other hand, for all $t\{u/a\} \xrightarrow{c} t'$ there is a $t' \xrightarrow{c}\xrightarrow{d}\xrightarrow{b} t''$ such that $t'' \not\xrightarrow{b}$: in particular, after $t\{u/a\} \xrightarrow{a} (d \mathbin{|||} c;d;b) \parallel_{c,d} (d \mathbin{|||} c;d;b)$ $(= t')$, which intuitively corresponds to the transition above, we can continue with

  $$t' \xrightarrow{c} (d \mathbin{|||} d;b) \parallel_{c,d} (d \mathbin{|||} d;b) \xrightarrow{d} (\mathbf{1} \mathbin{|||} d;b) \parallel_{c,d} (d \mathbin{|||} b) \xrightarrow{b} (\mathbf{1} \mathbin{|||} d;b) \parallel_{c,d} (d \mathbin{|||} \mathbf{1}) \quad (= t'')$$

  such that $t'' \not\xrightarrow{b}$.

Table 13: Auto-concurrent actions and determinism

t	$\mathcal{C}(t)$	$\mathcal{D}(t)$
$\mathbf{0}$	$\emptyset$	**true**
$\mathbf{1}$	$\emptyset$	**true**
α	$\{\alpha \mid \alpha \neq \tau\}$	$\alpha \neq \tau$
$t_1 + t_2$	$\mathcal{C}(t_1) \cup \mathcal{C}(t_2)$	$\mathcal{D}(t_1) \wedge \mathcal{D}(t_2) \wedge \nexists a\colon (t_1 \xrightarrow{a}) \wedge (t_2 \xrightarrow{a})$
$t_1; t_2$	$\mathcal{C}(t_1) \cup \mathcal{C}(t_2)$	$\mathcal{D}(t_1) \wedge \mathcal{D}(t_2)$
$t_1 \parallel_A t_2$	$\mathcal{C}(t_1) \cap \mathcal{C}(t_2)$ $\cup\, (\mathcal{C}(t_1) \cup \mathcal{C}(t_2)) \setminus A$ $\cup\, (\mathcal{S}(t_1) \cap \mathcal{S}(t_2)) \setminus A$	$\mathcal{D}(t_1) \wedge \mathcal{D}(t_2) \wedge (\mathcal{S}(t_1) \cap \mathcal{S}(t_2) \subseteq A)$
t_1/A	$\mathcal{C}(t_1) \setminus A$	$\mathcal{D}(t_1) \wedge (\mathcal{S}(t_1) \cap A = \emptyset)$
$t_1[a \to t_2]$	$\begin{cases} (\mathcal{C}(t_1) \setminus a) \cup \mathcal{S}(t_2) & \text{if } a \in \mathcal{C}(t_1) \\ \mathcal{C}(t_1) \cup \mathcal{C}(t_2) & \text{if } a \in \mathcal{S}(t_1) \setminus \mathcal{C}(t_1) \\ \mathcal{C}(t_1) & \text{otherwise} \end{cases}$	$\mathcal{D}(t_1) \wedge (a \notin \mathcal{S}(t_1) \vee \mathcal{D}(t_2))$
x	$\mathcal{C}_x$	$\mathcal{D}_x$
$\mu x.t_2$	$\mathcal{C}_x$	$\mathcal{D}_x$

In [69] we discussed necessary and sufficient conditions under which refinement and substitution of a synchronising action give rise to event bisimilar models. (To be precise, in [69] we considered isomorphism of configurations; as we stated before, however, this actually coincides with event bisimilarity.) The crucial part is to find conditions for the following distributivity property (which is analogous to the rule for action substitution in Table 12):

$$(t_1 \parallel_{A \uplus \{a\}} t_2)[a \to u] \sim_{\mathsf{ev}} t_1[a \to u] \parallel_{A \uplus \mathcal{S}(u)} t_2[a \to u] \ .$$

This is indeed precisely the property that fails to be satisfied in both instances of Example 6.9. The solution given in [69] consists of a number of fairly involved semantic constraints, which are in general undecidable; rather than repeating the constraints here, we give a decidable "approximation" or "estimate", which is a simplification of the one in [69].

The decidable approximations, which are given in Table 13 in the form of functions over *Lang*, provide sufficient conditions for the actual semantic constraints; the idea is the same as for the sort of a term (see Tables 11 and 12), which provides an approximation of the actions that may be performed by the term (see Proposition 6.2). The intention of the functions defined in Table 13 is as follows:

- $\mathcal{C}(t) \subseteq \mathit{Act}$ approximates the set of auto-concurrent actions of t, i.e., the ones that are executed in more than one parallel component without being synchronised. This is an over-estimate, due to the fact that one or more occurrences of an action may fail to be executed (as for a in $a \parallel_b b; a$) or are serialised (as for a in $a; b \parallel_b b; a$).
- $\mathcal{D}(t) \in \mathbb{B}$ is a boolean indicating whether t is deterministic and contains no internal actions. This is an under-estimate: terms for which $\mathcal{D}(t)$ does not hold may yet be deterministic —for example, $a \parallel_b b; a$.

For recursive terms, the definition of these functions relies on pre-defined sets $\mathcal{C}_x$ and predicates $\mathcal{D}_x$ for all $x \in \mathit{Var}$, just as the definition of the sort in Table 11. We extend the well-sortedness condition to imply that these sets and predicates are in correspondence with the recursion body:

Extended well-sortedness. t is well-sorted if for all subterms $\mu x.u$, in addition to $\mathcal{S}(u) \subseteq \mathcal{S}_x$ we also have $\mathcal{C}(u) \subseteq \mathcal{C}_x$ and $\mathcal{D}_x \Rightarrow \mathcal{D}(u)$.

In the remainder of this section, we only consider terms that are well-sorted in this extended sense. The following proposition expresses the semantic properties that are guaranteed by the functions in Table 13.

Proposition 6.10 *Let $t \in \mathit{Lang}$ be arbitrary.*

- *$a \in \mathcal{C}(t)$ if there exists a t' reachable from t such that $t' \xrightarrow{e_1,a}\xrightarrow{e_2,a}$ and $t' \xrightarrow{e_2,a}\xrightarrow{e_1,a}$ (in the event-based semantics of Section 4).*
- *$\mathcal{D}(t)$ implies that for all t' reachable from t, $t' \not\xrightarrow{\tau}$ and if $t' \xrightarrow{a} t_1''$ and $t' \xrightarrow{a} t_2''$ then $t_1'' = t_2''$.*

For the proof see [69]. An important further property of these functions is that they are insensitive to the flattening of terms, in the following sense:

Proposition 6.11 *If $t \in \mathit{Lang}$ such that $\mathit{flat}(t)$ is defined, then $\mathcal{S}(t) = \mathcal{S}(\mathit{flat}(t))$, $\mathcal{C}(t) = \mathcal{C}(\mathit{flat}(t))$ and $\mathcal{D}(t) \Leftrightarrow \mathcal{D}(\mathit{flat}(t))$.*

For the proof see [69]. We now define the concept of *u/a-compatibility* over the flat fragment of *Lang*, which establishes a sufficient condition guaranteeing that the refinement of a by u and the substitution of a by u coincide.

1. t/A is u/a-compatible if $a \notin A$, $\mathcal{S}(u) \cap A = \emptyset$ and t is u/a-compatible;
2. $t = t_1 \parallel_A t_2$ is u/a-compatible if t_1 and t_2 are u/a-compatible and one of the following holds:
   (a) $a \notin A$ and $\mathcal{S}(u) \cap A = \emptyset$;
   (b) $a \in A \setminus \mathcal{S}(t)$ and $\mathcal{S}(u) \cap \mathcal{S}(t) = \emptyset$;
   (c) $a \in A \setminus \mathcal{C}(t)$, $\mathcal{S}(u) \cap \mathcal{S}(t) = \emptyset$ and $\mathcal{D}(u)$;
   (d) $a \in A$ and $u = \sum_{i=1}^{n} a_i$ such that $\mathcal{S}(u) \cap \mathcal{S}(t) = \emptyset$;
3. $\mu x.t$ is u/a-compatible if t is u/a-compatible;
4. For all other operators op, $\mathit{op}(t_1, \ldots, t_n)$ is u/a-compatible if all t_i are u/a-compatible.

The interesting part is the requirement for parallel composition. This consists of the case where the substituted action is not in the synchronisation set (Clause 3(a)), or, if it is in the synchronisation set, the case where it is never performed (Clause 3(b)), where it is performed but not auto-concurrent, and the refinement is deterministic (Clause 3(c)) and where it is auto-concurrent and the refinement is a choice of atomic actions (Clause 3(d)). As a consequence of the main theorem of [69], we have the following property:

Proposition 6.12 *If $t \in \mathit{Lang}$ is flat and u/a-compatible, then $t\{u/a\}$ is defined and $t[a \to u] \sim_{\mathsf{ev}} t\{u/a\}$.*

Building on substitution compatibility, we now define the property of *reducibility* of terms, which establishes a sufficient condition guaranteeing that every action refinement operator in a term can be converted into an action substitution (i.e., the term can be flattened) without changing its semantics.

- $\mu x.t$ is reducible if t is reducible;
- $t[a \to u]$ is reducible if t and u are reducible and $\mathit{flat}(t)$ is u/a-compatible.
- For all other operators op, $op(t_1, \ldots, t_n)$ is reducible if all t_i are reducible.

Here, the interesting case is that of refinement. The following theorem extends Theorem 6.6 to *Lang*: a term and its flattening coincide *if* the term is reducible. The proof is by induction on the term structure, using Proposition 6.12.

Theorem 6.13 *If $t \in \mathit{Lang}$ is reducible, then $\mathit{flat}(t)$ is defined and $t \sim_{\mathsf{ev}} \mathit{flat}(t)$.*

By "tuning" the functions $\mathcal{S}$, $\mathcal{C}$ and $\mathcal{D}$, it is, of course, possible to come to an improved characterisation of reducibility, i.e., one which provides a better estimate of the underlying semantic property; however, as mentioned above, decidable necessary and sufficient criteria for the coincidence of syntactic and semantic refinement do not exist.

6.5 Application: A simple data base

We apply Theorem 6.13 to the example of Sections 4.6 and 5.5. This consists of a process $\mathit{Data}_S := \mathit{State} \parallel_{\mathit{upd}} \mathit{Backup}$, where

$$\begin{aligned} \mathit{State} &:= (\mathit{qry} + \mathit{upd}); \mathit{State} \\ \mathit{Backup} &:= (\mathit{copy} + \mathit{upd}); \mathit{Backup} \ . \end{aligned}$$

We then refine *upd* and *copy*, resulting in $\mathit{Data}_I := \mathit{Data}_S[\mathit{upd} \to \mathit{req}; \mathit{cnf}][\mathit{copy} \to \mathit{back}; \mathit{copy}]$. We show Data_I to be reducible, in the sense defined above. For this purpose, we must check that the terms $\mathit{Data}_S[\mathit{upd} \to \mathit{req}; \mathit{cnf}]$ and $\mathit{back}; \mathit{copy}$ are reducible and that $\mathit{flat}(\mathit{Data}_S[\mathit{upd} \to \mathit{req}; \mathit{cnf}])$ is $(\mathit{back}; \mathit{copy})/\mathit{copy}$-compatible.

- To see that $\mathit{Data}_S[\mathit{upd} \to \mathit{req}; \mathit{cnf}]$ is reducible, we must check that Data_S and $\mathit{req}; \mathit{cnf}$ are reducible (which is trivially true, since both are already flat) and that $\mathit{flat}(\mathit{Data}_S)$ $(= \mathit{Data}_S)$ is $(\mathit{req}; \mathit{cnf})/\mathit{upd}$-compatible. The latter comes down to checking that $\mathit{State} \parallel_{\mathit{upd}} \mathit{Backup}$ (the body of Data_S) is $(\mathit{req}; \mathit{cnf})/\mathit{upd}$-compatible.

  For this purpose, we must check that *State* and *Backup* are $(\mathit{req}; \mathit{cnf})/\mathit{upd}$-compatible (which is trivially true, since neither contains parallel composition) and that *upd* and $\mathit{req}; \mathit{cnf}$ are compatible with the synchronisation set. This is indeed the case, since $\mathcal{C}(\mathit{State} \parallel_{\mathit{upd}} \mathit{Backup}) = \emptyset$ and hence $\mathit{upd} \in \{\mathit{upd}\} \setminus \mathcal{C}(\mathit{State} \parallel_{\mathit{upd}} \mathit{Backup})$, and furthermore, $\mathcal{D}(\mathit{req}; \mathit{cnf})$; hence compatibility clause 3(c) holds.

- $\mathit{back}; \mathit{copy}$ is trivially reducible, since it is a flat term.

- $\mathit{flat}(\mathit{Data}_S[\mathit{upd} \to \mathit{req}; \mathit{cnf}]) = \mathit{Data}'$ where

  $$\begin{aligned} \mathit{Data}' &:= \mathit{State}' \parallel_{\mathit{req},\mathit{cnf}} \mathit{Backup}' \\ \mathit{State}' &:= (\mathit{qry} + \mathit{req}; \mathit{cnf}); \mathit{State}' \\ \mathit{Backup}' &:= (\mathit{back}; \mathit{copy} + \mathit{req}; \mathit{cnf}); \mathit{Backup}' \ . \end{aligned}$$

  To see that this is $(\mathit{back}; \mathit{copy})/\mathit{copy}$-compatible, the main point is checking that $\mathit{State}' \parallel_{\mathit{req},\mathit{cnf}} \mathit{Backup}'$ is so. This is indeed the case, since $\mathit{copy} \notin \{\mathit{upd}\}$; hence compatibility clause 3(a) holds.

According to Theorem 6.13, since $Data_I$ is reducible it can be flattened, i.e., it is equivalent (event bisimilar, and hence also ST-bisimilar and interleaving bisimilar) to the process obtained by carrying out the refinements syntactically, given by $Data_I^{flat} := State^{flat} \parallel_{req,cnf} Backup^{flat}$ where

$$\begin{array}{rcl} State^{flat} & := & (qry + req; cnf); State^{flat} \\ Backup^{flat} & := & (back; copy + req; cnf); Backup^{flat} \ . \end{array}$$

This can be verified by comparing the behaviour of $Data_I^{flat}$ with that of $Data_I$ as depicted in Figure 6 (taking the event-based semantics modulo event bisimilarity) and Figure 9 (taking the ST-semantics modulo standard bisimilarity).

7 Dependency-based action refinement

Although intuitively, action refinement is closely connected to the concept of top-down design, in practice the steps one would like to make in top-down design do not always correspond completely to the constructions allowed by action refinement. An example is the sequential ordering of tasks at different levels of abstraction. On an abstract level of design, one may specify that two activities, which on that level are regarded as atomic, are causally ordered —for instance because one of them produces an output that is consumed by the other. When one refines this specification in a top-down fashion, taking into account, among other things, that the previously atomic actions actually consist of several sub-activities, then the causal dependency does not necessarily hold between *all* respective sub-activities; rather, it may be the case that the output is produced somewhere *during* the refinement of the first action (not necessarily at the very end), and consumed *during* the refinement of the second (not necessarily at the very beginning).

Motivating example As a clarifying example (originally inspired by [99]), consider a buffer of capacity 1, which allows the alternating *put* and *get* of a data value. In fact, consider the case where only a single *put* and *get* occur. Abstractly, it is clear that a causal dependency exists between *put* and *get*, induced by the flow of data. Now consider a design step driven by the fact that the data values are too large for a single buffer cell, and consequently it is decided to use two buffers in parallel: data values are split in two and a piece is put in either (implementation-level) buffer. This comes down to refining *put* by $put_1 \mid\mid\mid put_2$ and *get* by $get_1 \mid\mid\mid get_2$.

If we treat this design step through the application of the refinement operator to the original specification, $put; get$, we arrive at the behaviour

$$(put; get)[put \rightarrow put_1 \mid\mid\mid put_2][get \rightarrow get_1 \mid\mid\mid get_2] \sim_{\mathsf{ev}} (put_1 \mid\mid\mid put_2); (get_1 \mid\mid\mid get_2) \ .$$

This, however, is *not* necessarily the intended implementation-level behaviour: there is an ordering between put_1 and get_2 and between put_2 and get_1, whereas the data flow only imposes a causal ordering between put_i and get_i for $i = 1, 2$.

In this section, we review the solutions that have been proposed for this problem, first by Janssen, Poel and Zwiers [91] in a linear time setting and later in [89, 90, 136, 126] for branching time. These solutions are based on an idea taken from Mazurkiewicz traces [103, 104], namely to recognise explicit *dependencies* between actions. That is, rather than working with actions that are (from the point of view of the formalism) completely uninterpreted, as before, now we assume some further knowledge about them, in the form of a (reflexive and symmetric) *dependency relation* over the action universe *Act*. Action refinement takes dependencies into account by imposing causal ordering only between dependent actions, regardless of the ordering specified on the abstract level.

For instance, in the example above, put_i and get_i are dependent (for $i = 1, 2$) but put_1 and get_2 as well as put_2 and get_1 are independent. Thus, the behaviour of the refinement, given by the term $(put_1 \mathbin{|||} get_1); (put_2 \mathbin{|||} get_2)$, would be equivalent to $put_1; get_1 \mathbin{|||} put_2; get_2$, which, in view of the assumptions, is a reasonable design. On the other hand, if the more complete ordering above was actually intended (for whatever reason), this can still be obtained by explicitly specifying that all get_i-actions are dependent on all put_j-actions.

7.1 Dependencies in linear time

The idea of action dependencies was transferred to a process-algebraic setting by Janssen, Poel and Zwiers [91], using an approach again advocated in [142]. We can reconstruct the basic elements of the setup of [91] as follows.

- They assume a global dependency relation $D \subseteq Act \times Act$, which is reflexive and symmetric but not necessarily transitive. The underlying intuition is that $a\ D\ b$ if a and b *share resources*, and so the order in which they are executed may influence the outcome of the computation. We also write $a\ D\ A$ if $\exists b \in A\colon a\ D\ b$.

  The complement of D is given by $I \subseteq Act \times Act$. It follows that $a\ I\ b$ only if a and b can be executed independently (or in either order). We also write $a\ I\ A$ if $\neg(a\ D\ A)$, i.e., if $\forall b \in A\colon a\ I\ b$.

- They define the *weak sequential composition* of two processes t and u, denoted $t{\cdot}u$, in such a way that only the dependent actions from first and second component are ordered. As a special case, if $a\ I\ b$ for all $a \in \mathcal{S}(t)$ and $b \in \mathcal{S}(u)$, then the semantics of $t{\cdot}u$ and $t \mathbin{|||} u$ coincide; or more generally, if the sorts of t_1 and u_2, respectively u_1 and t_2 are independent then the following *communication closed layers* law holds:

  $$(t_1 \mathbin{|||} u_2){\cdot}(t_2 \mathbin{|||} u_2) \;=\; (t_1{\cdot}t_2) \mathbin{|||} (u_1{\cdot}u_2) \;\;.$$

  In fact, in [91, 142] strong and weak sequential composition are both present within the same formalism; since the novelty is in the latter, we ignore the former in this section.

- They require that *refinement preserves independencies*, in the sense that if a is refined into u and $a\ I\ b$ for some b, then also $b\ I\ \mathcal{S}(u)$. (This requirement is not explicitly stated in [91], but it is necessary for the theory to be sound.)

- They do not consider invisible actions or hiding. The technical reason for this is that the information added by the dependency relation would be lost upon hiding. Although one can envisage an approach in which there is a family of invisible actions in which the dependency relation is somehow retained, this has not been worked out. In the remainder of this section, we therefore ignore hiding and invisible actions.

This results in a semantics where refinement distributes over weak sequential composition:

$$(t_1{\cdot}t_2)[a \to u] = t_1[a \to u]{\cdot}t_2[a \to u] \;\;.$$

Taking the example at the beginning of this section and interpreting the abstract sequential composition $put; get$ as the weak $put{\cdot}get$ instead, and assuming $put_i\ I\ get_j$ for $i \neq j$, it follows that refinement does not impose more ordering than necessary:

$$(put{\cdot}get)[put \to put_1 \mathbin{|||} put_2][get \to get_1 \mathbin{|||} get_2] = (put_1 \mathbin{|||} put_2){\cdot}(get_1 \mathbin{|||} get_2) = put_1{\cdot}get_1 \mathbin{|||} put_2{\cdot}get_2 \;\;.$$

Janssen et al. [91] use a compositional, *linear-time* semantics. The linear time nature has to do with the fact that they consider sets of traces as their semantic model, in which the moment of choice is not recorded. The traces are partially ordered (hence pomsets, see Section 5.1) in a special manner, closely related to Mazurkiewicz traces: two occurrences are ordered if and only if the associated actions are dependent. This means that even if parallel execution is specified, dependent actions in different operands are interleaved; thus, the traces of $a \mathbin{|||} b$ with $a\ D\ b$ are given by $\boxed{a \to b}$ and $\boxed{b \to a}$ and not by $\boxed{\begin{matrix} a \\ b \end{matrix}}$. As usual in a trace-based semantics, the choice operator is modelled by taking the union of trace sets.

Atomic refinement. The above informal discussion of the refinement operator does not precisely correspond to the solution of [91]. Namely, within the setting of action dependencies, one has once more the choice between *atomic* and *non-atomic* refinement, discussed at length earlier in this chapter (Sections 1.3 and 3). It is the former that has been worked out in [91]; in particular, they also include the atomiser $\langle t \rangle$ we discussed in Section 3. This gives rise to a language $\mathit{Lang}_D^{\mathit{lin}}$ with weak sequential composition and atomiser, generated by the following grammar:[3]

$$\mathsf{T} ::= \mathbf{0} \mid \mathbf{1} \mid a \mid \mathsf{T} + \mathsf{T} \mid \underline{\mathsf{T} \cdot \mathsf{T}} \mid \mathsf{T} \mathbin{|||} \mathsf{T} \mid \langle \mathsf{T} \rangle \mid \mathsf{T}[a \to \mathsf{T}] \mid x \mid \mu x.\mathsf{T} \ .$$

See also Table 16 (Page 85) for a complete overview of the different languages used in this chapter. In the linear-time setting of [91], there is no need to forbid termination in choice or refinement terms; correspondingly, we have not distinguished "virgin operands" in the above grammar. Instead, as stated above, the theory is sound only if we impose as a well-formedness condition that *refinement has to be D-compatible*:

Definition 7.1 *$t[a \to u]$ is said to be D-*compatible *if $a\ I\ b$ implies $\mathcal{S}(u)\ I\ b$ for all $b \in \mathit{Act}$.*

In this setting, the atomic execution of a term t comes down to the following. In a term of the form $\langle t \rangle \mathbin{|||} u$, those actions a of u that are *dependent* on some action (in the sort) of t are scheduled either weakly before or weakly after t, where the adjective "weakly" implies that a can still overlap with any initial or final fragment of t of which it is completely independent; but not in the middle (that is, after one action b of t with $a\ D\ b$ but before another action c of t with $a\ D\ c$). On the other hand, if a is *independent* of t, i.e., $a\ I\ \mathcal{S}(t)$, then it may be executed at any time during the execution of $\langle t \rangle$.[4]

Example 7.2 *If $b\ D\ a_i$ for $i = 1, 2$, then $a_1\, b\, a_2$ is not an allowed execution sequence of either $\langle a_1 \cdot a_2 \rangle \mathbin{|||} b$ or $\langle a_1 \mathbin{|||} a_2 \rangle \mathbin{|||} b$; if furthermore $b\ I\ a_3$ then $a_1\, a_2\, b\, a_3$ is an allowed trace of $\langle a_1 \cdot a_2 \cdot a_3 \rangle \mathbin{|||} b$. Finally, if $c\ I\ a_i$ for $i = 1, 2, 3$ then $\langle a_1 \cdot a_2 \cdot a_3 \rangle \mathbin{|||} c$ puts no constraints whatsoever on the moment of execution of c with respect to the a_i.*

The main motivation given in [91] for choosing atomic over non-atomic action refinement is to make refinement distribute over parallel composition, i.e., to satisfy

$$(t_1 \mathbin{|||} t_2)[a \to u] = t_1[a \to u] \mathbin{|||} t_2[a \to u] \ .$$

[3] In fact, only a special case of recursion is considered in [91], namely the Kleene star for weak sequential composition. We conjecture, however, that the theory carries over to full recursion without a problem.

[4] Thus, in terms of the discussion in Section 3, here the execution of an atomic process is non-interruptible for dependent actions but not for independent ones, and in contrast to the operator worked out in Section 3, here the execution of an atomic process is not all-or-nothing.

The following example shows why this is incompatible with non-atomic refinement. (Recall also Section 1.5, where we made exactly the same point in connection with standard interleaving semantics.)

Example 7.3 *Consider $Act = \{a, a_1, a_2, b\}$ with $a\ D\ b$ and $a_i\ D\ b$ for $i = 1, 2$. As we saw above, the semantics of $a \mathbin{|||} b$ is given by $\{\boxed{a \to b}, \boxed{b \to a}\}$; therefore $(a \mathbin{|||} b)[a \to a_1 \cdot a_2]$ is modelled by $\{\boxed{a_1 \to a_2 \to b}, \boxed{b \to a_1 \to a_2}\}$. In other words, the semantics satisfies*

$$(a \mathbin{|||} b)[a \to a_1 \cdot a_2] = a_1 \cdot a_2 \cdot b + b \cdot a_1 \cdot a_2 \ .$$

On the other hand, if we interpret action refinement non-atomically (meaning that $a[a \to t] = t$), then the above distributivity property implies

$$(a \mathbin{|||} b)[a \to a_1 \cdot a_2] = a[a \to a_1 \cdot a_2] \mathbin{|||} b[a \to a_1 \cdot a_2] = (a_1 \cdot a_2) \mathbin{|||} b \ .$$

This contradicts the previous equality, since $(a_1 \cdot a_2) \mathbin{|||} b \neq a_1 \cdot a_2 \cdot b + b \cdot a_1 \cdot a_2$. With atomic refinement (which satisfies $a[a \to t] = \langle t \rangle$ instead), we obtain $(a \mathbin{|||} b)[a \to a_1 \cdot a_2] = \langle a_1 \cdot a_2 \rangle \mathbin{|||} b$, which is fine, since indeed $\langle a_1 \cdot a_2 \rangle \mathbin{|||} b = a_1 \cdot a_2 \cdot b + b \cdot a_1 \cdot a_2$.

The details of the semantics of [91] are outside the scope of this paper; suffice it to say that it gives rise to a pre-order $\sqsubseteq^D_{\mathsf{tr}} \subseteq Lang^{lin}_D \times Lang^{lin}_D$ with the following property:

Proposition 7.4 *$\sqsubseteq^D_{\mathsf{tr}}$ is a pre-congruence over $Lang^{lin}_D$.*

Furthermore, just as in the case of $Lang^{atom}$, action refinement distributes over the other operators (see [91, Theorem 3.2]) and thus corresponds to atomised action substitution; that is, the following equality holds up to $\simeq^D_{\mathsf{tr}} = \sqsubseteq^D_{\mathsf{tr}} \cap \sqsupseteq^D_{\mathsf{tr}}$ (compare with [91, Corollary 3.3]):

$$t[a \to u] = t\{\langle u \rangle / a\} \ .$$

This implies that the following also holds (compare with Theorem 6.7):

Theorem 7.5 *$t \simeq^D_{\mathsf{tr}} flat^{atom}(t)$ for any $t \in Lang^{lin}_D$.*

7.2 Application: Critical sections

As an example of the use of action dependencies, consider again the system of Section 3.4: $Sys_S := Proc_1 \mathbin{|||} Proc_2$ where for $i = 1, 2$

$$Proc_i := ncs_i ; cs_i ; Proc_i \ .$$

The non-critical sections, ncs_i, do not depend on any action of the other process; i.e., we have $ncs_i\ I\ ncs_j$ and $ncs_i\ I\ cs_j$ for $i \neq j$. As a consequence, in the semantics of Sys_S the occurrences of ncs_i are unordered with respect to cs_j (for $i \neq j$); for instance, a valid trace is $\boxed{\begin{array}{lll} ncs_1 \to & cs_1 & \\ & \searrow & \\ & ncs_2 \to & cs_2 \end{array}}$.

Now consider the same refinement as in Section 3.4: $Sys_I := Sys_S[cs_1 \to a_1 ; b_1][cs_2 \to b_2 + c_2]$. In order for the refinements to be D-consistent, we must have $ncs_i\ I\ a_j$ whenever $i \neq j$; moreover, we assume all actions of the critical sections are mutually dependent. This time, the implementation does not suffer from the problem discussed in Section 3.4, where we found that the non-critical sections may not overlap with the critical sections. For instance, a valid trace of Sys_I is $\boxed{\begin{array}{llll} ncs_1 \to & a_1 \to & b_1 & \\ & & \searrow & \\ & & ncs_2 \to & b_2 \end{array}}$, where ncs_2 is unordered with respect to a_1 and b_1 and hence may be scheduled in between them.

7.3 Dependencies in branching time

As we have seen above (Example 7.3), the main motivation in [91] for considering atomic refinement was to obtain distributivity over parallel composition. If we abandon this property, there is no intrinsic difficulty in modelling either non-atomic refinement or parallel composition with synchronisation in a setting with action dependencies. For this purpose, we define the language Lang_D^{bra}, with the following grammar:

$$\mathsf{T} ::= \underline{\mathbf{0}_A} \mid a \mid \mathsf{T} + \mathsf{T} \mid \underline{\mathsf{T} \cdot \mathsf{T}} \mid \mathsf{T} \parallel_A \mathsf{T} \mid \underline{\mathsf{T}[\vec{a} \rightarrow \vec{\mathsf{T}}]} \mid x \mid \mu x . \mathsf{T} \enspace .$$

(Note that internal actions and hiding are again omitted, for reasons discussed above. See also Table 16 (Page 85) for a complete overview of the different languages used in this chapter.) The new constant $\mathbf{0}_A$ denotes *partial termination* and generalises both $\mathbf{0}$ and $\mathbf{1}$; it is discussed below in some detail. In contrast to $\mathbf{1}$, however, $\mathbf{0}_A$ is not considered an auxiliary operator; in fact, Lang_D^{bra} has no auxiliary operators and hence the concept of "virgin operand" plays no role in the well-formedness of terms of Lang_D^{bra}. Instead, we impose several well-formedness conditions to do with the dependency relation; see below. Another new aspect of Lang_D^{bra} is that its refinement operator maps a *vector* of actions to refining agents, instead of just one: the motivation for this is also given below.

Operational semantics. Table 14 contains a branching time operational semantics for Lang_D^{bra}, insofar it deviates from the semantics given before (Tables 2 and 6). The semantics for the flat fragment of Lang_D^{bra} was developed in [125] and extended with refinement in [136]; the solution presented here is based on an improved version in [126]. We briefly discuss the salient points.

Partial termination. Compared with the linear time case, there are some interesting complications having to do with the interplay between weak sequential composition and choice.

Example 7.6 *Consider actions $a, b, c \in \mathit{Act}$ such that $a \; D \; c$ and $b \; I \; c$, and consider the process $(a + b) \cdot c$. The linear time behaviour of this process consists of a trace where a and c are executed in sequence, and a trace where b and c are executed independently. The corresponding branching time behaviour implies that c can be executed initially, but* this resolves the choice *between a and b.*

In capturing these effects operationally, one has to take into account that

- the second operand of weak sequential composition can be active even if the first has not yet terminated;
- actions performed by the second operand of weak sequential composition may resolve choices in the first.

For this purpose, we have introduced the concept of *partial termination*: instead of a deadlock constant $\mathbf{0}$ and a termination constant $\mathbf{1}$, we now have a family of constants $\mathbf{0}_A$, where A stands for the *deadlock alphabet.* $\mathbf{0}_A$ is deadlocked for all actions a that are dependent on some $b \in A$, and terminated for all other actions. The notion of "termination for a given action" is captured by a family of new transition labels $\checkmark_a$ for $a \in A$: $t \xrightarrow{\checkmark_a} t'$ expresses that t is terminated for a, where invoking the termination possibly resolves some choices by which the term becomes t'.

For instance, we have that $a \xrightarrow{\checkmark_b} a$ if $b \; I \; a$ and $\mathbf{0}_A \xrightarrow{\checkmark_a} \mathbf{0}_A$ if $a \; I \; A$. The latter implies that the deadlock constant $\mathbf{0}$ of Lang corresponds to $\mathbf{0}_{\mathit{Act}}$ (no termination at all) whereas the termination constant $\mathbf{1}$ corresponds to $\mathbf{0}_\emptyset$ (termination for all actions). We continue using $\mathbf{0}$ and $\mathbf{1}$ as abbreviations for these special cases.

Table 14: Transition rules for $Lang_{D}^{bra}$.

$$\frac{b\ I\ a}{a \xrightarrow{\surd_b} a} \qquad \frac{a\ I\ A}{\mathbf{0}_A \xrightarrow{\surd_a} \mathbf{0}_A}$$

$$\frac{t \xrightarrow{\surd_a} t' \quad u \xrightarrow{\surd_a} u'}{t + u \xrightarrow{\surd_a} t' + u'} \qquad \frac{t \xrightarrow{\surd_a} t' \quad u \not\xrightarrow{\surd_a}}{t + u \xrightarrow{\surd_a} t'} \qquad \frac{t \not\xrightarrow{\surd_a} \quad u \xrightarrow{\surd_a} u'}{t + u \xrightarrow{\surd_a} u'}$$

$$\frac{t \xrightarrow{\surd_a} t' \quad u \xrightarrow{\surd_a} u'}{t{\cdot}u \xrightarrow{\surd_a} t'{\cdot}u'} \qquad \frac{t \xrightarrow{\surd_a} t' \quad u \xrightarrow{\surd_a} u'}{t \parallel_A u \xrightarrow{\surd_a} t' \parallel_A u'} \qquad \frac{t \xrightarrow{\surd_a} t'}{t[\vec{a} \rightarrow \vec{u}] \xrightarrow{\surd_a} t'[\vec{a} \rightarrow \vec{u}]}$$

$$\frac{a\ I\ \mathcal{S}_x}{\mu x.t \xrightarrow{\surd_a} \mu x.t} \qquad \frac{t\{\mu x.t/x\} \xrightarrow{\surd_a} t' \quad a\ D\ \mathcal{S}_x}{\mu x.t \xrightarrow{\surd_a} t'}$$

$$\frac{t \xrightarrow{a} t'}{t{\cdot}u \xrightarrow{a} t'{\cdot}u} \qquad \frac{t \xrightarrow{\surd_a} t' \quad u \xrightarrow{a} u'}{t{\cdot}u \xrightarrow{a} t'{\cdot}u'}$$

$$\frac{t \xrightarrow{b} t' \quad b \notin \{\vec{a}\}}{t[\vec{a} \rightarrow \vec{u}] \xrightarrow{b} t'[\vec{a} \rightarrow \vec{u}]} \qquad \frac{t \xrightarrow{a_i} t' \quad u_i \xrightarrow{b} u'}{t[\vec{a} \rightarrow \vec{u}] \xrightarrow{b} u'{\cdot}t'[\vec{a} \rightarrow \vec{u}]}$$

From the rules in Table 14, it can be seen that partial termination resolves choice if precisely one of the operands terminates and the other cannot. In Example 7.6, for instance, $b \xrightarrow{\surd_c} b$ but $a \not\xrightarrow{\surd_c}$, and hence $a + b \xrightarrow{\surd_c} b$; we obtain the derivation

$$\dfrac{\dfrac{a \not\xrightarrow{\surd_c} \qquad \dfrac{b\;I\;c}{b \xrightarrow{\surd_c} b}}{a + b \xrightarrow{\surd_c} b} \qquad \dfrac{}{c \xrightarrow{c} \mathbf{1}}}{(a + b)\cdot c \xrightarrow{c} b\cdot\mathbf{1}}$$

which is indeed the expected behaviour. (To avoid generating infinite state spaces for virtually all recursive terms, one then also has to add a mechanism to get rid of superfluous **1**'s, such as in the target term of the above transition; this can be done for instance by normalising terms with respect to some structural congruence that includes the axiom $t\cdot\mathbf{1} \equiv t$.)

Recursion. The termination behaviour of recursion has a problem connected to the negative premise in two of the rules for choice, if one takes the natural extension of the standard operational rule for recursion to partial termination:

$$\dfrac{t\{\mu x.t/x\} \xrightarrow{\surd_a} t'}{\mu x.t \xrightarrow{\surd_a} t'}$$

Example 7.7 *Let* $a, b, c \in Act$ *with* $a\;I\;c$ *and* $b\;I\;c$*, and consider the process* $t = \mu x.(a\cdot x + b)$ *with* $\mathcal{S}_x = \{a, b\}$*. If we assume* $t \not\xrightarrow{\surd_c}$*, we obtain a contradiction:*

$$\dfrac{\dfrac{\dfrac{t \not\xrightarrow{\surd_c}}{a\cdot t \not\xrightarrow{\surd_c}} \qquad \dfrac{b\;I\;c}{b \xrightarrow{\surd_c} b}}{a\cdot t + b \xrightarrow{\surd_c} b}}{t \xrightarrow{\surd_c} b}$$

On the other hand, no transition of the form $t \xrightarrow{\surd_c} t'$ *can be derived, since this would require an "infinite unfolding" of the recursion.*

Problems due to negative premises have been studied in depth in [80, 58]. A crucial point in our solution is to take the sort of process variables (see Table 11) into account within the operational rules:

- A recursive term always terminates, without unfolding, for any action that is independent of the entire sort of the recursion variable. For instance, for the term $t = \mu x.(a\cdot x + b)$ in Example 7.7 we obtain $t \xrightarrow{\surd_c} t$.
- The previous rule (in Example 7.7) for the partial termination of recursion is restricted to $\surd_a$-transitions with $a\;D\;\mathcal{S}_x$, because otherwise we could unfold recursive terms arbitrarily often in the derivation of a $\surd_a$-transition; for instance, for the same term t above

$$\dfrac{\dfrac{\dfrac{t \xrightarrow{\surd_c} t}{a\cdot t \xrightarrow{\surd_c} a\cdot t}}{a\cdot t + b \xrightarrow{\surd_c} a\cdot t + b}}{t \xrightarrow{\surd_c} a\cdot t + b}$$

A related problem is that, due to the changed nature of termination, the standard notion of guardedness (Definition 2.1) is not appropriate any more, but needs to be strengthened.

Definition 7.8 *We first define what it means for a term to be a D-*guard *for an action $a \in Act$.*

- $\mathbf{0}_A$ *is a D-guard for a if $a\ D\ b$ for some $b \in A$;*
- x *is a D-guard for a if $a\ D\ b$ for some $b \in \mathcal{S}_x$;*
- b *is a D-guard for a if $a\ D\ b$;*
- $t + u$ *is a D-guard for a if both t and u are D-guards for a;*
- $t[b \rightsquigarrow u]$ *is a D-guard for a if t is a D-guard for a;*
- *For all other operators op, $op(t_1, \ldots, t_n)$ is a D-guard for a if one of the t_i is a D-guard for a.*

Next, we define what it means for a variable to be D-guarded in a term.

- x *is D-guarded in y ($\in Proc$) if $x \neq y$;*
- x *is D-guarded in $t; u$ if x is D-guarded in t and either t is a D-guard for all $a \in \mathcal{S}_x$ or x is D-guarded in u;*
- x *is D-guarded in $\mu y.t$ if either $x = y$ or x is D-guarded in t;*
- *For all other operators op, x is D-guarded in $op(t_1, \ldots, t_n)$ if x is D-guarded in all t_i.*

For instance, the term $\mu x.(a{\cdot}x + b)$ in Example 7.7 is D-guarded if $a\ D\ b$ but not if $a\ I\ b$ (because then $a \xrightarrow{\surd_b} a$ but $b\ D\ b \in \mathcal{S}_x$). We require D-guardedness as a necessary condition for well-formedness.

Refinement. The operational rules for refinement in Table 14 are essentially the same as for the sequential language $Lang^s eq$ (see Table 2), except that we now use weak rather than strong sequential composition. However, for these rule to make sense, the refinement should be compatible with the dependency relation, in a stronger sense than in the linear time case (Definition 7.1).

Definition 7.9 $t[a \rightsquigarrow u]$ *is said to be* strictly D-compatible *if it is D-compatible and, moreover, for all $b \in Act$, $a\ D\ b$ implies (i) $u \not\xrightarrow{\surd_b}$ and (ii) $c\ D\ b$ for all $u \xrightarrow{c}$.*

The additional condition expresses that dependencies should be preserved as well as independencies: if a and b are dependent then u must be deadlocked for b (ensuring that if b has been specified sequentially after a then b cannot occur before u has started) and b must be deadlocked for all initial actions of u (implying the dual property, namely that if a is to occur after b then no initial action of u can occur before b).

The interesting thing about the current setting is that the operational rules remain correct even in the context of parallel composition with synchronisation: interleaving bisimulation is a congruence for action refinement. (Of course, what we have here is not really the standard interleaving setting, since the action dependencies provide additional information just like the event annotations in the event-based semantics of Section 4; however, since here the information is provided globally rather than locally in each individual transitions, it is less visible.)

Unfortunately, strict D-compatibility is more restrictive than one would like. For instance, the term $t[get \rightsquigarrow u]$ where $t = (put{\cdot}get)[put \rightsquigarrow put_1 \mid\mid\mid put_2]$ and $u = get_1 \mid\mid\mid get_2$ is not well-formed: according to D-compatibility, due to the fact that $get_2\ D\ put_2$ we must have $get\ D\ put_2$; however, since $u \xrightarrow{get_1}$ but nevertheless $get_1 \; I \; put_2$, condition (ii) of strong D-compatibility is violated. A similar violation occurs if we first refine *get* and then *put*. In fact, this particular design step can only be understood if we assume both refinements to take place *at the same time*. Looking back, this is in fact what we silently assumed in the discussion.

Thus, rather than considering single-action refinements $\_[a{\to}u]$, we should allow simultaneous refinements $\_[a_1{\to}u_1, \ldots, a_n{\to}u_n]$ (with $a_i \neq a_j$ for $i \neq j$), also written in vector notation as $\_[\vec{a}{\to}\vec{u}]$. Strict D-compatibility is extended accordingly.

Definition 7.10 *$t[\vec{a}{\to}\vec{u}]$ is said to be strictly D-compatible if for all $b \in Act$*

1. *$a_i \; I \; b$ implies $\mathcal{S}(u_i) \; I \; b$;*
2. *$a_i \; D \; b$ implies (i) $u_i \not\xrightarrow{\surd_b}$ and (ii) $c \; D \; b$ for all $u_i \xrightarrow{c}$.*

This indeed holds for $t = get; put$ and the refinement $put{\to}put_1 \mathbin{|||} put_2, get{\to}get_1 \mathbin{|||} get_2$ under consideration; for instance, $get \; D \; put$ and indeed for all $r(put) \xrightarrow{c}$ (meaning $c = put_i$ for $i \in \{1,2\}$) we have $r(get) \not\xrightarrow{\surd_c}$.

Well-formedness. To summarise the well-formedness conditions on $Lang_D^{bra}$:

- Recursion is required to be D-guarded;
- Refinement is required to be strictly D-compatible.

Behavioural semantics. Strong bisimilarity (Definition 2.6) is once more a congruence; for the proof see [126].

Proposition 7.11 *$\sim$ is a congruence over $Lang_D^{bra}$.*

As an example, consider again the buffer discussed at the start of this section, and let $put_i \; I \; get_j$ for $i \neq j$. Using the operational rules, we can derive for instance

$$\begin{array}{ll} \multicolumn{2}{l}{(put{\cdot}get)[put{\to}put_1 \mathbin{|||} put_2, get{\to}get_1 \mathbin{|||} get_2]} \\ \quad \xrightarrow{put_1} & (\mathbf{1} \mathbin{|||} put_2){\cdot}(\mathbf{1}; get)[put{\to}put_1 \mathbin{|||} put_2, get{\to}get_1 \mathbin{|||} get_2] \\ \quad \xrightarrow{get_1} & (\mathbf{1} \mathbin{|||} put_2){\cdot}(\mathbf{1} \mathbin{|||} get_2){\cdot}(\mathbf{1}; \mathbf{1})[put{\to}put_1 \mathbin{|||} put_2, get{\to}get_1 \mathbin{|||} get_2] \; . \end{array}$$

This shows that a low-level get_i-action can occur directly after the corresponding put_i-action, without having to wait for other put_j-actions to be executed as well; thus, the desired flexibility is achieved.

Another issue reported in [126] is the development of a denotational partial-order model for $Lang_D^{bra}$, for the purpose of showing that the above interleaving semantics is compatible with existing interpretations of action refinement. The denotational model is event-based, based on the families of posets model of [117], extended to take partial termination into account. The details are beyond the scope of this section; suffice it to say that one can define an isomorphism over the denotational model that is a congruence for $Lang_D^{bra}$ and strictly stronger than $\sim$, and moreover, there is a straightforward transformation from the denotational model into labelled transition systems (including $\surd_a$-transitions).

The fact that in the presence of action dependencies, the interleaving paradigm is strong enough to give rise to a compositional model for action refinement was discussed separately in [70].

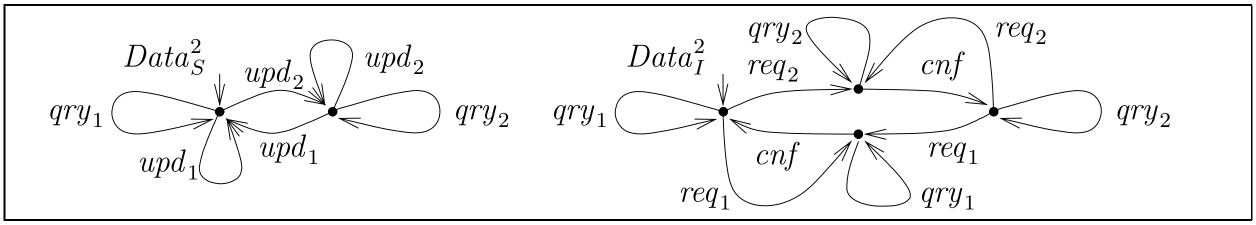


Figure 10: Refinement of Data_S^2 using action dependencies: $\mathit{qry}_i\ D\ \mathit{req}_j$ but $\mathit{qry}_i\ I\ \mathit{cnf}$

7.4 Application: A simple data base

Once more consider the data base example of Section 2.5 (which does not include the *copy*-operation of Section 4.6). The 2-state version of this data base was specified by $\mathit{Data}_S^2 := \mathit{State}_1$ where for $i = 1, 2$

$$\mathit{State}_i := \mathit{qry}_i; \mathit{State}_i + \mathit{upd}_1; \mathit{State}_1 + \mathit{upd}_2; \mathit{State}_2 \ .$$

We then refined upd_i into $\mathit{req}_i; \mathit{cnf}$ for $i = 1, 2$, where req_i is a request to change the state to i and *cnf* is the confirmation that this has been done. In the standard setting of Section 2.5 this automatically implies that *qry* is disabled in between req_i and *cnf* (see Figure 2). If one wants to allow *qry* also at that point (because the change to the state has already been made, so it is safe to read it), this can be achieved using action dependencies by specifying $\mathit{qry}_i\ I\ \mathit{cnf}$ for $i = 1, 2$ (and all other actions mutually dependent), and setting $r = (\mathit{upd}_1 \rightarrow \mathit{req}_1 \cdot \mathit{cnf}, \mathit{upd}_2 \rightarrow \mathit{req}_2 \cdot \mathit{cnf})$ as before. Note that r is strongly D-consistent: we have $\mathit{qry}_i\ D\ \mathit{upd}_j$ for all i, j, and indeed (i) $r(\mathit{upd}_j) \not\xrightarrow{\surd_{\mathit{qry}_i}}$ and (ii) $\mathit{qry}_i\ D\ a_j$ for all $r(\mathit{upd}_j) \xrightarrow{a_j}$ (namely, $a_j = \mathit{req}_j$). The behaviour of the system Data_S^2 and its refinement $\mathit{Data}_I^2 := \mathit{Data}_S^2[r]$ is depicted in Figure 10.

It should be noted that the previous solution, which did not allow the overlap, can still be derived if we set $\mathit{qry}_i\ D\ \mathit{cnf}$ instead.

7.5 The dual view: localities

Another way to integrate the concept of action dependencies into process algebra is by taking the dual view of *localities*. In the context of action refinement, this was worked out by Huhn in [89, 90]. We briefly recapitulate the main points of this approach.

The basic assumption is that a system is described by the parallel composition of a finite, indexed set of sequential sub-systems, such that all actions performed by any given sub-system are mutually dependent, and this is the only possible source of dependencies. Actions are refined *locally* at each sequential sub-system, possibly differently at each location; refinement has to preserve and reflect the locality of actions.

More precisely, one considers a finite set of locations *Loc* and a mapping $\ell: \mathit{Act} \to \mathbf{2}^{\mathit{Loc}}$ defining at what localities an action may occur; then $a\ D\ b$ for $a, b \in \mathit{Act}$ if and only if $\ell(a) \cap \ell(b) \neq \emptyset$. Let us denote $\mathit{Act}_l = \{a \in \mathit{Act} \mid l \in \ell(a)\}$ for the alphabet of location l. Overall system behaviour is specified by terms of the form $\Pi_{l \in \mathit{Loc}} t_l$, denoting the parallel composition of sequential processes t_l, where $\mathcal{S}(t_l) \subseteq \mathit{Act}_l$ for all $l \in \mathit{Loc}$, with implicit synchronisation over all common actions. The semantics of such a parallel composition is given by an ordinary labelled transition system.

Action refinement is interpreted in this context by a *family* of images for a given abstract action, namely one for each location that partakes in the action. Thus, in general, an action is mapped simultaneously to a different concrete process at each location. Operationally, therefore, the effect of a global action refinement is captured by the local action refinements, which in turn are refinements of a sequential system and hence analogous to the treatment in Section 2. In particular, at each location $l \in \ell(a)$, the essential rule for the behaviour of a refined system is given by

$$\frac{t_l \xrightarrow{a} t'_l \quad u_l \xrightarrow{b} u'_l}{t_l[a \rightarrowtail u_l] \xrightarrow{b} u'_l; t'_l[a \rightarrowtail u_l]}$$

where the subscript $_l$ indicates that we are dealing with the process at location l. There are some restrictions in [90] on the allowable (families of) refinement functions, which are closely related to the preservation of (in)dependencies imposed in the previous sub-section.

- Only the locations $l \in \ell(a)$ may refine a, and $\mathcal{S}(u_l) \subseteq Act_l$ for all $l \in Loc$. This implies that refinement is D-compatible (Definition 7.1).
- None of the local refinements may map onto a terminated process. This goes somewhat in the direction of strict D-compatibility: If $a\ D\ b$ then a and b share a location, say l; since $r_l(a) \not\xrightarrow{\checkmark}$ for all such shared l, it follows that b cannot "overtake" $r(a)$.

Refinement on the system level is interpreted by distribution to the local level: that is, one may specify a global refinement $\_[a \rightarrowtail \vec{u}]$ where $\vec{u}$ is a vector of local images u_l; $(\Pi_l t_l)[a \rightarrowtail \vec{u}]$ is then defined to correspond to $\Pi_l(t_l[a \rightarrowtail u_l])$.

If we use $Lang^{loc}$ to denote the resulting locality-based language, then the following result is an immediate consequence of the format of the operational rules:

Proposition 7.12 *$\sim$ is a congruence over $Lang^{loc}$.*

Furthermore, the fact that action refinement is essentially performed on local, sequential systems implies that action refinement and action substitution coincide (see Section 6.2):

Theorem 7.13 *$t \sim flat(t)$ for any $t \in Lang^{loc}$.*

The motivating example of this section can be dealt with by considering $Loc = \{1, 2\}$, i.e., a system with two localities, and assuming $\ell(put) = \ell(get) = \{1, 2\}$. The specification is then given by $put; get \mid\mid\mid put; get$ (one sequential term for each location), and the local refinements by $put \rightarrowtail put_l$ and $get \rightarrowtail get_l$ for $l = 1, 2$. Refinement then yields

$$(put; get)[put \rightarrowtail put_1][get \rightarrowtail get_1] \mid\mid\mid (put; get)[put \rightarrowtail put_2][get \rightarrowtail get_2]$$

which is equivalent to the desired solution, $put_1; get_1 \mid\mid\mid put_2; get_2$.

8 Vertical implementation

Finally, we present a different, more radical solution to the lack of flexibility of the action refinement operator that we noted in the previous section. The presentation is based on [119, 123]. It consists of re-interpreting action refinement so that it does not take the form of an *operator* (effectively a function $\_[a \rightarrowtail u]\colon Lang \to Lang$ for all u) but rather of a *relation* $\leq^{a \rightarrowtail u} \subseteq Lang^{flat} \times Lang^{flat}$, where $Lang^{flat}$ is the flat fragment of $Lang$ (see Page 29), i.e., without the refinement operator. (See also Table 16 (Page 85) for a complete overview of the different languages used in this chapter.) That is, if a system is abstractly specified by a term t_1, and a design step is taken that maps an abstract action a to a term u, the result is not uniquely determined but instead can be any term t_2 satisfying the requirement

$$t_1 \leq^{a \rightarrowtail u} t_2 \ .$$

$\leq^{a\to u}$ is called a *vertical implementation relation*, in analogy with the usual implementation relations (i.e., pre-orders) such as trace or failure inclusion (see, e.g., [27, 43]). This notion is interesting especially if we consider *simultaneous* refinements (see also Section 7.3); thus, actually we will consider indexed relations of the form $\leq^{\vec{a}\to\vec{u}} \subseteq \mathit{Lang}^{\mathit{flat}} \times \mathit{Lang}^{\mathit{flat}}$ where $\vec{a}\to\vec{u}$ is a finite list of refinement mappings. In this section, we will often denote such vectors by r, and occasionally interpret r as a function $r\colon \mathit{Act} \to \mathit{Lang}^{\mathit{flat}}$ that maps all a_i onto u_i and is the identity everywhere else.

For instance, consider again the small buffer example used as motivation in the previous section. We started with a specification $\mathit{put}; \mathit{get}$ and a (simultaneous) refinement given by $r = \mathit{put}\to\mathit{put}_1 \mathbin{|||} \mathit{put}_2, \mathit{get}\to\mathit{get}_1 \mathbin{|||} \mathit{get}_2$; by fixing a dependency relation, we arrived at a uniquely determined implementation. Using the concept of vertical implementation, on the other hand, there could be a number of correct implementations; for instance

$$\begin{aligned} \mathit{put}; \mathit{get} &\leq^r (\mathit{put}_1 \mathbin{|||} \mathit{put}_2); (\mathit{get}_1 \mathbin{|||} \mathit{get}_2) \\ \mathit{put}; \mathit{get} &\leq^r (\mathit{put}_1; \mathit{get}_1 \mathbin{|||} \mathit{put}_2); \mathit{get}_2 \\ \mathit{put}; \mathit{get} &\leq^r \mathit{put}_1; \mathit{get}_1 \mathbin{|||} \mathit{put}_2; \mathit{get}_2 \ . \end{aligned}$$

A major advantage of this approach is that we can stay in the realm of interleaving semantics, just as in the case of dependency-based refinement. This is due to the fact that the congruence question simply disappears once we have abandoned the idea of refinement as an operator. For instance, the following design steps all give rise to perfectly valid vertical implementations (compare Section 1.3):

$$\begin{aligned} a \mathbin{|||} b &\leq^{a\to a_1;a_2} a_1; a_2 \mathbin{|||} b \\ a \mathbin{|||} b &\leq^{a\to a_1;a_2} a_1; a_2; b + b; a_1; a_2 \\ a; b + b; a &\leq^{a\to a_1;a_2} a_1; a_2 \mathbin{|||} b \\ a; b + b; a &\leq^{a\to a_1;a_2} a_1; a_2; b + b; a_1; a_2 \ . \end{aligned}$$

As a consequence, it is possible to formulate a single, unified framework in which the standard concept of implementation (which, in contrast, one might call *horizontal*) and vertical implementation are integrated. In fact, given that there are many variations on the standard concept of implementation (based, e.g., on traces, testing or bisimulation), one can imagine analogous variations on vertical implementation.

Below, we propose a specific vertical implementation relation, based on delay bisimulation (see Section 2.4). Another vertical implementation relation, based on weak bisimulation, was worked out in [123, 124]; a testing-based relation was proposed in [118]. We then formulate some properties that one would naturally expect a vertical implementation relation to satisfy, inspired by properties of horizontal implementation (e.g., transitivity, monotonicity); we assert that vertical delay bisimulation indeed satisfies these properties.

8.1 Refinement functions.

Vertical implementation relations are a new area of study, and the results that have been achieved hold only for a restricted class of refinement functions. For the purpose of this section, we therefore impose some strong restrictions on the allowable refinements. See [124] for a more extensive discussion.

First of all, we only consider refinement images in a restricted sub-language $\mathit{Lang}^{\mathit{ref}}$, generated by the following grammar:

$$\mathsf{T} ::= a \mid \mathsf{T}; \mathsf{T} \mid \mathsf{T} + \mathsf{T} \ .$$

(Note that this limits actions to $a \in \mathit{Act}$, disallowing invisible actions.) Furthermore, we impose as a well-formedness condition on $\mathit{Lang}^{\mathit{ref}}$ that terms must be *distinct*, in the following inductively defined sense:

- Every action $a \in \mathit{Act}$ is said to be distinct;
- $t_1; t_2$ and $t_1 + t_2$ are said to be distinct if t_1 and t_2 are distinct and $\mathcal{S}(t_1) \cap \mathcal{S}(t_2) = \emptyset$.

Furthermore, given a set of actions $A \subseteq \mathit{Act}$, a simultaneous refinement $r\colon \mathit{Act} \to \mathit{Lang}^{\mathit{ref}}$ (using only well-formed terms as images) is called *distinct on A* if $\mathcal{S}(r(a)) \cap \mathcal{S}(r(b)) = \emptyset$ for $a \in A$ and $b \in \mathit{Act} \backslash \{a\}$. This has the consequence that, given a b-transition performed by the r-image of some action $a \in A$, both the action a and the position in $r(a)$ where b occurs are uniquely determined. In the remainder of this section, we will implicitly only consider vertical implementation relations $\leq^r$ between pairs t_1 and t_2 such that r is distinct on $\mathcal{S}(t_1)$.

Note that the image of $\mathit{put} \to \mathit{put}_1 \mathbin{|||} \mathit{put}_2$ is not in $\mathit{Lang}^{\mathit{ref}}$ (and neither is the image of get); hence we cannot treat the motivating example in its current form. Below, we consider the somewhat simpler case of $r = (\mathit{put} \to \mathit{put}_1; \mathit{put}_2$ and $\mathit{get} \to \mathit{get}_1; \mathit{get}_2)$; we still expect the following valid vertical implementations.

$$\begin{array}{lll} \mathit{put}; \mathit{get} & \leq^r & \mathit{put}_1; \mathit{get}_1; \mathit{put}_2; \mathit{get}_2 \\ \mathit{put}; \mathit{get} & \leq^r & \mathit{put}_1; (\mathit{get}_1 \mathbin{|||} \mathit{put}_2); \mathit{get}_2 \ . \end{array}$$

We furthermore use the following refinement function-related concepts:

- For every refinement function r, the function $\mathcal{S}_r\colon \mathit{Act} \to \mathbf{2}^{\mathit{Act}}$ maps all actions to the alphabet of their r-images; that is, for all $a \in A$

  $$\mathcal{S}_r\colon a \mapsto \mathcal{S}(r(a)) \ .$$

  This is also extended to sets of actions $A \subseteq \mathit{Act}$:

  $$\mathcal{S}_r\colon A \mapsto \textstyle\bigcup_{a \in A} \mathcal{S}_r(a) \ .$$

- For every refinement function r and set of actions A, r/A denotes the function that is the identity on A and coincides with r elsewhere; that is,

  $$r/A\colon a \mapsto \begin{cases} a & \text{if } a \in A \\ r(a) & \text{otherwise.} \end{cases}$$

- For all refinement functions r_1 and r_2, $r_1 + r_2$ denotes the function that maps all $a \in \mathit{Act}$ to the choice between the r_i-images:

  $$(r_1 + r_2)\colon a \mapsto r_1(a) + r_2(a) \ .$$

8.2 Vertical delay bisimulation

We develop a vertical implementation relation based on a variant of weak bisimulation called *delay bisimulation*, which we already introduced in Definition 2.11.[5] We actually drop the qualifier "delay" in the remainder of this section, when this does not give rise to confusion.

[5] Previously (in [123]), we took weak bisimulation as a basis for vertical extension; in doing so, we encountered some problems that we conjectured to be related the fact that weak bisimulation is not a congruence for the refinement operator in the sequential language $\mathit{Lang}^{\mathit{seq}}$ (see Section 2). Since delay bisimulation is the coarsest congruence for refinement contained in weak bisimulation (Proposition 2.12), its vertical extension is smoother than that of weak bisimulation.

An important extension with respect to the standard notion of bisimulation is that we have to take into account that in any given state of the implementation, there may be associated refined actions whose execution has not yet terminated. These will be collected in a (multi)set of *residual* (or *pending*) *refinements* and will be used to parameterise the bisimulation. Thus, bisimulations are not binary but ternary relations.[6] Furthermore, in contrast to standard bisimulation, the simulation directions from specification to implementation and vice versa are no longer symmetric. To simulate the abstract transitions of the specification by the implementation (with the set of pending refinements as an intermediary), we define the concept of *down-simulation*; the other direction is called an *up-simulation*. Finally, we also require a *residual simulation* which captures how the pending refinements are processed by the implementation.

Residual sets. A residual set for r will be a multiset of non-terminated proper derivatives of r-images, formally represented by a function $R\colon \mathit{Lang}^{seq} \to \mathbb{N}$. We will write $t \in R$ if $R(t) > 0$. To be precise, the collection of residual sets for r is defined as follows:

$$\mathit{RS}(r) = \{R\colon \mathit{Lang}^{seq} \to \mathbb{N} \mid \forall u' \in R\colon \exists a \in \mathit{Act}, \sigma \in \mathit{Act}^+\colon r(a) \overset{\sigma}{\Longrightarrow} u' \not\xrightarrow{\checkmark}\} \enspace .$$

We use the following constructions over residual sets:

$$\begin{aligned}
\emptyset\colon &\quad u \mapsto 0 \\
[t]\colon &\quad u \mapsto \begin{cases} 1 & \text{if } u = t \text{ and } t \not\xrightarrow{\checkmark} \\ 0 & \text{otherwise} \end{cases} \\
R_1 \oplus R_2\colon &\quad u \mapsto R_1(u) + R_2(u) \\
R_1 \ominus R_2\colon &\quad u \mapsto \mathit{max}\{R_1(u) - R_2(u), 0\}
\end{aligned}$$

The behaviour of a residual set corresponds to the synchronisation-free parallel composition of its elements. Formally:

$$R \xrightarrow{\alpha} R' \;:\Leftrightarrow\; \exists t \in R\colon \exists (t \xrightarrow{\alpha} t')\colon R' = (R \ominus [t]) \oplus [t']$$

Note that terminated terms do not contribute to the residual set, and $\alpha \neq \tau$. The reason why we can ignore terminated terms is that (due to well-formedness) it is certain that such terms no longer display any operational behaviour (in terms of Section 2, termination is final).

Notation for ternary relations. In the following, we will often work with ternary relations of the form $\rho \subseteq N \times N \times \mathit{RS}(r)$. We use the notation $n_1\; \rho^R\; n_2$ to abbreviate $(n_1, n_2, R) \in \rho$; in other words, ρ^R is interpreted as the binary relation $\{(n_1, n_2) \mid (n_1, n_2, R) \in \rho\}$.

Definition 8.1 *Let $\mathcal{T}$ be a labelled transition system. A* strict down-simulation up to r *over $\mathcal{T}$ is a ternary relation $\rho \subseteq N \times N \times \mathit{RS}(r)$ such that for all $n_1\; \rho^\emptyset\; n_2$, if $n_1 \xrightarrow{\alpha} n_1'$ then one of the following holds:*

1. *$\alpha = \tau$ and $n_1'\; \rho^\emptyset\; n_2$;*
2. *$\alpha \in \{\tau, \checkmark\}$ and $\exists (n_2 \Rightarrow\xrightarrow{\alpha} n_2')$ such that $n_1'\; \rho^\emptyset\; n_2'$;*
3. *$\alpha \in \mathit{Act}$ and $\forall (r(\alpha) \xrightarrow{c} u')\colon \exists (n_2 \Rightarrow\xrightarrow{c} n_2')$ such that $n_1'\; \rho^{[u']}\; n_2'$.*

[6] Other ternary bisimulation-based relations are, for instance, *history-preserving* bisimulation [62], and *symbolic* bisimulation [86].

Note that down-simulation only imposes conditions on triples (n_1, n_2, R) where $R = \emptyset$, i.e., there are no pending refinements. For such triples, apart for the simulation of internal and termination transitions (which are standard for delay bisimulation), it is required that any action a of the specification (n_1) is refined into a term $r(a)$ of which all initial actions are simulated by the implementation (n_2), such that the successors of n_1 and n_2 are again related, with the successor of $r(a)$ pending. The reason for the additive "strict" will become clear later this section.

Definition 8.2 *Let $\mathcal{T}$ be a labelled transition system. An* up-simulation up to r *over $\mathcal{T}$ is a ternary relation $\rho \subseteq N \times N \times RS(r)$ such that for all $n_1 \; \rho^R \; n_2$, if $n_2 \xrightarrow{\gamma} n_2'$ then one of the following holds:*

1. *$\gamma = \tau$ and $n_1 \; \rho^R \; n_2'$;*
2. *$\gamma \in \{\tau, \checkmark\}$ and $\exists(n_1 \Rightarrow\xrightarrow{\gamma} n_1')$ such that $n_1' \; \rho^R \; n_2'$;*
3. *$\exists a \in Act$: $\exists(n_1 \Rightarrow\xrightarrow{a} n_1')$ and $\exists(r(a) \xrightarrow{\gamma} u')$ such that $n_1' \; \rho^{R\oplus[u']} \; n_2'$;*
4. *$\exists(n_1 \Rightarrow n_1')$ and $\exists(R \xrightarrow{\gamma} R')$ such that $n_1' \; \rho^{R'} \; n_2'$.*

Thus, if the implementation's move γ is not an internal or termination transition, either it corresponds to the initial concrete action of a refined abstract action a (in which case, since r is distinct, a is uniquely determined), or it is an action of a residual refinement (which again is uniquely determined), in which case the specification is allowed to move silently. Note that these two cases (Clauses 3 and 4 in the definition) are mutually exclusive, again due to the distinctness of r.

Definition 8.3 *Let $\mathcal{T}$ be a labelled transition system. A* strict residual simulation *over $\mathcal{T}$ is a ternary relation $\rho \subseteq N \times N \times RS(r)$ such that for all $R \neq \emptyset$, if $n_1 \; \rho^R \; n_2$ then $\forall(R \xrightarrow{\gamma} R')$: $\exists(n_2 \Rightarrow\xrightarrow{\gamma} n_2')$ such that $n_1 \; \rho^{R'} \; n_2'$.*

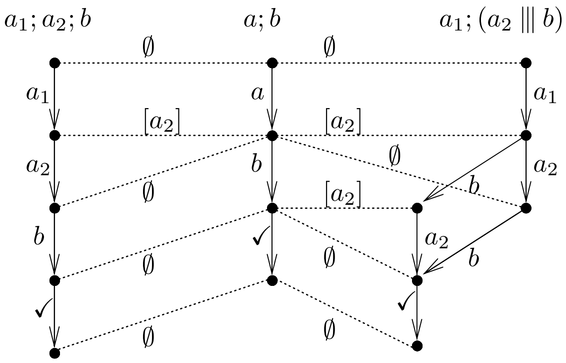
This specifies that any move of a pending refinement has to be (delay-)simulated by the implementation, while the specification remains as it is. This implies that pending refinements can be "worked off" in any possible order, or indeed in parallel, by the implementation. This property can be construed as an operational formulation of *non-interruptability*: that which is started can always be finished (see also Section 3). The combination of down-simulation, up-simulation and residual simulation gives rise to vertical bisimulation.

Definition 8.4 *Let $\mathcal{T}$ be a labelled transition system.*

- *A* strict vertical bisimulation up to r *over $\mathcal{T}$ is a ternary relation $\rho \subseteq N \times N \times RS(r)$ that is both a strict down-simulation up to r, an up-simulation up to r and a strict residual simulation.*
- Strict vertical bisimilarity up to r *over $\mathcal{T}$, denoted $\precsim^r_\forall$, is the largest strict vertical bisimulation up to r, and* rooted strict vertical bisimilarity up to r, *denoted $\preceq^r_\forall$, is the largest delay root of $\precsim^{r,\emptyset}_\forall$.*

The subscript "$\forall$" is related to the adjective "strict" and the $\forall$-quantifiers in the definitions of strict down- and residual simulation; see also below. The following is an example of an actual strict vertical bisimulation.

Example 8.5 *Let $r: a \mapsto a_1; a_2$. The following figure shows strict vertical bisimulations proving $a; b \preceq^r_\forall a_1; a_2; b$ and $a; b \preceq^r_\forall a_1; (a_2 \mathbin{|||} b)$, where the dotted lines connect related states and their* *labelling is the residual set indexing the relation:*



However, this does not yet quite allow the design step we used in our (modified) motivating example: that is, $put_1; (get_1 \mid\mid\mid put_2); get_2$ is not a vertical implementation of $put; get$. The reason is that the sequence

$$put_1; (get_1 \mid\mid\mid put_2); get_2 \xrightarrow{put_1} (get_1 \mid\mid\mid put_2); get_2 \xrightarrow{get_1} (\mathbf{1} \mid\mid\mid put_2); get_2$$

of the implementation cannot be strictly bisimulated: the only possibility would be

$$put; get \xrightarrow{put} get \xrightarrow{get} \mathbf{1}$$

with residual sets $R_0 = \emptyset$ (initially), $R_1 = [put_2]$ (after the first step) and $R_2 = [put_2] \oplus [get_2]$ (after the second step). Since $R_2 \xrightarrow{get_2}$, strict residual simulation requires that the implementation can do get_2 already, which is not the case.

To improve on this, we formulate a second, weaker notion of down-simulation and residual simulation, which state that not *all* transitions of a refinement image, respectively *all* transitions of the residual have to be simulated, but rather *at least one*.

Definition 8.6 *Let $\mathcal{T}$ be a labelled transition system and assume $\rho \subseteq N \times N \times RS(r)$.*

- *ρ is a* lax down-simulation up to r *if for all $n_1 \; \rho^{\emptyset} \; n_2$ and $n_1 \xrightarrow{\alpha} n_1'$, one of the following holds:*

  1. *$\alpha = \tau$ and $n_1' \; \rho^{\emptyset} \; n_2$;*
  2. *$\alpha \in \{\tau, \checkmark\}$ and $\exists(n_2 \Rightarrow\xrightarrow{\alpha} n_2')$ such that $n_1' \; \rho^{\emptyset} \; n_2'$;*
  3. *$\alpha \in Act$ and $\exists(r(\alpha) \xrightarrow{c} u') \colon \exists(n_2 \Rightarrow\xrightarrow{c} n_2')$ such that $n_1' \; \rho^{[u']} \; n_2'$.*

- *ρ is a* lax residual simulation *if for all $R \neq \emptyset$, if $n_1 \; \rho^R \; n_2$ then $\exists(R \xrightarrow{\gamma} R') \colon \exists(n_2 \Rightarrow\xrightarrow{\gamma} n_2')$ such that $n_1 \; \rho^{R'} \; n_2'$.*

- *ρ is a* lax vertical bisimulation up to r *if it is both a lax down-simulation up to r, an up-simulation up to r and a lax residual simulation.*

- Lax vertical bisimilarity up to r, *denoted $\precsim^r_\exists$, is the largest lax vertical bisimulation up to r, and* rooted lax vertical bisimilarity up to r, *denoted $\preceq^r_\exists$, is the largest delay root of $\precsim^{r,\emptyset}_\exists$.*

Note that the $\forall$-quantifiers in Definitions 8.1 and 8.3 have been turned into $\exists$; this is precisely what makes $\preceq^r_\exists$ weaker than $\preceq^r_\forall$. Now we indeed have $put; get \preceq^r_\exists put_1; (get_1 \mid\mid\mid put_2); get_2$ (where $r = (put{\to}put_1; put_2, get{\to}get_1; get_2)$ as before); witness Figure 11.

Due to the nature of our refinement functions, which are distinct and therefore cannot deadlock, the lax relation is indeed (strictly) weaker than the strong one.

Proposition 8.7 *For all distinct r, $\preceq^r_\forall \subseteq \preceq^r_\exists$.*

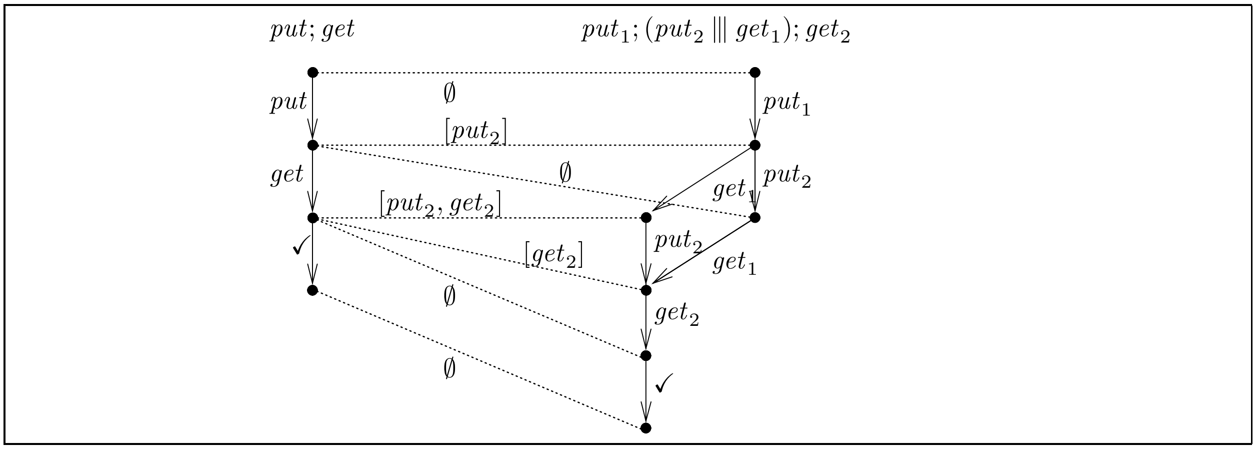


Figure 11: A lax vertical bisimulation

8.3 Requirements for vertical implementation

Apart from our one motivating example, one would like to have objective criteria to decide between the strict and lax versions of vertical bisimulation; and indeed to know whether these relations can be made part of a larger framework for design, as discussed at the start of this section. To answer these questions, we formulate a set of general proof rules for vertical implementation, and investigate to what degree either version of vertical bisimulation satisfies them.

The rules are collected in Table 15. Note that they feature a generic relation symbol $\sqsubseteq^r$ standing for any vertical implementation relation, and a generic symbol $\sqsubseteq$ standing for a horizontal implementation relation. In this context, $\sqsubseteq$ is called the *basis* of $\sqsubseteq^r$, and $\sqsubseteq^r$ a *vertical extension* of $\sqsubseteq$. We briefly discuss the rules.

- The first group of rules (R_1–R_3) expresses our basic assumption of working modulo the horizontal implementation relation. R_1 states that every term implements itself as long as no refinement takes place; R_2 says that $\sqsubseteq^{id}$ (where *id* is the identity refinement, mapping each action to itself) is compatible with the horizontal implementation relation; while R_3 explains the interplay between horizontal and vertical implementation. Note that, as a consequence, we also have the derived rule

  $$\frac{t \sqsubseteq u}{t \sqsubseteq^{id} u}$$

  that, in conjunction with R_2, ensures that $\sqsubseteq$ and $\sqsubseteq^{id}$ are indeed the same relation.

  Note also that R_1 and R_2 imply that $\sqsubseteq$ is reflexive, whereas R_1–R_3 together imply that $\sqsubseteq$ is transitive; hence $\sqsubseteq$ is a pre-order, which indeed is the standard requirement for horizontal implementation relations.

- The second group of rules (R_4–R_{10}) essentially express congruence of vertical implementation with respect to the operators of $\mathit{Lang}^{\mathit{flat}}$. For instance, if the refinement functions in these rules are set to *id*, then the properties expressed by the rules collapse to the standard pre-congruence properties of $\sqsubseteq$ for the operators of $\mathit{Lang}^{\mathit{flat}}$. (In other words, the horizontal implementation relation $\sqsubseteq$ needs to be a pre-congruence, at least.)

  Rules R_4 and R_5 simply express that deadlock and termination are independent of the abstraction level. R_6 is the core of the relationship between the refinement function r and the

Table 15: Proof rules for vertical implementation

$\dfrac{}{t \sqsubseteq^{id} t}\,\mathrm{R}_1$	$\dfrac{t \sqsubseteq^{id} u}{t \sqsubseteq u}\,\mathrm{R}_2$	$\dfrac{t \sqsubseteq t' \quad t' \sqsubseteq^r u' \quad u' \sqsubseteq u}{t \sqsubseteq^r u}\,\mathrm{R}_3$	
$\dfrac{}{\mathbf{0} \sqsubseteq^r \mathbf{0}}\,\mathrm{R}_4$	$\dfrac{}{\mathbf{1} \sqsubseteq^r \mathbf{1}}\,\mathrm{R}_5$	$\dfrac{}{\alpha \sqsubseteq^r r(\alpha)}\,\mathrm{R}_6$	$\dfrac{t_1 \sqsubseteq^r u_1 \quad t_2 \sqsubseteq^r u_2}{t_1 + t_2 \sqsubseteq^r u_1 + u_2}\,\mathrm{R}_7$
$\dfrac{t_1 \sqsubseteq^r u_1 \quad t_2 \sqsubseteq^r u_2}{t_1 ; t_2 \sqsubseteq^r u_1 ; u_2}\,\mathrm{R}_8$	$\dfrac{t \sqsubseteq^r u}{t/A \sqsubseteq^{r/A} u/\mathcal{S}_r(A)}\,\mathrm{R}_9$	$\dfrac{t_1 \sqsubseteq^r u_1 \quad t_2 \sqsubseteq^r u_2}{t_1 \parallel_A t_2 \sqsubseteq^r u_1 \parallel_{\mathcal{S}_r(A)} u_2}\,\mathrm{R}_{10}$	
$\dfrac{r(a) = u_1 ; u_2 \quad t \sqsubseteq^r v}{a ; t \sqsubseteq^r u_1 ; (u_2 \mathbin{\vert\vert\vert} v)}\,\mathrm{R}_{11}$	$\dfrac{r(a) = u_1 ; u_2 \quad t \sqsubseteq^r v_1 ; v_2}{a ; t \sqsubseteq^r u_1 ; (u_2 \mathbin{\vert\vert\vert} v_1) ; v_2}\,\mathrm{R}_{12}$	$\dfrac{t \sqsubseteq^{r_1} u}{t \sqsubseteq^{r_1 + r_2} u}\,\mathrm{R}_{13}$	

vertical implementation relation: it expresses the basic expectation that $r(a)$ should be an implementation for a. R_7–R_{10} are quite obvious, as they inductively go into the structure of the components. Note that in R_{10}, the synchronisation set A of the specification is refined in the implementation.

Rule R_9 is similar, with the proviso that the refinement images of the actions that are hidden is set to the identity. An interesting special case of R_9 is given by the following derived rule:

$$\frac{t \sqsubseteq^r u \quad A = \{a \mid r(a) \neq a\}}{t/A \sqsubseteq^{id} u/\mathcal{S}_r(A)}$$

Hence, by hiding all the actions that are properly refined, the vertical implementation is turned back into a horizontal implementation relation.

- The last group of rules (R_{11}–R_{13}) describes design steps that are different from the ones using the congruence-like rules of the previous group. In fact, these rules capture the gain over the traditional refinement operator: R_{11} and R_{12} allow certain parts of refinement images to overlap in the implementation, even if the specification imposes an ordering between the corresponding abstract actions, whereas R_{13} allows refinements to be implemented partially.

It seems natural to require that a vertical implementation relation satisfies at least rules R_1–R_{10}, and as many of the other rules as possible. In [124], we proved that this is the case for rooted vertical weak bisimulation; the proof carries over to rooted strict vertical bisimulation ($\preceq^r_\forall$).

Theorem 8.8 *$\preceq^r_\forall$ satisfies the rules in Table 15 except* R_{12} *and* R_{13}*.*

Unfortunately, lax vertical bisimilarity (Definition 8.6) does not satisfy the parallel composition rule.

Example 8.9 *Consider $r\colon a \mapsto a_1 + a_2$; let $t = a$ and $u_i = a_1$ for $i = 1, 2$. We have $t \preceq^r_\exists u_1$ and $t \preceq^r_\exists u_2$; however, $a_1 \parallel_{a_1,a_2} a_2 \simeq_{\mathsf{d}} \mathbf{0}$ and hence $t \parallel_a t \simeq_{\mathsf{d}} a \not\preceq^r_\exists \mathbf{0} \simeq_{\mathsf{d}} u_1 \parallel_{\mathcal{S}_r(a)} u_2$.*

The reason lies in the existential nature of the lax down-simulations (and also of the lax residual simulations, although the above example does not show that): the implementations of the parallel components both have to have *some* matching transition, but they need not be the same. However, this is the only proof rule that is problematic, and then only for synchronisation actions.

Theorem 8.10 *$\preceq^r_\exists$ satisfies the rules in Table 15, where* R_{10} *is replaced by the following special case:*

$$\frac{t_1 \sqsubseteq^r u_1 \quad t_2 \sqsubseteq^r u_2 \quad r/A = id}{t_1 \parallel_A t_2 \sqsubseteq^r u_1 \parallel_A u_2}$$

The situation is not yet very satisfactory: for lax vertical implementation, the failure of R_{10} is a grave disadvantage, since synchronisation is the basic interaction mechanism between system components: if one cannot synchronise on an action after its refinement, the formalism is not very useful. On the other hand, strong vertical implementation does not allow some design steps that seem intuitively reasonable and attractive —such as the one of our motivating example.

The situation is alleviated, however, by the fact that strong and lax refinement can be *combined* so as to merge the advantages of both to a large degree. Namely, it is possible to implement one operand of a synchronisation using the lax vertical bisimulation, and the other using the strong version; the resulting implementation will still be a vertical implementation, in the lax version. Formally:

Theorem 8.11 *Let $A \subseteq Act$ and assume $r/A = id$. If $t_1 \preceq^r_\forall u_1$ and $t_2 \preceq^r_\exists u_2$, then $t_1 \parallel_A t_2 \preceq^r_\exists u_1 \parallel_{\mathcal{S}_r(A)} u_2$.*

Example 8.12 *To see the possible interplay between strict and lax vertical implementation, consider a booking agent, whose function on an abstract level consists of a continuous series of book actions, and two customers that concurrently invoke this action. Algebraic specifications are given by*

$$\begin{array}{rcl} Agent_S & := & book; Agent_S \\ Users_S & := & book \mathbin{|||} book \ . \end{array}$$

Now consider a refinement r: $book \mapsto req; (yes + no)$, specifying that a booking consists of a request, which may be granted (yes) or refused (no). A possible implementation that is lax for the agent and strict for the users is given by

$$\begin{array}{rcl} Agent_I & := & req; (yes \mathbin{|||} req); Empty \\ Empty & := & (no \mathbin{|||} req); Empty \\ Users_I & := & req; (yes + no) \mathbin{|||} req; (yes + no) \ . \end{array}$$

The agent will receive the request of a second customer in parallel with the reply to the first, but replies yes to the first one only. The customers take either yes or no for an answer. We have $Agent_S \preceq^r_\exists Agent_I$ and $Users_S \preceq^r_\forall Users_I$; thus, according to Theorem 8.11, $Agent_S \parallel_{book} Users_S \preceq^r_\exists Agent_I \parallel_{req,yes,no} Users_I$.

Informally, the reason why this works is that every (down- or residual) simulated transition that exists on the lax side exists on the strict side as well; hence they can be combined into a simulated transition of the synchronised behaviour. We arrive at the following schema regarding the synchronisation of vertical implementations:

- Strict with strict yields strict;
- Strict with lax yields lax;
- Lax with lax does not yield a valid implementation.

8.4 Further developments

We briefly review two issues regarding vertical implementation, worked out in [124] for the strict case, that have been omitted from the presentation above.

Open terms and recursion. Table 15 contains no proof rule for recursion. Thus, for instance, we cannot prove the relation $\mathit{Agent}_S \preceq^r_\exists \mathit{Agent}_I$ in Example 8.12 algebraically. Intuitively, a proof rule would take the form

$$\frac{t \sqsubseteq^r u}{\mu x.t \sqsubseteq^r \mu x.u} \ .$$

However, here t and u are (in general) open terms; we have not discussed how to interpret vertical bisimulation over such. In fact, the most natural definition, which says that $t \sqsubseteq^r u$ iff all closed instances of t and u are related, is not satisfactory: for then the relation $x \sqsubseteq^r x$ would be inconsistent except for $r = id$. In fact, in $t \sqsubseteq^r u$, the occurrences of x in t (on the abstract side) should be interpreted differently from the occurrences of x in u (the concrete side). The solution developed for this in [124] (which we will not go into here) consists of adding an environment to the proof rules which records precisely these differences in interpretation. The above proof rule for recursion, extended appropriately with such environments, is sound for strict and lax vertical bisimilarity, provided we restrict ourselves to strongly guarded terms (meaning that any occurrence of the recursion variable in the body of a recursive term is guarded by a visible action).

Action substitution. When the rules in Table 15 are extended with a rule for recursion as discussed above, it can be seen that (simultaneous) action substitution in the line of Section 6 will automatically give rise to a correct implementation (with respect to any vertical implementation relation that satisfies all these rules). More precisely: let $t\{\vec{u}/\vec{a}\}$ denote the simultaneous substitution of all a_i-occurrences in t by u_i, defined through a straightforward generalisation of Table 11. Then for any vertical implementation relation $\leq^{\vec{a}\to\vec{u}}$ and any term $t \in \mathit{Lang}^{\mathit{flat}}$, we have

$$t \leq^{\vec{a}\to\vec{u}} t\{\vec{u}/\vec{a}\} \ .$$

This provides a link to the interpretation of action refinement as an operator, since in Section 6 we have seen that this operator can in many cases also be mimicked by action substitution.

8.5 Application: A simple data base

Let us consider once more the data base example of Section 2.5 (not including the *copy* operation considered in Section 4.6). We will show that, under *lax* vertical refinement, the n-state version of the data base, Data^n_S, is an implementation of the state-less (or 1-state) version, Data^1_S. The specifications are

$$\begin{array}{rcl} \mathit{Data}^1_S & := & (\mathit{qry} + \mathit{upd}); \mathit{Data}^1_S \\ \mathit{Data}^n_S & := & \mathit{State}_1 \\ \mathit{State}_i & := & \mathit{qry}_i; \mathit{State}_i + \sum_{k=1}^n \mathit{upd}_k; \mathit{State}_k \quad (\text{for } i = 1, \ldots, n) \ . \end{array}$$

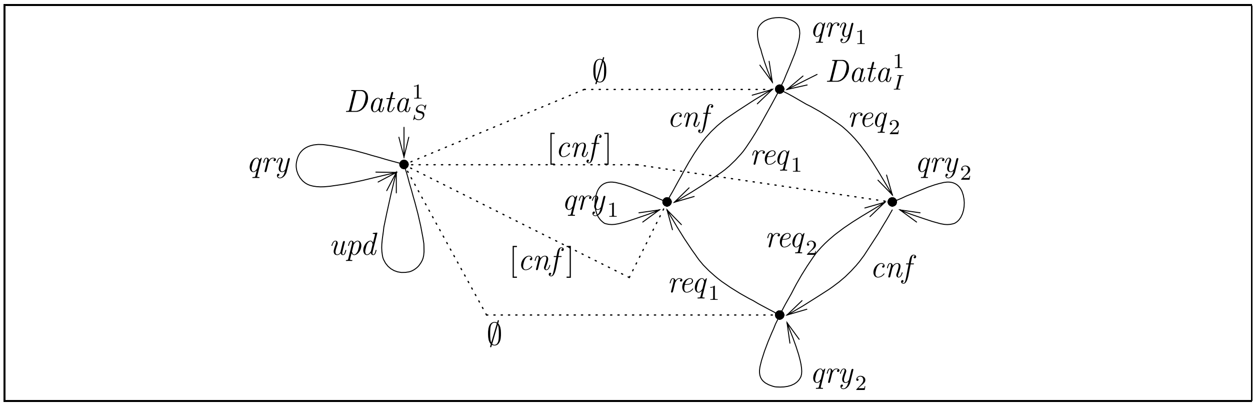


Figure 12: Lax vertical implementation of a state-less as a 2-state data base, while refining *upd*

Let r map qry to $\sum_{i=1}^{n} qry_i$ and upd to $\sum_{i=1}^{n} upd_i$; then it follows that $Data_S^1 \preceq_\exists^r Data_S^n$. In fact, this implementation relation can be proved algebraically, using the rules of Table 15 extended to cope with recursion as in [124] (briefly discussed above). The main point is that (in lax refinement) $qry \preceq_\exists^r qry_i$ for arbitrary i, i.e., it is not necessary to implement all *qry*-actions whenever *qry* is specified on the abstract level.

If we also consider a user of the data base who actually queries it, this can be modelled by a process $User_S$ synchronising with $Data_S^1$ over qry. Since $Data_S^n$ is a lax vertical implementation of $Data_S^1$, in order to refine the interaction with the user, we have to implement $User_S$ in the *strong* sense, meaning that whenever $User_S$ specifies a query, this must be implemented as the choice between all possible qry_i-actions. In fact, this is a reasonable requirement: since the user does not know the state of the data base at the moment he issues the query, he must accept all possible answers.

In a sense, the above example is not a case of proper action refinement, since all r-images terminate after a single action. Alternatively, also the *refined* n-state database can be derived as an implementation of $Data_S^1$. For instance, for the case where $n = 2$, both the implementation $Data_I^{2,seq}$ obtained in Section 2.5 (see Figure 2) and $Data_I^{2,D}$ obtained in Section 7.4 (see Figure 10) are valid lax implementations of $Data_S^1$ under the refinement r defined by

$$\begin{aligned} upd &\mapsto (req_1 + req_2); cnf \\ qry &\mapsto qry_1 + qry_2 \ . \end{aligned}$$

For instance, Figure 12 shows the lax vertical bisimulation proving $Data_S^1 \preceq_\exists^r Data_I^{2,D}$.

References

[1] L. Aceto. Full abstraction for series-parallel pomsets. In S. Abramsky and T. S. E. Maibaum, eds., *TAPSOFT '91, Volume 1*, vol. 493 of *Lecture Notes in Computer Science*, pp. 1–25. Springer-Verlag, 1991.

[2] L. Aceto. *Action Refinement in Process Algebras.* Cambridge University Press, 1992. PhD thesis of the University of Sussex.

[3] L. Aceto. History-preserving, causal and mixed-ordering equivalence over stable event structures. *Fundamenta Informaticae*, 17(4):319–331, 1992.

Table 16: Overview of language fragments and well-formedness conditions

Language	*Lang*	*seq*	*atom*	*Evt*	ST	D, *lin*	D, *bra*	*loc*	*flat*	*ref*
Section (Page)	4(29)	2(13)	3(23)	4(30)	5.4(49)	7.1(66)	7.3(68)	7.5(74)	8(74)	8(75)
$\mathbf{0}$	✓	✓	✓	✓	✓	✓		✓	✓	
$\mathbf{0}_A$							✓			
$\mathbf{1}$ 1)	✓	✓	✓	✓	✓	✓		✓	✓	
α	✓	✓	✓		✓				✓	
${}_e\alpha$				✓						
α^- 1)					✓					
a						✓	✓	✓		✓
$\mathsf{V} + \mathsf{V}$	✓	✓	✓	✓2)	✓	✓3)	✓3)	✓	✓	✓4)
$\mathsf{T}; \mathsf{V}$	✓	✓	✓	✓2)	✓			✓	✓	✓4)
$\mathsf{T} \cdot \mathsf{T}$						✓	✓			
$\mathsf{T} \parallel_A \mathsf{T}$	✓			✓2)			✓		✓	
$\mathsf{T} \parallel\mkern-3mu\mid \mathsf{T}$			✓5)			✓				
$\mathsf{T} \parallel_{A,M} \mathsf{T}$					✓					
$\Pi\vec{\mathsf{T}}$								✓		
$\langle \mathsf{V} \rangle$			✓			✓3)				
$*\mathsf{T}$ 1)			✓							
T/A	✓	✓	✓	✓	✓				✓	
$\mathsf{T}[a \rightsquigarrow \mathsf{V}]$	✓6)	✓6)	✓6)			✓3)7)				
$\mathsf{T}[a \rightsquigarrow \mathsf{V}, \vec{e} \rightsquigarrow \vec{\mathsf{T}}]$				✓2)						
$\mathsf{T}[a \rightsquigarrow \mathsf{V}, \vec{\mathsf{T}}]_M$					✓					
$\mathsf{T}[\vec{a} \rightsquigarrow \vec{\mathsf{V}}]$							✓3)8)			
$\mathsf{T}[a \rightsquigarrow \vec{\mathsf{V}}]$								✓		
x	✓	✓	✓		✓	✓	✓	✓	✓	
${}_e x$				✓						
${}_e[\mathsf{T}]$				✓						
$\mu x.\mathsf{V}$ 9)	✓10)	✓10)	✓	✓	✓	✓3)	✓3)11)	✓	✓	

Well-formedness conditions: (conditions in the left hand column apply to all languages)

1) auxiliary operator; may not occur in virgin operands (V in the grammar)
2) event annotations are compatible (Page 30)
3) operands are *not* virgin (i.e., may contain auxiliary operators)
4) terms are distinct (Page 76)
5) either operand is abstract (Page 24)
6) in Section 6, both operands are closed terms
7) refinement is D-compatible (Definition 7.1)
8) refinement is strictly D-compatible (Definition 7.10)
9) x is guarded in V (Definition 2.1); in Section 6, $\mu x.\mathsf{V}$ is well-sorted (Page 58)
10) not allowed in the fragments $Lang^{fin}$ and $Lang^{seq,fin}$
11) x is *D-guarded* in V (Definition 7.8)

[4] L. Aceto, B. Bloom, and F. W. Vaandrager. Turning SOS rules into equations. *Information and Computation*, 111(1):1–52, May 1994.

[5] L. Aceto and U. Engberg. Failures semantics for a simple process language with refinement. In S. Biswas and K. V. Nori, eds., *Foundations of Software Technology and Theoretical Computer Science*, vol. 590 of *Lecture Notes in Computer Science*, pp. 89–108. Springer-Verlag, 1991.

[6] L. Aceto, W. Fokkink, and C. Verhoef. Structural operational semantics. In *this volume*. Elsevier Science Publishers B.V., 2000.

[7] L. Aceto and M. C. B. Hennessy. Termination, deadlock, and divergence. *J. ACM*, 39(1):147–187, Jan. 1992.

[8] L. Aceto and M. C. B. Hennessy. Towards action-refinement in process algebras. *Information and Computation*, 103:204–269, 1993.

[9] L. Aceto and M. C. B. Hennessy. Adding action refinement to a finite process algebra. *Information and Computation*, 115:179–247, 1994.

[10] R. J. R. Back. A method for refining atomicity in parallel algorithms. In E. Odijk, M. Rem, and J.-C. Syre, eds., *PARLE '89, Volume I: Parallel Architectures*, vol. 366 of *Lecture Notes in Computer Science*, pp. 199–216. Springer-Verlag, 1989.

[11] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[12] J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, eds. *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, vol. 354 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[13] J. W. de Bakker and E. P. de Vink. Bisimulation semantics for concurrency with atomicity and action refinement. *Fundamenta Informaticae*, 20:3–34, 1994.

[14] H. P. Barendregt. *The Lambda Calculus*, vol. 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam etc., second edition, 1984.

[15] T. Basten. Branching bisimilarity is an equivalence indeed! *Information Processing Letters*, 58:141–147, 1998.

[16] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

[17] E. Best, R. Devillers, and J. Esparza. General refinement and recursion operators for the Petri box calculus. In P. Enjalbert, A. Finkel, and K. W. Wagner, eds., *STACS 93*, vol. 665 of *Lecture Notes in Computer Science*, pp. 130–140. Springer-Verlag, 1993.

[18] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. *J. ACM*, 42(1):232–268, Jan. 1995.

[19] G. Boudol. Atomic actions. *Bull. Eur. Ass. Theoret. Comput. Sci.*, 38:136–144, June 1989.

[20] G. Boudol and I. Castellani. On the semantics of concurrency: Partial orders and transition systems. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, eds., *TAPSOFT '87, Volume 1*, vol. 249 of *Lecture Notes in Computer Science*, pp. 123–137. Springer-Verlag, 1987.

[21] G. Boudol and I. Castellani. Flow models of distributed computations: Three equivalent semantics for CCS. *Information and Computation*, 114(2):247–314, Oct. 1994.

[22] W. Brauer, R. Gold, and W. Vogler. A survey of behaviour and equivalence preserving refinements of Petri nets. In G. Rozenberg, ed., *Advances in Petri Nets 1990*, vol. 483 of *Lecture Notes in Computer Science*, pp. 1–46. Springer-Verlag, 1990.

[23] M. Bravetti, M. Bernardo, and R. Gorrieri. Towards performance evaluation with general distributions in process algebras. In D. Sangiorgi and R. De Simone, eds., *Concur '98: Concurrency Theory*, vol. 1466 of *Lecture Notes in Computer Science*, pp. 405–422. Springer-Verlag, 1998.

[24] M. Bravetti and R. Gorrieri. Deciding and axiomatizing ST bisimulation for a process algebra with recursion and action refinement. Technical Report UBLCS-99-1, Department of Computer Science, University of Bologna, Feb. 1999.

[25] M. Bravetti and R. Gorrieri. Interactive generalized semi-markov processes. In J. Hillston and M. Silva, eds., *Proc. of the 7th Int. Workshop on Process Algebras and Performance Modeling*, pp. 83–98. Prensas Universitarias de Zaragoza, 1999. URL: `http://www.cs.unibo.it/~bravetti/papers/papm99.ps`.

[26] E. Brinksma, B. Jonsson, and F. Orava. Refining interfaces of communicating systems. In S. Abramsky and T. S. E. Maibaum, eds., *TAPSOFT '91, Volume 2*, vol. 494 of *Lecture Notes in Computer Science*, pp. 297–312. Springer-Verlag, 1991.

[27] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, July 1984.

[28] M. Broy. (inter-)action refinement: The easy way. In M. Broy, ed., *Program Design Calculi*, vol. 118 of *NATO ASI Series F: Computer and System Sciences*, pp. 121–158, 1993.

[29] M. Broy. Compositional refinement of interactive systems. *J. ACM*, 44(6):850–891, Nov. 1997.

[30] N. Busi. Raffinamento di azioni in linguaggi concorrenti. Master's thesis, Università degli Studi di Bologna, 1992. In italian.

[31] N. Busi, R. J. van Glabbeek, and R. Gorrieri. Axiomatising ST bisimulation equivalence. In E.-R. Olderog [112], pp. 169–188.

[32] L. Castellano, G. De Michelis, and L. Pomello. Concurrency vs. interleaving: An instructive example. *Bull. Eur. Ass. Theoret. Comput. Sci.*, 31:12–15, 1987. Note.

[33] F. Cherief. *Contribution à la Sémantique du Parallélisme: Bisimulations pour le Raffinement et le vrai Parallélisme.* PhD thesis, Univerisity of Grenoble, 1992. In french.

[34] F. Cherief and P. Schnoebelen. τ-bisimulation and full abstraction for refinement of actions. *Information Processing Letters*, 40:219–222, 1991.

[35] W. R. Cleaveland, ed. *Concur '92*, vol. 630 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[36] R. Costantini. *Abstraktion in ereignisbasierten Modellen verteilter Systeme.* PhD thesis, University of Hildesheim, 1995. In german.

[37] J. P. Courtiat and D. E. Saïdouni. Relating maximality-based semantics to action refinement in process algebras. In D. Hogrefe and S. Leue, eds., *Formal Description Techniques VII*, pp. 293–308. IFIP WG 6.1, Chapman-Hall, 1994.

[38] I. Czaja, R. J. van Glabbeek, and U. Goltz. Interleaving semantics and action refinement with atomic choice. In G. Rozenberg [127], pp. 89–109.

[39] P. Darondeau and P. Degano. Causal trees. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, eds., *Automata, Languages and Programming*, vol. 372 of *Lecture Notes in Computer Science*, pp. 234–248. Springer-Verlag, 1989.

[40] P. Darondeau and P. Degano. Causal trees = interleaving + causality. In I. Guessarian [82], pp. 239–255.

[41] P. Darondeau and P. Degano. About semantic action refinement. *Fundamenta Informaticae*, XIV:221–234, 1991.

[42] P. Darondeau and P. Degano. Refinement of actions in event structures and causal trees. *Theoretical Comput. Sci.*, 118:21–48, 1993.

[43] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Comput. Sci.*, 34:83–133, 1984.

[44] R. De Simone. Higher-level synchronising devices in Meije-SCCS. *Theoretical Comput. Sci.*, 37:245–267, 1985.

[45] P. Degano, R. De Nicola, and U. Montanari. On the consistency of 'truly concurrent' operational and denotational semantics. In *Third Annual Symposium on Logic in Computer Science*, pp. 133–141. IEEE, Computer Society Press, 1988.

[46] P. Degano, R. De Nicola, and U. Montanari. Partial orderings descriptions and observations of nondeterministic concurrent processes. In J. W. de Bakker et al. [12], pp. 438–466.

[47] P. Degano, R. De Nicola, and U. Montanari. A partial ordering semantics for CCS. *Theoretical Comput. Sci.*, 75:223–262, 1991.

[48] P. Degano and R. Gorrieri. Atomic refinement for process description languages. In A. Tarlecki [129], pp. 121–130. Extended abstract; full version: Technical Report 17–91, Hewlett-Packard Pisa Science Center.

[49] P. Degano and R. Gorrieri. An operational definition of action refinement. Technical Report TR–28/92, Università di Pisa, 1992.

[50] P. Degano and R. Gorrieri. A causal operational semantics of action refinement. *Information and Computation*, 122:97–119, 1995.

[51] P. Degano, R. Gorrieri, and G. Rosolini. A categorical view of process refinement. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, eds., *Semantics: Foundations and Applications*, vol. 666 of *Lecture Notes in Computer Science*, pp. 138–153. Springer-Verlag, 1992.

[52] R. Devillers. Maximality preserving bisimulation. *Theoretical Comput. Sci.*, 102:165–183, 1992.

[53] J. V. Echagüe Zappettini. *Sémantique des Systèmes Réactifs: Raffinement, Bisimulations et Sémantique Opérationnelle Structurée dans les Systeèmes de Transitions Asynchrones.* PhD thesis, Institut National Polytechnique de Grenoble, 1993. In french.

[54] R. Gerth, R. Kuiper, and J. Segers. Interface refinement in reactive systems. In W. R. Cleaveland [35], pp. 77–93.

[55] J. L. Gischer. The equational theory of pomsets. *Theoretical Comput. Sci.*, 61:199–224, 1988.

[56] R. J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions.* PhD thesis, Free University of Amsterdam, 1990.

[57] R. J. van Glabbeek. The refinement theorem for ST-bisimulation semantics. In *Programming Concepts and Methods.* IFIP, North-Holland Publishing Company, 1990.

[58] R. J. van Glabbeek. The meaning of negative premises in transition system specifications II. In F. Meyer auf der Heide and B. Monien, eds., *Automata, Languages and Programming*, vol. 1099 of *Lecture Notes in Computer Science*, pp. 502–513. Springer-Verlag, 1995. Full report version: STAN-CS-TN-95-16, Department of Computer Science, Stanford University.

[59] R. J. van Glabbeek. The linear time – branching time spectrum, parts I and II. In *this volume.* Elsevier Science Publishers B.V., 2000.

[60] R. J. van Glabbeek and U. Goltz. Equivalence notions for concurrent systems and refinement of actions. In A. Kreczmar and G. Mirkowska, eds., *Mathematical Foundations of Computer Science 1989*, vol. 379 of *Lecture Notes in Computer Science*, pp. 237–248. Springer-Verlag, 1989. Revised and extended in [64].

[61] R. J. van Glabbeek and U. Goltz. A deadlock-sensitive congruence for action refinement. SFB-Bericht 342/23/90 A, Technische Universität München, Institut für Informatik, Nov. 1990.

[62] R. J. van Glabbeek and U. Goltz. Equivalences and refinement. In I. Guessarian [82], pp. 309–333. Revised and extended in [64].

[63] R. J. van Glabbeek and U. Goltz. Refinement of actions in causality based models. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, eds., *Stepwise Refinement of Distributed Systems — Models, Formalisms, Correctness*, vol. 430 of *Lecture Notes in Computer Science*, pp. 267–300. Springer-Verlag, 1990. Revised and extended in [64].

[64] R. J. van Glabbeek and U. Goltz. Refinement of actions and equivalence notions for concurrent systems. Hildesheimer Informatik-Bericht 6/98, University of Hildesheim, 1998.

[65] R. J. van Glabbeek and F. W. Vaandrager. Petri Net models for algebraic theories of concurrency. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, eds., *PARLE — Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, vol. 259 of *Lecture Notes in Computer Science*, pp. 224–242. Springer-Verlag, 1987.

[66] R. J. van Glabbeek and F. W. Vaandrager. The difference between splitting in n and $n+1$. *Information and Computation*, 136(2):109–142, 1997.

[67] R. J. van Glabbeek and W. P. Weijland. Refinement in branching time semantics. In *Proceedings AMAST Conference*, pp. 197–201. Iowa State University, 1989.

[68] R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43(3):555–600, May 1996.

[69] U. Goltz, R. Gorrieri, and A. Rensink. Comparing syntactic and semantic action refinement. *Information and Computation*, 125(2):118–143, Mar. 1996.

[70] U. Goltz and H. Wehrheim. Modelling causality via action dependencies in branching time semantics. *Information Processing Letters*, 59:179–184, 1996.

[71] R. Gorrieri. *Refinement, Atomicity and Transactions for Process Description Languages.* PhD thesis, Università di Pisa, 1991. Report no. TD–2/91.

[72] R. Gorrieri. A hierarchy of system descriptions via atomic linear refinement. *Fundamenta Informaticae*, 16:289–336, 1992.

[73] R. Gorrieri and C. Laneve. The limit of split$_n$-bisimulations for CCS agents. In A. Tarlecki [129], pp. 170–180.

[74] R. Gorrieri and C. Laneve. Split and ST bisimulation semantics. *Information and Computation*, 116(1):272–288, Jan. 1995.

[75] R. Gorrieri, S. Marchetti, and U. Montanari. A^{2}CSS: Atomic actions for CCS. *Theoretical Comput. Sci.*, 72:203–223, 1990.

[76] R. Gorrieri and U. Montanari. Towards hierarchical specification of systems: A proof system for strong prefixing. *Int. Journal of Foundations of Computer Science*, 11(3):277–293, 1990.

[77] R. Gorrieri and U. Montanari. On the implementation of concurrent calculi in net calculi: Two case studies. *Theoretical Comput. Sci.*, 141:195–252, 1995.

[78] R. Gorrieri, M. Roccetti, and E. Stancampiano. A theory of processes with durational actions. *Theoretical Comput. Sci.*, 140:73–94, 1995.

[79] J. Grabowski. On partial languages. *Fundamenta Informaticae*, IV(2):427–498, 1981.

[80] J. F. Groote. Transition system specifications with negative premises. *Theoretical Comput. Sci.*, 118:263–299, 1993.

[81] J. F. Groote and F. W. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100:202–260, 1992.

[82] I. Guessarian, ed. *Semantics of Systems of Concurrent Processes*, vol. 469 of *Lecture Notes in Computer Science.* Springer-Verlag, 1990.

[83] J. I. den Hartog, E. P. de Vink, and J. W. de Bakker. Full abstractness of an interleaving semantics for action refinement. Rapport IR–443, Faculteit der Wiskunde en Informatica, Vrije Universiteit Amsterdam, Feb. 1998.

[84] M. C. B. Hennessy. Axiomatising finite concurrent processes. *SIAM J. Comput.*, 15(5):997–1017, Oct. 1988.

[85] M. C. B. Hennessy. Concurrent testing of processes. *Acta Informatica*, 32(6):509–543, 1995.

[86] M. C. B. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138(2):353–389, Feb. 1995.

[87] M. C. B. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 23(1):137–161, Jan. 1985.

[88] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[89] M. Huhn. Action refinement and property inheritance in systems of sequential agents. In U. Montanari and V. Sassone, eds., *Concur '96: Concurrency Theory*, vol. 1119 of *Lecture Notes in Computer Science*, pp. 639–654. Springer-Verlag, 1996.

[90] M. Huhn. *On the Hierarchical Design of Distributed Systems*. PhD thesis, University of Hildesheim, 1997.

[91] W. Janssen, M. Poel, and J. Zwiers. Action systems and action refinement in the development of parallel systems. In J. C. M. Baeten and J. F. Groote, eds., *Concur '91*, vol. 527 of *Lecture Notes in Computer Science*, pp. 298–316. Springer-Verlag, 1991.

[92] L. Jategaonkar. *Observing "True" Concurrency*. PhD thesis, Massachusetts Institute of Technology, 1993. Available as report MIT/LCS/TR-618.

[93] L. Jategaonkar and A. R. Meyer. Testing equivalences for Petri nets with action refinement. In W. R. Cleaveland [35], pp. 17–31.

[94] L. Jategaonkar and A. R. Meyer. Self-synchronization of concurrent processes. In *Eighth Annual IEEE Symposium on Logic in Computer Science*, pp. 409–417. IEEE, Computer Society Press, 1993.

[95] B. Jónsson. Arithmetic of ordered sets. In I. Rival, ed., *Ordered Sets*, pp. 3–41. Reidel Pub. Co., 1982.

[96] S. Katz. Refinement with global equivalence proofs in temporal logic. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 29:59–78, 1997.

[97] P. M. W. Knijnenburg and J. N. Kok. The semantics of the combination of atomized statements and parallel choice. *Formal Aspects of Computing*, 9(5/6):518–536, 1997.

[98] L. Lamport. On interprocess communication, part I: Basic formalisms. *Distributed Computing*, 1:77–85, 1986.

[99] R. Langerak. *Transformations and Semantics for LOTOS*. PhD thesis, University of Twente, Nov. 1992.

[100] R. Langerak. Bundle event structures: a non-interleaving semantics for LOTOS. In M. Díaz and R. Groz, eds., *Formal Description Techniques — V*, pp. 203–218. IFIP WG 6.1, North-Holland Publishing Company, 1993.

[101] K. S. Larsen. A fully abstract model for a process algebra with refinements. Master's thesis, Aarhus University, 1988.

[102] R. Loogen and U. Goltz. Modelling nondeterministic concurrent processes with event structures. *Fundamenta Informaticae*, XIV:39–73, 1991.

[103] A. Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Report PB–78, Aarhus University, 1977.

[104] A. Mazurkiewicz. Basic notions of trace theory. In J. W. de Bakker et al. [12], pp. 285–363.

[105] A. Meyer. Observing truly concurrent processes. In M. Hagiya and J. C. Mitchell, eds., *Theoretical Aspects of Computer Software*, vol. 789 of *Lecture Notes in Computer Science*, p. 886. Springer-Verlag, Apr. 1994.

[106] A. Meyer. Concurrent process equivalences: Some decision problems. In E. W. Mayr and C. Puech, eds., *STACS 95*, vol. 900 of *Lecture Notes in Computer Science*, p. 349. Springer-Verlag, 1995.

[107] R. Milner. A modal characterisation of observable machine-behaviour. In E. Astesiano and C. Böhm, eds., *CAAP '81*, vol. 112 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[108] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[109] F. Moller. *Axioms for Concurrency.* PhD thesis, University of Edinburgh, 1989. Available as report CST–59–89 and ECS–LFCS–89–84.

[110] M. Nielsen, U. Engberg, and K. S. Larsen. Fully abstract models for a process language with refinement. In J. W. de Bakker et al. [12], pp. 523–549.

[111] A. Obaid and L. Logrippo. An atomic calculus of communicating systems. In H. Rudin and C. H. West, eds., *Protocol Specification, Testing, and Verification, VII.* IFIP WG 6.1, North-Holland Publishing Company, 1987.

[112] E.-R. Olderog, ed. *Programming Concepts, Methods and Calculi*, vol. A–56 of *IFIP Transactions.* IFIP, 1994.

[113] E.-R. Olderog and C. A. R. Hoare. Specification-oriented semantics for communicating processes. *Acta Inf.*, 23:9–66, 1986.

[114] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[115] V. R. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[116] A. Rabinovich and B. A. Trakhtenbrot. Behaviour structure and nets. *Fundamenta Informaticae*, XI(4):357–404, Dec. 1988.

[117] A. Rensink. Posets for configurations! In W. R. Cleaveland [35], pp. 269–285.

[118] A. Rensink. *Models and Methods for Action Refinement.* PhD thesis, University of Twente, Enschede, Netherlands, Aug. 1993.

[119] A. Rensink. Methodological aspects of action refinement. In E.-R. Olderog [112], pp. 227–246.

[120] A. Rensink. Causal traces. Hildesheimer Informatik-Bericht 39/95, Institut für Informatik, University of Hildesheim, Dec. 1995.

[121] A. Rensink. An event-based SOS for a language with refinement. In J. Desel, ed., *Structures in Concurrency Theory*, Workshops in Computing, pp. 294–309. Springer-Verlag, 1995.

[122] A. Rensink. Algebra and theory of order-deterministic pomsets. *Notre Dame Journal of Formal Logic*, 37(2):283–320, 1996.

[123] A. Rensink and R. Gorrieri. Action refinement as an implementation relation. In M. Bidoit and M. Dauchet, eds., *TAPSOFT '97: Theory and Practice of Software Development*, vol. 1214 of *Lecture Notes in Computer Science*, pp. 772–786. Springer-Verlag, 1997. Revised in [124].

[124] A. Rensink and R. Gorrieri. Vertical bisimulation. Hildesheimer Informatik-Bericht 9/98, University of Hildesheim, June 1998. To appear in Information & Computation. URL: `http://www.cs.utwente.nl/~rensink/HIB98-9.ps.gz`.

[125] A. Rensink and H. Wehrheim. Weak sequential composition in process algebras. In B. Jonsson and J. Parrow, eds., *Concur '94: Concurrency Theory*, vol. 836 of *Lecture Notes in Computer Science*, pp. 226–241. Springer-Verlag, 1994.

[126] A. Rensink and H. Wehrheim. Dependency-based action refinement. In I. Prívara and P. Ruzicka, eds., *Mathematical Foundations of Computer Science 1997*, vol. 1295 of *Lecture Notes in Computer Science*, pp. 468–477. Springer-Verlag, 1997. Extended report version: CTIT Technical Report 99–02, University of Twente; URL: `http://www.cs.utwente.nl/~rensink/CTIT-99-02.ps.gz`.

[127] G. Rozenberg, ed. *Advances in Petri Nets 1992*, vol. 609 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[128] D. E. Saïdouni. *Sémantique de Maximalité: Application au Raffinement d'Actions dans LOTOS*. PhD thesis, LAAS-CNRS, Toulouse, 1996. In french.

[129] A. Tarlecki, ed. *Mathematical Foundations of Computer Science 1991*, vol. 520 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[130] R. Valette. Analysis of Petri Nets by stepwise refinement. *J. Comput. Syst. Sci.*, 18:35–46, 1979.

[131] W. Vogler. Failures semantics based on interval semiwords is a congruence for refinement. *Distributed Computing*, 4:139–162, 1991.

[132] W. Vogler. *Modular Construction and Partial Order Semantics of Petri Nets*, vol. 625 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[133] W. Vogler. Bisimulation and action refinement. *Theoretical Comput. Sci.*, 114:173–200, 1993.

[134] W. Vogler. The limit of Split$_n$-language equivalence. *Information and Computation*, 127(1):41–61, 1996.

[135] D. J. Walker. Bisimulation and divergence. *Information and Computation*, 85:202–241, 1990.

[136] H. Wehrheim. Parametric action refinement. In E.-R. Olderog [112], pp. 247–266. Full report version: Hildesheimer Informatik-Berichte 18/93, Institut für Informatik, University of Hildesheim, Nov. 1993.

[137] H. Wehrheim. *Specifying Reactive Systems with Action Dependencies*. PhD thesis, University of Hildesheim, 1996.

[138] G. Winskel. Synchronization trees. *Theoretical Comput. Sci.*, 34:33–82, 1984.

[139] G. Winskel. Event structures. In W. Brauer, W. Reisig, and G. Rozenberg, eds., *Petri Nets: Applications and Relationships to Other Models of Concurrency*, vol. 255 of *Lecture Notes in Computer Science*, pp. 325–392. Springer-Verlag, 1987.

[140] G. Winskel. An introduction to event structures. In J. W. de Bakker et al. [12], pp. 364–397.

[141] N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, Apr. 1971.

[142] J. Zwiers and W. Janssen. Partial order based design of concurrent systems. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, eds., *A Decade of Concurrency*, vol. 803 of *Lecture Notes in Computer Science*, pp. 622–684. Springer-Verlag, 1994.