HILDESHEIMER INFORMATIK-BERICHTE

ISSN 0941-3014

Abstraction and Refinement in Configuration Structures

Ruggero Costantini
Arend Rensink

18/92 (November 1992)

UNIVERSITÄT HILDESHEIM

Dieser Bericht ist herausgegeben vom

Institut für Informatik

Postfach 10 13 63
31113 Hildesheim

(This page intentionally left blank)

Abstraction and Refinement in Configuration Structures

Ruggero Costantini, University of Hildesheim
costa@informatik.uni-hildesheim.de

Arend Rensink, University of Twente[i]
rensink@cs.utwente.nl

Abstract

An abstraction operator for configuration structures is defined and it is proven that it is left inverse to the traditional refinement operator. The abstraction operator describes how concrete behaviour looks when observed from a more abstract level, where the difference between concrete and abstract is given by a *transformation mapping*. This generates a notion of implementation: $\mathcal{L}$ is said to implement $\mathcal{H}$ iff $\mathcal{L}$ is mapped to $\mathcal{H}$ by the abstraction operator. The implementation *relation* generated by the abstraction operator is strictly more general than the implementation *function* defined by a refinement operator, thus allowing a more flexible design process for distributed systems.

1 Introduction

Many of the formalisms that have been proposed and used for the description of concurrent systems are based on the notion of *actions*. An action can be seen as a name for a certain small piece of behaviour that can be invoked repeatedly. It turns out to entail considerable simplification in models and theory to assume that actions are *atomic*, i.e. not decomposable. On the other hand, it would seem that the concept of *procedures*, borrowed from sequential programming, could in principle be applied to actions, which would mean that the actions one has specified are replaced by procedures, i.e. more complex behaviours, in later phases of the design. This principle is commonly called *action refinement*. Because it crucially relies on the idea that actions are decomposable, which is clearly at odds with the assumption that actions are atomic, most if not all of the work regarding action refinement has been done in *causality based* behaviour models which do not rely on the latter assumption.

It is important to be aware of a certain overloading of the term *refinement*: it is used both for the *principle* of decomposing an action into a more complex behaviour and for the *operation* that one can define on the basis of this principle. Such an operation is usually based on some sort of *substitution* whereby single actions are replaced (in some model) by more complex objects. This type of operation is the central issue in much of the research on action refinement; cf. [1, 2, 3, 9, 18, 19, 22, 20, 23, 11, 15, 26, 25].

In this paper we study the principle on the basis of a *different* operation, viz. *abstraction*, which allows one to regard complex subbehaviour as a single action. Intuitively, abstraction is the inverse of refinement: if one refines a given behaviour and subsequently abstracts it, one would expect to get back the same behaviour. We will show that our operation is indeed

[i] This work was done while the author was on leave at the University of Hildesheim

the *left inverse* of the traditional refinement operation. Moreover, we show on the basis of a host of examples how the abstraction operation may be used to formulate a notion of *design by action refinement* which is more flexible than what can be obtained from a refinement operation.

Because this is only a first study, we do not deal with action refinement in full generality: we limit ourselves to *event refinement*, which can be seen as a simplified version of action refinement. An *event* corresponds to the *execution* of an action; hence there is a one-to-many relation between actions and events (or equivalently, a total function from events to actions). The effect is that events occur only once in any given run (although they may occur in different circumstances during different runs). This means that we can be much more precise about *which* event is being refined. This simplifies matters appreciably. We will show that it nevertheless gives rise to a nontrivial and interesting notion of design.

A further restriction is that we only investigate the refinement of *one event at a time*. For finite behaviour, this restriction is not so important since in order to refine all events one may simply repeat the process of refining a single one (although it turns out that one must be careful about the order in which the events are treated). For infinite behaviour however more powerful refinements may be necessary.

Refinement will be based on so-called *transformation pairs* $(t, \mathcal{T})$, where t is the event to be refined and $\mathcal{T}$ the behaviour that it is refined into. (We have chosen to call this a *transformation* pair to avoid further overloading of the term *refinement*.) On the basis of such a pair we define an *abstraction operator*, which yields for a given concrete, *low-level* behaviour $\mathcal{L}$ the corresponding abstract, *high-level* behaviour $\mathcal{H}$, if it exists.

1.1 System design

We digress slightly to discuss a general approach to system design known from the literature; see e.g. [5, 17]. We then show how event refinement can be made to fit into this general approach.

Formally, the description of behaviour consists of a process term in some language, *plus an implementation relation* over the language. In general, the latter is a *preorder*; however, in this paper we are interested only in the case where the implementation relation is an *equivalence relation*, denoted $\simeq$. This relation expresses which terms are considered to describe the same behaviour.[1] We will use letters $\mathcal{I}, \mathcal{S}$ etc. to denote terms. $\mathcal{I}$ is said to *implement* $\mathcal{S}$ if $\mathcal{I} \simeq \mathcal{S}$. A *design step* consists of moving from a given term to an implementation of it; i.e. going from $\mathcal{S}$ to $\mathcal{I}$ is a design step.

Due to the fact that we have chosen to work with an equivalence relation, this notion of design is in itself not very interesting, since it is symmetric: going from $\mathcal{I}$ back to $\mathcal{S}$ is also a design step. However, one can introduce some *side constraints* in the design process concerning, for instance, the syntactic structure of terms. This effectively means that candidates for $\mathcal{I}$ should satisfy some additional criteria apart from $\mathcal{I} \simeq \mathcal{S}$. A prime example of this principle is the implementation of *constraint oriented style* in *resource oriented style* described in [6, 24, 17].

We illustrate the principle of side constraints with a design problem documented in [7]. A given behaviour $\mathcal{S}$ is to be distributed over two components $\mathcal{I}_1$ and $\mathcal{I}_2$ which communicate synchronously over some local set of channels A, such that the actions of $\mathcal{S}$ are partitioned

[1] This is the same as saying that the behaviour is specified by an *equivalence class* of process terms.

over the components in a predetermined way. Hence apart from satisfying $\mathcal{I} \simeq \mathcal{S}$, the implementation $\mathcal{I}$ is required to be of the form (in LOTOS notation)

$$\mathcal{I} = \textbf{hide } A \textbf{ in } \mathcal{I}_1 \parallel_A \mathcal{I}_2$$

where the components $\mathcal{I}_1$ and $\mathcal{I}_2$ have fixed label sets disjoint from A which partition the label set of $\mathcal{S}$. This is an example of the resource oriented style mentioned above: the reason for requiring this structure of $\mathcal{I}$ is that it supposedly reflects the resources of the system that it will be implemented on.

This principle can also be applied to design by event refinement. In this paper, implementing a given term $\mathcal{S}$ using event refinement will mean to come up with a model $\mathcal{H} \simeq \mathcal{S}$ satisfying the following additional constraint:

$$\mathcal{H} = abs\ \mathcal{L}$$

where *abs* denotes an *abstraction operator* and $\mathcal{L}$ some *low-level behaviour*. The abstraction operator will be indexed by the transformation pair $(t, \mathcal{T})$ denoting the behaviour transformation involved. Hence there is again a side constraint to the design step, in the form of a fixed syntactic format.

In this paper we do not actually deal with a language but rather on the level of models. To turn the above principle into practice, we define an (indexed family of) *abs* operator(s) over *configuration structures*, which constitute a very general event-based model. Abstraction consists of regarding a part of the low-level behaviour, possibly consisting of many events, as a single high-level event. The equivalence relation $\simeq$ that we use will be equality of configuration structures. A design step in this setup will be a *triple* consisting of the high- and low-level behaviour and the transformation pair involved.

Now let us contrast this with the notion of design by (event) refinement induced by the *refinement operators* investigated extensively in the literature. Given an abstract term $\mathcal{S}$ and a transformation mapping of the kind mentioned above, one defines the *refinement* of $\mathcal{S}$ as a concrete model

$$\mathcal{L} = ref\ \mathcal{S}$$

where *ref* is an (indexed family of) operator(s) replacing high-level events in $\mathcal{S}$ by their transformed behaviour. The question that is being asked is then typically whether the refinement operator and the equivalence relation are such that models equivalent to $\mathcal{S}$ are refined into models equivalent to $\mathcal{L}$, i.e. whether the following holds:

$$\mathcal{S} \simeq \mathcal{S}' \Longrightarrow ref\ \mathcal{S} \simeq ref\ \mathcal{S}' \ .$$

This yields a more restrictive notion of design than the one we are proposing. Given a *ref* operator of the traditional kind, one may define a partial operator *abs* as its left inverse. With this operator one can proceed in the way discussed above, with the following effect: the model $\mathcal{L}$ constructed by directly refining $\mathcal{S}$ is indeed an implementation, since

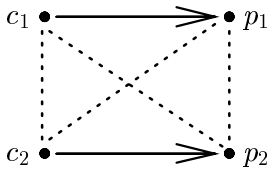
$$abs\ \mathcal{L} = abs\ ref\ \mathcal{S} \simeq \mathcal{S}$$

by definition of *abs*. Hence the traditional approach to refinement does provide a design step in our sense. On the other hand, given an *abs* operator there does not exist a *ref* which characterizes every valid implementation, because abstraction will in general not be injective up to $\simeq$.

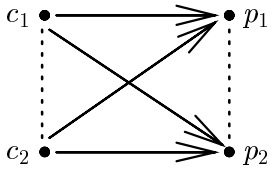
1.2 Example

Let us illustrate the difference with a small example, informally presented, which gives a taste of a larger example discussed more formally in Section 5. The example concerns *printing a file*. Assume that to print the file one first has to convert it from one format (e.g. DVI) to another (e.g. postscript) with an event c, and then print with in an event p; this behaviour $\mathcal{S}$ may be modelled by

The arrow in this picture signifies a *causal relation*: the printing is *causally preceded* by the conversion. Now suppose that this is implemented by a system consisting of *two* smart printers, each of which can do the conversion and the printing of a given file consecutively. Hence c is refined into the choice between c_1 and c_2, and p is refined into the choice between p_1 and p_2. The concrete system $\mathcal{L}$ may be modelled by



The dotted lines signify that there is a *conflict* between events: only one of every pair of events related so can actually occur. Below we define an abstraction operator which will indeed convert the second behaviour into the first; hence this is a valid design step. The traditional *refinement operator* however does not construct this system but instead the following:



Here there is a different connection between the conversion and print events: the file may be converted by one printer and printed by the other![2] Now this may in some cases be just what one would like, and in fact our abstraction operator converts this model into the high-level one also. The point is however that there is nothing optional about the refinement operator: it will always yield the same system, regardless whether this is what the designer had in mind or not.

We should mention a paper formalizing a notion of abstraction much like ours: Janssen, Poel and Zwiers [10]. Their approach is based on a concrete interpretation of the notion of causality: one event *must be* causally related to another *if and only if* they share so-called "resources." An event transformation is valid if and only if the high-level event and its low-level behaviour have exactly the same set of resources; *given the distribution of the resources*

[2] A complete and formal definition of the operational intuition behind flow event structures will be given later in this paper. Let us here just remark that the events p_1 and p_2 are enabled to occur if *one* of their conflicting causes c_1 and c_2 has occured.

in the low-level behaviour, the causality on the low level is completely determined —hence with that additional information they in fact have a refinement operator. Their approach is however limited to a trace-like model where conflict is not an issue, whereas the example above already shows that our approach has the capability to deal with conflict as well as causality.

The remainder of the paper is structured as follows. Section 2 defines the class of models that we are working with, and Section 3 defines the abstraction operator we are using, relative to a given transformation pair. A number of examples are included to clarify our intuitions. Section 4 shows that the traditional refinement as used by Van Glabbeek and Goltz in e.g. [21] is compatible to our abstraction operator —in other words, that abstraction is a left inverse of refinement. Finally, in Section 5 we present a couple of slightly larger examples, one of which is an extended and formalized version of the one presented above, whereas the other has to do with the design of a *semaphore algorithm*.

2 Definitions

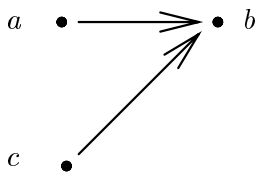
There are two kinds of models we use in this paper: *flow event structures* and their underlying *configuration structures*. The flow event structures are convenient to demonstrate our intuitions about design-by-refinement, whereas the definition of the abstraction operator has at present only been worked out on the level of configuration structures.

Let $\mathbf{E}$ be a fixed set of events.

2.1 Definition. A *flow event structure* is a tuple $\mathcal{E} = \langle E, \prec, \# \rangle$ where

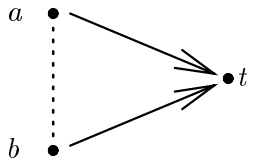
- $E \subseteq \mathbf{E}$ is a set of *events*;
- $\prec \subseteq E \times E$ is an irreflexive *flow relation*;
- $\# \subseteq E \times E$ is a symmetric *conflict relation*;

The components of a flow event structure $\mathcal{E}$ are denoted $E_{\mathcal{E}}$, $\prec_{\mathcal{E}}$ etc. The class of flow event structures will be denoted $\mathbf{F}$ and ranged over by $\mathcal{E}, \mathcal{F}$. Flow event structures can convienently be represented by pictures where causal relationships $e \prec f$ are depicted by arrows $e \bullet \longrightarrow \bullet f$ and conflicts $e \# f$ by dotted lines $e \bullet \cdots\cdots \bullet f$. For instance, the following picture represents a system that first performs events a and c independently, and then b which causally depends on both a and c:



(2.2)

On the other hand, the following picture represents a *choice* between a and b; whichever of the two is chosen, event t is performed afterwards.



(2.3)

The operational intuition behind flow event structures is formally defined by the *configurations* that it may execute, as follows:

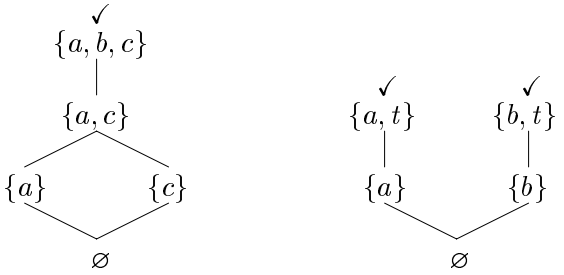
2.4 Definition. Let $\mathcal{E}$ be a flow event structure and $F \subseteq E_{\mathcal{E}}$ a subset of the events of $\mathcal{E}$. F is a *configuration of* $\mathcal{E}$ if it satisfies the following conditions:

- F does not contain flow-cycles, i.e. $(\prec_{\mathcal{E}} \cap (F \times F))^+$ (the transitive closure of $\prec_{\mathcal{E}}$) is irreflexive;
- F is conflict-free, i.e. $\neg\exists d, e \in F.\, d \#_{\mathcal{E}} e$;
- F is closed under non-conflicting causes: $\forall d \in (E_{\mathcal{E}} \setminus F).\, \forall e \in F.\, d \prec_{\mathcal{E}} e \implies \exists d' \in F.\, d \#_{\mathcal{E}} d' \prec_{\mathcal{E}} e$.

If F is a configuration of $\mathcal{E}$ then F is called *complete* if it satisfies

- all events not in F are in conflict with some event in F: $\forall e \in (E_{\mathcal{E}} \setminus F).\, \exists d \in F.\, d \# e$.

The following represents the sets of configurations derived from the flow event structures pictured in resp. (2.2) and (2.3), where the inclusion of configurations is represented by connecting lines, and the complete configurations are marked with ticks ✓.



(2.5)

Causality and conflict are reflected in the configuration structure in the following way: $d \# e$ implies that there is no configuration F such that $\{d, e\} \subseteq F$, whereas $d \prec e$ implies that there is no *pair* of configurations F, G such that $d \in F \subseteq G$ and $e \in G \setminus F$, i.e. such that e occurs "later" than d. The latter implies that if $d \prec e$ and $d, e \in F$ for a given configuration F, then $F \setminus \{d\}$ is not a configuration.

The set of configurations of a flow event structure has a certain order-theoretic structure that makes it a so-called *configuration structure*, defined below. In this way, configuration structures form an underlying semantic model for flow event structures —cf. Boudol and Castellani [4], Van Glabbeek and Goltz [21].

2.6 Definition. A *configuration structure* is a tuple $\mathcal{C} = \langle C, \checkmark \rangle$ where

- $C \subseteq \mathit{Fin}(\mathbf{E})$ is a set of finite *configurations* such that
  1. $\varnothing \in C$;
  2. $\forall F, G, H \in C.\, F \cup G \subseteq H \implies F \cup G \in C$;
  3. $\forall F \in C.\, \forall d, e \in F.\, d \neq e \implies \exists G \in C.\, G \subseteq F \wedge (d \in G \iff e \notin G)$;
- $\checkmark \subseteq C$ is a *termination predicate*, used in postfix notation, such that
  4. $\forall F \in C.\, (\exists G \in C.\, F \subset G) \implies \neg F\checkmark$;

  in other words, all terminated configurations are maximal w.r.t. $\subseteq$.

We will call $\mathcal{C}$ *nonempty* if $C \neq \{\varnothing\}$ and *terminating* if the implication in condition 4 can be turned around; i.e. if every maximal configuration of C is terminated. The class of all configuration structures is denoted $\mathbf{C}$ and ranged over by the calligraphic letters $\mathcal{C}, \mathcal{H}, \mathcal{L}, \mathcal{T}$. Now the statement above about the relation between flow event structures and configuration structures can be formalized.

2.7 Proposition. If $\mathcal{E}$ is a flow event structure, then $\langle C_{\mathcal{E}}, \checkmark_{\mathcal{E}} \rangle$ is a configuration structure, where

- $C_{\mathcal{E}}$ is the set of configurations of $\mathcal{E}$;
- $\checkmark_{\mathcal{E}}$ is the set of *complete* configurations of $\mathcal{E}$.

An important fact is that not *all* configuration structures can be obtained in this way; see e.g. [4, 16]. In Section 3 we give an example of a configuration structure for which no flow event structure exists.

2.8 Definition. Let $\mathcal{C}$ be a configuration structure and $F \in C_{\mathcal{C}}$ a configuration. A subset $E \subseteq F$ is *maximal* in F if $F \smallsetminus E \in C_{\mathcal{C}}$.

If events are thought to take a positive amount of time to complete, then only the *maximal* events of a configuration may be "busy," i.e. uncompleted. On the other hand, if a set of events E is *not* maximal in a configuration F then conceptually (some of) the events in $F \smallsetminus E$ have been *caused* by (some of) the events in E.

We will call an *event* $e \in F$ maximal in F if $\{e\}$ is maximal in F. Maximality of events is related to the flow relation as follows: if $\mathcal{E}$ is a flow event structure and $F \in C_{\mathcal{E}}$, then $e \in F$ is maximal if and only if it is maximal (in F) w.r.t. $\prec_{\mathcal{E}}$. For instance in the left hand side of (2.5), a is maximal in $\{a, c\}$, because $\{a, c\} \smallsetminus \{a\}$ $(= \{c\})$ is a configuration. In the corresponding flow event structure in (2.2), $\{a, c\}$ is unordered w.r.t. $\prec$ so that a is certainly maximal w.r.t. $\prec$. However, a is not maximal in $\{a, b, c\}$: $\{a, b, c\} \smallsetminus \{a\}$ is not a configuration in (2.5) and $a \prec_{\mathcal{E}} b$ in (2.2).

3 Abstraction

As discussed in the introduction, we will define an operation between systems which conceptually exist on different levels of abstraction. The basis of the operation is a relation between *events* of the high-level system and *entire subsystems* of the low-level system. In most of this paper we deal only with *single-event transformations*, which is to say that the transformation applies to a single high-level event only. Hence our transformations will in fact be *pairs* consisting of a single high-level event and its corresponding low-level behaviour.

3.1 Definition. An *event transformation* is a pair $(t, \mathcal{T})$ where

- $t \in \mathbf{E} \setminus E_{\mathcal{T}}$ is the high-level event to be transformed;
- $\mathcal{T} \in \mathbf{C}$ is a *nonempty* and *terminating* configuration structure, representing the low-level behaviour into which it is transformed.

We will often use just t to denote the pair $(t, \mathcal{T})$. The restriction of $\mathcal{T}$ to nonempty systems is standard; the argument is that specified high-level behaviour should not simply disappear during implementation. The restriction to terminating $\mathcal{T}$ can be argued from an intuitive and from a more technical point of view:

- The high-level event t always terminates. Replacing it with deadlocking behaviour is somehow counter-intuitive. Applying such a transformation to a system will in general change the behaviour quite drastically, since everything that depends on the high-level event will be prohibited on the low level. This does not fit in with the applications of transformation that we have in mind, where the specification and implementation are in a sense equivalent.

- We wish our family of *abstraction operators* (to be defined below) to be a left inverse of the refinement operators as defined by Van Glabbeek and Goltz [21]. However, if $\mathcal{T}$ may deadlock then the derived refinement operator is in general not injective, and hence does not have a left inverse.

We now have collected all the ingredients for the *abstraction operator* discussed in the introduction. The basic underlying thought is that *causality and conflict should be preserved and reflected by abstraction.* This implies that we should have an intuition about what it means for refined, "non-atomic" events (or more precisely *sets* of events) to be related by causality or conflict. Let E_1, E_2 be arbitrary sets of events in a given event structure $\mathcal{E}$.

- Intuitively and informally, $E_1 \prec E_2$ if there *exist* events $e_i \in E_i$ for $i = 1, 2$ such that $e_1 \prec_{\mathcal{E}} e_2$. If $E_1 \cup E_2 \subseteq F$ for a given configuration F, then this is equivalent to saying that $E_1 \prec E_2$ if there does *not* exist a configuration $G \subseteq F$ such that $E_2 \subseteq G$ and $G \cap E_1 = \varnothing$.

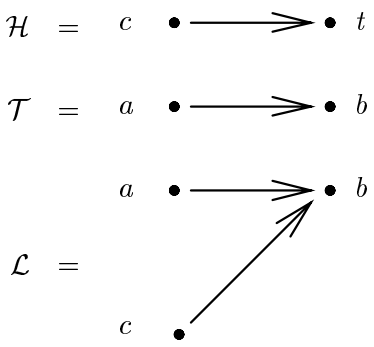
- On the other hand, $E_1 \# E_2$ if $e_1 \#_{\mathcal{E}} e_2$ for *all* events $e_i \in E_i$ for $i = 1, 2$, or in terms of configurations, if there is no configuration F such that $F \cap E_1 \neq \varnothing$ and $F \cap E_2 \neq \varnothing$.

This can be used as a basis for the notion of "reflection and preservation of causality and conflict" in the sense meant above. Note that because we refine only single events at a time, in the refined systems we actually compare *single* events to sets of events, and not *sets* of events to sets of events. Let e, t be two events in a high-level system $\mathcal{H}$, where t is an event to be transformed into $\mathcal{T}$.

- $e \prec_{\mathcal{H}} t$ if and only if $\{e\} \prec F$ in the corresponding low-level behaviour for all $F \in C_{\mathcal{T}}$; symmetrically for $t \prec_{\mathcal{H}} e$.
- $e \#_{\mathcal{H}} t$ if and only if $\{e\} \# F$ in the low level behaviour for all $F \in C_{\mathcal{T}}$.

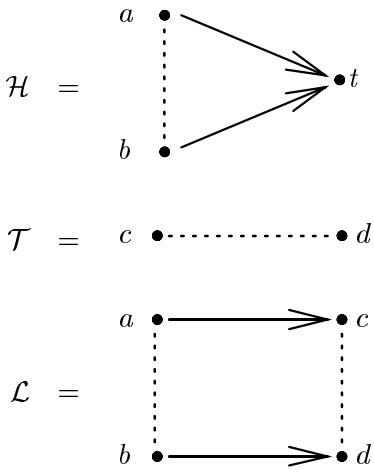
If $\mathcal{T}$ itself contains conflicts, then these conflicts cannot be visible in $\mathcal{H}$, because $\mathcal{T}$ is abstracted into the single event t for which "internal choices" cannot be represented. It follows that in the low-level behaviour, any conflict within $\mathcal{T}$ should be "inessential" in the sense that it reflects a choice actually made outside $\mathcal{T}$.

Some examples may be useful at this point. Consider the following triple of high-level system $\mathcal{H}$, transformation pair $(t, \mathcal{T})$ and low-level system $\mathcal{L}$:



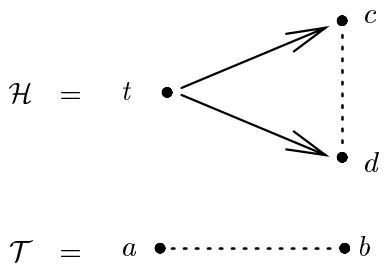
This satisfies our intuition about the transformation of causality: the high-level causality $c \prec_{\mathcal{H}} t$ is preserved by the low-level causality $c \prec_{\mathcal{L}} b$, but it is *not* the case that $c \prec_{\mathcal{L}} e$ for *all* $e \in E_{\mathcal{T}}$.

With respect to transforming conflict, consider the following:



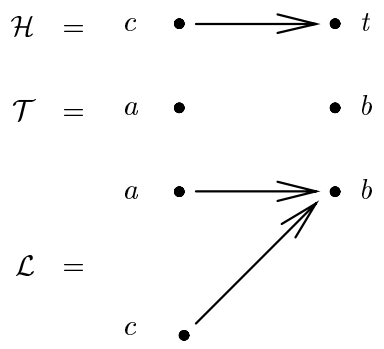
This also corresponds to a valid abstraction. The choice between c and d on the low lever has disappeared on the high level, but this choice is not "essential" in the sense discussed above: the corresponding "essential" choice is the one between a and b. In fact, trying to

abstract away from *that* choice is *not* valid: i.e. if $\mathcal{L}$ is as above then the following is not a valid abstraction.

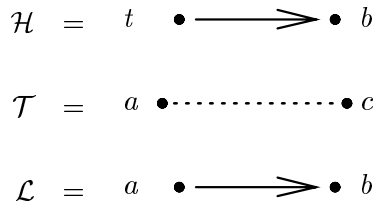


(3.2)

A final noteworthy feature of our notion of abstraction is that not all the behaviours allowed by $\mathcal{T}$ have to be actually present in the low-level behaviour. We give two examples to illustrate this.



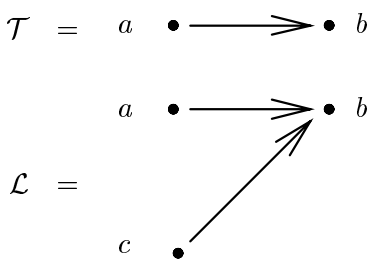
This is very similar to the first example above, except that the transformation itself does not specify that a and b have to be ordered on the low level; the execution of b before the execution of a is allowed by $\mathcal{T}$. In this specific $\mathcal{L}$ however, they *are* ordered and hence b may *not* occur before a has occured; this does not invalidate the abstraction.



In this case, $\mathcal{T}$ has an option c that is never realized in the low-level system. Again, not all the behaviours allowed by $\mathcal{T}$ are allowed in $\mathcal{L}$.

Let us now formally define the abstraction operator. If $F \in C_{\mathcal{L}}$ is a set of events such that $F \cap E_{\mathcal{T}} \neq \varnothing$, which is to say that part of the low-level behaviour of t has occurred in F, then in the abstraction a corresponding configuration $(F \setminus E_{\mathcal{T}}) \cup \{t\}$ will appear. However, we have to be careful about the "moment" when t becomes visible. For instance, if some small initial part of $\mathcal{T}$ has happened but $\mathcal{T}$ is as yet unable to terminate, due to some necessary

causal predecessors that have not yet occurred, then it would be incorrect to say that t has occurred already. For instance, consider once more



Now if we choose to "see" t on the high level when only a has happened on the low level, i.e. already in the configuration $\{a\}$, then $\mathcal{L}$ is abstracted into

which violates our intuition, because now the (low-level) causality $c \prec_{\mathcal{L}} \{a, b\}$ is not present on the high level. The solution is to make t visible only if some "essential part" of $\mathcal{T}$ has happened, where "essential" means that either $\mathcal{T}$ has been a cause of some other event of the low-level behaviour or F contains a terminated part of $\mathcal{T}$. This is captured in the following definition:

3.3 Definition. Let $\mathcal{L}$ be a low-level behaviour and $(t, \mathcal{T})$ a transformation. t is *visible* in a configuration $F \in C_{\mathcal{L}}$, denoted $t \lhd_{\mathcal{L}} F$, if one of the following conditions holds:

1. $(F \setminus E_{\mathcal{T}}) \notin C_{\mathcal{L}}$;
2. $(F \cap E_{\mathcal{T}})\checkmark_{\mathcal{T}}$.

Condition 1 represents the situation that the events in $\mathcal{T}$ are not *maximal* in F; this implies that $\mathcal{T}$ has caused something in F (cf. Definition 2.8). Condition 2 encodes the fact that F contains a terminated configuration of $\mathcal{T}$. Now we can use the relation $F \lhd_{\mathcal{L}} \mathcal{T}$ to define formally the abstraction of a single configuration:

$$F \Uparrow_{\mathcal{L}} t := \begin{cases} (F \setminus E_{\mathcal{T}}) \cup \{t\} & \text{if } t \lhd_{\mathcal{L}} F \\ (F \setminus E_{\mathcal{T}}) & \text{otherwise.} \end{cases} \tag{3.4}$$

Abstraction as an operator over configuration structures is only partially defined: not every low-level configuration structure can be abstracted according to every transformation pair. The following notion of *compatibility* captures the conditions under which abstraction *is* defined.

3.5 Definition. If $\mathcal{L}$ is a configuration structure and $(t, \mathcal{T})$ a transformation pair, then $(t, \mathcal{T})$ is *compatible* with $\mathcal{L}$ if the following conditions hold:

1. $t \notin E_{\mathcal{L}}$;
2. $(F \cap E_{\mathcal{T}}) \in C_{\mathcal{T}}$ for all $F \in C_{\mathcal{L}}$;
3. $(F \cap E_{\mathcal{T}}) = \varnothing \vee (F \cap E_{\mathcal{T}})\checkmark_{\mathcal{T}}$ for all $F\checkmark_{\mathcal{L}}$;
4. $t \lhd (F \cup \{e\}) \in C_{\mathcal{L}} \Longrightarrow \exists X\checkmark_{\mathcal{T}}.\ (F \cup X) \in C_{\mathcal{L}}$ for all $F \in C_{\mathcal{L}}$;
5. $F\Uparrow \subseteq G\Uparrow \Longrightarrow \exists H \in C_{\mathcal{L}}.\ F \subseteq H \wedge H\Uparrow = G\Uparrow$ for all $F, G \in C_{\mathcal{L}}$.

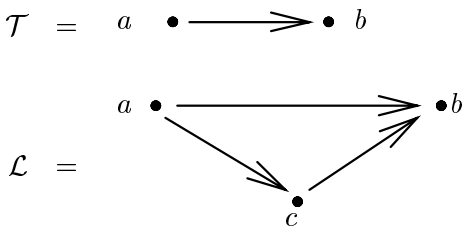
The conditions in this definition have the following intuitions:

1. The high-level event t will represent $\mathcal{T}$ on a higher level of abstraction and hence it cannot be a part of the low-level system $\mathcal{L}$.

2. This says that the $\mathcal{T}$-part of $\mathcal{L}$ should not exhibit any behaviour that is not part of $\mathcal{T}$ itself. For example if two events are in conflict within $\mathcal{T}$, they should not occur in the same computation of $\mathcal{L}$.

3. This says that if the transformed behaviour has started at all in a terminated configuration of the low-level system, then it has also terminated. A terminated configuration represents a state of the system where the computation has been completed succsessfully and it should not contain some started but not completed part of $\mathcal{T}$.

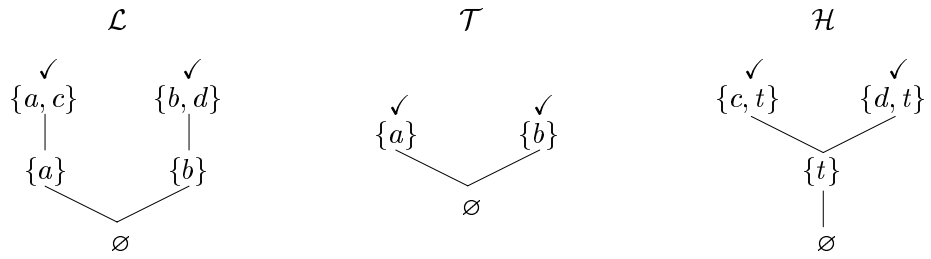
4. This states that whenever there is an event e such that $F \cup \{e\}$ is a configuration which contains an essential part of $\mathcal{T}$ —presumably because $\mathcal{T}$ has caused e— then $\mathcal{T}$ should not depend on anything in the rest of $\mathcal{L}$ in order to terminate directly (doing X). This rules out the situation that $\mathcal{T}$ and e depend on each other:



   This violates the condition for $F = \{a\}$ and $e = c$. Note that $E_{\mathcal{T}} = \{a, b\} \prec c$ and $c \prec E_{\mathcal{T}}$.

5. This states that the abstraction may not destroy "essential choices" in the low-level system: if the abstraction of a configuration F is contained in the abstraction of G, then on the low level F should also be able to proceed to G, or at least to a configuration H that is indistinguishable from G after abstraction. Let us restate example (3.2) in

terms of configuration structures:



Here we have $\{a\}\Uparrow = \{t\} \subseteq \{d,t\} = \{b,d\}\Uparrow$ but there does not exist a configuration $H \in \mathcal{L}$ such that $\{a\} \subseteq H$ and $H\Uparrow = \{d,t\}$ holds. The choice between a and b has disappeared in the abstraction.

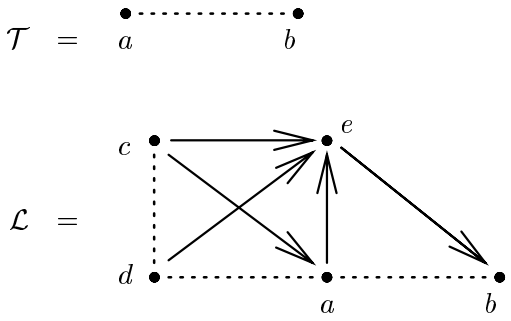
3.6 Definition. Let $\mathcal{L}$ be a configuration structure and $(t,\mathcal{T})$ a transformation compatible with $\mathcal{L}$. The *abstraction* of $\mathcal{L}$ according to t is defined by:

$$\mathcal{L} \Uparrow t := \langle \{\, F \Uparrow_{\mathcal{L}} t \mid F \in C_{\mathcal{L}} \,\}, \{\, F \Uparrow_{\mathcal{L}} t \mid F\checkmark_{\mathcal{L}} \,\} \rangle \ .$$

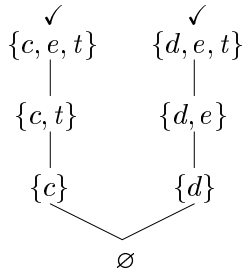
For the consistency of our setup it is important that configuration structures are closed under abstraction. This is formulated in the following proposition. The proof is quite tedious; it is contained in Appendix A.

3.7 Proposition. If $\mathcal{L}$ is a configuration structure and $(t,\mathcal{T})$ a transformation compatible with $\mathcal{L}$, then $\mathcal{L} \Uparrow t$ is a configuration structure.

Note that the proposition is in terms of configuration structures, *not* flow event structures. As mentioned in the previous section, there are in fact configuration structures that cannot be obtained from any flow event structure. Moreover, even if $\mathcal{L}$ and $\mathcal{T}$ are (obtained from) flow event structures, $\mathcal{L} \Uparrow t$ may not be, as the following example shows:

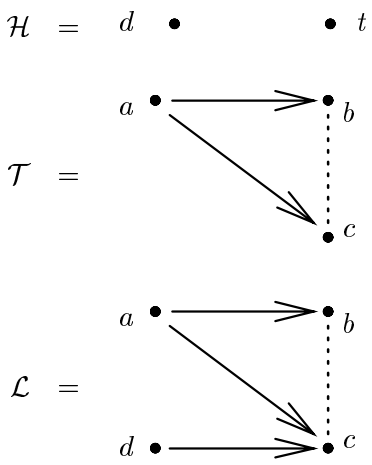


Here $\mathcal{L} \Uparrow t$ is the following configuration structure:



There is no corresponding flow event structure: it would need $e \prec t$ because t is maximal in $\{c, e, t\}$ and $t \prec e$ because e is maximal in $\{d, e, t\}$; but then $\{c, e, t\}$ would contain a flow cycle and thus not be a configuration.

The last example of this section shows that some unexpected things may happen when causality and conflict are combined. The following is a valid abstraction:



Here we see that two independent high-level events are causally related in *some* low-level configurations. In a sense, this is a "non-essential causality:" if for some reason d is postponed, the configuration $\{a, b\}$ can always be executed independently from d, so that t becomes visible.

4 Refinement operators

In this section we relate the abstraction operator defined in Definition 3.6 to the *refinement operator* over configuration structures known from the literature. The following definition is based on Van Glabbeek and Goltz [22]. Some notation: if $F \subseteq E \times E$ is a set of pairs, then

$$\begin{array}{rcl} \pi_i(F) & := & \{\, e \mid \exists (e_1, e_2) \in F.\, e = e_i \,\} \\ F(e) & := & \{\, e' \mid (e, e') \in F \,\} \end{array}$$

4.1 Definition ([22, Definition 3.6]). Let $\mathcal{H}$ be a (high-level) configuration structure and $f\colon E_{\mathcal{H}} \to \mathbf{C}$ a function from events to (transformed) configuration structures. A *pre-refinement pair* is a pair (F, g) such that

1. $F \in C_{\mathcal{H}}$ is a high-level configuration;
2. $g\colon F \to \mathbf{2^E}$ is a function such that $g(e) \in C_{f(e)}$ for all $e \in F$;
3. Let $F^{\checkmark} := \{\, e \in F \mid g(e)\checkmark_{f(e)} \,\}$; then $F^{\checkmark} \subseteq G \subseteq F$ implies $G \in C_{\mathcal{H}}$.

The *refinement* of $\mathcal{H}$ by f is given by $\mathcal{H}[f] = \langle C, \checkmark \rangle$ such that

$$\begin{array}{rcl} E_C & := & \bigcup_{e \in E_{\mathcal{H}}} \big(\{e\} \times E_{f(e)} \big) \\ C & := & \{\, G \subseteq E_C \mid (\pi_1(G), g\colon e \mapsto G(e)) \text{ is a pre-refinement pair} \,\} \\ \checkmark & := & \{\, G \in C \mid \pi_1(G)\checkmark_{\mathcal{H}} \wedge \forall e \in \pi_1(G).\, G(e)\checkmark_{f(e)} \,\} \end{array}$$

Pre-refinement pairs form a convenient intermediate step in the construction of the refined configurations. They consist of a set F of events corresponding to a configuration of the high-level system, plus a function selecting a possible refinement for each event $e \in F$ out of the behaviour given by $f(e)$. The conditions over pre-refinement pairs can be explained as follows.

1. F should be a high-level configuration.
2. Each high-level event $e \in F$ is refined into a configuration $g(e)$ allowed by $f(e)$.
3. Apart from the *terminated part* $F^{\checkmark} \subseteq F$ all events of F are "busy" and hence should be maximal in $\mathcal{H}$ (Definition 2.8).

Refinement constructs a low-level behaviour description out of a given high-level one. Conceptually, if $\mathcal{H}$ and f are given then the refinement $\mathcal{H}[f]$ is an *implementation* of $\mathcal{L}$. We will show that this notion of implementation is consistent with the one engendered by the *abstraction* operator of the previous section, so that $\mathcal{H}[f]$ always corresponds to an implementation of $\mathcal{H}$ in our sense, by showing that in principle the following holds:

$$\mathcal{H}[f] \Uparrow f = \mathcal{H} \ .$$

Unfortunately as it is formulated here this equality fails, because the left hand side is undefined: in our treatment of abstraction we have dealt only with single-event transformations and not full functions. However we can simulate the refinement operator above with a sequence of single-event transformations, at least as long as $|E_{\mathcal{H}}|$ is finite. For these single-event refinements we can then prove that abstraction is defined and yields the original structure.

4.2 Definition. If $\mathcal{H}$ is a configuration structure and $(t, \mathcal{T})$ a transformation pair such that $E_{\mathcal{T}} \cap E_{\mathcal{H}} = \varnothing$, then $\mathcal{H} \Downarrow t := \langle C_1 \cup C_2 \cup C_3, \checkmark_1 \cup \checkmark_2 \rangle$ such that

$$\begin{array}{rcl} C_1 & := & \{\, F \in C_{\mathcal{H}} \mid t \notin F \,\} \\ C_2 & := & \{\, (F \smallsetminus \{t\}) \cup X \mid (t \in F \in C_{\mathcal{H}}) \wedge (F \smallsetminus \{t\}) \in C_{\mathcal{H}} \wedge X \in C_{\mathcal{T}} \,\} \\ C_3 & := & \{\, (F \smallsetminus \{t\}) \cup X \mid (t \in F \in C_{\mathcal{H}}) \wedge X \checkmark_{\mathcal{T}} \,\} \\ \checkmark_1 & := & \{\, F \checkmark_{\mathcal{H}} \mid t \notin F \,\} \\ \checkmark_2 & := & \{\, (F \smallsetminus \{t\}) \cup X \mid (t \in F \checkmark_{\mathcal{H}}) \wedge X \checkmark_{\mathcal{T}} \,\} \ . \end{array}$$

The following proposition now expresses the desired relation between refinement by transformation functions (Definition 4.1) and refinement by transformation pairs (Definition 4.2).

4.3 Proposition. Let $\mathcal{H}$ be a (nonempty) high-level configuration structure and $f\colon E_{\mathcal{H}} \to \mathbf{C}$ a function from events to refined configuration structures. If $E_{\mathcal{H}} = \{t_1, \ldots, t_n\}$ then

$$\mathcal{H}[f] = (\cdots((\mathcal{H} \Downarrow t_1) \Downarrow t_2) \cdots) \Downarrow t_n$$

where for all $1 \le i \le n$, $(t_i, \mathcal{T}_i)$ is a transformation pair such that

$$\mathcal{T}_i = \langle\, \{\, \{t_i\} \times F \mid F \in C_{f(t_i)} \,\}, \{\, \{t_i\} \times F \mid F \checkmark_{f(t_i)} \,\} \,\rangle \ .$$

Proof. Let $\mathcal{H}$ be an arbitrary but fixed high-level configuration structure. We define for $\mathcal{H}$ a sequence of *approximations* $\mathcal{H}_i$ which correspond to refining only $t_1 \cdots t_i$, so that

$$\mathcal{H}_0 = \mathcal{H} \tag{4.4}$$
$$\mathcal{H}_i = \mathcal{H}_{i-1} \Downarrow t_i \quad \text{for } 1 \le i \le n \tag{4.5}$$
$$\mathcal{H}_n = \mathcal{H}[f] \ . \tag{4.6}$$

A straightforward induction on n then proves the proposition.

The construction is as follows. For all $0 \le i \le n$ let $E_i := \{t_1, \ldots, t_i\}$. An *$i$-level pre-refinement triple* is a pair (F_1, F_2, g) such that

1. $F_1 \cup F_2 \in C_{\mathcal{H}}$ such that $F_1 \cap E_i = \varnothing$ and $F_2 \subseteq E_i$;
2. g is a function over F_2 such that $g(e) \in C_{f(e)}$ for all $e \in F_2$;
3. Let $F^{\checkmark} := F_1 \cup \{\, e \in F_2 \mid g(e) \checkmark_{f(e)} \,\}$; then $F^{\checkmark} \subseteq G \subseteq F_1 \cup F_2$ implies $G \in C_{\mathcal{H}}$.

Conceptually, F_1 corresponds to the *non-refined part* and F_2 to the *refined part* of a high-level configuration. (F_1, F_2, g) *generates* the set

$$F := F_1 \cup \{\, \{e\} \times g(e) \mid e \in F_2 \,\} \ .$$

Now let $\mathcal{H}_i := \langle C_i, \checkmark_i \rangle$ such that

$$\begin{array}{rcl} C_i & := & \{\, F \mid F \text{ is generated by an } i\text{-level pre-refinement triple} \,\} \\ \checkmark_i & := & \{\, F \in C_i \mid ((F \cap E_{\mathcal{H}}) \cup \pi_1(F \smallsetminus E_{\mathcal{H}})) \checkmark_{\mathcal{H}} \wedge \forall e \in \pi_1(F \smallsetminus E_{\mathcal{H}}).\ F(e) \checkmark_{f(e)} \,\} \ . \end{array}$$

We prove that these constructions satisfy the conditions above.

(4.4) If $i = 0$ then $E_i = \varnothing$, hence all i-level refinement triples are of the form $(F, \varnothing, \varnothing)$ where $F \in C_{\mathcal{H}}$; hence $\mathcal{H}_i = \mathcal{H}$.

(4.5) The proof consists of two parts.

$\subseteq$ Assume $F \in C_i$; then there is an i-level pre-refinement triple (F_1, F_2, g) generating F. If $t_i \notin F_2$ then (F_1, F_2, g) is also $(i-1)$-level, hence $t_i \notin F \in C_{i-1} = C_{\mathcal{H}_{i-1}}$; then also $F \in C_{H_{i-1}\Downarrow t_i}$. Otherwise there is an $(i-1)$-level pre-refinement triple (F_1', F_2', g') such that

$$\begin{array}{rcl} F_1' & := & F_1 \cup \{t_i\} \\ F_2' & := & F_2 \setminus \{t_i\} \\ g' & := & g \restriction F_2' \end{array}$$

Let $F' \in C_{i-1}$ be the configuration generated by (F_1', F_2', g'); then $t_i \in F'$ and $F = (F' \setminus \{t_i\}) \cup (\{t_i\} \times g(t_i))$. Now either $g(t_i)\checkmark_{f(t_i)}$, implying $(\{t_i\} \times g(t_i))\checkmark_{\mathcal{T}_i}$, or $\neg g(t_i)\checkmark_{f(t_i)}$, implying $t_i \notin F^{\checkmark}$ and hence $(F_1 \cup F_2) \setminus \{t_i\} \in C_i$. In either case $F \in C_{H_{i-1}\Downarrow t_i}$ (case C_3 resp. C_2 of Definition 4.2).

If in addition $F\checkmark_i$ then either $t_i \notin F_2$ or $F'\checkmark_{i-1}$ and $g(t_i)\checkmark_{f(t_i)}$; $F\checkmark_{H_{i-1}\Downarrow t_i}$ follows by cases $\checkmark_1$ resp. $\checkmark_2$ of Definition 4.2.

$\supseteq$ Assume $F \in C_{H_{i-1}\Downarrow t_i}$; by construction (Definition 4.2) there are three cases.

- $F \in C_{i-1}$ and $t_i \notin F$, hence there is an $(i-1)$-level pre-refinement triple (F_1, F_2, g) generating F; because $t_i \notin F_1$, (F_1, F_2, g) is also an i-level pre-refinement triple, hence $F \in C_i$.
- $F = (F' \setminus \{t_i\}) \cup X$ for some F' such that $t_i \in F' \in C_{i-1}$ and $(F' \setminus \{t_i\}) \in C_{i-1}$ and $X \in C_{\mathcal{T}_i} = \{\, \{t_i\} \times H \mid H \in C_{f(t_i)} \,\}$. Let (F_1, F_2, g) be the $(i-1)$-level pre-refinement triple generating F'; then $(F_1 \setminus \{t_i\}, F_2, g)$ is an $(i-1)$-level pre-refinement triple generating $F' \setminus \{t_i\}$. Now define g' over $F_2 \cup \{t_i\}$ such that $g' \restriction F_2 = g$ and $g'(t_i) = X$; then $(F_1 \setminus \{t_i\}, F_2 \cup \{t_i\}, g')$ is an i-level pre-refinement triple generating F; hence $F \in C_i$.
- Similar to the above.

If in addition $F\checkmark_{H_{i-1}\Downarrow t_i}$, then there are two cases similar to the above, derived from the construction in Definition 4.2; in both cases it follows that $F\checkmark_i$.

(4.6) If $i = n$ then $E_i = E_{\mathcal{H}}$, hence the construction of $\mathcal{H}_i$ reduces to the definition of $\mathcal{H}[f]$ in Definition 4.1, which ensures $\mathcal{H}_i = \mathcal{H}[f]$. □

The following theorem now expresses the fact that our abstraction operator is indeed a left inverse of the traditional refinement operator. This concludes the formalization of the statement made in the introduction that the notion of implementation-by-abstraction advocated in this paper is more general than the traditional notion of implementation-by-refinement. In the next section we will give an example which shows that it is in fact *strictly* more general.

4.7 Theorem. If $\mathcal{H}$ is a configuration structure and $(t, \mathcal{T})$ a transformation pair such that $E_{\mathcal{T}} \cap E_{\mathcal{H}} = \varnothing$, then $(\mathcal{H} \Downarrow t) \Uparrow t = \mathcal{H}$.

Proof. Let $\mathcal{L} := \mathcal{H} \Downarrow t$. We first prove that $\mathcal{T}$ is *compatible* with $\mathcal{L}$ (Definition 3.5).

1. Assume $F \in C_{\mathcal{L}}$. If $F \in C_1$ then $F \cap E_{\mathcal{T}} = \varnothing \in C_{\mathcal{T}}$. If $F \in C_2 \cup C_3$ then $(F \cap E_{\mathcal{T}}) = X \in C_{\mathcal{T}}$.

2. Asume $F{\checkmark_{\mathcal{L}}}$. If $F{\checkmark_1}$ then $F \cap E_{\mathcal{T}} = \varnothing$. If $F{\checkmark_2}$ then $F \cap E_{\mathcal{T}} = X{\checkmark_{\mathcal{T}}}$.

3. Assume $t \lhd_{\mathcal{L}} (F \cup \{e\}) \in C_{\mathcal{L}}$. It follows that either $(F \cup \{e\}) \smallsetminus E_{\mathcal{T}} \notin C_{\mathcal{L}}$ or $((F \cup \{e\}) \cap E_{\mathcal{T}}){\checkmark_{\mathcal{T}}}$. In the second case, $X := (F \cup \{e\}) \cap E_{\mathcal{T}}$ satisfies the condition. In the first case, if $e \in E_{\mathcal{T}}$ then $(F \cup \{e\}) \smallsetminus E_{\mathcal{T}} = F \smallsetminus E_{\mathcal{T}} \notin C_{\mathcal{L}}$, hence $F \in C_3$ and $(F \cap E_{\mathcal{T}}){\checkmark_{\mathcal{T}}}$; otherwise $e \notin E_{\mathcal{T}}$, from which it follows that $(F \cup \{e\}) \in C_3$, and hence $((F \cup \{e\}) \cap E_{\mathcal{T}}) = (F \cap E_{\mathcal{T}}){\checkmark_{\mathcal{T}}}$. Either way $X := F \cap E_{\mathcal{T}}$ satisfies the condition.

4. Assume $(F \Uparrow_{\mathcal{L}} t) \subseteq (G \Uparrow_{\mathcal{L}} t)$. If $F \subseteq G$ then $H := G$ satisfies the condition; otherwise $F \cap E_{\mathcal{T}} \neq \varnothing \neq G \cap E_{\mathcal{T}}$ and $(F \smallsetminus E_{\mathcal{T}}) \subseteq (G \smallsetminus E_{\mathcal{T}})$. If $t \notin (G \Uparrow_{\mathcal{L}} t)$ then $G \in C_2$, hence $(G \smallsetminus E_{\mathcal{T}}) \cup X \in C_{\mathcal{L}}$ for all $X \in C_{\mathcal{T}}$, hence $H := (G \smallsetminus E_{\mathcal{T}}) \cup (F \cap E_{\mathcal{T}})$ satisfies the condition. Otherwise $G \in C_3$, hence $(G \smallsetminus E_{\mathcal{T}}) \cup X \in C_{\mathcal{L}}$ for all $X{\checkmark_{\mathcal{T}}}$; let X be such that $(F \cap E_{\mathcal{T}}) \subseteq X{\checkmark_{\mathcal{T}}}$, then $H := (G \smallsetminus E_{\mathcal{T}}) \cup X$ satisfies the condition.

Now let $\mathcal{D} := \mathcal{L} \Uparrow t$.

$$\begin{aligned}
F \cup \{t\} \in C_{\mathcal{D}} &\iff \exists G \in C_{\mathcal{L}}.\, (F \smallsetminus \{t\} = G \smallsetminus E_{\mathcal{T}}) \wedge (G \smallsetminus E_{\mathcal{T}} \notin C_{\mathcal{L}} \vee (G \cap E_{\mathcal{T}}){\checkmark_{\mathcal{T}}}) \\
&\iff \exists G \in C_3.\, F \smallsetminus \{t\} = G \smallsetminus E_{\mathcal{T}} \\
&\iff F \cup \{t\} \in C_{\mathcal{H}} \\
F \smallsetminus \{t\} \in C_{\mathcal{D}} &\iff \exists G \in C_{\mathcal{L}}.\, (F \smallsetminus \{t\} = G \smallsetminus E_{\mathcal{T}}) \wedge (G \smallsetminus E_{\mathcal{T}} \in C_{\mathcal{L}}) \\
&\iff \exists G \in C_{\mathcal{L}}.\, (F \smallsetminus \{t\} = G \smallsetminus E_{\mathcal{T}}) \wedge (G \smallsetminus E_{\mathcal{T}} \in C_1) \\
&\iff F \smallsetminus \{t\} \in C_{\mathcal{H}}
\end{aligned}$$

Because $F = F \cup \{t\}$ or $F = F \smallsetminus \{t\}$ for every set F, it follows that $C_{\mathcal{D}} = C_{\mathcal{H}}$.

$$\begin{aligned}
(F \cup \{t\}){\checkmark_{\mathcal{D}}} &\iff \exists G{\checkmark_{\mathcal{L}}}.\, (F \smallsetminus \{t\} = G \smallsetminus E_{\mathcal{T}}) \wedge (G \cap E_{\mathcal{T}}){\checkmark_{\mathcal{T}}} \\
&\iff \exists G{\checkmark_2}.\, F \smallsetminus \{t\} = G \smallsetminus E_{\mathcal{T}} \\
&\iff (F \cup \{t\}){\checkmark_{\mathcal{H}}} \\
(F \smallsetminus \{t\}){\checkmark_{\mathcal{D}}} &\iff \exists G{\checkmark_{\mathcal{L}}}.\, (F \smallsetminus \{t\} = G \smallsetminus E_{\mathcal{T}}) \wedge (G \cap E_{\mathcal{T}} = \varnothing) \\
&\iff \exists G{\checkmark_{\mathcal{L}}}.\, (F \smallsetminus \{t\} = G) \wedge G{\checkmark_1} \\
&\iff (F \smallsetminus \{t\}){\checkmark_{\mathcal{H}}}
\end{aligned}$$

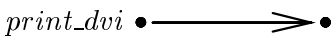
Because $F = F \cup \{t\}$ or $F = F \smallsetminus \{t\}$ for every set F, it follows that ${\checkmark_{\mathcal{D}}} = {\checkmark_{\mathcal{H}}}$. □

5 Examples

We will illustrate our approach to refinement in detail by two examples and explain the difference between our approach and the usual approach to refinement as proposed e.g. in [21] and defined in Definition 4.1 and Definition 4.2. The first example presented here is an elaborated version of the example given in the introduction. It gives a taste of how our definitions work and why the more general notion of refinement we are proposing is useful in practice. The second example tries to illustrate the increased power of the design method on an example that is closer connected to a real design problem, namely the implementation of a semaphore described by Lamport [12, 13, 14].

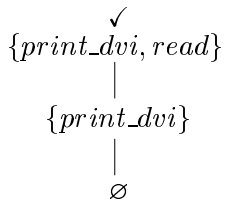
5.1 Printing a file

This example is inspired by the example of the design of a sender presented in [22]. Assume that one wants to read a certain text encoded in DVI-format. In order to do this one first has to print it with an event *print_dvi*, and then read it with an event *read*. It is clear that *print_dvi* must causally precede *read*; hence this behaviour is represented by the following flow event structure:



(5.1)

According to the definitions of Section 2 the behaviour of (5.1) is equivalently represented by the following configuration structure:

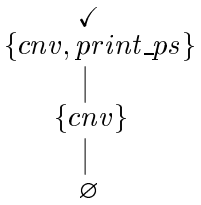


(5.2)

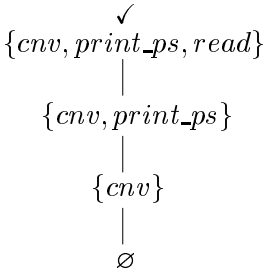
On a lower level of abstraction one may discover that printing a DVI-file is implemented as two consecutive steps where first the DVI-format is converted into postscript format in an event *cnv* and then the converted file is printed in an event *print_ps*; hence *cnv* is a causal predecessor of *print_ps*. The relationship between two different levels of abstraction is given by the transformation pair:

$$t_1 = (print\_dvi, (cnv \prec print\_ps)) \ .$$

Here $(cnv \prec print\_ps)$ is used as a short notation for the following configuration structure:



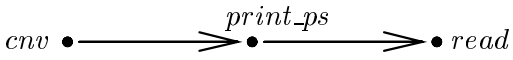
The transformation pair t_1 defines by Definition 3.6 an abstraction operator $\Uparrow t_1$ which maps the low-level system



(5.3)

to the high-level system (5.2); hence (5.3) is a valid refinement of (5.2).

Although the refinement and abstraction operators are formally defined on configuration structures we will — for the sake of a compact presentation — in the following only show flow event structures. Every flow event structure determines (by Definition 2.4) a configuration structure. For example (5.3) is represented by a flow event structure in a more compact way as follows:[3]

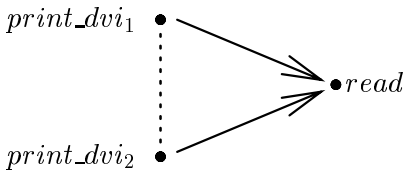


(5.4)

Let us consider an alternative design step, where printing a DVI-file is implemented by a system consisting of *two* smart printers, each of which can do the printing of a DVI-file. Hence *print_dvi* is transformed into the choice $(print\_dvi_1 \ \# \ print\_dvi_2)$. Thus we have another transformation pair

$$t_2 = (print\_dvi, (print\_dvi_1 \ \# \ print\_dvi_2)) \ .$$

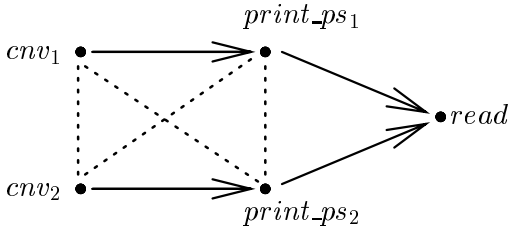
Then according to Definition 3.6 the system



(5.5)

is mapped to (5.1) by the abstraction operator $\Uparrow t_2$; hence (5.5) is a valid refinement of (5.1).

These two design steps encoded in t_1 and t_2 are to a certain degree *orthogonal*: they do not affect each other and it is possible to combine them in any order and refine the printing of a DVI-file into the following system:



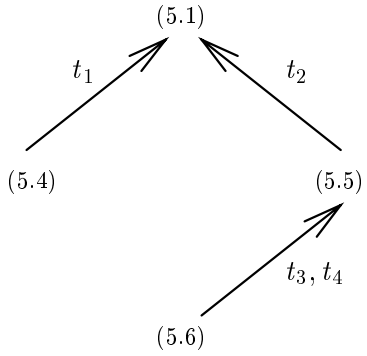
(5.6)

[3] As a consequence we will talk of (5.1) as an abstraction of (5.4) or of (5.4) as a valid refinement of (5.1) instead of referring to their configuration structures (5.2) and (5.3) directly.

In order to abstract (5.6) formally to (5.5) we make use of the following transformation pairs:

$$\begin{aligned} t_3 &= (print\_dvi_1, (cnv_1 \prec print\_ps_1)) \\ t_4 &= (print\_dvi_2, (cnv_2 \prec print\_ps_2)). \end{aligned}$$

Applying the corresponding abstraction operators $\Uparrow t_3$ and $\Uparrow t_4$ in either order will map (5.6) to (5.5), validating the design step. Hence system (5.6) is a refinement of (5.1) incorporating both design steps that were leading to the intermediate design of the systems (5.4) and (5.5) respectively.[4] We can depict the present state of our design process by the following *design diagram*:

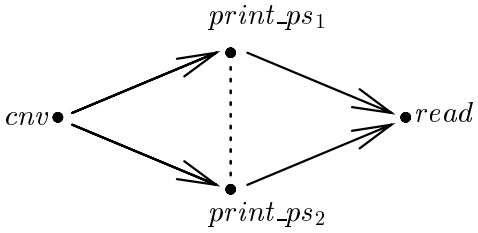


(5.7)

Now it is natural to expect that the diagram (5.7) can be completed and that (5.6) is a refinement of (5.4) as well. Consider therefore the transformation pairs:[5]

$$\begin{aligned} t_5 &= (cnv, (cnv_1 \# cnv_2)) \\ t_6 &= (print\_ps, (print\_ps_1 \# print\_ps_2)). \end{aligned}$$

The corresponding abstraction operators $\Uparrow t_5$ and $\Uparrow t_6$ can be applied to the system (5.6). But here one has to be careful in which order the abstraction operators $\Uparrow t_5$ and $\Uparrow t_6$ are applied. Using first $\Uparrow t_5$ would violate condition 4 of Definition 3.6 because in the system



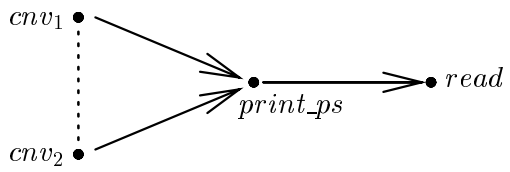
(5.8)

the configuration $\{cnv_2\}\Uparrow = \{cnv\}$ is a subset of the configuration $\{cnv_1, print\_ps_1\}\Uparrow = \{cnv, print\_ps_1\}$, but there exists no superset H of $\{cnv_2\}$ such that $H\Uparrow = \{cnv, print\_ps_1\}$ because after the execution of cnv_2 the occurence of $print\_ps_1$ becomes impossible. The choice between cnv_1 and cnv_2 is essential and hence we may not abstract from it.

[4] It is obvious that t_1 and t_3, t_4 encode the same conceptual design step, namely that printing a DVI-file consists of the two consecutive steps *cnv* and *print_ps*.

[5] Note that again t_2 and t_5, t_6 encode the same conceptual design step.

However, $\Uparrow t_6$ is defined on system (5.6) and its application yields the following system:



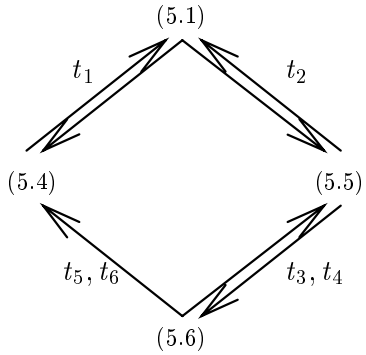
(5.9)

By applying $\Uparrow t_5$ then one gets back to the system (5.4). Hence the refinement of the system (5.4) into the system (5.6) is a valid design step in our framework.

So far we have only considered our abstraction operator, and shown some intuitively justified design steps that were formally verifiable. Now consider refinement instead. It is easy to check that the following hold:

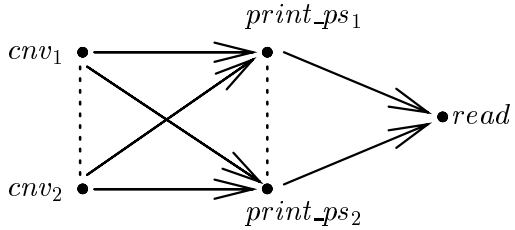
$$\begin{aligned}(5.4) &= (5.1) \Downarrow t_1\\ (5.5) &= (5.1) \Downarrow t_2\\ (5.6) &= ((5.5) \Downarrow t_3) \Downarrow t_4 = (((5.1) \Downarrow t_2) \Downarrow t_3) \Downarrow t_4\end{aligned}$$

Using upwards arrows for proofs by our approach and downwards arrows for proofs by the traditional approach we can represent the relationsships between the different levels of abstraction established so far in our example by the following design diagram:



(5.10)

However, the refinement operators $\Downarrow t_5$ and $\Downarrow t_6$ do *not* construct the system (5.6) as a traditional refinement of (5.4) but — applied in either order — instead the system

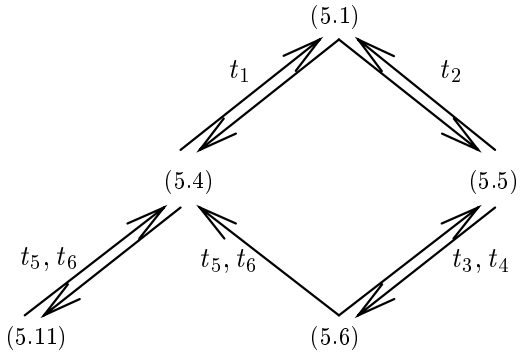


(5.11)

because the causality between cnv and $print\_ps$ is inherited by the subsystems ($cnv_1 \# cnv_2$) and ($print\_ps_1 \# print\_ps_2$).

System (5.11) is a sensible implementation of the specification (5.1), but it is important to note that it describes a quite different behaviour than (5.6): In (5.6) there is only the choice between using printer 1 or 2 for both converting and printing the file, but in (5.11) the file may be converted by one printer and printed by the other. Now this may in some cases be just what one would like, and in fact our abstraction operator maps (5.11) to the high-level system (5.4) also.[6]

Extending the diagram (5.10) by the relationship between (5.4) and (5.11) we obtain the following diagram:

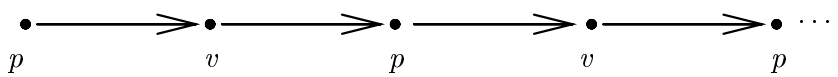


The picture makes visible a notable difference between the two frameworks: (5.6) is treated "symmetrically" by our approach but not by the traditional approach. If a designer using the traditional notion of implementation starts with the decision to apply t_1 to (5.1) (transforming *print_dvi* into $(cnv \prec print\_ps)$, which would seem neutral as far as further decisions are concerned) he will arrive at (5.4). But now in this design methodology he has lost the option to use (5.6) as his next implementation: it cannot be obtained as a refinement of (5.4). The design by abstraction offers greater flexibility in this respect.

5.2 Implementation of a Semaphore

A *semaphore* is a synchronization primitive which provides two operations called p and v. These operations are used to implement the *mutually exclusive* execution of so called *critical sections* of concurrent processes. A typical example of such a critical section is the usage of a resource that cannot be accessed by two processes simultaneously and hence may only be accessed by one process at a time.

The solution of the mutual exclusion problem with the p and v operations of the semaphore is accomplished in the following manner. The semaphore ensures that all the p and v operations can only be executed in an alternating manner, starting with a p operation. Hence the behaviour of the semaphore can be modelled as follows:[7]



(5.12)

[6] Because of Theorem 4.7 we know that $(5.4) = ((5.11) \Uparrow t_5) \Uparrow t_6 = ((5.11) \Uparrow t_6) \Uparrow t_5$ holds.

[7] This is also known as a *binary* semaphore, contrasting it to other types of semaphore which allow larger numbers of consecutive p's.

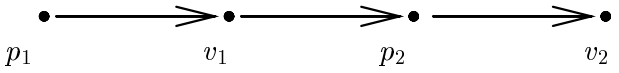
Now if every process executes a p operation before entering the critical section and a v operation afterwards then it is not possible that two processes are excuting their critical sections at the same time: if process P_1 executes the first p and enters its critical section then a second process P_2 has to wait for his execution of the p operation until P_1 leaves the critical section and executes the subsequent v.

A question tackled typically in the theory of operating systems is how a semaphore might be implemented by means of a more basic mechanism. One proposed implementation is the *bakery algorithm* [12, 13, 14]. This algorithm is inspired by the orderly manner in which the customers of American bakeries are apparently served. Each customer on entering the shop draws a *ticket* with a number on it that increases with each successively drawn ticket. There is a display behind the counter which holds the number of the customer being currently served. Being served is the critical section; one may enter it as soon as the display is increased to the number of one's own ticket; when one has payed, one leaves the shop and the critical section, and the display is increased to allow the next customer in line to be served.

In the bakery algorithm a slight variation on this principle is used: it is possible for two customers to have the same ticket value, in which case (but only in this case) the decision whom to serve is based on a further *inherent ordering* of the customers; say that in case of doubt, the mayor is always served first, then the sheriff, etc. Also, there is no explicit display; instead the customers are asked to decide for themselves who has the lowest number. In this paper however we will model the algorithm on a slightly more abstract level, without referring to values explicitly: we will understand the algorithm as a *set-and-test scheme* with the following three operations:

- *set*, corresponding to drawing a ticket and thereby establishing one's position in the line of customers;
- *test*, corresponding to testing whether one holds the ticket with the lowest number. This operation finishes only when the test is successful, that is when one has discovered that one indeed holds the lowest number;
- *reset*, corresponding to leaving the shop, thereby withdrawing one's ticket from the competition.

We can use this scheme to implement a semaphore by transforming the p operation into *set* followed by *test*, and v into *reset*. To keep the example simple we restrict ourself to two executions of the p and v operations; to distinguish them as events we simply assign a number to them. Hence the intended behaviour of the semaphore is represented as follows:

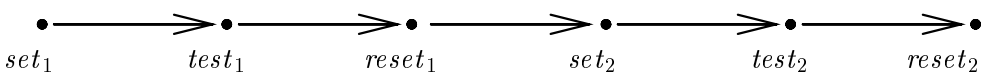


(5.13)

The proposed implementation gives rise to the following transformation pairs ($i = 1, 2$):

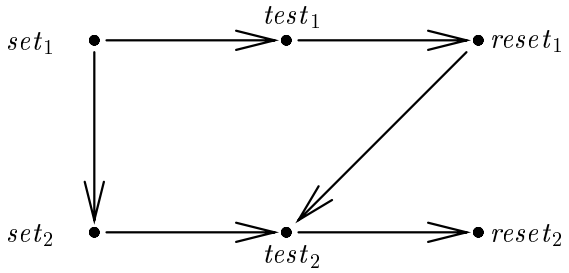
$$(p_i, (set_i \prec test_i))$$
$$(v_i, (reset_i))$$

We can immediately apply traditional refinement to (5.13) using these transformation pairs, resulting in the following behaviour:



(5.14)

This is however not the implementation we have in mind: it suggests that to draw a ticket (set_2) the second customer has to wait until the first customer has left the shop ($reset_1$). Instead, we would like to see the following behaviour:



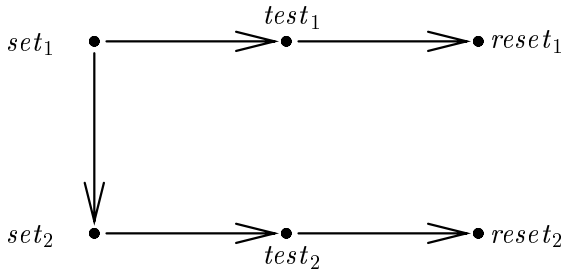
(5.15)

Here the second customer can draw his ticket (set_2) immediately after the first one has done so (set_1) —or to be precise it is the other way around: the decision who is first and who second is established precisely by the order in which the tickets are drawn (plus the inherent ordering mentioned above; in fact this inherent ordering may be seen as a "last resort" decision criterion to use if two customers draw concurrently and somehow get the same ticket). Subsequently each customer starts testing if he has the lowest number; only $test_1$ will succeed immediately, allowing the first customer to be served. The second customer will continue testing until the first customer leaves the shop ($reset_1$).

The additional flexibility of our implementation by abstraction, compared to the traditional implementation by refinement, now allows us to state that the (5.15) is a correct implementation of (5.13):

$$((((5.15) \Uparrow p_1) \Uparrow p_2) \Uparrow v_1) \Uparrow v_2 = (5.13) \ .$$

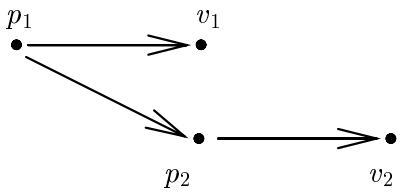
One might perhaps start to wonder if our approach is not in fact *too* flexible. We offer the following observation as an (inconclusive) argument against this fear. Consider what happens if the second customer doesn't test his ticket number (or doesn't respect the inherent ordering, being from out of town and not knowing the other is the sheriff), with the effect that he does not wait for the first customer to leave the shop. This may be represented by the following behaviour:



(5.16)

Now the algorithm breaks down, and *this is reflected in our approach*: abstracting this be-

haviour using the above transformation pairs yields



(5.17)

Hence both the out-of-town customer and the sheriff can perform p operations without waiting for the other to perform v, and they will get into a fight about whose turn it is. The out-of-town customer will spend a night in jail, and he won't make the same mistake again. (This is called the *stabilizing property* of the bakery algorithm.)

6 Conclusions

On the basis of a general philosophy of system design we have argued that it would be an interesting idea to define an *abstraction* operator indexed by a *transformation mapping* (also called *refinement function* between abstract actions and concrete behaviour. This shoud be contrasted to the more traditional *refinement operator* indexed by the same kind of mapping. The setting in which we have worked is characterized by the following conditions:

- our models are *event-based* with implicit notions of *causality* and *conflict*;
- action labels have not been considered;
- equivalences have not been considered;
- our transformation mappings are actually *pairs*, i.e. we only transform one abstract event at a time.

In this setting we have been able to do the following:

- define an abstraction operator satisfying our intuition about the relation between low- and high-level behaviour;
- prove that this operator is left inverse to the traditional refinement operator;
- show that the notion of implementation engendered by the abstraction operator allows one to take interesting design steps which are impossible using standard refinement techniques.

We have found that even in our restrictive setting, the notion of abstraction can have very subtle consequences which take a lot of examples to work out. For this reason we have included many of our small examples in the paper. We hope to extend the results to a more general setting where instead of applying one transformation pair at a time, the application of a *set* of transformation pairs or equivalently a transformation *mapping* is possible.

The generalized abstraction operator would then allow to transform infinitely many events. Hence the generalized abstraction operator would be a left inverse to the traditional refinement operator in the general case where infinitely many events are refined. Moreover using the definition of the abstraction operator presented here the *order* in which the transformation pairs are applied is important. Not every order of application is possible because it might be the case that a transformation is not *valid* right in the beginning but only after some intermediate transformations[8]. An example is the abstraction of (5.6) on Page 24. A generalized abstraction operator would overcome this incommodity.

Configuration structures did allow us to find a sound definition for the abstraction operator relatively easily, but they do not reflect concepts like causality or conflict in an intuitive way as for example flow event structures, families of posets [16] or prime event structures do. Defining abstraction operators for different models and comparing the definitions will give us a deeper insight into the concepts behind the implementation of distributed behaviour.

[8] If two different orders are applicable the transformations will always produce the same structures.

A Proofs

First an auxiliary result that gives an alternative characterization of configuration structures:

1.1 Proposition. If $C \subseteq \mathit{Fin}(\mathbf{E})$ satisfies condition 1 and 2 of Definition 2.6, then it satisfies condition 3 of Definition 2.6 if and only if

$$\forall F \in C \setminus \{\varnothing\}.\ \exists G \in C.\ G \subset F \wedge |F \setminus G| = 1 \ . \tag{1.2}$$

Proof.

$\Rightarrow$ Suppose $C \subseteq \mathit{Fin}(\mathbf{E})$ satisfies condition 1, 2 and 3 of Definition 2.6. If C contains only the empty set, (1.2) is trivially fulfilled. Otherwise let $F \in C \setminus \{\varnothing\}$ be abitrary. Define $n := \min\{\, |F \setminus G| \mid G \subset F, G \in C \,\}$. Because $\varnothing \subset F$ and $\varnothing \in C$, n is defined and $n \geq 1$.

If $n = 1$, the proof is done. If $n > 1$ then let $H \in C$ be such that $H \subset F$ and $|F \setminus H| = n$, and let $d, e \in F \setminus H$ with $d \neq e$. From condition 3 it follows that $\exists H' \in C. H' \subseteq F \wedge (d \in H' \iff e \notin H')$. Hence $(H \cup H') \subseteq F$ and using condition 2 we conclude that $(H \cup H') \in C$. But now $H \subset (H \cup H') \subset F$ and $|F \setminus (H \cup H')| < |F \setminus H| = n$ because $d \in (H \cup H')$ or $e \in (H \cup H')$ but $d, e \notin H$. This contradicts the construction of n. It follows that $n \not> 1$.

$\Leftarrow$ Suppose $C \subseteq \mathit{Fin}(\mathbf{E})$ satisfies condition 1 and 2 of Definition 2.6 and (1.2), and let $F \in C$. If $|F| = n$ then we can inductively construct a sequence $\varnothing = F_0, \ldots, F_n = F$ such that $F_i \subset F_{i+1}$ and $|F_{i+1} \setminus F_i| = 1$ for all $0 \leq i < n$. It follows that $F = \{e_i\}_{1 \leq i \leq n}$ where e_i is uniquely determined by $e_i \in F_i \setminus F_{i-1}$ for all $1 \leq i \leq n$. Hence if $d, e \in F$ are such that $d \neq e$, then $d = e_i$ and $e = e_j$ for some i, j such that $i \neq j$. Now $G := F_{\min\{i,j\}}$ fulfills the requirements of condition 3 of Definition 2.6. □

3.7 Proposition. If $\mathcal{L}$ is a configuration structure and $(t, \mathcal{T})$ a transformation compatible with $\mathcal{L}$, then $\mathcal{L} \Uparrow t$ is a configuration structure.

Proof. Let $\mathcal{H} := \mathcal{L} \Uparrow t$. By abuse of notation we will use $F \in \mathcal{L}$ as a shorthand for $F \in C_{\mathcal{L}}$ and also $F \in \mathcal{H}$ instead of $F \in C_{\mathcal{H}}$. We will prove that the conditions of Definition 2.6 hold for $\mathcal{H}$. In the proof we will use the following facts:

$$\forall X \in \mathcal{H} \ : \ X \cap E_{\mathcal{T}} = \varnothing \tag{1.3}$$

$$\forall X \in \mathcal{H} \ : \ t \notin X \Longrightarrow X \in \mathcal{L} \tag{1.4}$$

$$\forall F \in \mathcal{L} \ : \ (F \cap E_{\mathcal{T}}) = \varnothing \Longrightarrow F = F\Uparrow \in \mathcal{H} \tag{1.5}$$

$$\forall X \in \mathcal{H} \ : \ t \in X \Longrightarrow \exists X' \in \mathcal{L}.\ (X'\Uparrow = X\Uparrow \wedge (X' \cap E_{\mathcal{T}})\checkmark_{\mathcal{T}}) \tag{1.6}$$

(1.3), (1.4) and (1.5) are direct consequences of (3.4), and (1.6) follows immediately from condition 3 of Definition 3.5.

1. $\varnothing \in \mathcal{L}$; hence $\varnothing\Uparrow = \varnothing \in \mathcal{H}$.

2. We prove that condition 2 of Definition 2.6 holds. Let $F\Uparrow, G\Uparrow, H\Uparrow \in \mathcal{H}$ be arbitrary such that $F\Uparrow \cup G\Uparrow \subseteq H\Uparrow$. We prove $F\Uparrow \cup G\Uparrow \in \mathcal{H}$ by studying the three cases where t is contained in none, one or both of $F\Uparrow$ and $G\Uparrow$. For the case t is contained in one of $F\Uparrow$ and $G\Uparrow$, we will assume without loss of generality that $t \notin F\Uparrow \wedge t \in G\Uparrow$. Whenever $t \notin F\Uparrow$ we can assume $F\Uparrow = F$ and $F \cap E_{\mathcal{T}} = \varnothing$ without loss of generality because of (1.3), (1.4) and (1.5). Moreover (1.6) implies that we can always assume $(F \cap E_{\mathcal{T}})\checkmark_{\mathcal{T}}$ without loss of generality, if $t \in F\Uparrow$ holds.

   $t \notin F\Uparrow \cup G\Uparrow$: In this case we know by (1.4) that $F\Uparrow, G\Uparrow \in \mathcal{L}$ and because of $t \notin F\Uparrow \cup G\Uparrow \subseteq H\Uparrow$ we know also $F\Uparrow \cup G\Uparrow \subseteq H \setminus E_{\mathcal{T}} \subseteq H \in \mathcal{L}$. But then by condition 2 of Definition 2.6 we know $F\Uparrow \cup G\Uparrow \in \mathcal{L}$ and hence by (1.3) and (1.5) it is obvious that $F\Uparrow \cup G\Uparrow \in \mathcal{H}$.

   $t \notin F\Uparrow, t \in G\Uparrow$: Let us first prove the fact

$$F\Uparrow \cup G\Uparrow \subseteq H\Uparrow \wedge F \cap E_{\mathcal{T}} = \varnothing \Longrightarrow F \cup G \in \mathcal{L} \tag{1.7}$$

   From $G\Uparrow \subseteq H\Uparrow$ it follows by condition 4 of Definition 3.5 that theres is some $G' \in \mathcal{L}$ such that $G \subseteq G'$ and $G'\Uparrow = H\Uparrow$ holds. And because of (1.5) we have $F = F\Uparrow$ and hence $F \subseteq H\Uparrow = G'\Uparrow \subseteq G' \cup \{t\}$ which implies $F \subseteq G'$ because $t \notin F$. Then by condition 2 of Definition 2.6 we know $F \cup G \in \mathcal{L}$ and (1.7) is proved.

   Because $t \notin F\Uparrow$ we can assume $F\Uparrow = F$ and $F \cap E_{\mathcal{T}} = \varnothing$ without loss of generality. Then $F \cup G \in \mathcal{L}$ because (1.7) applies. We assume further $(G \cap E_{\mathcal{T}})\checkmark_{\mathcal{T}}$ without loss of generality because $t \in G\Uparrow$. Then $(G \cap E_{\mathcal{T}})\checkmark_{\mathcal{T}}$ implies $((F \cup G) \cap E_{\mathcal{T}})\checkmark_{\mathcal{T}}$ and then by (3.4) we have $t \in (F \cup G)\Uparrow$. But then $F\Uparrow \cup G\Uparrow = (F \cup G)\Uparrow \in \mathcal{H}$.

   $t \in F\Uparrow \cap G\Uparrow$: If $F' := F \setminus E_{\mathcal{T}} \in \mathcal{L}$ then by (3.4) we know $t \notin F'\Uparrow = F' = F \setminus E_{\mathcal{T}}$ and hence also $F'\Uparrow \subseteq F\Uparrow \subseteq H\Uparrow$. But for the case $t \notin F'\Uparrow$ and $t \in G\Uparrow$ we just have proved that from $F'\Uparrow, G\Uparrow \subseteq H\Uparrow$ follows $F'\Uparrow \cup G\Uparrow \in \mathcal{H}$. Now $F\Uparrow = F'\Uparrow \cup \{t\}$ which implies $F\Uparrow \cup G\Uparrow = F'\Uparrow \cup G\Uparrow \in \mathcal{H}$.

   Let be $F \setminus E_{\mathcal{T}} \notin \mathcal{L}$ and symmetrically $G \setminus E_{\mathcal{T}} \notin \mathcal{L}$. We have to construct a set $K \in \mathcal{L}$ such that $F\Uparrow \cup G\Uparrow = K\Uparrow \in \mathcal{H}$. To construct K we will need to construct auxilliary sets F_{1-4} and K_{1-3}:

$$\begin{aligned} F_1 &:= \bigcup \{\, X \in \mathcal{L} \mid X \subseteq F \wedge (X \cap E_{\mathcal{T}}) = \varnothing \,\} \\ F_2 &:= \bigcup \{\, X \in \mathcal{L} \mid X \subseteq F \wedge X \setminus E_{\mathcal{T}} \in \mathcal{L} \,\} \end{aligned}$$

   Note that since configurations are finite sets, both of these unions are finite. By a generalization of condition 2 of Definition 2.6 to arbitrary finite unions it follows that $F_1, F_2 \in \mathcal{L}$. It should be clear that $F_1 \subseteq F_2 \subseteq F$ and $F_1 = F_2 \setminus E_{\mathcal{T}}$; moreover $F_2 \neq F$ because $(F \setminus E_{\mathcal{T}}) \notin \mathcal{L}$ but $(F_2 \setminus E_{\mathcal{T}}) \in \mathcal{L}$. Let $e \in F \setminus F_2$ be such that $(F_2 \cup \{e\}) \in \mathcal{L}$;[9] then $(F \cup \{e\}) \setminus E_{\mathcal{T}} \notin \mathcal{L}$ by construction of F_2. Hence by condition 3 of Definition 3.5 there is some $X\checkmark_{\mathcal{T}}$ such that $(F_2 \cup X) \in \mathcal{L}$. Let $F_3 := F_2 \cup X$. It follows that $F_3 = F_1 \cup X$ because X is maximal in $C_{\mathcal{T}}$ and $F_3 \setminus E_{\mathcal{T}} = F_2 \setminus E_{\mathcal{T}} = F_1$; hence $F_3\Uparrow = F_1\Uparrow \cup \{t\} \subseteq F\Uparrow$. Now by condition 4 of Definition 3.5 there is some $F_4 \in \mathcal{L}$ such that $F_3 \subseteq F_4$ and $F_4\Uparrow = F\Uparrow$.

[9] The existence of e is a consequence of condition 3 of Definition 2.6

$F_1\Uparrow \subseteq H\Uparrow$ and $F_1 \cap E_\mathcal{T} = \varnothing$ implies $K_1 := F_1 \cup G \in \mathcal{L}$ because of (1.7). We can assume $(G \cap E_\mathcal{T})\checkmark_\mathcal{T}$ wich implies $((F_1 \cup G) \cap E_\mathcal{T})\checkmark_\mathcal{T}$ an therefore $t \in K_1\Uparrow$. Then we know $K_1\Uparrow = F_1\Uparrow \cup G\Uparrow$; hence from $F_3\Uparrow = F_1\Uparrow \cup \{t\}$ and $t \in G\Uparrow$ follows $F_3\Uparrow \subseteq K_1\Uparrow$. Then due to condition 4 of Definition 3.5 there exists a configuration $K_2 \in \mathcal{L}$ such that $F_3 \subseteq K_2$ and $K_2\Uparrow = K_1\Uparrow$. For the same reason $F_4\Uparrow = F\Uparrow \subseteq H\Uparrow$ implies the existance of $K_3 \in \mathcal{L}$ such that $F_4 \subseteq K_3$ and $K_3\Uparrow = H\Uparrow$.

$F_3 \subseteq K_2$ and $F_3 \subseteq F_4 \subseteq K_3$ hold by construction and because $(F_3 \cap E_\mathcal{T})$ is maximal in $E_\mathcal{T}$ it follows that $(K_2 \cap E_\mathcal{T}) = (F_3 \cap E_\mathcal{T}) = (K_3 \cap E_\mathcal{T})$; in addition we have $K_2 \setminus E_\mathcal{T} \subseteq K_3 \setminus E_\mathcal{T}$ because $K_2\Uparrow = K_1\Uparrow \subseteq H\Uparrow = K_3\Uparrow$ is implied by $F_1\Uparrow \subseteq F\Uparrow$; hence $K_2 \subseteq K_3$ and by construction we had $F_4 \subseteq K_3$ which implies by condition 2 of Definition 2.6 that $K := (F_4 \cup K_2) \in \mathcal{L}$. Now $(K_2 \cap E_\mathcal{T})\checkmark_\mathcal{T}$ implies $(K \cap E_\mathcal{T})\checkmark_\mathcal{T}$ hence $t \in K\Uparrow$ and therefore $K\Uparrow = F_4\Uparrow \cup K_2\Uparrow$ because $t \in K_2\Uparrow$. We have that $F_4\Uparrow = F\Uparrow$ and $K_2\Uparrow = K_1\Uparrow = F_1\Uparrow \cup G\Uparrow$ and because $F_1\Uparrow \subseteq F\Uparrow$ it follows $F\Uparrow \cup G\Uparrow = K\Uparrow \in \mathcal{H}$.

3. Using Proposition 1.1 we only have to show that (1.2) holds. Let $F \in \mathcal{H} \setminus \{\varnothing\}$ be arbitrary. We prove $\exists G \in \mathcal{H}.\, G \subset F \wedge |F \setminus G| = 1$ by studying the two cases $t \in F$ and $t \notin F$:

   $t \notin F$: (1.4) implies $F \in \mathcal{L}$ and then (1.2) implies $\exists G \in \mathcal{L}.\, G \subset F \wedge |F \setminus G| = 1$. From (1.3) follows $F \cap E_\mathcal{T} = \varnothing$ which implies $G \cap E_\mathcal{T} = \varnothing$, hence (1.5) implies that $G = G\Uparrow \in \mathcal{H}$ satisfies (1.2).

   $t \in F$: Define $F\Downarrow := \{\, H \in \mathcal{L} \mid ((H \setminus E_\mathcal{T}) \cup \{t\} = F) \wedge (H \setminus E_\mathcal{T} \notin \mathcal{L}) \,\}$.

   There must exist some configuration $H \in \mathcal{L}$ such that $H\Uparrow = (H \setminus E_\mathcal{T}) \cup \{t\} = F$ and H cannot be empty because $t \in H\Uparrow$. $F\Downarrow = \varnothing$ implies that $H \setminus E_\mathcal{T} \in \mathcal{L}$ and then (3.4) implies $(H \cap E_\mathcal{T})\checkmark_\mathcal{T}$. Because $H \neq \varnothing$ we know by (1.2) that there is a configuration $G \in \mathcal{L}$ and an event $e \notin G$ such that $G \cup \{e\} = H$. If $e \in E_\mathcal{T}$ then $G\Uparrow = F \setminus \{t\} \in \mathcal{H}$ because $G \setminus E_\mathcal{T} = H \setminus E_\mathcal{T}$ and from $(G \cap E_\mathcal{T}) \subset (H \cap E_\mathcal{T})$ follows by condition 4 of Definition 2.6 that $(G \cap E_\mathcal{T})\checkmark_\mathcal{T}$ does not hold; hence $t \notin G\Uparrow$. If $e \notin E_\mathcal{T}$ then $G\Uparrow = F \setminus \{e\} \in \mathcal{H}$ because $G \setminus E_\mathcal{T} = (F \setminus \{e\}) \setminus E_\mathcal{T}$ and $(G \cap E_\mathcal{T}) = (H \cap E_\mathcal{T})$ implies $(G \cap E_\mathcal{T})\checkmark_\mathcal{T}$; hence $t \in G\Uparrow$ and $e \notin G\Uparrow$. In either case $G\Uparrow$ satisfies (1.2).

   Let $F\Downarrow$ be non-empty and choose $H \in \min F\Downarrow$. Then $(H \setminus E_\mathcal{T}) \notin \mathcal{L}$ implies $H \neq \varnothing$, hence (1.2) implies $\exists G \in \mathcal{L}.\, \exists e \notin G.\, G \cup \{e\} = H$. Now $(H \setminus E_\mathcal{T}) \notin \mathcal{L}$ implies $t \in (G \cup \{e\})\Uparrow$ and by condition 3 of Definition 3.5 we know that there is a configuration $X\checkmark_\mathcal{T}$ such that $G \cup X \in \mathcal{L}$. Then $((G \cup X) \cap E_\mathcal{T})\checkmark_\mathcal{T}$ implies that $(G \cup X)\Uparrow = (G \cup X) \setminus E_\mathcal{T} \cup \{t\} = G \setminus E_\mathcal{T} \cup \{t\}$ is in $\mathcal{H}$. Because H is minimal in $F\Downarrow$ we cannot have $G \setminus E_\mathcal{T} = H \setminus E_\mathcal{T}$, hence $(G \cup X)\Uparrow = F \setminus \{e\}$ satisfies (1.2).

4. Let $F\Uparrow \in H$ be arbitrary such that $F\Uparrow\checkmark_\mathcal{H}$. Then by Definition 3.6 we know that $F\checkmark_\mathcal{L}$ which implies by Definition 2.6 that F is maximal in $\mathcal{L}$. If we assume a configuration $G\Uparrow \in H$ such that $F\Uparrow \subset G\Uparrow$ we can infer by condition 4 of Definition 3.5 that $\exists H \in \mathcal{L}.\, F \subseteq H \wedge H\Uparrow = G\Uparrow$. Because F is maximal in $\mathcal{L}$ we know $F = H$ and $F\Uparrow = G\Uparrow$ which contradicts our assumption. □

References

[1] L. Aceto. *Action Refinement in Process Algebras.* PhD thesis, University of Sussex, Feb. 1991. report no. 3/91.

[2] L. Aceto and M. Hennessy. Towards action-refinement in process algebras. Computer Science Report 3/88, University of Sussex, Apr. 1988.

[3] L. Aceto and M. Hennessy. Adding action refinement to a finite process algebra. In J. Leach Albert, B. Monien, and M. R. Artalejo, editors, *Automata, Languages and Programming*, number 510 in Lecture Notes in Computer Science, pages 506–519. Springer-Verlag, 1991.

[4] G. Boudol and I. Castellani. Flow models of distributed computations: Event structures and nets. Rapports de Recherche 1482, INRIA, July 1991.

[5] E. Brinksma. What is the method in formal methods? Memoranda Informatica 91–79, University of Twente, 1989.

[6] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS specifications, their implementations and their tests. In B. Sarikaya and G. von Bochmann, editors, *Protocol Specification, Testing, and Verification, VI*, pages 349–360. IFIP WG 6.1, Elsevier Science Publishers B.V., 1987.

[7] P. Broekroelofs. Bipartitioning of LOTOS specifications. Master's thesis, University of Twente, May 1992.

[8] W. R. Cleaveland, editor. *Concur '92*, volume 630 of *Lecture Notes in Computer Science.* Springer-Verlag, 1992.

[9] U. Engberg. Failures semantics for a simple process language with refinement, May 1991.

[10] W. Janssen, M. Poel, and J. Zwiers. Actions systems and action refinement in the development of parallel systems. In J. C. M. Baeten and J. F. Groote, editors, *Concur '91*, volume 527 of *Lecture Notes in Computer Science*, pages 298–316. Springer-Verlag, 1991.

[11] L. Jateaonkar and A. Meyer. Testing equivalences for Petri nets with action refinement. In Cleaveland [8], pages 17–31.

[12] L. Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Trans. Prog. Lang. Syst.*, 1(1):84–97, July 1979.

[13] L. Lamport. Reasoning about nonatomic operations. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 28–37. ACM, 1983.

[14] L. Lamport. *win* and *sin*: Predicate transformers for concurrency. *ACM Trans. Prog. Lang. Syst.*, 12(3):396–428, July 1990.

[15] M. Nielsen, U. Engberg, and K. G. Larsen. Fully abstract models for a process language with refinement. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 523–549. Springer-Verlag, 1989.

[16] A. Rensink. Posets for configurations! In Cleaveland [8], pages 269–285.

[17] J. Schot. *The Role of Architectural Semantics in the Formal Approach of Distributed Systems Design*. PhD thesis, University of Twente, Feb. 1992.

[18] R. van Glabbeek. The refinement theorem for ST-bisimulation semantics. Report RvG-8906, Centre for Mathematics and Computer Science, 1989.

[19] R. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions*. PhD thesis, Free University of Amsterdam, 1990.

[20] R. van Glabbeek and U. Goltz. Equivalences and refinement. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes*, volume 469 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[21] R. van Glabbeek and U. Goltz. Equivalences and refinement. SFB-Bericht 432/12/90 A, Technische Universität München, Institut für Informatik, July 1990.

[22] R. van Glabbeek and U. Goltz. Refinement of actions in causality based models. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems — Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 267–300. Springer-Verlag, 1990.

[23] R. van Glabbeek and W. P. Weijland. Refinement in branching time semantics. Report ACMCS-R8922, Centre for Mathematics and Computer Science, Amsterdam, May 1989.

[24] C. A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. On the use of specification styles in the design of distributed systems. *Theoretical Comput. Sci.*, 89(1):179–206, Oct. 1991.

[25] W. Vogler. Bisimulation and action refinement. SFB-Bericht 342/10/90 A, Technische Universität München, May 1990.

[26] W. Vogler. Failures semantics based on interval semiwords is a congruence for refinement. In C. Choffrut and T. Lengauer, editors, *STACS 90*, volume 415 of *Lecture Notes in Computer Science*, pages 285–297. Springer-Verlag, 1990.