Action Systems and Action Refinement in the Development of Parallel Systems

An Algebraic Approach

Wil Janssen, Mannes Poel, & Job Zwiers

University of Twente,
Dep. of Computer Science
P.O. Box 217,
7500 AE Enschede,
The Netherlands
E-mail:{janssenw,infpoel,zwiers}@cs.utwente.nl

Abstract

A new notion of refinement and several other new operators are proposed that allow for a compositional algebraic characterization of action systems and serializability in distributed database systems. A simple design language is introduced and is provided with a semantics essentially based on partial order models.
Various algebraic transformation rules are introduced. They are applied in the design process of a simple system, starting from a specification of functional behaviour via the intermediate step in the form of a "true concurrency" based initial design to an actual implementation on a two processor architecture.

1 Introduction

Two different trends can be observed in recent developments around the specification, verification and design of parallel systems:

One school of thought derives from the *compositionality principle*. A basic assumption here is that complex systems are *composed* out of smaller parts. The compositionality principle requires that a *specification* of some composed system can be derived from *specifications* of the constituent parts, i.e. without knowledge of the internal (syntactic) structure of those parts. An essential requirement for a compositional or modular approach is that a system has a suitable *algebraic* structure, which forms the basis for the analysis of that system.

A second school of thought rejects the algebraic compositional style, claiming that a system should be regarded rather as a collection of (guarded) actions, without any apparent algebraic structure. Such *action systems* as they are called are used by for instance Chandy and Misra [CM] and by Back [Back]. The rationale in the book by Chandy and Misra [CM] is that the algebraic syntactic structure of programming languages is too close to the actual architectures of (parallel) machines and that this aspect should not influence the (initial) design of systems. Questioning of the usefulness of the syntactic structure for analysis and design of parallel systems is also present in the work on *communication closed layers* by Elrad and Francez [EF], and related work on the verification of distributed protocols by Gallager, Humblet and Spira [GHS], and Stomp and de Roever [SR]. Although the *physical* structure of a distributed protocol has the form of a parallel composition of sequential programs, the *logical* structure of a protocol can often be described as a sequential composition of a number of parallel *layers* corresponding to the different phases of the protocol, which does not fit in the syntactic program structure as indicated. A third example of noncompositional reasoning can be found in *serializability theory* as described in the literature on distributed databases ([BHG]).

The main contribution of this paper is that we show how action systems and communication closed layers and concurrent database transactions can be designed and analyzed *in an algebraic, compositional framework*. This is accomplished by introducing a *new notion of refinement of actions* in a model of parallelism strongly related to *partial orders of events*, namely *directed acyclic graphs*. The relationship between refinement and an operation called *conflict composition* is clarified. The latter operation can be seen as a syntactic means to describe "layers" as in [EF] et al. It is a weak form of sequential composition, with nicer algebraic properties however, allowing for useful *transformation rules* for parallel systems. We envisage the role of operations such as conflict composition as being an adequate language for what software engineers call the *initial design stage* of a system. Such operations allow one to make the step from specifications of observable behaviour to *algorithms*, without premature commitment to any particular system architecture. The latter should be taken into account only in the stages following the initial design. The transformation of an initial design employing operations specifying what is called "true concurrency" to an implementation on a given architecture, with for instance a fixed number of processors, is a highly interesting, but largely unsolved problem. [CM][Hooman] We bring together the various techniques in an example where we design a parallel program for calculating binomial coefficients. We show the initial design from a specification of the required functional behaviour, resulting in a maximal parallel solution, followed by transformation to a system suited for a limited number of processors. The example employs a specialization of our algebraic language tailored to a shared variables model. At certain transformation stages we combine algebraic laws with state-based reasoning.

After the short sketch above of the contents of the paper we would like to discuss refinement and related operations in some more detail.

Partial order based descriptions of concurrent systems and refinement of atomic actions in partial orders have been studied extensively [Pratt] [DNM] [NEL] [GG] [Pomello]. We start from such partial order models but impose some extra structure in the form of *conflicts*[1] between events. Conflicting events in a single run cannot occur simultaneously, they have to be ordered. Our notion of conflict is an abstraction of conflicts between transactions in distributed database systems, where transactions are said to be in conflict if they access some common database item. In essence such a notion of conflict occurs also in [Pratt] where it is called colocation of events. The rationale for introducing conflicts is that it allows for the formulation of an interesting notion of *refinement of actions*. Our refinement notion differs from action refinement as e.g. in [GG] in that the refinement of action a inherits only those causal relationships which are imposed by conflicts on the concrete (refined) level (fig. 1). For action refinement as in [GG] an action, say b, in the refinement of some higher level action a inherits *all* ordering relations from that a action.

This minimization of causal relationships allows one to model *serializable* executions of distributed database transactions $P_0, P_1, \cdots, P_n$, ([BHG]) as a refinement of executions of atomic actions $T_0, \cdots, T_n$ thus:

$$\textbf{ref } T_0 = P_0, T_1 = P_1, \cdots, T_n = P_n \textbf{ in}$$

$$T_0 \parallel T_1 \parallel \cdots \parallel T_n$$

On the abstract level one sees sequential executions of the form $T_{i_0} ; T_{i_1} ; \cdots ; T_{i_n}$, where $\{i_1, \ldots, i_n\}$ is some permutation of $\{0, \ldots, n\}$. Refinement with inheritance of *all* causal relations would result in *serial* executions of the form

$$P_{i_0} ; \cdots ; P_{i_n}$$

Our notion of refinement however results in *serializable* executions which are of the form (using *conflict composition*) as

$$P_{i_0} \cdot P_{i_1} \cdot \ldots \cdot P_{i_n}$$

The *conflict composition* $P_1 \cdot P_2$ is defined as follows:
For computations P_1 and P_2, $P_1 \cdot P_2$ results in causally ordering only those elements between P_1

[1] Terminology differs here from what is used to describe event structure as in for instance [Winskel], where "conflicting" events cannot occur in a single run

and P_2 which are conflicting; however – unlike parallel composition – one only wants to introduce ordering from P_1 to P_2, indicating the asymmetric nature of the construct. Our notion of *conflict composition* is related to the notion of *D-local concatenation* in [Gaifman].

As this database example indicates, action refinement does *not* commute with sequential composition. In a general *partial order* framework, it turns out that conflict composition *does not* commute with refinement either because a problem arises with *transitivity* as we will illustrate now. Whereas on the abstract level sequential execution of atomic transactions can be dictated by conflicts, after refinement the order implied by transitivity on the abstract level is no longer justified. Take for example the transactions sketched in figure 1. On the abstract level we have a serial execution of

Figure 1: Refinement of database transactions

three transactions, where T_1 accesses some item x, T_2 accesses items x and y, and T_3 accesses an item y (fig. 1a). This is implemented by refining T_2 into two independent actions, accessing x and y. (fig. 1b) I.e., whereas when executing atomic actions causal order implies temporal order, after refinement this isn't necessarily the case; e.g. in the example above T_3 may very well be executed before T_1!
Our solution.to this problem is not to work with (transitive) partial orders, but with non-transitive orders, described as *directed acyclic graphs* (DAG's). For this class of models we show that action refinement has appropriate algebraic properties, such as commutativity with *sequential composition*, and the property that nested refinement is equivalent to simultaneous refinement.

We formulate algebraic laws for refinement and the different types of composition. An important paradigm in parallel program development, *"communication closed layers"* ([EF]), can now be algebraically characterized (for example in the case of two layers):

$$\text{(CCL)} \qquad (S_1 \cdot S_2) \parallel (T_1 \cdot T_2) = (S_1 \parallel T_1) \cdot (S_2 \parallel T_2)$$

provided no causal dependency between S_1 and T_2, and between S_2 and T_1 exists. The same law with conflict composition replaced by *sequential composition* does *not* hold.

For program development, algebraic equalities do not suffice. Two extra ingredients are required:

- *Reducing nondeterminism.* In accordance with this principle, $S_1 \cdot S_2$ should be a legal implementation of $S_1 \parallel S_2$, for every DAG in the semantics of $S_1 \cdot S_2$ is an element of the semantics of $S_1 \parallel S_2$.
- *Reducing the degree of parallelism.* E.g. when reducing the degree of parallelism to 1 (due to a single processor implementation), $S_1 ; S_2$ might also be seen as a legal implementation of $S_1 \parallel S_2$.

The structure of this paper is as follows: in the next section we introduce the algebraic process language, and a version of this language for shared variables based on guarded assignments and predicates that is used throughout the paper. Next we give a number of algebraic laws and implementation relations that are used in the design of a parallel algorithm to compute binomial coefficients. We also define a semantics for our language, justifying these laws and relations. We conclude with some final remarks and a discussion of possible future work.

2 A general process language

In this paper we use a process language based on a set of *actions* and the notion of *conflicts* between actions. Apart from well-known notions such as *sequential composition* and *parallel composition*,

iteration and *choice* the language includes interesting operators like *refinement*, *atomicity* and *contraction*, and a more liberal form of sequential composition, named *conflict composition*.

This gives a language that allows us to express algebraic properties as well as programming constructs found in current programming languages. We also give an interpretation of the abstract actions and the conflict relation based on shared variables, with guarded assignments as basic actions. We will use the shared variables interpretation as a basis for the examples in this paper. It is however also possible to give other interpretations of these abstract notions as long as there is a sensible notion of *conflicts* between actions.

We first define the abstract syntax, which we extend to shared variables and actions described by predicates. Furthermore an intuition for the operators is given.

2.1 Syntax and intuition of the language

Assume we have a class of *actions* $\mathcal{A}ct$, with typical element a. We assume a symmetric relation *conflict* on actions.
It should be set at the outset that our *conflict* relation abstracts from the notion of conflicts between transactions in distributed databases, where conflicts are generated by accesses to some common database item. So, conflicting actions *can* occur in a single "run" of a system, but in that case they must be *causally ordered*.

The syntax of the language is the following:

$S \in \mathcal{P}rocess$

$$S ::= a \mid \mathbf{empty} \mid \mathbf{skip} \mid S_0 \parallel S_1 \mid S_0 \cdot S_1 \mid S_0 ; S_1 \mid S_0 \ \mathbf{or}\ S_1 \mid S^{\odot} \mid S^{*} \mid \mathbf{aug}(S) \mid \langle S \rangle \mid \mathbf{contr}(S) \mid \mathbf{ref}\ a = S_0\ \mathbf{in}\ S_1$$

Most of the operators have an intuitively simple meaning. The most important operators are the three composition operators: *parallel composition* $S_0 \parallel S_1$, *conflict composition* $S_0 \cdot S_1$ and *sequential composition* $S_0 ; S_1$. In a sequential composition $S_0 ; S_1$ all actions in S_0 precede all actions in S_1. Conflict composition is a more liberal form of sequential composition where only *conflicting* actions are ordered. The iterated versions of conflict composition and sequential composition are denoted by $S^{\odot}$ and S^{*} respectively.
The **skip** statement acts as a unit element for all three types of composition, e.g.:

$$S_0 \parallel \mathbf{skip} = \mathbf{skip} \parallel S_0 = S_0$$

The process **empty** is the unit element for non-deterministic choice (**or**).

We also included a few non standard processes. The process $\langle S \rangle$ denotes execution of S where no interfering actions from other processes are allowed. The means that no action from some other process T may be "in between" two actions from S, where "being in between" refers to *causal ordering* of actions, not to *temporal ordering*.
Contrary to atomic brackets in interleaving models of concurrency, execution of S does allow for parallel activity, provided that such an action a either:

1. does not conflict at all with S actions, i.e. a and S are independent, or
2. causally precedes some S actions, but does not *follow* any, or
3. follows some S actions, but does not precede any.

Related to the atomic processes (and to refinement) is the *contract* operator. The process $\mathbf{contr}(S)$ denotes the atomic execution of S, mapped into *a single action*. Its main purpose is to relate the functional behaviours of processes, independent of their implementation. We do not allow the **contr** operator to appear within the scope of a refinement, as contraction can "hide" the actions to be refined in a single event.

In order to be able to relate processes with respect to their ordering relation, we have the *augment* operator $\mathbf{aug}(S)$. This denotes the process in which the ordering between actions of S is arbitrarily extended.

Finally we have refinement. The **ref** construct is a form of refinement where only *necessary ordering* is included in the refinement. In a process **ref** $a = S_0$ **in** S_1 the action a is executed *atomically* in S_1. All actions in S_0 that are in conflict with actions in S_1 are ordered according to the ordering of A in S_1, but non-conflicting actions are not.

2.2 An interpretation in a shared variables model

In our examples in this paper we use a version of the abstract language based on shared variables. To define this language we have to fill in the abstract notion of actions and the notion of conflict.

The basic actions are guarded multiple assignments, and abstract — uninterpreted — actions, and first order predicate logic formulae representing actions. Assume that we have given a finite set of variables $(x \in) \mathcal{V}ar$, a class of expressions $(f \in) \mathcal{E}xp$ and a class of boolean expressions $(b \in) \mathcal{B}exp$. For every variable $x \in \mathcal{V}ar$ we also have a "hooked" variable $\overleftarrow{x}$ that can be used in formulae to denote the initial value of x. The variables occurring in expression f or boolean expression b are denoted by $var(f)$ and $var(b)$. A set set of variables is denoted by vector notation, i.e. $\bar{x} \in \mathcal{P}(\mathcal{V}ar)$. All variables and expressions are of the same type, which is assumed to be "integer" in examples. Also assume that we have given a class of process names $\mathcal{P}roc$, with typical element A. The syntax of actions is:

$a \in \mathcal{A}ct$

$a \ ::= \ b \,\&\, \bar{x} := \bar{f} \ \mid \ A \ \mid \ \mathit{read}\ \bar{y}, \mathit{write}\ \bar{z} : \phi(\overleftarrow{x}, x)$

The intuition of an action $b \,\&\, \bar{x} := \bar{f}$ is that the process waits until the boolean guard b evaluates to true, thereafter simultaneously assigning f_i to x_i, for each x_i in $\bar{x}$. The action is executed as a single action, i.e. atomically.
Abstract actions A are left uninterpreted, but we assume they have a (syntactically determined) base denoting the variables read and written by the action.

We can define pure guards and simple assignments as $b \,\&\, x_b := x_b$, where x_b is a variable in $var(b)$, and $\mathit{true} \,\&\, \bar{x} := \bar{f}$ respectively.

The notion of *conflicts* can now be based on accesses to variables. There are several possibilities of doing so. We can define conflicts to be *read–read & read–write* conflicts, which means that any two actions that access a common variable are conflicting. In the examples however we take the more liberal notion of *read–write conflicts* only, which implies that two actions are not causally related if they read some common variable, but neither of them writes it.
Let a_1 be the assignment $b_1 \,\&\, \bar{x}_1 := \bar{f}_1$ and a_2 the assignment $b_2 \,\&\, \bar{x}_2 := \bar{f}_2$. We can define the *readset* $R(a)$ and the *writeset* $W(a)$ of an assignment a in the in the normal way, e.g. $R(a_1) = var(b_1) \cup var(\bar{f}_1)$ and $W(a_1) = \bar{x}_1$. We can define conflict in terms of these read- and writesets as follows:

$$\mathit{conflict}(\, a_1, a_2 \,) \stackrel{\text{def}}{=} (R(a_1) \cup W(a_1)) \cap W(a_2) \neq \emptyset \ \vee \ W(a_1) \cap (R(a_2) \cup W(a_2)) \neq \emptyset$$

As actions that occur simultaneously at the beginning of a process can all read some variable x we assume that all variables are initialized to some arbitrary value that is read by all of them.

In order to be able to relate processes and specifications we include first order predicate logic formulae as actions in the language. A first order logic predicate $\mathit{read}\ \bar{y}, \mathit{write}\ \bar{z} : \phi(\overleftarrow{x}, x)$ specifies an action where the variables $\overleftarrow{x}$, like in VDM, denote the initial values of variables in the action and x the final value [Jones]. The sets $\bar{y}$ and $\bar{z}$ determine the base related to the action, being the variables read and written. For example, the predicate

$$\mathit{read}\ x, \mathit{write}\ x : \forall n\ (\overleftarrow{x} = n^2 \Rightarrow x = n)$$

denotes an action calculation the square root of an integer (if it exists). In fact, even guarded assignments can be expressed as predicates. We have

$$(b \,\&\, x := f) \;=\; (\mathit{read}\,(\mathit{var}(b) \cup \mathit{var}(f)), \mathit{write}\; x : b[\overleftarrow{x}\,/x] \wedge x = f[\overleftarrow{x}\,/x])$$

where $b[\overleftarrow{x}\,/x]$ is the boolean expression b in which every free variable x is substituted by $\overleftarrow{x}$.

3 Algebraic laws and implementation relations

In this section we discuss a number of general algebraic laws that hold for our language. The purpose of these laws is *not* to give a complete algebraic axiomatization of the language, but instead they are essential in the construction and derivation of a process from a specification.
In first subsection we state simple algebraic laws concerning associativity, commutativity, and distributivity of the operators. The validity of these laws is a result of the definition of the semantics. Afterwards we will give some relevant laws for the refinement operator. Finally the algebraic law concerning the *communication closed layers* studied by Elrad and Francez [EF] is stated.
In the second section we will also define two important relations $\subseteq$ and $\sqsubseteq$ which are closely connected with the ideas of reduction of *nondeterminism* and reduction of *parallelism*, respectively.

3.1 Algebraic laws

First we will look at some laws concerning the choice operator and parallel, sequential, and conflict composition. It should be remarked that "=" denotes semantical equality. From the semantics, which is defined in the next section, trivially follows:

Lemma 3.1
Let P, Q and R be processes.

- Associativity and Distributivity

Let *op* be $\cdot$ or ; or $\|$ or **or** . Then:

$$(P \mathit{\ op\ } Q) \mathit{\ op\ } R \;=\; P \mathit{\ op\ } (Q \mathit{\ op\ } R)$$

$$P \mathit{\ op\ } (Q \textbf{ or } R) \;=\; (P \mathit{\ op\ } Q) \textbf{ or } (P \mathit{\ op\ } R)$$

$$(P \textbf{ or } Q) \mathit{\ op\ } R \;=\; (P \mathit{\ op\ } R) \textbf{ or } (P \mathit{\ op\ } Q)$$

- Commutativity

$$Q \parallel P = P \parallel Q$$

$$Q \textbf{ or } P = P \textbf{ or } Q$$

- Unit element

Let *op* be $\cdot$ or ; or $\|$. Then:

$$P \mathit{\ op\ } \textbf{skip} = P$$

$$P \textbf{ or } \textbf{empty} = P$$

□

For refinement the situation is more subtle, one of the reasons is that refinement can eliminate order introduced by sequential composition.

Theorem 3.2 *(Algebraic laws for refinement)*
Let Q and P be processes, the refinement operator **ref** obeys the following laws:

$(\,\mathbf{ref}\ a = S\ \mathbf{in}\ a) = \langle S \rangle$

$(\,\mathbf{ref}\ a = S\ \mathbf{in}\ (Q\,;\,a)) = (Q \cdot \langle S \rangle)$

$(\,\mathbf{ref}\ a = S\ \mathbf{in}\ (a\,;\,Q)) = (\langle S \rangle \cdot Q)$

Let op be one of the syntactic operators **or** , $\cdot$, or $\|$ then

$(\,\mathbf{ref}\ a = S\ \mathbf{in}\ (P\ op\ Q)) = ((\,\mathbf{ref}\ a = S\ \mathbf{in}\ P)\ op\ (\,\mathbf{ref}\ a = S\ \mathbf{in}\ Q))$

$(\,\mathbf{ref}\ a = S\ \mathbf{in}\ \langle P \rangle) = \langle\langle\,\mathbf{ref}\ a = S\ \mathbf{in}\ P \rangle\rangle)$

Under the assumption that the events a_0, a_1, are not contained in S_1 and S_0 respectively we have

$(\,\mathbf{ref}\ a_0 = S_0\ \mathbf{in}\ (\,\mathbf{ref}\ a_1 = S_1\ \mathbf{in}\ P)) = (\,\mathbf{ref}\ a_1 = S_1\ \mathbf{in}\ (\,\mathbf{ref}\ a_0 = S_0\ \mathbf{in}\ P))$

$(\,\mathbf{ref}\ a = S\ \mathbf{in}\ P^{\odot}) = (\,\mathbf{ref}\ a = S\ \mathbf{in}\ P)^{\odot}$

However in general

$(\,\mathbf{ref}\ a = S\ \mathbf{in}\ P\,;\,Q) \neq ((\,\mathbf{ref}\ a = S\ \mathbf{in}\ P)\,;\,(\,\mathbf{ref}\ a = S\ \mathbf{in}\ Q))$

□

By induction on the structure of a process we immediately deduce from the above theorem

Corollary 3.3
Let P be a process constructed by only using the syntactic operators $\|$ and $\cdot$, then

$(\,\mathbf{ref}\ a = S\ \mathbf{in}\ P) = P[\langle S \rangle / a]$

where $P[Q/a]$ denotes simply substituting Q for a in P as usual. □

The contract operator **contr** allows us to relate the functional behaviour of processes. One important algebraic law is that the *functional behaviour* of a process cannot change by replacing *conflict composition* by *sequential composition*, or in general by augmenting the ordering between actions.

Theorem 3.4 *(Contraction laws)*
Let P and Q be processes. We then have:

$\mathbf{contr}(P \cdot Q) = \mathbf{contr}(P\,;\,Q) = \mathbf{contr}(\,\mathbf{contr}(P)\,;\,\mathbf{contr}(Q))$

$\mathbf{contr}(\,\mathbf{aug}(P)) = \mathbf{contr}(P)$

The following laws also hold:

$\mathbf{contr}(P\ \mathbf{or}\ Q) = \mathbf{contr}(P)\ \mathbf{or}\ \mathbf{contr}(Q)$

$\mathbf{contr}(\langle P \rangle) = \mathbf{contr}(P)$

Furthermore contraction relates in the following way to refinement:

$\mathbf{contr}(\,\mathbf{ref}\ a = P\ \mathbf{in}\ Q) = \mathbf{contr}(\,\mathbf{ref}\ a = \mathbf{contr}(P)\ \mathbf{in}\ Q) = \mathbf{contr}(P[\,\mathbf{contr}(S)/a])$

□

Let S_0, S_1, T_0 and T_1 be processes such that there is no *conflict* between S_0 and T_1. Also assume that there is no *conflict* between S_1 and T_0. The system $(S_0 \cdot S_1) \parallel (T_0 \cdot T_1)$ has the general semantic structure, sketched in figure 2. It can be divided into two communication closed layers: the semantic structure of $(S_0 \parallel T_0) \cdot (S_1 \parallel T_1)$ is the same. (Intuitively this immediately clear.) The precise result is

Theorem 3.5 *(Communication Closed Layers)*
Let S_0, S_1, T_0 and T_1 be processes such that there is no *conflict* between S_0 (S_1) and T_1 (T_0) respectively. Then

$(S_0 \cdot S_1) \parallel (T_0 \cdot T_1) = (S_0 \parallel T_0) \cdot (S_1 \parallel T_1)$

□

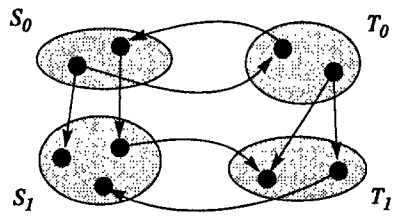

Figure 2: Communication closed layers

Taking one or more processes equal to **skip** leads to

Corollary 3.6
Let P, Q, and R be processes with no conflict between P and Q then

$$P \parallel (Q \cdot R) = Q \cdot (P \parallel R)$$
$$P \parallel (R \cdot Q) = (P \parallel R) \cdot Q$$
$$P \parallel Q = P \cdot Q$$

□

The law in Theorem 3.5 can however not be derived from these simpler laws. We continue with a number of useful implementation relations.

3.2 Implementation relations

In our setup the semantics of a process S, denoted by $[\![S]\!]$, is given by a set of possible *runs*. We can therefore define a *implementation relation*, sometimes called the satisfaction relation, based only on set inclusion and a *sequentialization relation* based on augmentation and set inclusion.

Definition 3.7 *(Implementation and Sequentialization)*
Let P and Q be processes, we say that process P *implements* process Q, denoted by

$P \subseteq Q$ if and only if $[\![P]\!] \subseteq [\![Q]\!]$

Process P *sequentializes* process Q, denoted by

$P \sqsubseteq Q$ if and only if $[\![P]\!] \subseteq [\![\, \mathbf{aug}(Q)]\!]$

i.e. if and only if $P \subseteq \mathbf{aug}(Q)$

□

Observe that the relations $\subseteq$ and $\sqsubseteq$ are transitive and are related as follows:

$$P \subseteq Q \Rightarrow P \sqsubseteq Q$$
$$(P \subseteq Q \sqsubseteq S \vee P \sqsubseteq Q \subseteq S) \Rightarrow (P \sqsubseteq S)$$

By augmenting a process Q to a process P, we cannot change it functional behaviour, only its degree of parallelism. This can be expressed as:

Proposition 3.8
Let P and Q be processes.

$$P \sqsubseteq Q \Rightarrow \mathbf{contr}(P) \subseteq \mathbf{contr}(Q)$$

□

For *interleaving models* of parallelism, one would have that $(P\,;Q) \subseteq (P \parallel Q)$. This is no longer the case for our models since $P\,;Q$ induces in general more ordering than $P \parallel Q$. However, we do have the following.

Proposition 3.9
For processes P and Q:

$$(P \cdot Q) \subseteq (P \parallel Q)$$

$$(P ; Q) \sqsubseteq (P \cdot Q)$$

and thus by the observations above

$$(P ; Q) \sqsubseteq (P \parallel Q)$$

□

In general there is no direct implementation relation between $(P ; Q)$ and $(P \cdot Q)$, apart from the fact that $(P ; Q)$ is an augment of $(P \cdot Q)$.

3.3 Mixed algebraic and state-based reasoning

Program verification for shared variables programming has been studied extensively.([OG], [Lamport], [MP]) It is not our intention to give a full account of such reasoning within our present framework. Rather we describe some useful transformation techniques where we combine algebraic reasoning with state-based reasoning.

A *local annotation* for a term S consists of first order formulae pre_S and $post_S$ called the (global) pre- and postcondition, together with, for each action a occurring in S, a pair $pre_a, post_a$. We require that the free variables of $pre_S, post_S$ are contained in $var(S)$, and similarly that the free variables in $pre_a, post_a$ are contained in $base(a)$. We denote an annotated term S by $\{pre_S\}\tilde{S}\{post_S\}$, where $\tilde{S}$ is like S accept that actions a have been replaced by $\{pre_a\}\ a\ \{post_a\}$. Such an annotation is called *valid* if for any execution of $\langle S \rangle$, starting with a (global) initial state satisfying pre_S, the event(s) corresponding to action a have an initial state satisfying pre_a and final state satisfying $post_a$ (for each action a). (Note that the restriction on free variables guarantees that we can assign a truth value to pre_a and $post_a$ in these local states indeed.) Moreover the global final state of the execution must satisfy $post_S$.

We indicate here how to prove correctness of a given local annotation for the special case that S is build up from action using *exclusively* conflict composition, sequential composition, iteration, and choice. For such S, let $S^;$ be the term obtained from S by replacing all conflict composition by sequential composition operations, and $\odot$ by $*$. The resulting term $S^;$ is a purely sequential program to which the well-known classical techniques for sequential program proving apply, such as Hoare's logic. In particular one can use a Hoare style *proof outline* $\{pre_S\}\tilde{S}^;\{post_S\}$ to show the correctness of the Hoare formula $\{pre_S\}S^;\{post_S\}$. Note that such a proof outline attaches first order assertions to all intermediate control points in $S^;$, each referring to the *global* state of $S^;$. In particular the proof outline attaches global assertions pre_a^G and $post_a^G$ to the points immediately before and after action a.

Claim: let $\{pre_S\}\tilde{S}\{post_S\}$ be a locally annotated term. Let $\{pre_S\}\tilde{S}^;\{post_S\}$ be the corresponding proof outline as indicated above which has been shown to be correct. Moreover assume that for each action a in S we have that

$$pre_a^G \Rightarrow pre_a$$

$$post_a^G \Rightarrow post_a.$$

Then the local annotation is valid. □

We use local annotations in the following program transformation technique. Let S be a term containing a subterm P of the form $P = Q \cdot R$. Assume that the only conflict between Q and R is caused by some action a_1 in Q and action a_2 in R. (The generalization to more conflicts is straightforward, if tedious.) Furthermore assume that we have a valid local annotation for S containing

$$\{\neg b\}a_1\{b\} \text{ and } \{b\}a_2\{\cdots\}$$

and that for all actions a in S except a_1 we have that

$$W(a) \cap var(b) = \emptyset$$

where $W(a)$ denotes the set of variables written by a.
Then we claim that by replacing P by $P' = Q \parallel R'$, where R' is R with action a_2 replaced by $b \& a_2$, in the local annotation results in again a valid annotation.

Proof: omitted.

Example. Let $S = (x := 2 \cdot y := x^2)$. If we have a valid local annotation containing

$$\{x = 0\}\ x := 2\ \{x > 0\}, \quad \{x > 0\}\ y := x^2\ \{\cdots\}$$

then we can replace the conflict composition within that annotation by

$$x := 2 \parallel (x > 0) \,\&\, y := x^2$$

4 An example of program development

We illustrate the algebraic laws and transformation techniques introduced above in the development of a simple program P calculating binomial coefficients. The idea is to calculate $\binom{n}{k}$ in shared variables $c(n,k)$ for indices $(n,k) \in I$, where I is the index set

$$I = \{(n,k) \mid 0 \leq n \leq m, 0 \leq k \leq n\}$$

and m is some number determining how "deep" we have to compute. We assume that the variables $c(n',k)$ are initialized to zero; an assumption that is made explicit in the following functional specification of P:

$$\mathbf{contr}(P) \subseteq (\bigwedge_I \overleftarrow{c}\,(n,k) = 0) \Rightarrow (\bigwedge_I c(n,k) = \binom{n}{k}) \tag{1}$$

(We abbreviate $\bigwedge_{(n,k)\in I} \varphi$ by $\bigwedge_I \varphi$.) The well-known recurrence relation:

$$\binom{n}{0} = \binom{n}{n} = 1, \text{ for } n \geq 0,$$

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \text{ for } n > 0, 0 < k < n,$$

suggests that we calculate $c(n,k)$ by executing action $a(n,k)$, defined as

$$a(n,k) \;=\; c(n,k) := 1, \text{ for } k = 0 \text{ and } k = n$$

$$a(n,k) \;=\; c(n,k) := c(n-1,k-1) + c(n-1,k), \text{ for } 0 < k < n$$

It will be clear that for $0 < k < n$, action $a(n,k)$ is in conflict (only) with actions $a(n-1,k-1)$ and $a(n-1,k)$, and moreover that those two actions should *precede* $a(n,k)$. This is illustrated in fig. 3. We can impose this precedence relation by denoting P as a (conflict) composition of layers $L(k)$, each of which executes the actions $a(n,k)$, for $n = k, k+1, \cdots, m$. So:

$$P \;=\; L(0) \cdot L(1) \cdot L(2) \cdot \ldots \cdot L(m)$$

where

$$L(k) \;=\; a(k,k) \cdot a(k+1,k) \cdot \ldots \cdot a(m,k)$$

In [CM] this example is given as an action system where all actions $a(n,k)$ are executed infinitely often in an arbitrary order, eventually reaching a stable situation, thereby avoiding architectural bias. A problem is that such a description has no compositional (algebraic) structure. Moreover, if one would like to add *control flow* in order to transform towards a more efficient implementation, boolean variables and guards have to be added to encode this flow of control, which is somewhat unsatisfactory.

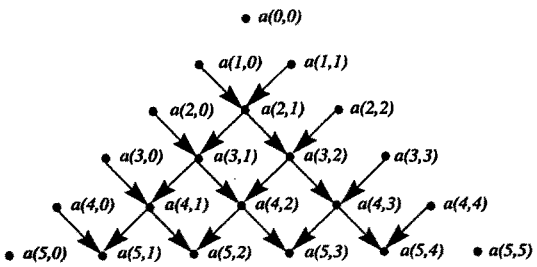

Figure 3:

Note that we have a structured (algebraic) description of the system without imposing unnecessary flow of control or architectural bias, however. Such an *initial design* can be transformed towards a design that suits a particular architecture as we show below. First however we prove *functional correctness* of our initial design, exploiting its compositional structure.

Action indices (i,j) occur *textually* in P in the order defined by

$$(i,j) \prec (k,l) \text{ iff } j < l \text{ or } (j = l \text{ and } i < k)$$

Note however that actions are not *executed* in this order; in fact, since we used conflict composition exclusively, action $a(i,j)$ and $a(k,l)$ will executed independently *except* when in conflict. In the latter case they execute in the order determined by "$\prec$." Note that $(n-1,k-1) \prec (n,k)$ and $(n-1,k) \prec (n,k)$, so indeed $a(n,k)$ will be preceded by $a(n-1,k-1)$ and $a(n-1,k)$.

To prove the correctness of (1) we must construct a local annotation of the form

$$\{(\bigwedge_I c(n,k) = 0)\}\ \tilde{P}\ \{(\bigwedge_I c(n,k) = \binom{n}{k})\} \qquad (*)$$

As discussed in section 3.3 it suffices here to consider the sequential program $\tilde{P}'$ obtained from P by replacing conflict composition by sequential composition, for which we must show:

$$\{(\bigwedge_I c(n,k) = 0)\}\ \tilde{P}'\ \{(\bigwedge_I c(n,k) = \binom{n}{k})\}$$

This is easily shown, using classical Hoare style logic for sequential programs. To be precise, the inductive assertion $\varphi(n,k)$ attached as precondition to action $a(n,k)$ can be chosen as follows: Let $J = \{(i,j) \in I \mid (i,j) \prec (n,k)\}$. Then

$$\varphi(n,k) \stackrel{\text{def}}{=} \bigwedge_J c(i,j) = \binom{i}{j} \wedge \bigwedge_{I-J} c(i,j) = 0$$

Apart from proving the functional correctness of the initial design, this also shows the validity of the *local* annotation $(*)$ if we annotate action $a(i,j)$ as:

$$\{c(i,j) = 0 \wedge c(i-1,j) > 0\}\ a(i,j)\ \{c(i,j) > 0\}$$

We use this local annotation in our next transformation step where we aim at gradually removing conflict composition in favour of parallel composition and sequential composition. We start with transforming each layer $L(k)$ into a number of parallel components, where the number of components depends upon the number of available processors. For simplicity we illustrate this for the case of two processors.

According to section 3.3 we can transform

$$a(k,k) \cdot \ldots \cdot \{\neg c(i,k) = 0\}\ a(i,k)\ \{c(i,k) > 0\}\ \cdot\ \{c(i,k) > 0\}\ a(i+1,k)\ \cdot \ldots \cdot a(m,k)$$

into

$$a(k,k) \cdot \ldots \cdot a(i,k)\ \|\ (c(i,k) > 0\ \&\ a(i+1,k)) \cdot \ldots \cdot a(m,k)$$

where i denotes the action in every layer where we split the layer in two. So we can transform $L(k)$ to $D(k) \parallel U(k)$ where

$$D(k) = a(k,k) \cdot \ldots \cdot a(i,k), \text{ and}$$

$$U(k) = (c(i,k) > 0)\ \&\ a(i+1,k)) \cdot a(i+2,k) \cdot \ldots \cdot a(m,k)$$

By doing so we have transformed P into

$$P_1 = (D(0) \parallel U(0)) \cdot (D(1) \parallel U(1)) \cdot \ldots \cdot (D(m) \parallel U(m))$$

Note that we have conflicts between $D(i)$ and $D(i+1)$, between $U(i)$ and $U(i+1)$, and between $D(i)$ and $U(i)$. However, no conflicts occur between $D(i)$ and $U(j)$ for $i \neq j$! So we can apply the *communication closed layers* law (3.5), and obtain the following program P_2:

$$P_2 = (D(0) \cdot D(1) \cdot \ldots \cdot D(m)) \parallel (U(0) \cdot U(1) \cdot \ldots \cdot U(m))$$

Finally we take into account that we have only two processors available. To this end we will intentionally decrease the amount of parallelism by introducing extra ordering between actions. Note that by law 3.8 this cannot effect the functional correctness. So we deduce

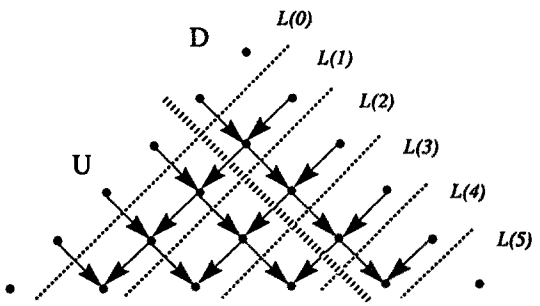

Figure 4:

$$\begin{aligned}
P_2 &= (D(0) \cdot D(1) \cdot \ldots \cdot D(m)) \parallel (U(0) \cdot U(1) \cdot \ldots \cdot U(m)) \\
&\sqsupseteq \{ \text{ proposition 3.9 } \} \\
&\quad (D(0)\,;\, D(1)\,;\, \ldots\,;\, D(m)) \parallel (U(0)\,;\, U(1)\,;\, \ldots\,;\, U(m)) \\
&\sqsupseteq \{ \text{ proposition 3.9 } \} \\
&\quad (D'(0)\,;\, D'(1)\,;\, \ldots\,;\, D'(m)) \parallel (U'(0)\,;\, U'(1)\,;\, \ldots\,;\, U'(m)) \\
&\stackrel{\text{def}}{=} P_3
\end{aligned}$$

where $D'(k)$ is obtained from $D(k)$ by replacing every conflict composition by sequential composition, and analogously for $U'(k)$. We take P_3 as our final solution.

5 Semantics of the language

In this section we present a semantics for the language used in this paper. This semantics relies upon the distinction between process and environment actions, resulting in a definition of the composition operators based on intersection.

Computations, or single "runs" of a system, are modelled by a pair $(E, <)$, where E is a set of *events* and $<$ denotes the successor relation, or *causal order*. The pair $(E, <)$ must form a *directed acyclic graph* (DAG), which means that the transitive closure of the successor relation defines a *partial order*.

First we introduce the domain of computations and some notation. Then the semantics of processes is defined. Finally we gives a semantics for the shared variables interpretation of the language.

5.1 Preliminaries

In our framework a computation consists of a pair $(E, <)$ where E is a set of events from some domain *Event* and $<$ is the successor relation or *causality relation*, i.e. if two events e_1 and e_2 are related ($e_1 < e_2$) then event e_1 must conceptually precede event e_2. This relation is irreflexive but not transitive; its transitive closure $<^+$ must be irreflexive too, which means that $(E, <^+)$ is an irreflexive partial order. For now, events are left uninterpreted.

Different events in a run can be occurrences of the same *action*, therefore E is a *multiset* of actions, rather than a set of actions. Formally speaking, we should define DAG-s over multisets as equivalence classes under isomorphism of 4-tuples $(E, \Sigma, <, \mu)$, where E is the set of events, $<$ the set of edges, Σ the set of possible actions and μ the labelling function mapping events to actions. If we assume all events can be distinguished we can restrict ourselves to the simpler model of pairs $(E, <)$.

In this paper we are only interested in *finite behaviour* of processes. Therefore the multiset E is always finite. It is also possible to give a model for diverging behaviour (which is done in [JPZ]) giving the possibility to discuss notions such as fairness. Here we restrict ourselves to the finite model.

A system can have many different runs due to nondeterminism or different initial states. The denotation of system is therefore a set of computations, where every computation denotes a possible run of the system.

We assume that a binary, symmetric relation *conflict* on events is given. Every pair of conflicting events must be ordered, as conflicting events can intuitively be seen as accesses to some common resource that have to take place under *mutual exclusion.*

A DAG $(E, <)$ has a certain *functional behaviour.* This functional behaviour can be represented as a *single event,* in some way analogously to the way a computation in the *interleaving framework* can be represented by a single state transformation. A computation in the interleaving framework consists of a *sequence of states.* This gives a contraction function with functionality $\mathcal{S}tate^* \rightarrow \mathcal{S}tate^2$:

$$(s_0, s_1, \ldots, s_{n-1}, s_n) \xrightarrow{contr} (s_0, s_n)$$

This can also be viewed as a function mapping a sequence of *pairs of states* (i.e. state transformers) to a single pair of states:

$$((s_0, s_1), (s_1, s_2), \ldots, (s_{n-1}, s_n)) \xrightarrow{contr} (s_0, s_n)$$

These state pairs can be viewed as *events*, i.e. $e_i = (s_i, s_{i+1})$, resulting in a mapping from a sequence of events to an event.

$$(e_0, e_1, \ldots, e_{n-1}) \xrightarrow{contr} e$$

This mapping for sequences of events can be generalized to our computations. We therefore want to define a function *contr* wich maps computations to events. This function is not defined for arbitrary DAG's, we need some notion of *well-formedness.* In the interleaving framework for example a sequence $((s_0, s_1), (s_2, s_3))$ such that $s_1 \neq s_2$ cannot be contracted to a single state transformation, as it has an inconsistent state pair.

A computation can be contracted if and only if it is *well-formed* and finite. (In this paper every computation is finite.) To define contraction we postulate a *partial function*

$$\rho : \mathcal{E}vent^2 \xrightarrow{p} \mathcal{E}vent$$

(analogously to the composition of binary relations). This function defines the contraction of a pair of events. Two events can be contracted to a single event if they are either *direct successors* or *unrelated.* I.e. e_1 and e_2 are contractable in a computation $(E, <)$ iff

$$\neg(\exists e \in E(e_1 <^+ e <^+ e_2 \ \vee \ e_2 <^+ e <^+ e_1))$$

If e_1 and e_2 are contractable and $\neg(e_2 <^+ e_1)$ then the function $\rho(e_1, e_2)$ is defined. The exact definition of the contraction depends upon the underlying model of events. We require ρ to be associative, and to be commutative for non-conflicting events.

We can define $contr(E, <)$ inductively as:

$$contr(\emptyset, \emptyset) \stackrel{\text{def}}{=} (\emptyset, \emptyset), \quad contr(\{e\}, \emptyset) \stackrel{\text{def}}{=} (\{e\}, \emptyset)$$

$$contr(E_1 \cup E_2, <_1 \cup <_2 \cup X) \stackrel{\text{def}}{=} contr(\{e_1, e_2\}, <), \text{ if } <_1 \subseteq E_1^2, <_2 \subseteq E_2^2, X \subseteq (E_1 \times E_2)$$

where $e_1 = contr(E_1, <_1), e_2 = contr(E_2, <_2)$, and $<= \{(e_1, e_2)\}$ if $X \neq \emptyset$, else $<= \emptyset$. Informally, if we can split a computation into two "layers" without any ordering in opposite direction, we then can compute the contractions of the layers seperately and compose them.

Thus, in light of the remarks above, we define our domain of computations *Comp* as

$$\begin{aligned} Comp \stackrel{\text{def}}{=} \{ (E, <) \quad | \quad & E \subseteq Event \wedge < \subseteq (E \times E) \wedge \forall e \in E\, (e \not< ^+ e) \wedge \\ & \forall e, e' \in E\, (conflict(e, e') \rightarrow (e < e' \vee e' < e)) \wedge \\ & (E, <) \text{ is well-formed } \} \end{aligned}$$

Our approach to define the semantics is to give the semantics of a process as computations consisting of two types of labelled events: *process* events and *environment* events. The process events give the semantics of the process in isolation, whereas the environment events model the (arbitrary) behaviour of other processes. An alternative would be to use process events only and to leave "gaps" to be filled by other processes. The approach taken in this paper enables us to define the semantics of e.g. atomic statements and refinement in an elegant way (see [JPZ] for comparison of the two semantics). This leads to a semantic function where we have to specify the name of the process and all processes in the environment. Formally:

$$[\![\cdot]\!] : Process \rightarrow (Pid \rightarrow (\mathcal{P}(Pid) \rightarrow \mathcal{P}(Comp)))$$

So the intention is that $[\![S]\!](P)(Env)$ is the set of all computations allowed by process P with body S in the context of a set of processes Env, where $P \notin Env$.
In the approach taken by [BKP] one would have a single environment label in the semantics. Our approach however facilitates the definition of parallel composition.

The semantics of a process *in isolation* (a complete system) can now be defined as function:

$$[\![S]\!] \stackrel{\text{def}}{=} [\![S]\!](P)(\emptyset)$$

Events are labelled with the name of the process that executes them. For an event e this label is denoted by $pid(e)$. We extend the definition of *pid* to computations:

$$pid(E, <) \stackrel{\text{def}}{=} \bigcup \{pid(e) \mid e \in E\}$$

Furthermore events are tagged with a name, corresponding with the action they belong to. This is used to recognize events in order to be able to refine them by other computations. The name of an event e is denoted by $name(e)$.

5.2 The semantics of processes

To ease the definition of the semantics of some processes we introduce some auxiliary processes, viz. the **step** and **maxpar** process. The former denotes a process where only a single process event is executed and the latter denotes all processes where the ordering relation between process and environment events is minimal in the sense that only *conflicting* events are ordered. This is a requirement that holds for every computation in the semantics of a process.
Their semantics can be defined as follows:

$$[\![\textbf{step}]\!](P)(Env) =$$

$$\{(E, <) \in Comp \mid \exists! e \in E (pid(e) = P) \wedge pid(E, <) \subseteq Env \cup \{P\} \}$$

where $\exists!$ denotes *unique* existence.

$$[\![\textbf{maxpar}]\!](P)(Env) = \{(E, <) \in Comp \mid pid(E, <) \subseteq Env \cup \{P\}$$
$$\forall e, e' \in E\, ((pid(e) = P \wedge pid(e') \neq P \wedge (e < e' \vee e' < e)) \rightarrow conflict(e, e'))\}$$

Actions a have a semantics consisting of computation containing a single process event labelled a. We assume the set of possible events corresponding with an action a is determined by a characteristic predicate φ. This leads to the following definition.

$$[\![a]\!](P)(Env) = [\![\textbf{maxpar}]\!](P)(Env) \cap [\![\textbf{step}]\!](P)(Env) \cap \{(E, <) \in Comp \mid \exists e \in E\ (\varphi(e) \wedge name(e) = a)\}$$

The semantics of the **empty** process simply consists of the empty set of computations.

$$[\![\textbf{empty}]\!](P)(Env) = \emptyset$$

The semantics of **skip** is simple: the process P itself performs no actions, whereas the environment can perform arbitrary actions.

$$[\![\textbf{skip}]\!](P)(Env) = \{h \in Comp \mid pid(h) \subseteq Env\}$$

The **or** -operator represents nondeterministic choice. Its semantics therefore simply boils down to set theoretic union.

$$[\![S_0 \textbf{ or } S_1]\!](P)(Env) = [\![S_0]\!](P)(Env) \cup [\![S_1]\!](P)(Env)$$

Our semantics have been constructed to be able to define parallel composition in a straightforward way. The environment events should model the behaviour of other components in a parallel composition. When defining the semantics of $S_0 \parallel S_1$ the environment of S_0 should include the behaviour of S_1 and vice versa. Taking the parallel composition then simply boils down to taking the *intersection* of the respective semantics.

$$[\![S_0 \parallel S_1]\!](P)(Env) = ([\![S_0]\!](P_0)(Env_0) \cap [\![S_1]\!](P_1)(Env_1))[P/P_0, P/P_1]$$

In this definition P_0 and P_1 are "fresh" process names and $Env_0 = Env \cup \{P_1\}$ and $Env_1 = Env \cup \{P_0\}$. The *renaming* $U[P/P_0, P/P_1]$ operates pointwise on the computations in U. It relabels the events labelled P_0 or P_1 into events labelled P. This renaming is necessary as after the parallel composition we do not want to distinguish between events of the two components, whereas it is necessary to distinguish them as *process* or *environment* events.

The conflict composition can again be defined analogously to the parallel composition, where all conflicting events are ordered in the correct manner. To do so we define

$$conflict\_order(P_1, P_2) \stackrel{\text{def}}{=} \{(E, <) \in Comp \mid \forall e, e' \in E(pid(e) = P_1 \wedge pid(e') = P_2 \wedge conflict(e, e') \rightarrow e < e')\}$$

The semantics of conflict composition is now

$$[\![S_0 \cdot S_1]\!](P)(Env) = ([\![S_0]\!](P_0)(Env_0) \cap [\![S_1]\!](P_1)(Env_1) \cap conflict\_order(P_0, P_1))[P/P_0, P/P_1]$$

It is more difficult to give the semantics of sequential composition in an analogous way, due to the fact that the ordering between process events and environment events is *minimal*, i.e. only conflicting events are ordered. We can however define the semantics of sequential composition as an augment (extension of the ordering) of the semantics of conflict composition: in this case all process events from the two components must be ordered.
We define a function $\alpha_{P \rightarrow Q}$ that performs that augment, which can be defined as:

$$\alpha_{P \rightarrow Q}(E, <) \stackrel{\text{def}}{=} (E, < \cup X)$$

where X is the extension of the ordering, consisting of pairs of events labelled P and Q respectively, i.e.

$$X \stackrel{\text{def}}{=} \{(e, e') \mid e, e' \in E \wedge pid(e) = P \wedge pid(e') = Q\}$$

Note that this function does not always result in a DAG in general, it may lead to cycles. In our definition of the semantics of sequential composition however it does.
This results in the following semantics for sequential composition:

$$[\![S_0 ; S_1]\!](P)(Env) = (\alpha_{P_0 \rightarrow P_1}([\![S_0]\!](P_0)(Env_0) \cap [\![S_1]\!](P_1)(Env_1) \cap conflict\_order(P_0, P_1)))[P/P_0, P/P_1]$$

One can also define the semantics of sequential composition in a more direct way, by taking the disjoint union of the two computations and augmenting the resulting order in the correct way, meaning that process actions from the two components must all be ordered in the same direction, whereas other conflicting but yet unordered events can be ordered arbitrarily.

The iterated versions of the conflict and sequential composition can intuitively be seen as a non-deterministic choice of zero or more times executing a statement S, using the right form of composition. We therefore inductively define the following statements:

$$S^{[0]} \stackrel{\text{def}}{=} \textbf{skip} \qquad S^{[i+1]} \stackrel{\text{def}}{=} S^{[i]} \,;\, S$$

$$S^{(0)} \stackrel{\text{def}}{=} \textbf{skip} \qquad S^{(i+1)} \stackrel{\text{def}}{=} S^{(i)} \,.\, S$$

The semantics can now be defined as

$$[\![S^*]\!](P)(Env) = \bigcup_{i \geq 0} [\![S^{[i]}]\!](P)(Env)$$

$$[\![S^\odot]\!](P)(Env) = \bigcup_{i \geq 0} [\![S^{(i)}]\!](P)(Env)$$

The intuition of atomic statements $\langle S \rangle$ is that during the execution of S no other processes may interfere; this does not preclude (conceptually) parallel events from other processes. This can be compared to refinement in the sense that the events in S can be seen as the result of the refinement of some abstract action A. This is the case if there are no cycles in the transitive ordering of A which is the case iff for every pair of process events e, e' there is no environment event e'' such that $e <^+ e''$ and $e'' <^+ e'$, giving $e <^+ e'$.
We therefore define:

$$\begin{aligned} [\![\langle S \rangle]\!](P)(Env) \;=\; & [\![S]\!](P)(Env) \cap \\ & \{(E, <) \in \mathit{Comp} \mid \forall e, e', e'' \in E \\ & \quad (\mathit{pid}(e) = \mathit{pid}(e') = P \wedge \mathit{pid}(e'') \neq P \rightarrow \neg(e <^+ e'' <^+ e')) \} \end{aligned}$$

The semantics of the *contract* operator is related to the semantics of the atomic statements. Informally, a process $\textbf{contr}(S)$ denotes the atomic execution of S, mapped into *a single process event*, analogously to the contract function *contr*, but for process events only! We denote this process contraction function as contr_P. It is defined as:

$$\mathit{contr}_P(E, <) \stackrel{\text{def}}{=} ((E - E_P) \cup \{e_p\}, (< \cap (E - E_P)^2) \cup X)$$

where

$$E_P = \{e \in E \mid \mathit{pid}(e) = P\}, \quad e_p = \mathit{contr}(E_P, < \cap E_P^2)$$

$$X = \{(e, e_p) \mid e \in E - E_P \wedge \exists e' \in E_p (e < e')\} \cup \{(e_p, e) \mid e \in E - E_P \wedge \exists e' \in E_p (e' < e)\}$$

This function is only defined for computations $(E, <)$ such that

$$\neg(\exists e_1, e_2, e_3 \in E (\mathit{pid}(e_1) = \mathit{pid}(e_3) = P \neq \mathit{pid}(e_2) \wedge e_1 <^+ e_2 <^+ e_3))$$

i.e. the set of P-events in $(E, <)$ can be regarded as *atomic*.

This gives the following semantics:

$$\begin{aligned} [\![\, \textbf{contr}(S)]\!](P)(Env) \;=\; & [\![\, \textbf{maxpar} \,]\!](P)(Env) \cap [\![\, \textbf{step} \,]\!](P)(Env) \cap \\ & \{\mathit{contr}_P(h) \mid h \in [\![\langle S \rangle]\!](P)(Env) \} \end{aligned}$$

Augment closure can of be defined by simply augmenting the order arbitrarily:

$$[\![\, \textbf{aug}(S)]\!](P)(Env) = \{(E, <) \in \mathit{Comp} \mid \exists (E, <') \in [\![S]\!](P)(Env) \; (<=<' \cup X)\}$$

where X is *any* relation on $E \times E$ such that $<' \cup X$ satisfies the restrictions imposed by *Comp*. This is implicit by defining $(E, <) \in \mathit{Comp}$.

The refinement construct has a more intricate semantics. Informally the semantics of a process $\textbf{ref}\ a = S_0\ \textbf{in}\ S_1$ can be given as follows. First we compute the semantics of S_1, where the semantics of the action a is given by computations consisting of a single a-labelled process event. In the

computations that are the result of this semantics, all abstract events named a are substituted by elements from the semantics of S_0, where the semantics of S_0 is computed in an *empty environment*, as to make sure that S_0 is executed *atomically*. This substitution is done in such a way that only *conflicting* events are ordered, and they are ordered in a manner corresponding to the ordering relation of the abstract event. The substitution should preserve the *functional behaviour* of the computation, i.e. the *contraction* of the computations that are substituted should be equal to the abstract event for which they are substituted. Figure 5 illustrates this definition of refinement. We

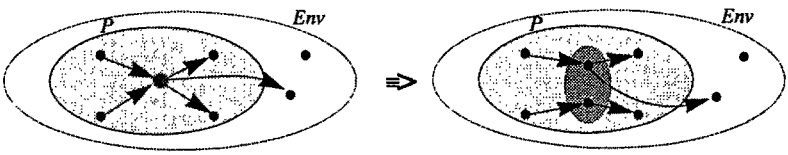

Figure 5: An example of refinement

can formalize this as follows. For computations $h_0 = (E_0, <_0)$ and $h_1 = (E_1, <_1)$, and process name a we define a refinement operator $h_0[a \Rrightarrow h_1]$ substituting elements from h_1 for abstract events named a in h_0. In this definition we assume that a occurs only once in a computation, the extension to multiple occurrences is straightforward but leads to a tedious definition.
For a multiset E let $e_a(E)$ denote the (unique) a-labelled event $e \in E$. Assume $e = e_a(E_0)$ exists and that $contr(h_1) = e$. We now define:

$$h_0[a \Rrightarrow h_1] \stackrel{\text{def}}{=} ((E_0 - \{e\}) \cup E_1,\ (<_0 \cap (E_0 - \{e\})^2) \cup <_1 \cup X)$$

where

$$\begin{aligned} X \;=\; & \{\, (e_0, e_1) \mid e_0 \in E_0 - \{e\} \wedge e_1 \in E_1 \wedge e_0 <_0 e \wedge conflict(e_0, e_1) \,\} \cup \\ & \{\, (e_1, e_0) \mid e_1 \in E_1 \wedge e_0 \in E_0 - \{e\} \wedge e <_0 e_0 \wedge conflict(e_0, e_1) \,\} \end{aligned}$$

We extend this definition to *sets* of computations D_0 and D_1 by pointwize extension:

$$\begin{aligned} D_0[a \Rrightarrow D_1] \;\stackrel{\text{def}}{=}\; & \{\, h_0[a \Rrightarrow h_1] \mid h_0 \in D_0 \wedge h_1 \in D_1 \wedge e_a(h_0) \text{ exists } \wedge contr(h_1) = e_a(h_0) \,\} \cup \\ & \{\, h_0 \in D_0 \mid e_a(h_0) \text{ does not exist } \} \end{aligned}$$

Now we can define the semantics of refinement as:

$$[\![\textbf{ref } a = S_1 \textbf{ in } S_0]\!](P)(Env) = [\![S_1]\!](P)(Env)[a \Rrightarrow [\![S_0]\!](P)(\emptyset)]$$

5.3 The semantics of the shared variables model

When we work in some fixed model such as the shared variables model, we can give an interpretation to the set of events, the conflict relation and the contraction function ρ. In this section we give such an interpretation and a semantics for actions.

The events model *state transformations* caused by executing guarded assignments $b \,\&\, \bar{x} := \bar{f}$. Such events are described as labelled state transitions e of the form:

$$e = (s \xrightarrow{P_i, \beta} s')$$

The label consists of the name P_i of the process that caused the event and a base β which is a pair consisting of the set of variables that are read and the set of variables that are written by the event. We denote these by $pid(e)$ and $base(e)$. The base of e is also the domain of the (local) states s and s', i.e. $s : base(e) \rightarrow \mathcal{V}al$ and $s' : base(e) \rightarrow \mathcal{V}al$, where $\mathcal{V}al$ is some given domain of values of expressions. The first component of a base β (the read component) is denoted by β_R, the write component by β_W. The name of an event e is denoted by $name(e)$.

In the informal explanation of the language the concept of *conflicts* between actions was defined to be *read-write* conflicts only. For two events e and e' we can define the predicate *conflict* as:

$$\begin{aligned} conflict(e, e') \;\stackrel{\text{def}}{=}\; & ((base_R(e) \cup base_W(e)) \cap base_W(e') \neq \emptyset \vee \\ & (base_R(e') \cup base_W(e')) \cap base_W(e) \neq \emptyset) \end{aligned}$$

For our shared variables model we have to define a partial function ρ mapping pairs of events to a single event. The domain of ρ, $dom(\rho)$, is defined as:

$$dom(\rho) = \{(e_1,e_2) \in \mathcal{E}vent^2 \mid \neg conflict(e_1,e_2) \wedge \forall x \in base_R(e_1) \cap base_R(e_2)(e_1(x) = e_2(x))\} \cup \{(e_1,e_2) \in \mathcal{E}vent^2 \mid conflict(e_1,e_2) \wedge \forall x \in base(e_1) \cap base(e_2)(e_1'(x) = e_2(x))\}$$

where $e(x)$ and $e'(x)$ denote the initial and final values of x respectively. The contraction $\rho(e_1,e_2)$ is now defined as: $\rho(e_1,e_2) = e$, where

$base(e) = (base_R(e_1) \cup base_R(e_2), base_W(e_1) \cup base_W(e_2))$ and $pid(e) = P$ and

$$e(x) \stackrel{\text{def}}{=} \begin{cases} e_1(x) & \text{if } x \in base(e_1) \\ e_2(x) & \text{if } x \in base(e_2) - base(e_1) \end{cases} \qquad e'(x) \stackrel{\text{def}}{=} \begin{cases} e_2'(x) & \text{if } x \in base(e_2) \\ e_1'(x) & \text{if } x \in base(e_1) - base(e_2) \end{cases}$$

We now give the semantics of actions. To do so we have to define a characteristic predicate for them. First of all guarded assignments. The intuition of guarded assignments is that in the initial state the guard should evaluate to true. The final state should represent the state that is the result of the multiple assignment. This gives a predicate φ defined as:

$$\varphi_{(b\,\&\,\bar{x}:=\bar{f})}(e) \stackrel{\text{def}}{=} base_R(e) = var(\bar{f}) \cup var(b) \wedge base_W(e) = \bar{x} \wedge b(e) \wedge \bar{x}'(e) = \bar{f}(e) \wedge \forall y \in base_R(e) - base_W(e)\,(y'(e) = y(e))$$

For abstract actions A we assume a base β is given, syntactically determined by the process it can be refined into. As we can only determine an upper bound for the base we assume the base of the event is a subset of β. Abstract events are left uninterpreted. Therefore, besides this base, no restrictions are placed on the functional behaviour of the actions, i.e. nothing is required for the initial and final states. This leads to a predicate φ_β:

$$\varphi_\beta(e) \stackrel{\text{def}}{=} base(e) \subseteq \beta$$

The semantics of predicates ϕ consists of all computations that consist of a single process action satisfying ϕ, with the base given by the *read* and *write* sets in ϕ. Let β be this base of ϕ. We can then define a predicate $\varphi_{\phi(\tilde{x},x)}$ characterizing these events as:

$$\varphi_{\phi(\tilde{x},x)}(e) \stackrel{\text{def}}{=} \phi(x(e),x'(e)) \wedge base(e) = \beta$$

This concludes the definition of our semantics.

6 Conclusion and future work

The introduction of refinement and operations such as conflict composition enable a compositional treatment of action systems, and allows for a smooth design process, from initial design to final implementation. We have provided several *algebraic* laws to characterize these new operations, and applied these laws in a transformation process, showing the feasibility of such a transformational approach. In the future we would like to supplement such algebraic laws by axioms and verification rules in the style of program logics like Hoare's logic. The approach taken in [ZR], where weakest preconditions are defined as adjoints of sequential composition operators, seems to be a good candidate for the axiomatization of the conflict composition operation. We are investigating a kind of "weakest precondition" [ZR] or "weakest prespecification" [HH] in the form of an adjoint to conflict composition.

The relation between our model and real-time is also a subject of current research. It appears that extending our model with an extra "temporal" ordering relation allows for an interesting notion of refinement in a real-time setting.

References

[Back] R.J.R. Back, A calculus of refinements for program derivations, *Acta Informatica 25*, 1988.

[BKP] H. Barringer, R. Kuiper and A. Pnueli, Now you may compose temporal logic specifications, *Proc. of the 16th ACM Symposium on Theory of Computing*, Washington, 1984, pp. 51-63.

[BHG] P.A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[CM] K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[DNM] P. Degano, R. De Nicola and U. Montanari, Partial orderings descriptions and observations of nondeterministic concurrent processes, *Proc. of the REX workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, 1988, LNCS 354, pp. 438-466.

[EF] T. Elrad and N. Francez, Decomposition of distributed programs into communication closed layers, *Science of Computer Programming 2*, 1982.

[Gaifman] H. Gaifman, Modeling Concurrency by Partial Orders and Nonlinear Transition Systems, *Proc. of the REX workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, 1988, LNCS 354, pp. 467-488.

[GHS] R.T. Gallager, P.A. Humblet and P.M. Spira, A distributed algorithm for minimum-weight spanning trees, *ACM TOPLAS 5-1*, 1983.

[GG] R. J. van Glabbeek and U. Goltz, *Equivalence Notions for Concurrent Systems and Refinement of Actions*, Arbeitspapiere der GMD, Number 366, GMD, 1989

[HH] C.A.R. Hoare and He Jifeng , The weakest prespecification, *IPL*, 1987.

[Hooman] J. Hooman, *Specification and Compositional Verification of Real-Time Systems*, Ph.D. Thesis, Eindhoven University of Technology, 1991.

[Jones] C.B. Jones, *Systematic software development using VDM*, Prentice–Hall, 1986.

[JPZ] W. Janssen, M. Poel and J. Zwiers, *Consistent alternatives of parallelism with conflicts*, Memorandum INF-91-15, University of Twente.

[KP] S. Katz and D. Peled, Interleaving set temporal logic, *Proc. of the 6th ACM Symposium on Principles of Distributed Computing*, Vancouver, 1987, pp. 178-190.

[Lamport] L. Lamport, The Hoare Logic of concurrent programs, *Acta Informatica 14*, 1980.

[MP] Z. Manna and A. Pnueli, Verification of concurrent programs: the temporal framework, In R.S. Boyer and J.S. Moore (eds), *The Correctness Problem in Computer Science*, Academic Press, 1981.

[NEL] M. Nielsen, U. Engberg and K.S. Larsen, Fully abstract models for a process language with refinement, *Proc. of the REX workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, 1988, LNCS 354, pp. 523-548.

[OG] S. Owicki and D. Gries, An axiomatic proof technique for parallel programs, *Acta Informatica 6*, 1976.

[Pomello] L. Pomello, Refinement of Concurrent Systems Based on Local State Transformations, *Proc. REX workshop on Stepwise Refinement of Distributed Systems*, 1989 LNCS 430, pp. 641-668.

[Pratt] V. Pratt, Modelling Concurrency with Partial orders, *International Journal of Parallel Programming 15*, 1986, pp. 33-71.

[SR] F.A. Stomp and W.P. de Roever, Designing distributed algorithms by means of formal sequentially phased reasoning, *Proc. of the 3rd International Workshop on Distributed Algorithms*, Nice, LNCS 392, Eds. J.-C. Bermond and M. Raynal, 1989, pp. 242-253.

[Winskel] G. Winskel, An introduction to event structures, *Proc. of the REX workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, 1988, LNCS 354, pp. 364-397.

[ZR] J. Zwiers and W.P. de Roever, Predicates are Predicate Transformers: a unified theory for concurrency, *Proc. of the conference on Principles of Distributed Computing*, 1989.