# Interface Groups for Analytic Execution Architectures

Jan A. Bergstra [a,b] Alban Ponse [a]

[a] *University of Amsterdam, Programming Research Group, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands*

[b] *Utrecht University, Department of Philosophy, Heidelberglaan 8, 3584 CS Utrecht, The Netherlands*

**Abstract**

Analytic execution architectures have been proposed as a means to conceptualize the cooperation between components such as programs, threads, states and services. Interface groups provide a technique to combine interface elements into interfaces with the flexibility to distinguish between permission and obligation and between promise and expectation which all come into play when component interfaces are specified.

The set of basic actions $A$ that underlies any program algebra or thread algebra generates the interface group for $A$ (in additive notation). The main principle that makes use of an interface group is that when composing a closed system of a collection of interacting components, the sum of their interfaces must be 0.

Interface groups provide a setting in which architectures, components and roles can be easily specified and discussed. As an example of this, we show that the program algebra PGA essentially needs an infinite number of primitive instructions to express all finite state threads.

*Key words:*
Interface, Interface group, Program, Thread, Service, Execution architecture.

## 1 Introduction

In [9] we proposed "analytic execution architectures" as a means to conceptualize the cooperation between key components such as programs, threads, states and services. Interfaces are a practical tool for the development of all but the most elementary architectural designs.

In this paper we propose interface groups as a technique to combine interface elements into interfaces with the flexibility to distinguish between permission and obligation and between promise and expectation, distinctions that all come into play when component interfaces are specified and analyzed.

As a vehicle to present and investigate interface groups we use the program algebra PGA as defined in [4] and thread algebra (abbreviated as TA, see e.g. [23]). A quick introduction to PGA and TA is given in Appendix A.

From the set of basic actions $A$ that underlies any program algebra or thread algebra a set $\mathit{ife}(A)$ of interface elements is derived. These generate the interface group for $A$ (in additive notation). The main principle that makes use of an interface group is that when composing a closed system of a collection of interacting components, the sum of their interfaces must be 0. The specification of various sorts of components is discussed in Section 3, and their combination into architectures in Section 4.

Interface groups provide a setting in which architectures, components and roles can be easily specified and discussed. As an example of this, we show in Section 5 that the program algebra PGA essentially needs an infinite number of primitive instructions to express all finite state threads.

The paper is ended with some discussion and concluding remarks in Section 6.

## 2 Interface elements and interface groups

In this section we introduce our basic technical ingredients: interface elements and interface groups. When working on the design of an analytic execution architecture, it is suggested that one is very precise about the interfaces of the components and that all interfaces are chosen as elements of an interface group.

### *2.1 Interface elements*

Given a finite collection $A$ of basic actions the following set of *interface elements* is introduced:

$$\mathit{ife}(A) = \{a/TF, a/T, a/F, a/* \mid a \in A\},$$

where $T$ stands for true and $F$ for false. The intended meaning of these elements and their complements (prefixed with $-$) is as follows:

$a/TF$ indicates the permission (option) of a component to issue request $a$ while expecting a reply in the set $B = \{T, F\}$.

$a/T$ indicates the permission (option) of a component to issue request $a$ while expecting a reply $T$.

$a/F$ indicates the permission (option) of a component to issue request $a$ while expecting a reply $F$.

$a/*$ indicates the permission (option) of a component to issue request $a$ while expecting no reply at all.

$-a/TF$ indicates the obligation (requirement) of a component to accept request $a$ while promising a reply in the set $B = \{T, F\}$.

$-a/T$ indicates the obligation (requirement) of a component to accept request $a$ while promising a reply $T$.

$-a/F$ indicates the obligation (requirement) of a component to accept request $a$ while promising a reply $F$.

$-a/*$ indicates the obligation (requirement) of a component to accept request $a$ while promising to produce no reply at all.

## 2.2 Interface groups

The *interface group* $\mathit{ifg}(A)$ is the commutative group with additive notation, freely generated from the set $\mathit{ife}(A)$ as generators.

We distinguish the following subsets of $\mathit{ife}(A)$ and their generated subgroups:

$\mathit{ife}_{ts}(A)$, the subset of $\mathit{ife}(A)$ that leaves out elements of the form $a/*$. The set $\mathit{ife}_{ts}(A)$ contains the interface elements for threads and services. Its subgroup $\mathit{ifg}_{ts}(A)$ is generated by $\mathit{ife}_{ts}(A)$.

$\mathit{ife}_B(A)$ $(B = \{T, F\})$. The set $\mathit{ife}_B(A)$ contains only the interface elements of the form $a/TF$. The subgroup $\mathit{ifg}_B(A)$ of $\mathit{ifg}_{ts}(A)$ is generated by $\mathit{ife}_B(A)$.

$\mathit{ife}_{B/T}(A)$, which excludes the interface elements of the form $a/F$, and its generated subgroup $\mathit{ifg}_{B/T}(A)$.

$\mathit{ife}_{B/F}(A)$, which excludes the interface elements of the form $a/T$, and its generated subgroup $\mathit{ifg}_{B/T}(A)$.

$\mathit{ife}_p(A)$, the set of process interface elements consists of the elements of the form $a/*$ and these generate the interface group $\mathit{ifg}_p(A)$ which can be used for the description of architectures involving components that are specified in a symmetric process algebra (rather than a polarized process algebra such as TA).

When working on the design of an analytic execution architecture it is now suggested that one aims at being precise about component interfaces and that one of the groups above is chosen and all interfaces are taken as elements of that group.

The intended meaning of interfaces derives from the intended meaning of positive and negative elements as described above. In a composed interface $I + J$ it is implied that the combination of two permissions is a permission to choose between them (i.e., the ability to do both) and the combination of two obligations is the obligation to accept both requests. Complementary elements are cancelled because the interface group mechanism incorporates the assumption that when composing two components, complementary interface elements are used for internal interaction as much as possible.

Note that an element such as $a/T{+}a/F$ just represents that $a$ can be requested in two ways, one that expects the single reply $T$ and one that expects the single reply $F$.

The partial ordering $<$ on interfaces in $\mathit{ifg}(A)$ is generated by these rules:

- $0 < p$ for all interface elements $p \in \mathit{ife}(A)$,
- $a/TF < a/T$,
- $a/TF < a/F$,
- $X < Y$ if and only if $X + Z < Y + Z$.

It is not assumed that $a/T < a/*$ because $a/*$ expresses the expectation that no reply will be returned (and needs to be received) which cannot be understood as a refinement of the expectation that $T$ will be returned, whereas the latter expectation indeed refines the expectation that either $T$ or $F$ will returned after completion of processing the request $a$ by another component. As ususal, we write $X \leq Y$ whenever $X < Y$ or $X = Y$, and common identities like $X \leq X + Y$ if and only if $0 \leq Y$ follow easily.

**Note 1** *We could incorporate interface elements of the form $a/TF*$, describing maximal freedom in the expectation of an answer, and of the form $a/T*$ and $a/F*$. Then, $\leq$ can be characterized by $a/v \leq a/w$ if and only if $V \supseteq W$ where $v$ $(w)$ is interpreted as a set $V$ $(W) \subseteq \{T, F, *\}$. Similarly, this holds for $<$, replacing $\supseteq$ by $\supsetneq$.*

Interfaces as modeled by interface groups have less structure than the signatures used as interfaces in the module algebra of [3]. This can be remedied by providing more structure to the basic action set (see Section 4.2). Module algebra, however fails to provide any concept of complementarity and for that reason it has a bias in the direction of the combination of services (rather than clients). Module algebra and similar approaches fail to provide the basic technical ingredients needed for the description of analytic execution architectures which are meant to combine various components such as clients and services in asymmetric ways.

## 3 Components, interfaces and embodiments

In this section we distinguish various types of components and classify these by the nature of their interfaces.

### *3.1 Components and compliance*

Several terms are used to indicate the working of components in a system. In previous work we used programs, program objects, instruction sequences and polarized processes (in [4,1]), and threads, services and multi-threads (e.g. in [7]). State machines occur in [8] and risk assessment services (architecture sensitive services) are used in [2]. In [6] a special form of state machines (state machine services) is used under the name of 'Maurer computers'.

What these terms have in common is that they make reference to descriptions of the functionality (behavior, inner structure, underlying mechanism) of parts of conceivable systems. These parts are either named by their role (thread, client or service) or by their mathematical identity (process, program object, polarized process).

It is tempting to view these references as references to actual, potential, designed or contemplated system components but we will propose not to do so. Instead we will propose to view a component as a pair

$$[I, E]$$

of an interface $I$ and an embodiment $E$. Threads, programs, services and so on are typical embodiments while the elements of the aforementioned interface groups may act as interfaces.

Given a component $[I, E]$ it is plausible to require in addition that the embodiment $E$ is compliant (in some sense to be specified depending on the kind of embodiment and the interface group used) with the interface $I$. As an alternative to 'embodiment' the term 'precomponent' will also be used. We will assume that a notion of compliance is available but it will not be assumed that compliance is a precondition for the definition of a component. The definition of compliance takes the following form: an operator $i(\_)$ which determines for a precomponent $E$ its derived interface, i.e., the coarsest interface in which it exists. So, for $E$ it is then required that $i(E) \leq I$. For example, for a PGA program $p$ containing (test) actions $a_1, ..., a_k$ we define

$$i(p) = \sum_{i=1}^{k} a_i/TF.$$

Note that in case $I$ contains negative interface elements, $i(E) \leq I$ implies that these majorize their counterparts in $i(E)$, e.g., if $-a/TF$ occurs in $I$, then $i(E)$ contains $-a/T$, $-a/F$ or $-a/TF$. Components with non-interface compliant embodiments are conceivable as well. In that case an implicit restriction to the embodiment such that it then fits the interface should be made. We will not discuss this case further in this paper.

Because embodiments invariably have their origin in one of a number of classes of system specifications it is also reasonable to classify components according to the same classification. The next two sections describe some simple examples.

### *3.2 Program and thread components*

A component can be called *active* whenever its compliant interface is composed of positive interface elements only. A *program component* is an active component embodied by a program. Some examples:

(1) $[a/TF + b/TF, +a; b; !]$ may be called a program component (i.e. a component embodied by a program).
(2) Similarly $[a/TF+b/F, +a; b; !]$ is a program component, though one which requires slightly more about its context (namely that $b$ always gives rise to reply $F$).

The interface $a/TF + b/TF + c/*$ is not a natural candidate for the program $c; +a; \#4; -b; !; \#0$ because upon execution, the action $c$ is seen as a request, although the actual reply does not control further execution.

Program components may claim more permissions than they will use, as e.g.,

$$[a/TF + b/TF + c/T + d/F, +a; -b; \#1; !].$$

This may be useful if design rules for closed system architectures are to be met. We return to this subject in Section 4.

A *thread component* is an active component embodied by a thread. Some examples:

(1) $[a/TF + b/TF + c/T + d/F, |{+a}; -b; \#2; -c; d; !|]$ is a thread component because the embodiment is presented in the form of a thread extraction $|{+a}; -b; \#2; -c; d; !|$ from a PGA program.
(2) $[a/TF+b/TF+c/T+d/F, ((\mathsf{S} \trianglelefteq c \trianglerighteq d \circ \mathsf{S}) \trianglelefteq b \trianglerighteq d \circ \mathsf{S}) \trianglelefteq a \trianglerighteq (\mathsf{S} \trianglelefteq c \trianglerighteq d \circ \mathsf{S})]$ is the same component in thread algebra notation.

**Note 2** *A reason for not requiring that $i(E) \leq I$ for a component $[I, E]$ may* *be that it is also reasonable to assume that by default the behavior of the body is restricted (encapsulated) by the capabilities listed in the interface. For instance,* $[a/TF + c/TF, +a; b; !]$ *may also be called a program component but now it is necessary that a semantic account is given of what a request* $b$ *amounts to, given that it is not permitted by the interface. A reasonable default is that the system deadlocks in that case. Dropping the requirement* $i(E) \leq I$ *can be called* interface restriction.

## *3.3 Service components and reactor components*

A component whose interface consists solely of negative interface elements is called a *service*. Upon requests, a service serves the operation of one or more active components. Some examples of common services:

(1) A boolean register service component

$$[-\texttt{set:true}/T - \texttt{set:false}/T - \texttt{eq:true}/TF - \texttt{eq:false}, H_{bool}]$$

(where $H_{bool}$ is the boolean register) works as follows: the request for a `set:true` or a `set:false` action always succeeds and sets the internal boolean value. The actions `eq:true` and `eq:false` test the current value. In this case it is necessary to prescribe some particular initialization of $H_{bool}$ (e.g., it has initial value `false`, or all $\texttt{eq:}b$ requests reply $F$ until the first $\texttt{set:}b$ action has been executed).

(2) Given data elements $1, ..., n$, the stack service component over these data elements is $[I, S]$, where

$$I = -\texttt{empty}/TF - \texttt{pop}/TF - (\Sigma_{i=1}^{n} \texttt{push:}i/T + \texttt{topeq:}i/TF)$$

and $S$ works as follows: initially $S$ is empty and the action `empty` replies $T$ in this state, while `pop` replies $F$; both these actions leave the stack empty. A $\texttt{push:}i$ action pushes value $i$ onto the stack and always replies $T$. The test actions $\texttt{topeq:}i$ test whether value $i$ is on top of the stack, and `pop` pops the non-empty stack with reply $T$.

In [8] a few more services (called state machines in that paper) and their possible use are explained. In [9] we consider services such as the Turing machine tape and rational agents, as well as the notion of a *reactor*, a service that may also be active with respect to other components or 'the environment'. Typically, the interface of a reactor is a composition of both positive and negative interface elements. Furthermore, we assume that such an interface is always given in normal form:

**Definition 1** *An interface is in* normal form *if all positive and negative interface elements are cancelled out.*

If the body of a reactor component is simply known as $X$, an interface may already be given, e.g.,

$$[-a1/TF - a2/TF - b/T - c/T + d/TF + e1/T + e2/T, X].$$

## 4 Combining components and describing architectures

In this section we discuss how components can be combined on the abstract level introduced thus far. We propose to describe analytic architectures using actions that have more structure (the focus.method notation introduced in [4]).

### *4.1 Combining components*

We start with an example: assume

$$I = a/TF + b/TF + c/T$$

is the interface of the program $p = c; +a; \#4; -b; !; \#0$, whereas

$$J = -a/TF - b/T - c/T$$

is the interface of a service $H$. When $p$ and $H$ are put together as a closed system (all requests issued by $p$ are meant for $H$ and all requests accepted by $H$ are issued by $p$) then the sum of both interfaces should vanish (equal 0). But that is not the case because

$$\begin{aligned} I + J &= (a/TF + b/TF + c/T) + (-a/TF - b/T - c/T) \\ &= b/TF - b/T, \end{aligned}$$

which differs from 0 (in fact, $b/TF - b/T < 0$). This is indicative of an architectural mismatch. This mismatch may for instance be resolved by replacing $J$ by $J + b/T - b/TF$ which weakens its promise to reply positively at request $b$ by the promise to reply either positive or negative.

Using components one may provide information about an architecture which has been formalized only in part. For instance a power control unit component $PCU$ (taking the name from the role of its body) may be postulated with interface

$$on/TF + off/* - quit/*,$$

representing that another part of the system can be switched 'on' while leading to a positive reply at success and a negative reply if the component at stake is already active and that it can be switched off (upon which no reply is generated) and that the system can itself decide to quit which is accepted as a message while no reaction is foreseen. Probably this is not entirely satisfactory because the option to have no reply in the case of a defect might be taken into account as well. To do so may require the addition of new interface elements, e.g. $a/TF*$ explicitly indicating the possibility of no-reply (cf. Note 1). The method of interface groups is sufficiently flexible to make these adaptations, though we will stick below to the given set of groups. The power control unit component can be presented as $[on/TF + \mathit{off}/* - quit/*, PCU]$ with the behavior $PCU$ yet unspecified.

The power control unit component $PCU$ can be combined with a program component which has been adapted to take the power control commands into account as e.g. in

$$[-on/TF - \mathit{off}/* + quit/* + a/TF + b/T, +a; b; !].$$

In the description of this component an informal explanation takes care of explaining the role of the $PCU$ interaction commands, whereas the behavior along the other part of the interface is given using thread extraction. Unavoidably this leads to subtleties that have to be resolved in a later stage, e.g., concerning the precise effects of the *off* request on various parts of a system in operation.

With $-a/TF - a/TF (= -2 \cdot a/TF)$ it is expressed that a component can accept (and reply to) request $a$ in two different ways (e.g. from two different other components), similarly $2 \cdot a/TF = a/TF + a/TF$ expresses the option for a component to issue request $a$ in two different ways (and to expect the corresponding replies so that they can be distinguished). If a service with body $H$ is shared by two other active components its interface requires duplication as in $[-2.a/TF - 2.b/TF - 2.c/T, H]$, if it is active as well this may lead to an active service component (a reactor), e.g.,

$$[-2.a/TF - 2.b/TF - 2.c/T + d/TF + e1/T + e2/T, H]$$

which combines single and double multiplicities.

## *4.2 Architecture descriptions and focus prefixing*

An architectural description provides a list of components (some of which will have token bodies because no behavior is known together with a directed graph structure on the components, where the polarized interaction arrow

$$C_1 \longrightarrow C_2$$

indicates that requests issued by $C_1$ may be accepted by $C_2$. If also $C_1 \longrightarrow C_3$, i.e.,

$$\begin{tikzcd} & C_1 \arrow[dl] \arrow[dr] & \\ C_2 & & C_3 \end{tikzcd}$$

a choice is made between both components preferably on the basis of the available interface information. It is a design rule that architectural descriptions resolve that form of ambiguity. If also $C_4 \longrightarrow C_2$, thus

$$\begin{tikzcd} C_4 \arrow[dr] & & C_1 \arrow[dl] \arrow[dr] & \\ & C_2 & & C_3 \end{tikzcd}$$

this implies that $C_2$ receives requests in two different ways. If both $C_1$ and $C_4$ may issue identical requests that will be reflected in the duplication of interface elements in $C_2$.

If a component is imported in an architecture as available under a *focus*, this will be indicated by a labeled polarized interaction arrow:

$$C_1 \xrightarrow{f} C_2$$

indicates that only requests carrying the prefix $f$. are meant to be processed by $C_2$. It is equivalent to write $C_1 \longrightarrow f.C_2$ where the focus prefixing operator $f.C$ prefixes each negative interface element of $C$ with an $f$ and a dot. (This is meaningful only on normal form interfaces where positive and negative elements have already been fully cancelled out). Below we shall use this so-called *focus.method* notation, which was introduced in [4].

If nothing is known about an external environment it may be simply represented by a token name *EEB* for external environment behavior: in the case of a program component $[a/TF + b/TF + c/T, +a; -b; c; !]$ this leads to a component

$$[-a/TF - b/TF - c/T, \mathit{EEB}].$$

In [9] we have proposed so-called *analytic execution architectures* as a model for the cooperation between a program and other system components. In particular a program may use a number of services and thereby produce a thread along its external interface. For example program component $[I, p]$ may be composed with service component $[J, H]$ available under focus $f$ and with service component $[K, H']$ available under focus $g$ in order to interact with an external environment behavior component $[I_e, \mathit{EEB}]$ that is placed in the analytic architecture under focus *ext*, i.e.,

$$\begin{tikzcd}
 & {[I,p]} \arrow[r, "ext"] \arrow[dl, "f"'] \arrow[dr, "g"] & {[I_e, EEB]} \\
{[J,H]} & & {[K,H']}
\end{tikzcd} \tag{1}$$

Under the assumption that this analytic execution architecture constitutes a closed system we have

$$I + ext.I_e + f.J + g.K = 0$$

which implies $ext.I_e = -(I + f.J + g.K)$.

This is a general model for explaining the operation of a program $p$ (or its extracted thread $|p|$) in the presence of auxiliary services with the objective to realize intended behavior along the external interface $I_e$. In [7] the external interface (or more precisely: $-ext.I_{ext}$) has been named the target interface (of $P$) and the interface $-(f.J+g.K)$ is referred to as $P$'s co-target interface, thus emphasizing the auxiliary nature of the components available under the foci $f$ and $g$. It is a definite advantage of the use of the interface group notation that explicit notations for these interfaces can be given so that these unusual mnemonic descriptions become additional rather than essential.

## 5 Analytical execution architectures

In this section we discuss the expressiveness of a few programming notations; a topic that arises in a context where architectures, components and roles can be easily specified and discussed. Technically it can be discussed without the use of an interface group but the explicit use of interfaces is definitely helpful to set the scene for the question that will be addressed.

### *5.1 Formal instruction sequences*

The program algebra PGA of [4] provides a practical notation for $P$ by writing $P = |p|$ with $p$ an instruction sequence expressed as a closed term in PGA. The term $p$ denotes a so-called program object. This is a finite or infinite (and repeating) instruction sequence over the collection of primitive instructions of PGA. Given a set $A$ of basic actions, PGA's primitive instruction set is 'universal' in the sense that it admits writing notations for all finite state threads over $A$. Now consider the analytic execution architecture depicted in (1) above. Whenever a finite state discretely timed mechanism can produce a thread along the external interface with the help of both auxiliary services and the service $H$ is finite state, this can also be achieved by an instruction sequence expressed in PGA that does not use the service component $[J, H]$. A key observation at this point is that PGA's primitive instruction set is infinite because it contains an infinite number of different jump instructions $\#k$ for $k \in \mathbb{N}$.

This raises the question whether or not a finite instruction set could have been used instead. Taking for granted that if a universal finite instruction set can be found it can also be reduced to two elements, there is now the following problem:

**Problem 2** *How to formalize what it means that finite or infinite boolean sequences can be used as a program notation instead?*

As an answer to this problem we take the following:

**Requirements 3** *To establish that boolean sequences can be used as a universal program notation (for finite state threads, using finite state means of execution) we fix the external interface $I_e$ as above in* (1) *as follows:*

$$-ext.I_e = ext.b/TF + ext.c/T + ext.d/T,$$

*so that we can express all finite state threads with interface $b/TF + c/T + d/T$. Further we encode the boolean sequence, say $\alpha$, in the service component $[K, H']$ as follows: let $K = -eof/TF - gnv/TF$ and $H' = H_\alpha$ produces $T$ on all requests after eof has been positively answered once, while until that stage eof leads to a negative reply without state change and gnv ('getnextvalue') dequeues the first boolean (or bit) from the sequence $\alpha$ while returning its value as a reply. We will refer to $\alpha$ in this context as a* formal instruction sequence*.*

*Then it suffices to find a single PGA program $p_{int}$ (interpreter program) and a finite state service component $[J, H]$ such that for each thread $t$ with interface $-ext.I_e$ there exists a finite or infinite boolean sequence $\alpha$ such that the open (= non-closed) analytic execution architecture made up from*

*(1) the program component $[I, p_{int}]$,*
*(2) the finite state auxiliary service component $[J, H]$ in focus $f$ with $[I, p_{int}] \xrightarrow{f} [J, H]$, and*
*(3) the formal instruction sequence provider component $[K, H_\alpha]$ in focus $g$ with $[I, p_{int}] \xrightarrow{g} [K, H_\alpha]$*

*computes the thread $t$ or more precisely, the thread component $[-ext.I_e, t]$.* (End Requirements 3.)

With these requirements, we have the following result:

**Theorem 4** *There exists no PGA program $p_{int}$ that meets Requirements 3.*

**PROOF.** Towards a contradiction, suppose that $p = p_{int}$ is an interpreter program from which all finite state threads over the given interface can be developed by feeding it with an appropriate $\alpha$ encoded as $H_\alpha$. As a first step the auxiliary service component $[J, H]$ is removed because it can be incorporated in the PGA program $p$ according to [8]. This of course transforms $p$, but in a uniform manner. We may then assume that $p$ has the form

$$p = (u_0; ...; u_{n-1})^\omega$$

with no jump counter exceeding $n$. [1]

Consider for $i \leq n$ the threads $T_i = T_{i+1} \unlhd b \unrhd c^i \circ d^\infty$ with $T_{n+1} = \mathsf{S}$. Let $\alpha$ be a bit sequence such that it generates $T_0$ in our architecture.

Now $n + 1$ different cases can be distinguished according to the number of times $b$ returns $T$ before returning $F$. Suppose that at its $j_{th}$ turn $b$ returns $F$, while processing instruction number $x_j.n + y_j$ (with $y_j < n$) having consumed the first $z_j$ values from $\alpha$ (including its repeating tail if it has already reported $T$ on request $g.eof$). Writing $\alpha_j$ for the remaining part of $\alpha$, we then find that the simplified architecture (without $[J, H]$) equipped with program component

$$q = u_{y_j+1}; ...; u_{n-1}; (u_0; ...; u_{n-1})^\omega$$

and formal instruction sequence provision component $[K, H_{\alpha_j}]$ computes $c^j \circ d^\infty$. Such a triple $x_j, y_j, z_j$ can be found for $n + 1$ different values of $j$. The $x_j$ turn out to be irrelevant but the $z_j$ are essential. Let $z_{j_0}$ be the maximum of the $z_j$. Then $\alpha_{j_0}$ is a tail of each of the other $\alpha_j$. Now one follows for each other value of $j$ the corresponding computation until the precise moment that it has consumed $z_{j_0}$ boolean values from $\alpha$, thus leaving it in state $\alpha_{j_0}$. This indicates that the computations for each of the triples pass a stage where the remaining tails are identical. Because they compute different threads there must be another difference and that is of course the instruction number $y_j + 1$ in the current cycle of $p$. But there are only $n$ different possibilities for $n + 1$ different threads. That is impossible and this contradiction concludes the proof. $\Box$

It has been proved in [4] that limiting PGA to bounded jump counters reduces its expressive power, but the result shown here is considerably stronger. It

[1] This can be seen in many ways, for instance following the *projection semantics* of [4]: for each PGA program $p$, `pglb2pga`(`pga2pglb`($p$)) has this form.

implies the result from [4] assuming that the above proof works for formal instruction sequences over a finite instruction set just as well.

It is obvious how to adapt the proof that there is no universal interpreter program for formal instruction bit-sequences to arbitrary finite instruction sets (interfaces). We may conclude that it is reasonable in hindsight that PGA has an infinite set of primitive instructions because no other design would allow one to reduce this to a finite number.

## *5.2 Structured Programming*

The program notation PGLS as defined in [4] represents the class of while-programs and qualifies as a structured program notation. It adds to PGA a finite number of instructions for conditional constructs and while loops. Because PGLS (of course) does away with jumps its programs can be represented as finite or infinite streams of instructions over a finite instruction alphabet. Although it was shown in [4] that PGLS has weaker expressive power than PGA it can be shown that a similar result as Theorem 4 holds for PGLS: no interpreter program can compute all PGLS expressible threads from a formal instruction sequence encoded as a bit stream in a formal instruction sequence provider service.

Thus we conclude two major facts:

(1) structured programming primitives allow the representation of a large number of threads using instruction streams over a finite instruction set, but:
(2) executing these programs cannot be done by finite state means (and no modification of the instruction set can change this).

As it turns out PGLS programs can be interpreted by a PGA program if that is allowed to make use of an auxiliary service that provides a boolean stack. This strategy of implementation uses an interpreter for PGLS. The alternative is to transform PGLS programs to PGA, which is the strategy that uses a compiler/assembler. But now there is a crucial disadvantage: either one has to accept that arbitrary jump counters come into play or one accepts that only a part of PGLS (or any other program notation with comparable expressive power) is correctly translated (which is the practical situation). Because both disadvantages are quite unsatisfactory and programs used in industry grow in size all the time, it is understandable that a way out of this dilemma is needed. This is obtained by taking a program notation of at least the expressive power of PGLS as the endpoint of compilation at large (e.g. in the case of CIL [13]), while limiting the role of compilation to executable form to so-called JIT (just in time) transformations that are applied to fragments of the code only, thus limiting the risk that jump counters grow too large.

## 6 Discussion and concluding remarks

The term interface group has been discussed in [22] and occurs widely in the literature about internet protocols; it was used by Keith Cheverst et. al. in the context of groupware description [14]. These uses of the phrase make no reference to the mathematical theory of groups. For that reason we consider it justified to propose the meaning assigned to 'interface group' in this paper for use in a theoretical context.

Both PSF (Process Specification Formalism [21]) and muCRL (micro Common Representation Language [17]) are tool-supported, formal description techniques developed for the ACP-style specification of concurrent systems. In PSF, muCRL process interfaces are declared but not used, in TCSP [11] the parallel composition operator depends on interface information requiring synchronization over shared actions. One may require that precomponents are computable in some sense derived from the concepts explained in e.g. [24] or more recently [19]. However, that requirement is not strictly needed because an embodiment need not be of a mechanistic nature, it may also be a functional specification that defeats automatic execution, interpretation or simulation.

Explicit work on interfaces and components can be found in several lines of research. In [16] a treatment of ACP-style process algebra is given which uses a subsystem PAP of ACP that further decouples the axioms from interface information than most previous presentations have done. We mention Java [10] where interfaces are a separate category but where the independent existence of bodies that may or may not be combined with an interface to form a software component is absent. A significant theory of interfaces and components is given by Scheben in [26]. Issued requests are referred to as 'required services', whereas accepted requests are referred to as 'provided services'. Scheben also designs a general notation for the description of component interfaces. In [27] interfaces are cast in terms of 'interface automata'. What is called a reply service in [2] and a state machine service in [8] are special cases of interface automata. In contrast to our proposals this line of work captures behavioral and temporal information as a part of interface descriptions. This we have deliberately rejected in order to have more orthogonality between interfaces and embodiments, forcing us, however, to accept mere behavioral specifications as embodiments, a decision that may of course be disputed.

A convincing example of interfaces are the so-called instruction set architectures for microprocessors, which can be given a theoretical basis by means of the theory of Maurer in [20]. Maurer instruction set architectures, as proposed in [6] combine an interface description with operational information. The importance of explicit interface descriptions also emerges from [18] where essentially a conventional RISC processor interface is extended with four additional instructions so that it can deal with a multi-thread rather than with a single thread. These multi-threads are executed by multiplexing on a single pipeline or by distribution over a number of pipelines. A part of the motivation for the work on interface groups stems from a systematic attempt to develop a theoretical account of the proposals made in [18] that we have set as a long term goal.

The constraint that a boolean is given when a request $a$ is accepted $(-a/TF)$ has been termed a 'promise'. Promises is a topic that Mark Burgess has been developing for the description of services in networks of autonomous components (see [12]). Interface groups might be helpful in the formalization of promises as well. We also expect interface groups to be useful for the description and further development of so-called sourcing architectures, for which a rudimentary diagrammatic display method has been developed and used by Rijsenbrij and Delen in [25] and subsequently in [15]. Their theory of atomic outsourceable units depends on the proper use of interfaces. It is in this area that we intend to work on practical applications of interface groups: starting from a flexible theory of interfaces, it might be possible to develop the equivalent of design patterns in the context of sourcing architectures.

## References

[1] J.A. Bergstra and I. Bethke. Polarized process algebra and program equivalence. In J.C.M. Baeten, J.K. Lenstra, J. Parrow, and G.J. Woeginger (editors), *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4*, LNCS 2719, pages 1-21, Springer-Verlag, 2003.

[2] J.A. Bergstra, I. Bethke and A. Ponse. Thread algebra and risk assessment services. To appear in the proceedings of the *Logic Colloquium 2005 (Athens)*, 2006.

[3] J.A. Bergstra, J. Heering and P. Klint. Module Algebra. *Journal of the ACM*, 37(2):335-372, 1990.

[4] J.A. Bergstra and M.E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125-156, 2002.

[5] J.A. Bergstra and C.A. Middelburg, Thread algebra for strategic interleaving. *Technical University Eindhoven report TU/e 04/35, To appear.*

[6] J.A. Bergstra and C.A. Middelburg, Maurer computers for pipelined instruction processing. *Technical University Eindhoven report TU/e 06/12*, 2006.

[7] J.A. Bergstra and C.A. Middelburg. A thread algebra with multi-level strategic interleaving. In S.B. Cooper, B. Loewe and L. Torenvliet (editors), CiE 2005, LNCS 3526, pages 35-48, Springer-Verlag, 2005.

[8] J.A. Bergstra and A. Ponse. Combining programs and state machines. *Journal of Logic and Algebraic Programming*, 51(2):175-192, 2002.

[9] J.A. Bergstra and A. Ponse. Execution architectures for program algebra. *Journal of Applied Logic*, to appear.

[10] G. Bracha, et al. *The Java Language Specification* (2nd edition). Addison Wesley, 2000. (First edition by James Gosling, Bill Joy, Guy Steele, 1996.)

[11] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(8):560–599, 1984.

[12] M. Burgess. An approach to understanding policy based on autonomy and voluntary cooperation. *16th IFIP/IEEE Distributed Systems Operations and Management (DSOM 2005)*, LNCS 3775, pages 97-108, Springer-Verlag, 2005.

[13] C# and Common Language Infrastructure Standards, Microsoft .NET Framework Developer Center, `http://msdn.microsoft.com/netframework/ecma/`, ©Microsoft Corporation, 2006.

[14] K. Cheverst, G. Blair, N. Davies and A. Friday. The support of mobile-awareness in collaborative groupware. *Personal Technologies*, 3(1-2):33-42, 1999.

[15] G.P.A.J. Delen. *Decision- en controlfactoren voor sourcing van IT (in Dutch)*. Ph.D. Thesis, University of Amsterdam (van Haren publishing Zaltbommel), 2005.

[16] W.J. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science. Springer-Verlag, 2000.

[17] J.F. Groote and A. Ponse. The syntax and semantics of muCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen (editors), *Algebra of Communicating Processes*, Utrecht 1994. Workshops in Computing, pages 26-62, Springer-Verlag, 1995. (See also `http://homepages.cwi.nl/~mcrl/`).

[18] C.R. Jesshope and B. Luo. Micro-threading, a new approach to future RISC. *In ADAC 2000, IEEE Computer Society Press*, 34-41, 2000.

[19] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation* (2nd edition). Prentice-Hall, 1997.

[20] W.D. Maurer. A theory of computer instructions. *Journal of the ACM*, 13(2):226-235, 1966.

[21] S. Mauw, G.J. Veltink. A Process Specification Formalism, *Fundamenta Informaticae* XIII:85-139, 1990. (See also `http://www.science.uva.nl/~psf/`).

[22] M. Olsen, E. Oskiewics and J. Warne. A model for interface groups. *JProc. 10th IEEE Symp. on Reliable Distributed Systems*, 98-107, (Pisa Italy) 1991.

[23] A. Ponse and M.B. van der Zwaag. An introduction to program and thread algebra. In A. Beckmann et al. (editors), CiE 2006, LNCS 3988, pages 445-458, Springer-Verlag, 2006.

[24] H. Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill Book Co., 1967. Reprinted, MIT Press, 1987.

[25] D.B.B. Rijsenbrij and G.P.A.J. Delen. Enterprise-architectuur is een noodzakelijke voorwaarde voor verantwoorde outsourcing (in Dutch). *in: IT service management best practices (red. J. van Bon), van Haren Publishing Zaltbommel*, 35-58, 2004

[26] U. Scheben. Hierarchical composition of industrial components. *Science of Computer Programming*, 56, 117-139, 2005.

[27] P. Völgyesi, M. Maróti, S. Dóra, E. Osses, and Á. Lédeczi. Software composition and verification for sensor networks. *Science of Computer Programming*, 56, 191-210, 2005.

## A Program Algebra and Thread Algebra

Program algebra (PGA, for ProGram Algebra) provides a rigid framework for the understanding of imperative sequential programming. Starting point is the perception of a program object as a possibly infinite sequence of instructions. PGA programs are composed from instructions (explained below) and two operators: concatenation $X; Y$ and repetition. Concatenation is an associative operator and usually we do not write brackets in repeated concatenations.

Execution of a program object is single-pass: the instructions are visited in order and are dropped after having been executed. Execution of a *basic* or *test instruction* is interpreted as a request to the execution environment: the environment processes the request and replies with a boolean value true ($T$) or false (notation $F$).

We start with some typical examples. Given a set $A$ of basic actions $a, b, c, ...$, the PGA program $+a; b; !$ prescribes the following behavior: first the *positive test instruction* $+a$ is executed, leading to action $a$; then upon reply $T$, the *basic instruction* $b$ is executed (action $b$) and termination follows (by the *termination instruction* !); upon reply $F$, $b$ is skipped and termination follows immediately. A *negative* test instruction $-a$ has the reverse effect: upon reply $T$ the next instruction is skipped. Furthermore, PGA contains a *jump instruction* $\#k$ for each $k \in \mathbb{N}$. For example, the PGA program

$$-a; \#3; b; !$$

executes $a$ and upon reply $T$ continues with $b$ and then terminates; if $a$ yielded reply $F$, execution continues with #3 (so jumps three instructions ahead), which results into deadlock because there there are not that many instructions left. The instruction #0 (jump 0 instructions ahead) always results into deadlock. A *primitive instruction* is just one of the above instructions, i.e. a basic, test, termination or jump instruction.

In PGA, the notation for repetition is $(\_)^\omega$. Repetition typically satisfies the identity $X^\omega = X; X^\omega$ (unfolding). For example, the PGA program $(a; b)^\omega$ executes $a$ followed by $b$ and repeats these two actions, as does $a; (b; a)^\omega$, while the program $(a; !; b; c)^\omega$ terminates after $a$ has been executed.

Execution of PGA programs is modeled as threads, i.e., as elements of Thread Algebra (TA). The primary operation of TA is *postconditional composition* $\_ \trianglelefteq a \trianglerighteq \_$ for each action $a$:

$$P \trianglelefteq a \trianglerighteq Q$$

stands for the execution of action $a$, followed by execution of $P$ if $T$ is returned and by $Q$ if $F$ is returned. In case $P = Q$, we use *action prefix* $a \circ P$ as a shorthand for this behavior, thus $a \circ P = P \trianglelefteq a \trianglerighteq P$.

Finite threads are built up from the constants $\mathsf{D}$ (deadlock) and $\mathsf{S}$ (termination) using postconditional composition. Thread extraction of a PGA program $p$, notation $|p|$, is defined by the following thirteen equations, where $a$ ranges over the basic instructions, and $u$ over the primitive instructions ($k \in \mathbb{N}$):

$$\begin{array}{llll}
(1) & |!| = \mathsf{S}, & (2) & |!; X| = \mathsf{S}, \\
(3) & |a| = a \circ \mathsf{D}, & (4) & |a; X| = a \circ |X|, \\
(5) & |{+a}| = a \circ \mathsf{D}, & (6) & |{+a}; X| = |X| \trianglelefteq a \trianglerighteq |\#2; X|, \\
(7) & |{-a}| = a \circ \mathsf{D}, & (8) & |{-a}; X| = |\#2; X| \trianglelefteq a \trianglerighteq |X|, \\
(9) & |\#k| = \mathsf{D}, & (10) & |\#0; X| = \mathsf{D}, \\
 & & (11) & |\#1; X| = |X|, \\
 & & (12) & |\#k{+}2; u| = \mathsf{D}, \\
 & & (13) & |\#k{+}2; u; X| = |\#k{+}1; X|.
\end{array}$$

For PGA programs, these equations yield either finite threads, or in the case that a non-empty loop occurs, finite state threads which can be captured by a system of recursive equations. E.g.,

$$\begin{aligned}
|(a;+b)^\omega| &= |a;+b;(a;+b)^\omega| \\
&= a \circ P, \text{ with} \\
P &= |+b;(a;+b)^\omega| \\
&= |(a;+b)^\omega| \trianglelefteq b \trianglerighteq P.
\end{aligned}$$

In the particular case that these equations applied from left to right generate a loop without any action, as for instance in

$$|(\#2;+a)^\omega| = |\#2;+a;(\#2;+a)^\omega| = |\#1;(\#2;+a)^\omega| = |(\#2;+a)^\omega|,$$

the extracted behavior is defined as $\mathsf{D}$. As an example, we find

$$|(+a;\#2)^\omega| = \mathsf{D} \trianglelefteq a \trianglerighteq |(+a;\#2)^\omega|.$$

Finite state threads can be characterized in various ways. For more information, see e.g. [4] (where threads are called *polarized processes*) or [23].