FORMAL VERIFICATION SUPPORT FOR ELLA

IED 4/1/1357

Deliverable D2.3c

An Algebraic Framework for Action and Process*

Howard Barringer Graham Gough Brian Monahan† Alan Williams ‡

Department of Computer Science

University of Manchester§

Oxford Road, Manchester, UK.

email: howard@cs.man.ac.uk

January 27, 1995

†Brian Monahan is now with Harlequin Ltd., Alderley Edge

‡Alan Williams acknowledges additional support from DRA (Malvern) and Harlequin Ltd (Cambridge)

§The University of Manchester acknowledges support from the SERC via research grant GR/F 38174.

1 Introduction

We began our work on ELLA semantics[BGMW93] by characterising it in terms of a synchronous model of Input/Output Automata[BGL$^+$91, LT88]; these were essentially an elaborate form of Mealy automata.

The aim of this work below is to generalise our basic model to provide a (very) general algebraic framework for semantics based upon process algebraic ideas. This framework is far more general than that required to capture semantics for ELLA. For instance, we hope to be able to describe synchronous and asynchronous systems equally in this framework.

We give a brief summary of the rest of the paper. The next section briefly discusses some of the ideas and considerations that lead us to taking a more algebraic approach. The following section defines the algebraic framework for *actions* and *bodies* — i.e. generic process terms. This section also introduces the notion of an algebra of *actions* for which concrete definitions are given within an instantiation of the framework. A transition based semantics is given for both *open* and *closed* bodies. In section 4, we define an algebra of synchronous actions abstracted from a general notion of typed transition. This algebra is of the form required for instantiating our framework. The final section briefly states some early conclusions from our work.

2 ELLA and Process Algebras

ELLA is a commercially used hardware description language[MDED91]. This application forces our model to have a number of characteristics that require extensions to more conventional process algebraic approaches (cf. LOTOS, CCS, SCCS).

Our basic requirement is to provide a semantically *accurate* approach to the verification of systems written in ELLA. Due to the complexities introduced by data, a more symbolic approach is required which allows both the concurrent and data aspects to be described. We have attempted to focus on approaches which help to separate these forms of structure. We hope that this may have the payoff of allowing verification tasks that can be more evenly split between these structures.

2.1 Processes and data-rich actions

The calculus that we shall develop below is mainly concerned with how actions and processes interact to produce system behaviour. Our actions will contain values from pure data-types and as such form a departure from a purely foundational analysis of process algebra. Applications will typically rely heavily on a mixture of data and control transaction rather than just pure synchronisation.

The extension of pure CCS/SCCS to include value passing (by use of indexed process terms) is well-known and understood. The approach that we shall be taking is similar to this. We will find it convenient to impose some general structure upon the actions to enable a general characterisation of the (synchronous) action product.

Because we shall add data types and values as an independently specified component of the actions that will label our transitions, we say that our calculus involves an *extrinsic* characterisation of data types. In precisely the same way, the value-passing versions of CCS and SCCS also depend upon an extrinsic characterisation of data types.

There are many examples of calculi that are sufficiently expressive that can internally describe general classes of data types and operations upon their values, including Turing Machines and (a little weaker) Polymorphic 2nd-Order λ-calculus. There are also pure process calculi, such as Milner's π-calculus, which are able to express general operations within a wide class of data types via synchronisation and concurrent computation. When any of these calculi are used in this way to encode data types, we may say that they are providing an *intrinsic* characterisation of data type.

The use of the terms *intrinsic* and *extrinsic* is not intended to assert anything about the expressiveness of the calculi since even powerful and expressive calculi may be used with externally specified data types (i.e. used in an *extrinsic* manner). These terms are used to indicate how the calculus interacts with data typing. It is perhaps fairest to say that each pure calculus may be used to characterise a class of data types and operations over them, although in some cases, this class may be rather restrictive.

2.2 Requirements for approach

The main differences between the required model and the basic approach taken in conventional synchronous process algebraic approaches are summarised here:

Broadcast communication :

Hardware devices generally have a large, effectively arbitrary, fanout. Accordingly, outputs from devices can be read by many observers.

In synchronous process algebras, the basic mode of communication is *point-to-point* handshaking, although multi-way synchronisation is permitted.

There are at least two ways of simulating the broadcast form of communication within synchronous process algebras. One way is for the source of a signal to carefully make enough copies of a signal for each consumer to use. The drawback of this is that it is a form of 'plumbing' that requires the sender to know all the users of each output signal. A similar and essentially equivalent way is to define explicit 'fanout' nodes that accept inputs and then rebroadcast them appropriately.

Another way is to change SCCS's product operation on actions to permit outputs to *satisfy* inputs, but not to be cancelled by their consumption (e.g outputs are *persistent*). This might be expressed by the following equation on SCCS *actions*:

$$a \star a^{-1} = a^{-1}$$

So, by requiring the broadcast model, we don't have a natural group structure on actions. In addition we further require multiple occurrences of any action to have the same effect as a single occurrence:

$$a \star a = a.$$

This means that we don't have a natural multiset structure, at least in the way that we intend to interpret what actions are.

Partiality and the use of channels:

Hardware systems have input and output channels for communication of signals and/or events with their external environment. Conceptually, this means that each channel is *named* and that whatever is read from a channel is whatever was written to that channel.

> The behaviour of systems is determined by the way in which they react in combination with other systems i.e. by the way that they respond to communication with their environment. So, assuming that we have *broadcast* communication, we discover that *inconsistent* combinations of actions could occur. These inconsistencies would arise as a result of action products that do not respect the intended meaning of channels (e.g. reading values different from those that were written). As such, this implies that such inconsistent combinations must be handled differently from those that do respect the meaning of channels.
>
> There are at least two approaches to dealing with this in synchronous process algebras such as SCCS. Perhaps the most obvious and straightforward way is to add side-conditions to the action product so as to ensure that only consistent combinations can be made. Another way is to add the notion of *improper* transitions whose effect is to 'mask' the partiality and this is the approach that we take below.

The above discussion shows that synchronous process algebras like SCCS are certainly capable of representing the model we want, but possibly at the expense of some encoding. As will be seen below, we try to reduce this encoding by adapting the standard treatment of the action monoid in SCCS by adding a 'zero' element (to handle partiality) and by ensuring that the action product accommodates broadcast communication. Unfortunately, this will mean that we will lose the elegant group structure as required by standard treatments of action products in SCCS. Nevertheless, our treatment remains consistent with Milner's most basic requirement that the action product be a symmetric monoid[Mil82].

However, we find some encouragement for our decision in the work on the *Algebra of Synchronous Systems* (ASP) by Bergstra and Klop, and by Weijland. ASP differs from SCCS in that it uses a multiset notation as the basis of parallelism, where each component of the multiset represents an atomic action upon a named channel. It turns out that ASP can easily incorporate both of the above requirements by specifying a suitable choice of *communication function* between actions. ASP also incorporates the concepts of well-termination and sequential composition. However, this is richer than we need as there is no accepted concept of well-termination of hardware. This further implies that sequential composition has no accepted standard interpretation in the context of hardware systems[1].

Clearly, the following work is strongly related to the approaches taken for both ASP and SCCS. In some sense, we develop and extend some concepts taken from ASP, but in the style of SCCS, with the intention of algebraically characterising the basic semantic model for ELLA.

2.3 Synchronous vs. Asynchronous Concurrency

There are several forms of concurrency, each of which depend upon the fundamental nature of *action* and *communication*. Each of these may be either synchronous or asynchronous, giving four possible combinations:

Synchronous communication: Communication proceeds by *handshaking* between participating processes. The participants engage in a simultaneous transfer of information from sender to receiver, happening at the same moment of time. Therefore communication necessarily places a constraint upon the flow of events within a given system.

[1]Unless one perhaps considers the notion of hardware that dynamically rewires itself (cf. neural networks)!

Synchronous action: Concurrent processes proceed in synchrony — the progress made by each process is in phase with others it could potentially communicate with. This model can support a notion of time flow, associated with the smallest granularity of action available.

Input/output statements must synchronise for information transfer to occur. This model can be used in situations where both input/output events are necessarily associated with every action made e.g. hardware components.

Alternatively, the model might also be used when components can *poll* additional status information to see if communication is possible. Resonance effects and other forms of race condition can occur in this model.

Asynchronous action: Concurrent processes can make progress independently of other processes within the system. There is no absolute notion of time flow within this model.

Input/output actions usually occur separately in each process.

Asynchronous communication: The acts of sending and receiving information is now separated in time and the sending process no longer participates directly in the reception of information by other processes. Usually, some form of synchronisation protocol is used as well in an attempt to regulate the progress of a system and prevent race conditions and deadlock.

Synchronous action: Input/output actions can be made, depending upon *status*. For example, reading can only take place when there is something to be read. The time taken for transmission of messages may become observable, assuming some global clocking scheme.

Asynchronous action:

Action and communication are only weakly coupled — all that is known is that messages can only be received after they have been sent.

The above is clearly not an exhaustive list of potential ways in which concurrent activity can manifest itself (e.g. models of concurrency), and really only touches on that subject. There are various other dimensions for comparison between process calculi, such as whether *multi-way broadcast* or *point-to-point* forms of communication are used.

It must be said that process calculi take the assumption of synchronous communication as primitive - asynchronous communication being then modelled by the introduction of an intermediary 'buffer' process representing the transport medium for signals.

2.4 Other work

A number of existing approaches have dealt with various aspects of this such as [Pon92, Win89, M.H91]. We briefly describe some work done by Winskel, although other work should also be discussed.

As noted by Winskel[Win84], the notion of synchronous and asynchronous action can be captured by the notion of a *synchronisation algebra*.

In [Win89], Winskel uses indexed categories to give a more abstract characterisation of transition systems than the previous treatment using synchronisation algebras[Win84].

As a fundamental notion, he allows a special *stuttering* step - what we have called ι. Winskel makes heavy use of *idle* transitions (where he uses $*$ for our ι action):

$$s \xrightarrow{*} s'$$

The $*$ element is a special improper label for which all Winskel's transition morphisms are strict.

We have also included our own version of the above as a primitive rule in the system, meaning that we always include ι stuttering. This means that the denotation of a process is essentially unaffected by the presence of these transitions - since every process is forced to have them by the above rule. As a consequence, they cannot be eliminated by mapping or restriction. They are included because of their possible effect in compositions and renaming. For example, the above rule (with ι replacing $*$) would still allow $*$ to appear on transitions between distinct process terms - this may be very useful in conjunction with process congruences other than general bisimilarity.

The basic upshot of Winskel's category theoretic approach is that it:

- shows elegant semantic correspondence between categories of synchronisation trees and labelled transition systems e.g. essentially synchronisation trees are normalised forms of general labelled transition systems.
- shows how different kinds of product are obtained by interactions between general product upon synchronisation trees and these label morphisms.
- shows how to construct a general logic over categories of labelled transition systems, exploiting the category theoretic structure in doing so. This construction is generic over a given category of propositions. It all works smoothly except for product.

3 General Process Framework

We give here a general framework for process calculi which is generic over an multi-sorted theory of *individuals*. The idea is that the framework captures common elements of different process calculi, each of which embody various concepts of action, communication and concurrency.

The general framework provides basic operators on processes and is dependent upon two major components : an algebra of actions and a theory of individuals. An instance of the framework is then obtained by providing each of these components. As an example, we shall use this framework to present a form of synchronous process calculus suitable for giving the semantics of ELLA.

3.1 Preliminaries

We take as given a basic collection of primitive sort constants, *SORT*. A signature, Σ, represents a binding from entity names to structural types — this information indicates how a given named entity may participate in the construction of terms. We let $\Sigma : s$ denote the signature obtained by restricting Σ to those bindings with result sort s.

In general, the equals symbol, '$=$', stands for semantic equality between terms of the same kind, where there is no possibility of ambiguity. We shall use '$\equiv$' to mean syntactic equivalence (i.e. identity upon term structure). Other forms of equivalence (congruence) will be decorated by subscripts as appropriate e.g. '$\equiv_{tr}$'.

Let Var be the collection of all sorted variables and let Var_s be the class of variables of sort s (i.e. $Var_s = Var \cap Tm(\Sigma : s)$). We let $Tm_V(\Sigma : s)$ be the class of terms of sort s, freely generated over signature Σ and set of variables $V \subseteq Var$. As special cases, we generally write for general terms, $Tm(\Sigma : s) = Tm_{Var}(\Sigma : s)$ and for ground terms, $GndTm(\Sigma : s) = Tm_{\emptyset}(\Sigma : s)$.

Some of the operators that are introduced in this section, such as the derived operator 'Parametric Choice', act as *variable binding operators* and introduce scopes for bound variables. We say that $FV(t)$ gives the collection of all free variables in term t and use $FV_s(t)$ to give the collection of all free variables of sort s in term t (i.e. $FV_s(t) = FV(t) \cap Var_s$). The notation $FV(t_1, \cdots, t_n)$ stands for $FV(t_1) \cup \cdots \cup FV(t_n)$. We use the following notation for (simultaneous) substitution: $t[\vec{a}/\vec{x}]$ of terms $\vec{a}$ for variables $\vec{x}$ throughout term t, taking care to avoid clashes between free and bound variables (i.e. by renaming bound variables as necessary). This operation is only defined when the variables and the terms that are to replace them have the same sort.

3.2 Basic Framework Definition

Consider a given theory of *individuals*, IND. Let Σ_{IND} denote the many-sorted signature of operations for the theory IND. We assume that this theory contains a number of sorts, Idv_i, and require that there is a distinguished sort of *actions*, Act. So let $\mathcal{S} = SORT(\Sigma_{\text{IND}})$ denote the class of all sorts in signature Σ_{IND}. By an abuse of notation, we shall use $Tm(\Sigma : \mathcal{S})$ to denote the set of all terms for every sort in $\mathcal{S} \subseteq SORT(\Sigma)$, with respect to the signature Σ_{IND}.

We further assume that the class of terms of sort Act, $Tm(\Sigma_{\text{IND}} : Act)$, be non-empty, and that the family of sorts $\mathcal{S}$ contains the standard Boolean sort: $\mathbb{B} \in \mathcal{S}$. The collection of Boolean terms on signature, Σ_{IND}, is denoted by $Tm(\Sigma_{\text{IND}} : \mathbb{B}) = Tm(\Sigma_{\text{IND}} : \mathbb{B})$. We assume that theory IND subsumes the standard theory of Booleans, with all the usual operators over Booleans and including the constants true and false.

3.2.1 Ground terms and normalisation

It turns out to be convenient to ensure that ground terms denoting individuals are considered to be identified according to appropriate algebraic congruences specified by theory IND. This amounts to considering ground individual terms of each sort $s \in \mathcal{S}$ as being identified with the equivalence class implied by the theory IND.

Formally, we assume that the theory IND provides a *complete* characterisation of equivalence of ground terms, so that each ground term can be taken to correspond to a unique congruence class. In particular, all ground terms are denoting. This is equivalent to assuming that the theory IND comes equipped with a family of (meta-level) *evaluation* functions:

$$\langle eval_s : GndTm(\Sigma_{\text{IND}} : s) \to GndTm(\Sigma_{\text{IND}} : s) \rangle_{s \in \mathcal{S}}$$

indexed by sorts s from $\mathcal{S}$, such that if $eval_s[\![t]\!] \equiv [\![h]\!]$ then:

(a). $\text{IND} \models t = h$

(b). $eval_s[\![h]\!] \equiv [\![h]\!]$

(c). $\text{IND} \models u = v$ iff $eval_s[\![u]\!] \equiv eval_s[\![v]\!]$

Clearly, there are many choices of evaluation function in general, for a given theory with complete ground congruences.

In practice, this means that we treat arbitrary ground term individuals as though they were in an appropriate normalised form. Accordingly, ground terms will be matched modulo such a normalisation.

3.2.2 Composition and Mutation Operators

Our framework allows for several kinds of composition operator and what we shall call 'mutation' operators at the process level. Composition operators will correspond to concurrency products over processes; the mutation operators are so called because they generalise the notion of hiding and restriction and allow general mappings upon the actions that label transitions.

We acknowledge that composition and mutation operators at the generic process level are induced by corresponding operators and relations over the actions that label transitions. To formalise this, we require that there are two non-empty collections of operator symbols *CompOpr* and *MuRel* such that, with respect to the signature Σ_{IND}:

$$Op \in CompOpr \;\Rightarrow\; Op : (Act^2 \rightarrow Act)$$
$$Op \in MuRel \;\Rightarrow\; Op : (Act \times s \times Act \rightarrow \mathbb{B}),\ \text{for some sort } s \in \mathcal{S}$$

We call symbols from *MuRel*, *mutation relations* and symbols from *CompOpr*, *composition operators*, and we assume that $\phi \in MuRel$ and $\theta \in CompOpr$.

3.2.3 Basic Framework

We now introduce the structure of the basic framework in terms of the syntax presentation below. We let Σ_{GPF} be the signature of all process operators introduced below by the schematic syntax definition (i.e. including 'composite' operators such as $\backslash_\phi$ and $|_\theta$).

$$
\begin{array}{rcll}
Context & ::= & \vec{D} & \\
D \in BodyDefn & ::= & p(\vec{xs}) \stackrel{def}{=} B & \\
B \in Body & ::= & a :: B_1 & \textit{Prefixing} \\
 & | & g \rightarrow B_1 & \textit{Guarding} \\
 & | & \sum_{i \in I} B_i & \textit{Meta Summation} \\
 & | & p(\vec{t}) & \textit{Continuation Call} \\
 & | & B_1 \backslash_\phi t & \textit{Generic Mutation} \\
 & | & B_1 \,|_\theta\, B_2 & \textit{Generic Composition} \\
xs \in FormalParam & ::= & x : s &
\end{array}
$$

where we have that:

$$a \in Tm(\Sigma_{\text{IND}} : Act), \quad g \in Tm(\Sigma_{\text{IND}} : \mathbb{B})$$
$$s \in \mathcal{S}, \quad x \in Var$$
$$p \in BodyName$$

We have used some *meta* notation above in defining Meta Summation: let I denote arbitrary, but at most *countable* indexing sets[2] and we let B_i stand for an indexed family of terms of sort *Body* — a function from $I \to Tm(\Sigma_{GPF} : Body)$.

A *process* is a *closed body*, a term B with *no* free individual variables. The set of closed bodies is defined to be:

$$ClsBody \triangleq \{B \in Tm(\Sigma_{GPF} : Body) \mid FV(B) = \emptyset\}$$

We let C range over *ClsBody*.

There are several well-formedness conditions, such as:

For $p(xs_1, \ \cdots \ , xs_k) \triangleq B$, we have that $FV(B) \subseteq FV(xs_1, \ \cdots \ , xs_k)$

3.3 Semantic requirements on actions

In order to develop the framework further by giving it a transition based semantics, we require that each action algebra *Act* contain the following:

$$\iota \ : Act$$
$$\boxtimes : Act$$

Intuitively, the ι action is an element meaning 'skip' from a local point of view but which allows the environment to make a move (c.f. asynchronous moves). The $\boxtimes$ action is a distinguished element meaning 'non-action' - every process is related to each other by this action (see transition semantics below).

We also require that each action composition operator $\theta \in CompOpr$ satisfy the following requirements:

associativity	$(a_1 \ \theta \ a_2) \ \theta \ a_3$	$= a_1 \ \theta \ (a_2 \ \theta \ a_3)$
commutativity	$a_1 \ \theta \ a_2$	$= a_2 \ \theta \ a_1$
zero	$\boxtimes \ \theta \ a$	$= \boxtimes$

Finally, we require that each mutation relation $\phi \in MuRel$ satisfies the following requirements, for every $x : s, a : Act$:

[2] A restriction on cardinality of the indexing set, I is used to ensure that the class of Body terms itself forms a *set*, rather than a proper class.

$$\boxed{\text{Vacuum}}\ \frac{}{C \xrightarrow{\boxtimes}_C C'} \qquad \boxed{\text{Skip}}\ \frac{}{C \xrightarrow{\iota}_C C} \qquad \boxed{\text{Prefix}}\ \frac{}{ga :: C \xrightarrow{ga}_C C}$$

$$\boxed{\text{Guard}}\ \frac{\vdash g, \quad C \xrightarrow{a}_C C'}{(g \to C) \xrightarrow{a}_C C'} \qquad \boxed{\text{Meta Summation}}\ \frac{\text{for some } j \in I,\ C_j \xrightarrow{a}_C C'}{\sum_{i \in I} C_i \xrightarrow{a}_C C'}$$

$$\boxed{\text{Eq}}\ \frac{C_1 \stackrel{def}{=} C_2, \quad C_1 \xrightarrow{a}_C C'}{C_2 \xrightarrow{a}_C C'} \qquad \boxed{\text{Mutation}}\ \frac{C \xrightarrow{a}_C C', \quad \vdash \phi(a,t,b)}{C \setminus_\phi t \xrightarrow{b}_C C' \setminus_\phi t}$$

$$\boxed{\text{Product}}\ \frac{C_1 \xrightarrow{a_1}_C C_1', \quad C_2 \xrightarrow{a_2}_C C_2'}{C_1 \mid_\theta C_2 \xrightarrow{a_1\, \theta\, a_2}_C C_1' \mid_\theta C_2'}$$

Table 1: Transition semantics for closed terms

zero $\quad \phi(\boxtimes,\, x, a) \;\Rightarrow\; a = \boxtimes$

To summarise, a completely specified action algebra provides the following information:

- The action sort, *Act*
- Interpretations for the constants $\iota,\ \boxtimes : Act$.
- A family of composition operators, one for each operator in *CompOpr* satisfying the above properties.
- A family of *mutation* relations, one for each operator in *MuRel* and which act upon transitions according to the above requirements.

3.4 Transition based semantics

We now give a general transition based semantics for the generic framework. This consists of firstly defining a transition relation for closed bodies first, and then generalising this semantics to arbitrary open bodies (i.e. generic process terms) by using instantiation with respect to ground substitutions.

The semantics of closed body terms is given as the *least* transition relation between closed body terms, indexed by actions from *Act*, satisfying the rules given in Table 1.

Note that the rule given for **Meta Summation** is an existential form of *infinitary* rule - the premisses are conditional on there being an index value, $j \in I$ selecting a *closed* term C_j equal to some C, such that $C \xrightarrow{a}_C C'$.

We now give a semantics for *open terms* as follows. Let $a \in Tm(\Sigma_{\mathrm{IND}} : Act)$ and $B,\ B' \in Tm(\Sigma_{GPF} : Body)$

$$B \xrightarrow{a}_O B' \;\Leftrightarrow\; \forall\, \sigma \in \mathit{GndSubst}(\mathit{FV}(B, B', a)) \bullet B\,[\,\sigma\,] \xrightarrow{a\,[\,\sigma\,]}_C B'\,[\,\sigma\,]$$

This extends our semantics for closed terms to those for open terms by quantifying over the collection of all well-typed *ground substitutions*.

What is interesting here are some of the differences between the rules given in Table 1 and those for earlier systems such as BPA and CCS. Of particular note is the *Vacuum* rule, which says that transitions labelled by the zero action, $\boxtimes$, may take a generic process B into *any* process B'. This is the process algebraic equivalent of the falsum element in logic. Transitions labelled by $\boxtimes$ are said to be *vacuous* and are factored out of equivalence notions involving the above semantics. Actions $a \in Act$ not equal to $\boxtimes$ are known as *proper* actions.

The introduction of *vacuous* transition has the advantage of permitting a uniform and pleasing treatment of both partial behaviour and hiding. In fact, the notion of hiding can be completely captured generically in terms of the application to transitions of appropriate hiding operators over actions, whose transition semantics can be stated concisely within a single, unqualified rule.

3.5 General Process Congruences

The above framework gives the general form of process calculi involving value passing and can support a variety of process congruence notions. The most basic of these is the notion of *general bisimilarity* which essentially forces process terms to denote the collection of all possible ways of continuing. This is of course essentially the same as saying that a process denotes a generalised form of synchronisation tree[Win84, Mil89a].

General bisimilarity is therefore the finest congruence on processes that we will wish to consider.

3.5.1 General Bisimilarity

Following Milner[Mil89b], we can now give definitions for the generic framework of the notion of *general bisimilarity*. Introduce the following binary relation over generic process terms:

$$(\_ \sim \_) \subseteq \mathit{ClsBody}^2$$

which we define to be the *greatest* relation satisfying the following:

$(C_1 \sim C_2) \;\Leftrightarrow$

For every **proper** action $a \in Act$,

(a) For every C_1' such that $C_1 \xrightarrow{a}_C C_1'$,
There exists C_2' such that $C_2 \xrightarrow{a}_C C_2'$ and $(C_1' \sim C_2')$

and

(b) For every C_2' such that $C_2 \xrightarrow{a}_C C_2'$,
There exists C_1' such that $C_1 \xrightarrow{a}_C C_1'$ and $(C_2' \sim C_1')$

The above captures the basic idea that two process terms are generally bisimilar when each can make corresponding moves to further process terms that are in turn generally bisimilar. As such, general bisimilarity is the weakest (or finest) congruence that we would wish to consider between process terms. So, we say that a *process congruence* can be any congruence over process terms containing *general bisimilarity*.

3.5.2 Trace equivalence

In later applications of this work to ELLA semantics, we need only trace equivalence of systems and this is briefly described here.

We define the following relation:

$$\_ \xrightarrow{\_}{}_C^{+} \_ \subseteq (ClsBody \times Act^{+} \times ClsBody)$$

to be the *transitive* closure of the closed transition relation, $\xrightarrow{\_}{}_C$. We now define the *language* accepted by a process to be the set of its *traces*, for $C \in ClsBody$:

$$Lang(C) \triangleq \{tr \in Act^{+} \mid \exists\, C' \in ClsBody \cdot C \xrightarrow{tr}{}_C^{+} C'\}$$

Using this, we can easily define the notion of *trace equivalence* and *trace containment*:

$$C_1 \equiv_{tr} C_2 \;\Leftrightarrow\; Lang(C_1) = Lang(C_2)$$
$$C_1 \subseteq_{tr} C_2 \;\Leftrightarrow\; Lang(C_1) \subseteq Lang(C_2)$$

The above can be generalised to infinitary traces in the usual way (e.g. consider prefix closed chains). Such a generalisation is useful when considering temporal properties.

3.6 Full Generic Framework

Using the framework above, we now add some additional notation for process bodies which is often useful in practice and which can be explained in terms of the earlier basic framework by use of Meta Summation.

We essentially add the following:

$$0 \quad : Body$$
$$(\_ + \_) : Body^2 \rightarrow Body$$

The inactive body, 0 is defined by:-

$$0 \triangleq \sum_{i \in \emptyset} B_i$$

for any family of bodies $\langle B_i \rangle_{i \in I}$.

and $(\_ + \_)$ is defined by:

$$\boxed{\text{Binary Sum (a)}}\ \frac{C_1 \xrightarrow{a}_C C_1'}{C_1 + C_2 \xrightarrow{a}_C C_1'} \qquad \boxed{\text{Binary Sum (b)}}\ \frac{C_2 \xrightarrow{a}_C C_2'}{C_1 + C_2 \xrightarrow{a}_C C_2'}$$

$$\boxed{\text{Parametric Choice}}\ \frac{C\,[\,t\,/\,x\,] \xrightarrow{a}_C C'}{\square_{xs}\, C \xrightarrow{a}_C C'}$$

$$\boxed{\text{General Choice}}\ \frac{\vdash g\,[\,t\,/\,x\,],\quad C\,[\,t\,/\,x\,] \xrightarrow{a}_C C'}{\square\{\ C \mid x : s\ \wedge\ g\ \} \xrightarrow{a}_C C'}$$

Table 2: Derived closed transitions for derived operators

$$B_1 + B_2 \triangleq \sum_{i \in \{1,2\}} U_i$$

where

$$U_1 = B_1$$
$$U_2 = B_2$$

We can introduce the following *variable binding operator*, 'Parametric Choice', as a *derived operator* defined by Meta Summation and uniform substitution:

$$\square_{xs} B \triangleq \sum_{t \in Tm(\Sigma_{\mathrm{IND}}\ :\ s)} U_t$$

where

$$U_t = B\,[\,t\,/\,x\,]$$

Finally, we also introduce the following shorthand:

$$\square\{\ B \mid x : s\ \wedge\ g\ \}$$

which is defined to be:

$$\square\{\ B \mid x : s\ \wedge\ g\ \} \triangleq \square_{x\,:\,s}\,(g \rightarrow B)$$

Using the above, the standard derived rules as given in Table 2 can be shown.

Note that rule for Parametric Choice makes use of an open term, B forming the scope of the binding operator. However, since the term $B\,[\,t\,/\,x\,]$ in the premiss must match a closed body, the free variables of B must be contained in $\{x\}$. Also note this rule is not *infinitary* as it stands, but is essentially defined by syntactically *uniform* substitution.

4 Synchronous Mealy Calculus

In this section, we give a *synchronous instantiation* of our general framework given above. The basic intuition for this is inspired by considering I/O automata[LT88] as a model of computation appropriate for describing hardware systems. Such systems make simultaneous input/output transactions continually with their environment. It would be inappropriate to consider these systems as making separate input and output transactions - no observation could separate them. As such we consider input and output as an indivisible, simultaneous action of the system.

The main requirements of any instantiation is to provide the following:

- a set of actions, denoted by *Act*.
- an interpretation for the constants $\iota,\ \boxtimes \in Act$.
- interpretations for any composition operators (e.g. the causal product over actions $(\_ \mid \_)$)
- interpretations for any mutation operators (e.g. the hiding operator $(\_ \setminus \_)$).

Our presentation below will adopt a layered approach that gradually instantiates the basic framework. For example, we do not need to fix the form of our most primitive constituents until the final subsection.

The following subsections are organised as follows. Section 4.1 introduces the basic structure of our synchronous actions. These are essentially composed of pairs of I/O component actions. However, we defer the discussion of the axiomatic properties of synchronous actions until Section 4.3. This follows Section 4.2 where we discuss the I/O component actions in detail. In particular, we define sufficient structure that enables an abstract definition of action product and hiding operator. Section 4.4 gives an instantiation of the primitive I/O component actions satisfying the requirements made by earlier subsections.

4.1 Introduction

Synchronous actions are used to label transitions. For us, a (proper) action is a composite object consisting of a pair, each of which are in turn structured objects, called *I/O actions*. We now provide a general algebraic presentation of this structure according to the following syntax:

Definition 4.1 : Synchronous Actions

$$\gamma \in IO ::= \iota \,\big|\, \boxtimes \,\big|\, (\alpha \,/\, \beta)$$

where $\alpha,\ \beta \in IOAct$, the I/O actions.

The above syntax presentations in effect introduce the following algebraic signature:

$$\begin{aligned} \iota &: Act \\ \boxtimes &: Act \\ (\_ \,/\, \_) &: IOAct^2 \to Act \end{aligned}$$

As usual, we require the initial (many sorted) term algebra over this signature.

We defer until Section 4.3 a detailed presentation of equational requirements over the above signature, after our discussion of the algebra of IO actions below.

4.2 I/O Component Actions

We show in this section how composite I/O Actions can be built from primitive atomic I/O actions. Assume that *pIOAct* denotes a fixed but *given* set of primitive actions, whose values are ranged over by variables $pa, pa_1, pa_2 \cdots \in pIOAct$. The appropriate term structure for I/O actions is presented by the following syntax:

Definition 4.2 : I/O Actions

$$\alpha,\ \beta \in IOAct ::= \boxtimes \,\big|\, \varepsilon \,\big|\, \alpha_1 \oplus \alpha_2 \,\big|\, pa$$

In general, both α and β range over I/O Actions.

The above syntax presentation introduces the following algebraic signature:

$$\begin{aligned} \boxtimes &: IOAct \\ \varepsilon &: IOAct \\ (\_ \oplus \_) &: IOAct^2 \rightarrow IOAct \end{aligned}$$

The intuition is that the summation operation $(\_ \oplus \_)$ is used to combine I/O actions consistently.

Axiom 4.3 : Actions :

$$\begin{array}{lrcl} \text{a.} & (\alpha_1 \oplus \alpha_2) \oplus \alpha_3 & = & \alpha_1 \oplus (\alpha_2 \oplus \alpha_3) \\ \text{b.} & \alpha_1 \oplus \alpha_2 & = & \alpha_2 \oplus \alpha_1 \\ \text{c.} & \varepsilon \oplus \alpha & = & \alpha \\ \text{d.} & \alpha \oplus \alpha & = & \alpha \\ \text{e.} & \boxtimes \oplus \alpha & = & \boxtimes \end{array}$$

These axioms say that the structure $\langle IOAct,\ \oplus,\ \varepsilon \rangle$ forms an absorptive, commutative monoid, with an annihilator element given by $\boxtimes$. Accordingly, *initiality* forces this structure to be freely generated by the primitive actions specified in *pIOAct*.

The following lemma provides some simple, but useful decomposition properties of the binding summation operator:

Lemma 4.4 : 1. $(\alpha_1 \oplus \alpha_2) \neq \boxtimes \;\Rightarrow\; (\alpha_1 \neq \boxtimes) \wedge (\alpha_2 \neq \boxtimes)$

2. $(\alpha_1 \oplus \alpha_2) = \varepsilon \;\Leftrightarrow\; (\alpha_1 = \varepsilon) \wedge (\alpha_2 = \varepsilon)$

The following properties concerning the generating elements from *pIOAct* are consequences of the initiality of the above algebra of I/O actions:

Lemma 4.5 : $\{\varepsilon,\ \boxtimes\} \cap pIOAct = \emptyset$.

Proof : These may be seen by noting that initiality requires term structures to be distinct, unless otherwise identified by the congruence induced by the above equations. As no equations specifically identify generated elements, the distinguished constants could not belong to *pIOAct*. $\square$

Furthermore, we also have the following results from initiality:

Lemma 4.6 : Definedness of $(\_ \oplus \_)$

$$(\alpha_1 \oplus \alpha_2) \neq \boxtimes \quad \Leftrightarrow \quad \boxtimes \notin \{\alpha_1,\ \alpha_2\}$$

This property says that $\boxtimes$ can only arise when it occurs explicitly as a summand.

Lemma 4.7 : Trivial compatibility of $(\_ \oplus \_)$

$$(\alpha_1 \oplus \alpha_2) = \alpha_2 \quad \Leftrightarrow \quad \exists\, \alpha_3 \in IOAct \bullet \alpha_2 = (\alpha_1 \oplus \alpha_3)$$

4.2.1 Interpretations for applications

The following sections are concerned with ways in which this basic structure can be used to provide interpretations that model applications. It turns out that such interpretations amount to imposing some particular congruence relation upon the above terms. However, the behaviour of this congruence relation is determined by the effects of a number of other relations that naturally in these applications. Accordingly, the remaining subsections concern themselves with how the appropriate congruence relation is *induced* from such relations.

4.2.2 Information pre-ordering and conflict freedom relations

We assume as our starting point that we are given the following pair of relations over I/O actions:

$$(\_ \preccurlyeq \_) \subseteq IOAct^2$$
$$(\_\ \mathsf{CF}\ \_) \subseteq IOAct^2$$

Furthermore, we assume that these relations possess certain properties that we present and motivate within the discussion below. But, before doing so, note that these relations act upon I/O actions as a whole and not just the primitive actions.

Information pre-ordering The intuition behind the $(\_ \preccurlyeq \_)$ relation is that it represents an "information" pre-order saying when elements have compatible content. By only requiring a pre-order here, we conveniently allow distinctly presented actions to have equivalent meaning, since equivalence relations are precisely symmetric pre-order relations. Another purpose is to allow primitive actions to include auxiliary information, such as typing, which may then influence compatibility appropriately. These considerations lead to the following axiomatic properties:

Axiom 4.8 : Information pre-ordering upon I/O actions

1. $(\_ \preccurlyeq \_)$ is a pre-order (i.e. reflexive and transitive) on *IOAct*.
2. $\alpha \preccurlyeq (\alpha \oplus \beta)$.
3. $(\alpha \preccurlyeq \beta) \;\Rightarrow\; (\alpha \oplus \beta) \preccurlyeq \beta$.

The first axiom above states that $\preccurlyeq$ is a pre-ordering, as discussed previously, whereas the last couple of axioms requires that $\oplus$ conserves the pre-ordering. In actual fact, they ensure that $\oplus$ behaves like a (weak) 'LUB' operation with respect to the given pre-order. These facts are formally stated and demonstrated later on.

Consequences Using the axioms given above, we now derive a number of properties of the information pre-order, $\preccurlyeq$. First of all, we show a symmetry and monotonicity result:

Lemma 4.9 : $\beta \preccurlyeq (\alpha \oplus \beta)$

Lemma 4.10 : Monotonicity of $(\_ \oplus \_)$ wrt. $(\_ \preccurlyeq \_)$.

$$\alpha_1 \preccurlyeq \alpha_2 \;\Rightarrow\; (\alpha_1 \oplus \beta) \preccurlyeq (\alpha_2 \oplus \beta)$$

As corollaries, we have by commutativity of $\oplus$:

Corollary 4.11 :
$$\alpha_1 \preccurlyeq \alpha_2 \wedge (\beta_1 \preccurlyeq \beta_2) \;\Rightarrow\; (\alpha_1 \oplus \beta_1) \preccurlyeq (\alpha_2 \oplus \beta_2)$$

Earlier, we alluded to the fact that $(\_ \oplus \_)$ represents a (weak) LUB operation with respect to $(\_ \preccurlyeq \_)$. This is now formally characterised through the following:

Lemma 4.12 : LUB property for $(\_ \oplus \_)$ wrt. $(\_ \preccurlyeq \_)$

$$\forall\, \beta \in \mathit{IOAct} \bullet (\alpha_1 \preccurlyeq \beta) \wedge (\alpha_2 \preccurlyeq \beta) \;\Rightarrow\; (\alpha_1 \oplus \alpha_2) \preccurlyeq \beta$$

Corollary 4.13 : $\alpha \preccurlyeq \beta \;\Leftrightarrow\; (\alpha \oplus \beta) \preccurlyeq \beta$

Using the above corollary, we can now show the following:

Lemma 4.14 :
1. $\varepsilon \preccurlyeq \alpha$
2. $\alpha \preccurlyeq \boxtimes$

Conflict freedom relation The intuition motivating the relation, $(\_\ \mathsf{CF}\ \_)$, is that it characterises when I/O actions are *conflict free*. Intuitively, this means that it says which pairs of I/O actions can be composed together so that the result is *not equivalent* to $\boxtimes$. So, in some sense, this suggests that $(\_\ \mathsf{CF}\ \_)$ acts like a pre-condition for the composition operation $(\_ \oplus \_)$. These considerations lead to the following axiomatic requirements:

Axiom 4.15 : Conflict Freedom relation upon I/O actions

1. $(\_\ \mathsf{CF}\ \_)$ is symmetric.
2. $\neg(\boxtimes\ \mathsf{CF}\ \boxtimes)$.
3. The Triangle Law
   $(\alpha_1 \oplus \alpha_2)\ \mathsf{CF}\ \alpha_3 \ \Leftrightarrow\ (\alpha_1\ \mathsf{CF}\ \alpha_3) \wedge (\alpha_2\ \mathsf{CF}\ \alpha_3) \wedge (\alpha_1\ \mathsf{CF}\ \alpha_2)$

In detail, some of the above axioms may be *intuitively* motivated from the following considerations. The symmetry of the $(\_\ \mathsf{CF}\ \_)$ relation follows from consideration of the commutativity of composition. For the second axiom, we clearly require that $\boxtimes$ cannot be consistently composed with itself, by strictness of composition with respect to $\boxtimes$. The final axiom characterises the consistency relationship of composition in terms of how this relation behaves with respect to composite I/O actions. Essentially, this axiom says that a composite Action is conflict free with another element *precisely* when each of the components are pairwise conflict free with each other.

Consequences We now briefly turn our attention to consequences for the $(\_\ \mathsf{CF}\ \_)$ relation.

Lemma 4.16 : Rotation property

$$(\alpha_1 \oplus \alpha_2)\ \mathsf{CF}\ \alpha_3 \ \Leftrightarrow\ (\alpha_2 \oplus \alpha_3)\ \mathsf{CF}\ \alpha_1 \ \Leftrightarrow\ (\alpha_3 \oplus \alpha_1)\ \mathsf{CF}\ \alpha_2$$

Lemma 4.17 : Self-consistency property

$$\alpha\ \mathsf{CF}\ \beta \ \Rightarrow\ \alpha\ \mathsf{CF}\ \alpha$$

From this, we get the following corollary:

Corollary 4.18 : $\neg(\boxtimes\ \mathsf{CF}\ \alpha)$

Finally, we make the following useful observation from the triangle law:

Lemma 4.19 :
$$\begin{aligned}(\alpha \oplus \beta)\ \mathsf{CF}\ \beta \ &\Leftrightarrow\ \alpha\ \mathsf{CF}\ \beta \\ &\Leftrightarrow\ (\alpha \oplus \beta)\ \mathsf{CF}\ \alpha\end{aligned}$$

Connecting the information pre-order and the conflict freedom relation The information pre-order and the conflict freedom relations have been considered separately without concern for any interdependencies that they ought to possess. This of course allows these to be specified independently. However, they should interact at some stage to induce the congruence relation appropriate to the application.

We can relate the $(\_\ \mathsf{CF}\ \_)$ and $(\_ \preccurlyeq \_)$ relations, by requiring a form of monotonicity property for CF, namely that CF be *lower closed* relative to the pre-order $\preccurlyeq$.

Axiom 4.20 : Lower closure of $(\_\ \mathsf{CF}\ \_)$ wrt. $(\_ \preccurlyeq \_)$

$$(\alpha_1 \preccurlyeq \alpha_2) \wedge (\alpha_2\ \mathsf{CF}\ \beta) \Rightarrow (\alpha_1\ \mathsf{CF}\ \beta)$$

From this axiom, we can now show:

Lemma 4.21 : $(\alpha\ \mathsf{CF}\ \beta) \Rightarrow \neg(\boxtimes \preccurlyeq \alpha)$

Proof : Suppose that, on the contrary, $\boxtimes \preccurlyeq \alpha$. By axiom, we have that $\alpha \preccurlyeq (\alpha \oplus \beta)$ and so by transitivity, we have that $\boxtimes \preccurlyeq (\alpha \oplus \beta)$. Using the triangle law for CF, we can see that $(\alpha \oplus \beta)\ \mathsf{CF}\ \alpha \Leftrightarrow \alpha\ \mathsf{CF}\ \beta$. Since we are given that $\alpha\ \mathsf{CF}\ \beta$, we see that $(\alpha \oplus \beta)\ \mathsf{CF}\ \alpha$ holds. By applying the lower-closure axiom, we then have that $\boxtimes\ \mathsf{CF}\ \alpha$. But, by an earlier lemma, this is false. Therefore, our original assumption is contradictory and so $\neg(\boxtimes \preccurlyeq \alpha)$. $\square$

Digression : Forcing lower closure of CF Within applications, it may turn out that the natural specification of the CF relation is not lower closed. We briefly digress below to show that such a situation can always be remedied in an optimal way by defining an appropriately lower-closed version of the CF relation, which also satisfies the axioms for CF. This result shows that the lower-closure property does not place an essential restriction on the useful CF relations, since the appropriate closure can always be applied.

Firstly, for fixed pre-order $\preccurlyeq$, we define the following predicate:

Definition 4.22 : Weakly defined actions

$$Ds(\_) \subseteq IOAct$$
$$Ds(\alpha) \Leftrightarrow \neg(\boxtimes \preccurlyeq \alpha)$$

This predicate specifies those I/O actions which are *not* above the element $\boxtimes$, and as such, correspond to the (weakly) defined elements in *IOAct*, all with respect to $(\_ \preccurlyeq \_)$. Significant observations about this predicate are that it is lower-closed and that it is not satisfied by $\boxtimes$. Formally, we have:

Lemma 4.23 :
1. $(\alpha \preccurlyeq \beta) \wedge Ds(\beta) \Rightarrow Ds(\alpha)$
2. $\neg Ds(\boxtimes)$

Now, suppose that we have a binary relation, $(\_\ \mathsf{CF'}\ \_) \subseteq IOAct^2$, which satisfies the axioms for CF, but with the possible exception of the above lower-closure axiom:

Assumption 4.1 : 1. $(\_\ \mathsf{CF'}\ \_)$ is symmetric.

2. $\neg(\boxtimes\ \mathsf{CF'}\ \boxtimes)$.

3. $(\alpha_1 \oplus \alpha_2)\ \mathsf{CF'}\ \alpha_3 \;\Leftrightarrow\; (\alpha_1\ \mathsf{CF'}\ \alpha_3) \wedge (\alpha_2\ \mathsf{CF'}\ \alpha_3) \wedge (\alpha_1\ \mathsf{CF'}\ \alpha_2)$

We now define the lower-closed version of CF':

Definition 4.24 :

$$
\begin{array}{l}
(\_\ \mathsf{CF'}\!\downarrow\ \_) \subseteq \mathit{IOAct}^2 \\
\forall\ \alpha,\ \beta \in \mathit{IOAct} \bullet \\
\quad (\alpha\ \mathsf{CF'}\!\downarrow\ \beta) \;\Leftrightarrow \\
\qquad \exists\ \alpha_1,\ \beta_1 \in \mathit{IOAct} \bullet \\
\qquad\quad (\alpha_1\ \mathsf{CF}\ \beta_1) \wedge \mathit{Ds}(\alpha_1) \wedge (\alpha \preccurlyeq \alpha_1) \wedge \mathit{Ds}(\beta_1) \wedge (\beta \preccurlyeq \beta_1)
\end{array}
$$

From this, we get the following three results:

Lemma 4.25 : 1. $\forall\ \alpha, \beta \in \mathit{IOAct} \bullet$

$$\mathit{Ds}(\alpha) \wedge \mathit{Ds}(\beta) \wedge (\alpha\ \mathsf{CF}\ \beta) \;\Rightarrow\; (\alpha\ \mathsf{CF'}\!\downarrow\ \beta)$$

2. $(\alpha_1 \preccurlyeq \alpha_2) \wedge (\alpha_2\ \mathsf{CF'}\!\downarrow\ \beta) \;\Rightarrow\; (\alpha_1\ \mathsf{CF'}\!\downarrow\ \beta)$

3. $(\alpha\ \mathsf{CF'}\!\downarrow\ \beta) \;\Rightarrow\; \mathit{Ds}(\alpha)$

We now show that $\mathsf{CF'}\!\downarrow$ satisfies all the axioms given earlier for CF':

Lemma 4.26 : 1. $(\_\ \mathsf{CF'}\!\downarrow\ \_)$ is symmetric.

2. $\neg(\boxtimes\ \mathsf{CF'}\!\downarrow\ \boxtimes)$.

3. $(\alpha_1 \oplus \alpha_2)\ \mathsf{CF'}\!\downarrow\ \alpha_3 \;\Leftrightarrow\; (\alpha_1\ \mathsf{CF'}\!\downarrow\ \alpha_3) \wedge (\alpha_2\ \mathsf{CF'}\!\downarrow\ \alpha_3) \wedge (\alpha_1\ \mathsf{CF'}\!\downarrow\ \alpha_2)$

This shows that $\mathsf{CF'}\!\downarrow$ always exists and can adequately function in place of CF'. However, there remains the question of whether $\mathsf{CF'}\!\downarrow$ is the least such lower-closed relation implied by CF'. We answer this by showing that if CF' is *already* lower-closed then the relations CF' and $\mathsf{CF'}\!\downarrow$ are in full agreement (i.e. $\mathsf{CF'}\!\downarrow$ requires no more nor less than CF' does). Formally, we have:

Theorem 4.27 : If:

$$\forall\ \alpha_1,\ \alpha_2,\ \alpha_3 \in \mathit{IOAct} \bullet (\alpha_1 \preccurlyeq \alpha_2) \wedge (\alpha_2\ \mathsf{CF'}\ \alpha_3) \;\Rightarrow\; (\alpha_1\ \mathsf{CF'}\ \alpha_3)$$

Then:

$$\forall\ \alpha,\ \beta \in \mathit{IOAct} \bullet (\alpha\ \mathsf{CF'}\ \beta) \;\Leftrightarrow\; (\alpha\ \mathsf{CF'}\!\downarrow\ \beta)$$

Principal Atoms and Projection It turns out that we need the following additional requirements on the basic structure so as to have appropriate congruence results of certain operators at a later stage.

We define first of all the *strict lower closure* of the Primitive elements:

$$PA = (\downarrow \{a \in pIOAct \mid \mathcal{D}(a)\}) \setminus \{\varepsilon\}$$

where for any $S \subseteq IOAct$:

$$\downarrow S \triangleq \{x \in IOAct \mid \exists\, s \in S \bullet x \preccurlyeq s\}$$

We now require the following axioms:

Axiom 4.28 : Leastness of ε :

$$\alpha \preccurlyeq \varepsilon \;\Rightarrow\; \alpha = \varepsilon$$

Projection Axiom :

$$\begin{array}{l}(\alpha \,\mathsf{CF}\, \beta) \wedge p \in PA \;\wedge\; p \preccurlyeq (\alpha \oplus \beta) \;\Rightarrow \\ \quad (p \preccurlyeq \alpha) \;\vee \\ \qquad (p \preccurlyeq \beta) \;\vee \\ \qquad\quad (\exists\, a, b \in PA \bullet \\ \qquad\qquad (a \preccurlyeq \alpha) \wedge (b \preccurlyeq \beta) \wedge (p = (a \oplus b)) \\ \qquad\quad)\end{array}$$

The leastness axiom is included for tidiness - there appears to be nothing lost by assuming it. The Projection axiom requires that the general LUB elements which cover atoms can always be pulled back into the lower closed subset *PA*.

Action pre-ordering and congruence To make further progress, we combine the given information pre-order and conflict freedom relations to produce a new pre-order which is then used to induce the required congruence relation.

First of all, we introduce a stronger notion of defined element that is directly based upon the conflict freedom relation:

Definition 4.29 : Strongly defined actions

$$\begin{array}{l}\mathcal{D}(\_) \;\subseteq\; IOAct \\ \mathcal{D}(\alpha) \;\Leftrightarrow\; (\alpha \,\mathsf{CF}\, \alpha)\end{array}$$

From this definition and earlier lemmas, we directly obtain the following:

Lemma 4.30 :
1. $\neg\, \mathcal{D}(\boxtimes)$
2. $(\alpha \preccurlyeq \beta) \;\wedge\; \mathcal{D}(\beta) \;\Rightarrow\; \mathcal{D}(\alpha)$
3. $(\alpha \,\mathsf{CF}\, \beta) \;\Rightarrow\; \mathcal{D}(\alpha)$

4. $\boxtimes \preccurlyeq \alpha \;\Rightarrow\; \neg\,\mathcal{D}(\alpha)$

Property 4 expresses the fact that the conflict freedom relation constrains the defined elements further than the given information pre-order does.

Using the above, we can now show a further link between $\mathcal{D}$ and CF, via $\oplus$, in the following theorem:

Theorem 4.31 : Conflict freedom is equivalent to strong definedness of composition.

$$\alpha \,\mathsf{CF}\, \beta \;\Leftrightarrow\; \mathcal{D}(\alpha \oplus \beta)$$

This theorem neatly captures the intuition that the conflict freedom relation represents the argument pairs for which the action composition is strongly defined.

As a direct corollary, we have that:

Corollary 4.32 : $\mathcal{D}(\alpha \oplus \beta) \;\Rightarrow\; \mathcal{D}(\alpha)$

From this, the following then becomes clear:

Corollary 4.33 : $\mathcal{D}(\varepsilon) \;\Leftrightarrow\; \exists\, \alpha \in \mathit{IOAct} \cdot \mathcal{D}(\alpha)$

We are now ready to introduce an adapted pre-order upon I/O actions which takes both $\preccurlyeq$ and CF into account.

Definition 4.34 : Induced action pre-ordering

$$(\_ \sqsubseteq \_) \;\subseteq\; \mathit{IOAct}^2$$
$$(\alpha \sqsubseteq \beta) \;\Leftrightarrow\; \neg\,\mathcal{D}(\beta) \;\vee\; (\alpha \preccurlyeq \beta)$$

We begin our examination of $\sqsubseteq$ with the following obvious but useful observation:

Lemma 4.35 : Over specification property

$$\neg\,\mathcal{D}(\beta) \;\Rightarrow\; (\alpha \sqsubseteq \beta)$$

This shows that $\sqsubseteq$ strictly adds something to $\preccurlyeq$ and that the "over specified" elements are all grouped together, above any strongly defined elements.

Using the above observation, we can easily check that $\sqsubseteq$ is indeed a pre-order:

Lemma 4.36 : $(\_ \sqsubseteq \_)$ is a pre-ordering of *IOAct*

Next, we satisfy ourselves that $\sqsubseteq$ conserves existing notions, via the following sequence of lemmas:

Lemma 4.37 : 1. $\alpha \preccurlyeq \beta \;\Rightarrow\; \alpha \sqsubseteq \beta$

2. $(\alpha \sqsubseteq \beta) \wedge \mathcal{D}(\beta) \Rightarrow \alpha \preccurlyeq \beta$

From this, we immediately obtain the following corollaries:

Corollary 4.38 : 1. $\varepsilon \sqsubseteq \alpha$

2. $\alpha \sqsubseteq \boxtimes$

3. $(\alpha \sqsubseteq \beta) \wedge \mathcal{D}(\beta) \Rightarrow \mathcal{D}(\alpha)$

The following lemma shows that $\oplus$ behaves well with respect to $\sqsubseteq$:

Lemma 4.39 : $(\alpha_1 \sqsubseteq \alpha_2) \Rightarrow (\alpha_1 \oplus \beta) \sqsubseteq (\alpha_2 \oplus \beta)$

As corollaries to this, we get the following:

Corollary 4.40 : 1. $\alpha \sqsubseteq (\alpha \oplus \beta)$

2. $\alpha \sqsubseteq \beta \Rightarrow (\alpha \oplus \beta) \sqsubseteq \beta$

3. $(\alpha_1 \sqsubseteq \alpha_2) \wedge (\alpha_2 \mathsf{CF} \beta) \Rightarrow (\alpha_1 \mathsf{CF} \beta)$

These then lead now to:

Lemma 4.41 : $\alpha \sqsubseteq \beta \Leftrightarrow (\alpha \oplus \beta) \sqsubseteq \beta$

Moreover, we then have that:

Lemma 4.42 : LUB property of $\oplus$ with respect to $\sqsubseteq$.

$$\begin{array}{l} \forall \beta \in \mathit{IOAct} \bullet \\ \quad (\alpha_1 \sqsubseteq \beta) \wedge (\alpha_2 \sqsubseteq \beta) \Rightarrow (\alpha_1 \oplus \alpha_2) \sqsubseteq \beta \end{array}$$

We are finally ready to introduce the induced congruence relation via the following natural definition:

Definition 4.43 : I/O action equivalence

$$\begin{array}{l} (\_ \simeq \_) \subseteq \mathit{IOAct}^2 \\ (\alpha \simeq \beta) \Leftrightarrow (\alpha \sqsubseteq \beta) \wedge (\beta \sqsubseteq \alpha) \end{array}$$

From the above we directly obtain the following substitution properties, using the fact that $\sqsubseteq$ is a pre-order and so transitive:

Corollary 4.44 : 1. $(\alpha_1 \simeq \alpha_2) \wedge (\beta \sqsubseteq \alpha_1) \Rightarrow (\beta \sqsubseteq \alpha_2)$

2. $(\alpha_1 \simeq \alpha_2) \wedge (\alpha_1 \sqsubseteq \beta) \Rightarrow (\alpha_2 \sqsubseteq \beta)$

The above lemmas allow us to 'transport' earlier results involving $\sqsubseteq$ and turn them into results concerning $\simeq$:

Lemma 4.45 : Congruence properties of $\simeq$.

1. $(\_ \simeq \_)$ is an equivalence relation.
2. $(\alpha_1 \simeq \alpha_2) \;\Rightarrow\; (\alpha_1 \oplus \beta) \simeq (\alpha_2 \oplus \beta)$
3. $(\alpha_1 \simeq \alpha_2) \wedge (\alpha_1 \; \mathsf{CF} \; \beta) \;\Rightarrow\; (\alpha_2 \; \mathsf{CF} \; \beta)$
4. $(\alpha_1 \simeq \alpha_2) \wedge \mathcal{D}(\alpha_1) \;\Rightarrow\; \mathcal{D}(\alpha_2)$

Using this congruence, we can 'characterise' all the principle notions we have introduced so far.

Theorem 4.46 : Characterisation theorem

1. $\mathcal{D}(\alpha) \;\Leftrightarrow\; \neg(\boxtimes \simeq \alpha)$
2. $\alpha \; \mathsf{CF} \; \beta \;\Leftrightarrow\; \neg(\boxtimes \simeq (\alpha \oplus \beta))$
3. $\alpha \sqsubseteq \beta \;\Leftrightarrow\; (\alpha \oplus \beta) \simeq \beta$

In other words, we could simply take a congruence relation upon our Action structure, and then have *defined* the relations we used along in the manner given above. This theorem has in some sense demonstrated the adequacy and generality of our approach.

As corollaries to the above, we note the following useful properties:

Corollary 4.47 :

1. $(\alpha_1 \sqsubseteq \alpha_2) \;\Leftrightarrow\; \exists\, \beta \in \mathit{IOAct} \bullet (\alpha_1 \oplus \beta) \simeq \alpha_2$
2. $(\alpha_1 \oplus \alpha_2) \sqsubseteq \beta \;\Leftrightarrow\; (\alpha_1 \sqsubseteq \beta) \wedge (\alpha_2 \sqsubseteq \beta)$
3. $(\beta \sqsubseteq \alpha_1) \vee (\beta \sqsubseteq \alpha_2) \;\Rightarrow\; \beta \sqsubseteq (\alpha_1 \oplus \alpha_2)$

Elimination operator We next introduce the *elimination operator* on actions. This operator will be used later to eliminate consistent overlaps between inputs and outputs for action product.

The basic idea is that this operator represents a kind of 'subtraction' operator which removes all parts from the first argument which 'vaguely resemble' parts from the second. Fortunately, it turns out that a suitable notion of 'vague resemblance' is definable in terms of concepts we already use, such as conflict freedom and the information pre-order.

Accordingly, we determine this operator essentially by induction over the term structure for *IOAct* given by $\langle \oplus, \varepsilon, \boxtimes \rangle$. Of course, the tricky part is to make sure that the resulting operation upon *IOAct* is a *congruent* operator relative to the action congruence, $\simeq$. This entails taking appropriate care with the operator's definition upon *atoms*. It is upon these that a basic notion of 'vague resemblance' is defined in terms of the ordering $\sqsubseteq$ restricted to atoms.

Definition 4.48 : Elimination operator

$(\_ \ominus \_) : IOAct^2 \rightarrow IOAct$

$$
\begin{array}{ll}
(1) & : \alpha \ominus \beta = \boxtimes \;\Leftrightarrow\; \alpha \oplus \beta \simeq \boxtimes \\
(2) & : ((\alpha = \varepsilon) \vee (\beta = \varepsilon)) \wedge (\alpha \,\mathsf{CF}\, \beta) \;\Rightarrow\; (\alpha \ominus \beta) = \alpha \\
(3) & : (\alpha_1 \sqsubseteq \alpha_2) \ominus \beta = (\alpha_1 \ominus \beta) \oplus (\alpha_2 \ominus \beta) \\
(4) & : \alpha \,\mathsf{CF}\, (\beta_1 \oplus \beta_2) \;\Rightarrow\; \alpha \ominus (\beta_1 \oplus \beta_2) = (\alpha \ominus \beta_1) \ominus \beta_2 \\
(5.1) & : \alpha, \beta \in PA \wedge (\alpha \,\mathsf{CF}\, \beta) \;\Rightarrow\; \alpha \gtreqless \beta \;\Rightarrow\; \alpha \ominus \beta = \varepsilon \\
(5.2) & : \alpha, \beta \in PA \wedge (\alpha \,\mathsf{CF}\, \beta) \;\Rightarrow\; \neg(\alpha \gtreqless \beta) \;\Rightarrow\; \alpha \ominus \beta = \alpha
\end{array}
$$

The above axioms constrain the possible functions that can act as a suitable elimination operator. If you prefer, one may rewrite the last two axioms into a simpler axiom using the conventional **if _ then _ else _** operator:

$$\alpha, \beta \in PA \wedge (\alpha \,\mathsf{CF}\, \beta) \;\Rightarrow\; \alpha \ominus \beta = \textbf{if } \alpha \gtreqless \beta \textbf{ then } \varepsilon \textbf{ else } \alpha$$

The notion of 'Vague Resemblance' The notion of Vague Resemblance required for the Elimination operator is summed up by the 'zig-zag' equivalence relation, $(\_ \gtreqless \_) \subseteq PA^2$, defined by the restriction:

$$(\_ \sqsubseteq_{PA} \_) = (\_ \sqsubseteq \_) \cap PA^2$$

Definition 4.49 : Zig Zag relation:

$$
\begin{array}{l}
(\_ \gtreqless \_) \subseteq PA^2 \\
(\_ \gtreqless \_) = ((\_ \sqsubseteq_{PA} \_) \cup (\_ \sqsubseteq_{PA}{}^{-1} \_))^{\star}
\end{array}
$$

In other words, $(\_ \gtreqless \_)$ is the symmetric, transitive closure of $(\_ \sqsubseteq_{PA} \_)$.

Lemma 4.50 :
1. $(\_ \sqsubseteq_{PA} \_)$ is a pre-order on PA.
2. $(\alpha \sqsubseteq_{PA} \beta) \;\Rightarrow\; (\alpha \gtreqless \beta)$
3. $(\_ \gtreqless \_)$ is the least equivalence relation containing $(\_ \sqsubseteq_{PA} \_)$.

Existence and Uniqueness It is possible to demonstrate existence of an Elimination operator, by carefully introducing a new function and then *defining* it according to strengthened versions of the above axioms. The form of these axioms would be such that they guarantee that the function exists. We would then show that this function satisfies the weaker axioms given above.

We continue with the following theorem:

Theorem 4.51 : Uniqueness Property.

If $F,\ G : IOAct^2 \rightarrow IOAct$ each satisfy the above axioms, we can then show that:

$$\forall\, \alpha_1,\ \alpha_2 \in IOAct \bullet F(\alpha_1, \alpha_2) = G(\alpha_1, \alpha_2)$$

Elimination is action congruent An auxiliary result ...

Lemma 4.52 : $\mathcal{D}(\beta) \wedge (\alpha \in PA) \Rightarrow (\alpha \ominus \beta = \alpha) \vee (\alpha \ominus \beta = \varepsilon)$

Lemma 4.53 : $\mathcal{D}(\beta) \Rightarrow (\alpha \ominus \beta) \sqsubseteq \alpha$

Now show monotonicity for the first argument ...

Lemma 4.54 : $\alpha_1,\ \alpha_2 \in PA \wedge (\alpha_1 \sqsubseteq \alpha_2) \Rightarrow (\alpha_1 \ominus \beta) \sqsubseteq (\alpha_2 \ominus \beta)$

Lemma 4.55 : $\mathcal{D}(\beta) \wedge (\alpha_1 \in PA) \wedge (\alpha_1 \sqsubseteq \alpha_2) \Rightarrow (\alpha_1 \ominus \beta) \sqsubseteq (\alpha_2 \ominus \beta)$

Lemma 4.56 : $\mathcal{D}(\beta) \wedge (\alpha_1 \sqsubseteq \alpha_2) \Rightarrow (\alpha_1 \ominus \beta) \sqsubseteq (\alpha_2 \ominus \beta)$

And now for the second argument (show anti-monotonicity), but first show a useful additional result ...

Lemma 4.57 : $(\alpha \ \mathsf{CF}\ \beta) \wedge (x, \beta \in PA) \wedge x \preccurlyeq (\alpha \ominus \beta) \Rightarrow (x \ominus \beta) = x$

Lemma 4.58 : $\beta_1,\ \beta_2 \in PA \wedge (\beta_1 \sqsubseteq \beta_2) \wedge (\alpha \ \mathsf{CF}\ \beta_2) \Rightarrow (\alpha \ominus \beta_2) \sqsubseteq (\alpha \ominus \beta_1)$

Lemma 4.59 : $\beta_1 \in PA \wedge (\beta_1 \sqsubseteq \beta_2) \wedge (\alpha \ \mathsf{CF}\ \beta_2) \Rightarrow (\alpha \ominus \beta_2) \sqsubseteq (\alpha \ominus \beta_1)$

Lemma 4.60 : $(\beta_1 \sqsubseteq \beta_2) \wedge (\alpha \ \mathsf{CF}\ \beta_2) \Rightarrow (\alpha \ominus \beta_2) \sqsubseteq (\alpha \ominus \beta_1)$

Putting the above lemmas together, we have our congruence result:

Theorem 4.61 : Congruence of Elimination

1. $\mathcal{D}(\beta_2) \wedge (\alpha_1 \sqsubseteq \alpha_2) \wedge (\beta_1 \sqsubseteq \beta_2) \Rightarrow (\alpha_1 \ominus \beta_2) \sqsubseteq (\alpha_2 \ominus \beta_1)$
2. $\mathcal{D}(\beta_2) \wedge (\alpha_1 \sqsubseteq \alpha_2) \wedge (\beta_1 \sqsubseteq \beta_2) \Rightarrow (\alpha_1 \ominus \beta_2) \sqsubseteq (\alpha_2 \ominus \beta_1)$

4.3 Synchronous I/O Algebra

In this section, we can now impose some requirements upon the form of synchronous actions, defined earlier in Section 4.1. These requirements depend upon the I/O component action algebra.

So, our first axiom upon our synchronous actions uses the Elimination operator defined upon *IOAct* to define a consistency axiom requiring that the inputs and outputs do not interfere, at least at a primitive level:

Axiom 4.62 : $(\alpha \,/\, \beta) = ((\alpha \ominus \beta) \,/\, \beta)$

This essentially says that consistent *IOAct* elements do not overlap.

The following pair of equations describes how the annihilator element $\boxtimes$ propagates from I/O Actions up to Synchronous Action elements:

Axiom 4.63 : Inconsistency Propagation:

$$(\boxtimes \,/\, \beta) = \boxtimes$$
$$(\alpha \,/\, \boxtimes) = \boxtimes$$

4.3.1 Causal Action Product

We can now *define* the *causal action product* $(\_ \star \_)$ on synchronous actions as follows:

Definition 4.64 : Causal Action Product

$$(\_ \star \_) : Act^2 \to Act$$

$$\boxtimes \star \gamma = \boxtimes = \gamma \star \boxtimes$$
$$\varepsilon \star \gamma = \boxtimes = \gamma \star \varepsilon$$
$$(\alpha_1/\beta_1) \star (\alpha_2/\beta_2) = ((\alpha_1 \oplus \alpha_2)/(\beta_1 \oplus \beta_2))$$

This causal action product is used to define *causal product* over synchronous I/O processes.

4.4 Representation using maps

In this section, we give non-trivial *concrete* instantiation of *pIOAct* which satisfies all requirements imposed by earlier subsections. The example is to take a primitive I/O action to be essentially a representation of *the state of a typed, named channel*. A typed channel, *ch* can be represented by a pair consisting of a Name, *nm* and a Type, *ty*. The state of channel *ch* can then be represented by the pairing of *ch* with its present value, *v*.

To do be precise, we shall make the following minimal assumptions concerning types and their values:

1. $TYPE \subseteq \mathbb{P}(VALUE)$
2. $\forall\, ty_1,\, ty_2 \in TYPE \bullet (ty_1 \cap ty_2) \in TYPE$
3. $\emptyset \in TYPE$

Generally, we shall use variables as follows:

$$
\begin{aligned}
v,\, v_1,\, v_2,\ \cdots &\in \mathit{VALUE} \\
ty,\, ty_1,\, ty_2,\ \cdots &\in \mathit{TYPE} \\
nm,\, nm_1,\, nm_2 &\in \\
\cdots &\in \mathit{Name}
\end{aligned}
$$

4.4.1 Primitive I/O actions

As indicated above, primitive I/O actions represent the state of typed, named channels:

Definition 4.65 : Primitive I/O action:

$$pa \in pIOAct \quad ::= \quad nm{:}\, ty \rightarrow v$$

We now define the primitive ordering and conflict-ordering relation:

$$(\_ \preccurlyeq_p \_),\ (\_ \,\mathsf{CF}_p\, \_) \ \subseteq\ pIOAct^2$$

defined by:

$$
\begin{aligned}
& nm_1{:}\, ty_1 \ \rightarrow\ v_1 \preccurlyeq nm_2{:}\, ty_2 \ \rightarrow\ v_2 \quad \Leftrightarrow \\
& \quad (nm_1 = nm_2) \ \wedge\ (ty_2 \ \subseteq\ ty_1) \ \wedge\ (v_1 = v_2)
\end{aligned}
$$

$$nm_1{:}\, ty_1 \ \rightarrow\ v_1 \ \mathsf{CF}\ nm_2{:}\, ty_2 \rightarrow\ v_2 \quad \Leftrightarrow \quad (nm_1 \neq nm_2)$$

Using these very simple definitions over channels, we can *generate* the appropriate structure over composite I/O actions, by taking an appropriate free construction to provide composite terms. We then extend the above relations on primitive actions to composite actions as follows:

The relation $(\_ \preccurlyeq \_) \subseteq IOAct^2$ is the least relation such that it:

- is a pre-ordering on *IOAct*
- extends $(\_ \preccurlyeq_p \_)$
- preserves existing LUB's with respect to $(\_ \preccurlyeq_p \_)$
- makes $(\_ \oplus \_)$ into a (weak) LUB operation on *IOAct*

When these properties are formalised appropriately, it can be shown that they can be characterised by a monotonic functional upon the complete lattice of binary relations on *IOAct*. By the Tarski-Knaster theorem, this functional has a least fixpoint, and so there exists such a least relation satisfying all the above properties.

We also need to extend the context freedom relation on primitive actions to composite actions. But this is straightforward, by using the triangle law and the above induced ordering with the desired monotonicity properties.

Using these relations, we can show the following as theorems, where $(\_ \simeq \_)$ is the induced congruence upon I/O actions:

Theorem 4.66 :

a. $nm{:}\,ty_1 \rightarrow v \oplus nm{:}\,ty_2 \rightarrow v \simeq nm{:}\,(ty_1 \cap ty_2) \rightarrow v$

b. $(v_1 \neq v_2) \Rightarrow nm{:}\,ty_1 \rightarrow v_1 \oplus nm{:}\,ty_2 \rightarrow v_2 \simeq \boxtimes$

To help show that this structure satisfies the Projection Axiom, we shall introduce some additional functions which determine the set of channel names associated with a given I/O action:

$$name\text{-}of : pIOAct \rightarrow Name$$
$$name\text{-}of(nm{:}\,ty \rightarrow v) = nm$$

$$Names : IOAct \rightarrow \mathbb{P}(Name)$$
$$Names(\alpha) \triangleq \{name\text{-}of(pa) \mid pa \in pIOAct \wedge pa \sqsubseteq \alpha\}$$

We can easily show the following, for $\alpha_1,\ \alpha_2 \not\simeq \boxtimes$:

Lemma 4.67 :

- $(\alpha_1 \oplus \alpha_2) \not\simeq \boxtimes \Rightarrow$
  $Names(\alpha_1 \oplus \alpha_2) = Names(\alpha_1) \cup Names(\alpha_2)$
- $\alpha_1 \simeq \alpha_2 \Rightarrow Names(\alpha_1) = Names(\alpha_2)$
- $\alpha_1 \sqsubseteq \alpha_2 \Rightarrow Names(\alpha_1) \subseteq Names(\alpha_2)$

Next, we show the following decomposition property, by induction:

Lemma 4.68 :

$$pa \sqsubseteq (\alpha_1 \oplus \alpha_2) \wedge name\text{-}of(pa) \notin Names(\alpha_1) \Rightarrow$$
$$pa \sqsubseteq \alpha_2$$

This is enough to show the Projection Property holds for this structure:

Theorem 4.69 : The Projection Property

$$
\begin{array}{l}
p \in pIOAct \wedge p \sqsubseteq (\alpha_1 \oplus \alpha_2) \not\simeq \boxtimes \Rightarrow \\
\quad (p \sqsubseteq \alpha_1) \vee \\
\quad\quad (p \sqsubseteq \alpha_2) \vee \\
\quad\quad\quad (\exists\, a, b \in PA \bullet \\
\quad\quad\quad\quad (a \sqsubseteq \alpha_1) \wedge (b \sqsubseteq \alpha_2) \wedge (p = (a \oplus b)) \\
\quad\quad\quad)
\end{array}
$$

4.4.2 Hiding operator

We briefly introduce the hiding operator at the mapping level by its action on primitive elements and then use the structure over composite I/O actions to extend the definition.

$$\begin{aligned}
&\textit{hide-prim-act} : \textit{pIOAct} \times \mathcal{F}(\textit{Name}) \to (\textit{pIOAct} \cup \{\boxtimes\})\\
&\textit{hide-prim-act}(\textit{pa}, \textit{nms}) =\\
&\quad \textbf{if } \textit{name-of}(\textit{nm}) \in \textit{nms} \textbf{ then } \boxtimes \textbf{ else } \textit{pa}
\end{aligned}$$

The function *hide-prim-act*(pa, nms) maps those primitive actions *pa* having names in *nms* to $\boxtimes$, but is otherwise the identity.

This is extended to composite I/O actions by the obvious structural inductive equations:

$$\begin{aligned}
&\textit{hide-io-act} : \textit{IOAct} \times \mathcal{F}(\textit{Name}) \to \textit{IOAct}\\
&\textit{hide-io-act}(\varepsilon, \textit{nms}) = \varepsilon\\
&\textit{hide-io-act}(\boxtimes, \textit{nms}) = \boxtimes\\
&\textit{hide-io-act}(\alpha_1 \oplus \alpha_2, \textit{nms}) = \textit{hide-io-act}(\alpha_1, \textit{nms}) \oplus \textit{hide-io-act}(\alpha_2, \textit{nms})\\
&\textit{hide-io-act}(\textit{pa}, \textit{nms}) = \textit{hide-prim-act}(\textit{pa}, \textit{nms})
\end{aligned}$$

We can show that the above axioms are consistent in the sense that there is a function satisfying them.

We can then use this to give hiding over synchronous actions as follows:

$$\begin{aligned}
&\textit{hide-act} : \textit{Act} \times \mathcal{F}(\textit{Name}) \to \textit{Act}\\
&\textit{hide-act}(\boxtimes, \textit{nms}) = \boxtimes\\
&\textit{hide-act}(\iota, \textit{nms}) = \iota\\
&\textit{hide-act}((\alpha \;/\; \beta), \textit{nms}) = (\textit{hide-act}(\alpha, \textit{nms}) \;/\; \textit{hide-act}(\beta, \textit{nms}))
\end{aligned}$$

4.4.3 Presentation

We now briefly present the instantiation of the general framework by the synchronous interpretation that has been developed here.

Actions : The sort of Actions in the framework (*Act*) is identified with the structure also called *Act* here.

Mutation relations : In this particular instantiation, there is precisely one mutation relation, defined by:

$$\begin{aligned}
&\textit{Hide} : \textit{Act} \times \mathcal{F}(\textit{Name}) \times \textit{Act} \to \mathbb{B}\\
&\textit{Hide}(\gamma, \textit{nms}, \gamma') \;\Leftrightarrow\; (\textit{hide-act}(\gamma, \textit{nms}) = \gamma')
\end{aligned}$$

Composition operators : In this particular instantiation, there is precisely one composition operator, defined by:

$$Cprd : Act^2 \rightarrow Act$$
$$Cprd(\gamma_1, \gamma_2) = (\gamma_1 \star \gamma_2)$$

Clearly, other instantiations could be developed which introduce further operators.

5 Conclusion

Our basic requirement was to provide a semantically *accurate* approach to the verification of systems written in ELLA. Due to the complexities introduced by data, a more symbolic approach is required which allows both the concurrent and data aspects to be described. We have attempted to focus on approaches which help to separate these forms of structure.

We have defined a general process algebraic framework which can incorporate data type specification frameworks. We have done this by formulating requirements for algebraic notions of 'action'. Rather than concentrating complexity within the process framework, this allows a better separation of concerns between the state control structure and functional value framework.

Future work involves understanding existing attempts by Hennessy, Milner and other to develop full value-passing forms of process algebra [Pon92, Win89, M.H91]. We hope to do so by addressing fundamental concerns connected to practical verification, such as the bundling of actions to further separate data and control.

6 References

References

[BGL$^+$91] H. Barringer, G. Gough, T. Longshaw, B. Monahan, M. Peim, and A. Williams. Semantics and verification for Boolean Kernel ELLA using IO automata. In *Advanced Research Workshop on Correct Hardware Design Methodologies (CHARME'91)*, Turin, Italy, May 1991.

[BGMW92] H. Barringer, G. Gough, B. Monahan, and A. Williams. A semantic framework for kernel ella. Technical Report D2.3a, Formal Verification Support for Ella, IED project 4/1/1357, September 1992.

[BGMW93] H. Barringer, G. Gough, B. Monahan, and A. Williams. A Semantics for CORE ELLA. Technical Report IED 4/1/1357 Deliverable, University of Manchester, May 1993.

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Inf. and Control*, 60:109–137, 1984.

[Hil91] M G Hill. Foundational transformations in kernel ella. Technical Report D1.2a, Formal Verification Support for Ella, IED project 4/1/1357, September 1991.

[HM92] M.G. Hill and J.D. Morison. Semantics and basic laws of ella. Technical Report D1.2, Formal Verification Support for Ella, IED project 4/1/1357, May 1992.

[LT88] N.A. Lynch and M.R. Tuttle. An Introduction to Input/Output Automata. Technical Report MIT/LCS/TM-373, MIT, November 1988.

[MDED91] University of Manchester and RSRE Malvern DRA Electronics Division. Semantics and Verification Foundations for ELLA. Technical Report IED 4/1/1357 Deliverable D1.1+D2.1, May 1991.

[M.H91] M.Hennessy and A.Ingolfsdottir. A theory of communicating processes with value-passing. Technical Report (to appear in *Information and Computation* 7/91, University of Sussex, August 1991.

[Mil82] R. Milner. On relating synchrony and asynchrony. *TCS*, 1982.

[Mil89a] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer Verlag, 1989.

[Mil89b] R. Milner. *Concurrency and Communication*. Prentice Hall, 1989.

[Pon92] Alban Ponse. *Process Algebras with Data*. PhD thesis, 1992.

[Win84] Glynn Winskel. Synchronisation trees. *TCS*, 34:33 – 82, 1984.

[Win89] Glynn Winskel. A compositional proof system on a category of labelled transition systems. Technical Report DAIMI PB-294, Computer Science Dept., Aarhus University, November 1989.

[W.P89] W.P.Weijland. The Algebra of Synchronous Processes. *Fund. Inf.*, 12:139 – 162, 1989.