An Interpretation of Cognitive Theory in Concurrency Theory (Long Version)*

Howard Bowman†

Computing Laboratory, University of Kent at Canterbury,
Canterbury, Kent, CT2 7NF, United Kingdom

Email: H.Bowman@ukc.ac.uk

WWW: http://www.cs.ukc.ac.uk/people/staff/hb5/

Abstract

Theories of concurrent systems have been extensively investigated in the computer science domain. However, these theories are very general in nature and hence, we would argue, are applicable to many disciplines in which concurrency arises. Furthermore, a number of existing theories of cognitive science are formulated in terms of concurrent subsystems interacting in solving cognitive tasks. In this paper we investigate the application of a (process calculi based) concurrency theory to modelling such a (concurrent) cognitive theory. The cognitive theory chosen is ICS (Interacting Cognitive Subsystems), which we interpret using our process calculus and then we verify some simple behavioural properties of the resulting interpretation. These properties concern the capabilities of the cognitive system to realise deictic reference.

1 Introduction

Concurrency Theory. In a very broad sense, the history of computer science can be characterised by two, often competing, views of how to build and understand computer systems: the *engineering* view and the *mathematical* view. According to the former, computer systems should be viewed as engineering artefacts which can be constructed using an "informal" process of design, implementation and testing. In contrast, the latter view advocates that formal, mathematically based, theories of computing should be developed. This brings

*Reference - Technical Report University of Kent at Canterbury, Number 8-98, July 1998.

†Howard Bowman is currently on leave at VERIMAG, Centre Equation, 2 rue Vignate, 38610 GIERES, France and CNR-Istituto CNUCE, Via S. Maria 36, 56126 - Pisa - Italy with the support of the European Commission's TMR programme.

a number of potential benefits, not least of which is that computer systems can be constructed in a more rigorous fashion leading to higher quality finished systems containing fewer errors.

In this paper we will be interested in this second view. In particular, we will consider techniques which can broadly be categorised as *formal methods* [26]. These provide notations for describing systems - *formal specification languages* and analysing these specifications - *formal verification techniques*. From within formal methods we will particularly focus on a set of techniques that have been developed as *mathematical models of concurrent systems*.

The majority of work on mathematical theories of computing has focused on systems, which can be categorised as *sequential* i.e. systems containing a single component which evolves by performing a sequence of operations one after the other. Such systems can typically be viewed as input to output transformers, i.e. given a set of input values the system generates a set of output values and importantly, such transformers can be mathematically viewed as functions (from inputs to outputs). In line with [34], we call such systems *transformational systems*. Although perfectly adequate in the sequential setting, such transformational interpretations have proved insufficient in the concurrent setting. Concurrency theory has sought to respond to this problem.

Concurrency theory studies systems containing a number of components that are evolving simultaneously. Such systems arise throughout computer science. To take a familiar example, the Internet is a concurrent system containing many thousands or even millions of components. In concurrent systems, the default behaviour of each component is to evolve completely independently of all other components[1]. However, components also *interact* with one another in performing certain tasks.

As suggested earlier, with transformational systems the key issue is what results the computation terminates with. With concurrent systems this is no longer the case. Think again of our example of the Internet, firstly, it is not clear whether the system will ever terminate and secondly, the outputs produced when it terminates (perhaps in the year 2000) are likely to be degenerate rather than useful. The interesting aspect of concurrent systems is rather their ongoing behaviour and how components respond to external stimuli throughout the system's life-time[2]. Thus, we will typically model concurrent systems in terms of the order in which they can perform external interactions.

Perhaps the earliest work in concurrency theory was that in the 1960's by Carl Petri, which yielded Petri Nets [45]. However, it was in the 1980's that the field reached maturity. This was inspired by Tony Hoare's development of Communicating Sequential Processes (CSP) [28, 27] and Robin Milner's development of the Calculus of Communicating Systems (CCS) [36, 37, 38]. A wealth of techniques resulted from this 80's renaissance, e.g. communicating automata [23], further Petri Nets research [45], Temporal Logics [34], however, here we

[1] In fact, there is a distinction between, so called *synchronous* and *asynchronous* concurrent systems and what we discuss here is most relevant to asynchronous systems.

[2] This interpretation has prompted the so called *reactive* [34] view of computing.

will focus on the approach most closely inspired by Hoare and Milner - Process Calculi[3]. We will describe this approach in section 3.

Although concurrency theory was developed with computer applications in mind the core concepts of concurrency theory are completely general and are thus applicable to modelling any variety of concurrent system. Although rare, some applications outside mainstream computer science have been made, e.g. to biological systems [51] and in Physics [22]. Thus, one objective of this paper is to publicise the concurrency theory techniques to a wider audience. In doing this we will focus on applying concurrency theory in the area of cognitive modelling.

Cognitive Theory. From very early in the history of computer science analogies between the mind and the digital computer were used in explaining human cognition, e.g. [12]. The resulting *information processing* paradigm [31] has come to dominate cognitive psychology, with many diverse computational explanations of cognitive behaviour, e.g. semantic networks [14], production systems [42] and connectionist models [48, 49]. Furthermore, some of these approaches are argued to be "Unified Theories" in the sense that they propose a "single system of mechanisms that operate together to produce the full range of human cognition" [41], e.g. the Soar Architecture [41]. However importantly, many of these cognitive theories are concurrent in nature, for example, Soar contains elements of concurrent behaviour [41].

In this paper we will focus on a particular "general purpose" cognitive theory called Interacting Cognitive Subsystems (ICS) [1, 3, 2]. This is a theory of working memory developed at the (UK) Medical Research Councils Applied Psychology Unit. The reasons for choosing this theory are two fold. Firstly, the model is highly concurrent in nature and is thus particularly amenable to the concurrency theory techniques we have in mind. Secondly, there has been previous work [18, 17, 19, 21] (in the domain of Human Computer Interaction) on modelling ICS with formal methods. This previous work has been called *syndetic modelling*; in it, both cognitive behaviour and the computer interface are modelled using the same formal method. Then properties of the composed system are determined.

ICS adopts a "top down" approach to the design of a cognitive theory by providing a framework containing a set of core components and mechanisms that it is argued give a "potential design of a complete mental mechanism" [2]. It describes cognitive tasks in terms of multiple flows of representations, which compete for cognitive resources. We introduce ICS in section 2.

Interpreting Cognitive Theory in Concurrency Theory. In illustrating the central tenet of this paper - that concurrency theory can be used to model and analyse cognitive theories - we will interpret ICS using a simple process calculus. Then we will analyse our specification of ICS to identify a number of simple behavioural properties. These properties focus on the capabilities of the system to realise certain forms of deictic reference. Typical forms of deictic reference that we consider are the capability of a computer system user to select

[3]The term Process Algebra is also often used.

from a list of items in a computer display while performing some other task, e.g. speaking.

An obvious question that arises is: why apply a (concurrent) formal method to cognitive modelling? We would argue that the formal approach brings two main benefits:

- *Rigorous Analysis.* Perhaps the most powerful argument for adopting formal methods in computing is that they yield a description of the system which is amenable to rigorous analysis, e.g. demonstrating that your system does not enter certain degenerate states, such as deadlock states. This analysis can take one of two general forms - *verification through proof* or *automated state space analysis*. Under the former an, often hand crafted, series of formal steps are exhibited which show that a particular property can be derived from the formal specification of the system. In contrast, in the latter case a tool is used which automatically analyses the state space of a specification to determine whether a certain property holds. The analysis we perform in section 3 will contain elements of both these approaches.

  Now how does the potential for rigorous analysis fit with cognitive science. Well, the standard cognitive science approach is to attempt to mimic cognitive behaviour by writing a computer program which when run simulates the particular cognitive task being considered. A rather obvious observation about such simulation is that it can never be exhaustive. In fact, in practice, such techniques will only be able to explore a small subset of the complete system behaviour[4]. One reason for the inexhaustiveness of simulation is that the behaviour of any non-trivial system will be infinite in some respect, e.g. through using infinite data sets or exhibiting infinite concurrent behaviour. Although, it would be wrong to over emphasize the power of formal methods in resolving this problem, in particular formal methods have their own constraining boundaries (e.g. the state explosion problem), they do at least offer the potential of a more exhaustive investigation of the behaviour of cognitive theories. In particular, infinite behaviour can be handled using inductive and co-inductive techniques [53].

  It is also important to note that formal methods do not preclude simulation, since tool suites for such methods typically allow specifications to be run using simulation and animation packages. These runs can either be step by step explorations of the state space or automatic generations of a single path. Such a tool will be used in section 7 where we will in fact use a mixture of formal reasoning and simulation techniques.

  Finally, it is also worth pointing out that tools are now available which enable programs which are executable in a traditional sense to be generated from formal specifications. For example, the TOPO tool generates C code from process calculi specifications [33].

[4] In this respect simulation is like testing, which as Djkstra so pertinently pointed out, can only demonstrate the presence of errors, but can never demonstrate the absence of errors.

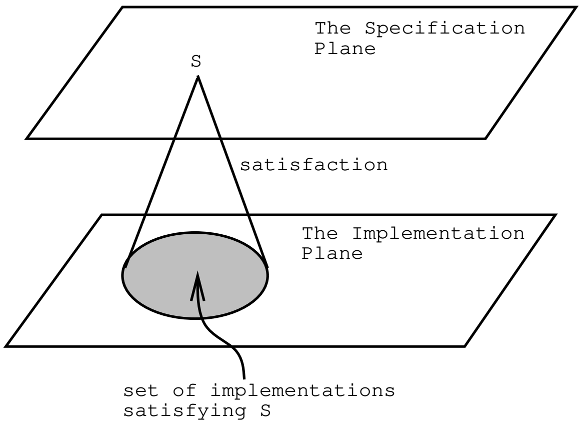

Figure 1: The Nature of Formal Specification

- *Avoiding Overspecification.* It is important to realise that formal specifications are in nature very different to computer programs or what we will more broadly call implementations. A formal specification is abstract in the sense that it characterises a set of possible implementations, while a program characterises a single implementation - itself. A satisfaction relation is typically used to associate sets of implementations with specifications, i.e. a specification characterises the set of implementations that satisfy it. This situation is shown in figure 1.

  Associated with this aspect is the desire not to overspecify (or in other terms to provide *loose specification*), i.e. that the nature of the specification language should not force the specifier to rule out acceptable implementations. Formal methods typically use the notion of non-determinism in order to realise such loose specification. We will introduce non-determinism in section 3.

  We believe this feature of formal methods is potentially very useful in the cognitive setting. In particular, many of the more general theories (as indeed ICS is) leave much unexplained since a complete mechanistic interpretation of cognitive behaviour is often not available. A difficult problem this raises is how to provide "executable" realizations of cognitive theories such as ICS which do not overprescribe what is known. We believe this avoidance of overprescription has much in common with avoidance of overspecification.

  Thus, in the sequel we will attempt to define the "strongest known" constraint about ICS. This will characterise many possible actual cognitive behaviours. Then we will see what properties we can determine of such a loosely specified cognitive theory.

An associated point is the so called *irrelevant specification problem* [46, 41]. In order to construct a working simulation program a large number of assumptions have to be made, leaving it unclear what aspect of the behaviour of the program corresponds to known cognitive behaviour and what arises from expediency. We would argue that the abstract nature of formal methods enable cognitive systems to be described in a manner that is more likely to avoid the irrelevant specification problem.

The Paper. We have attempted to make this article as stand-alone as possible. In particular, one of our goals has been to make it accessible to the wider cognitive psychology field. The prerequisites that we assume are a basic knowledge of discrete mathematics, in particular, set theory and logic and some broad knowledge of computer science. The theory that is not generally accessible we have placed in the appendix.

The paper is structured as follows. Section 2 describes the basic ICS model. Section 3 introduces the LOTOS specification notation that we use to describe ICS. Section 4 presents a complete LOTOS specification of ICS. Section 5 formulates a number of behaviour goals over ICS. These concern its capability to perform certain variaties of deictic reference. The goals are formulated in an interval temporal logic called Mexitl which we also introduce. Section 6 verifies the main negative goals that we consider. Section 7 presents simulation based validation of our positive goals. Then section 8 presents some concluding remarks and discusses further work. Section 9 is the appendix.

2 Interacting Cognitive Subsystems

As pointed out previously, ICS is a "general purpose" cognitive theory, which exhibits highly concurrent behaviour. It has been applied in a number of areas, e.g. performance of short term memory tasks [1], HCI [18, 19, 21], experimentation with theories of depression [4, 50].

It is beyond the scope of this paper to give a complete introduction to the ICS model, the interested reader is refered to [2]. What we present here follows in many places the exposition in [2], but in a summarised form. In particular, our presentation will largely concentrate on the control/behavioural aspects of ICS, while we will only give a cursory exposition of the data aspects of the model.

Subsection 2.1 works through the basic elements of the model, while subsection 2.2 presents some sample scenarios of how particular cognitive tasks can be performed in ICS. We finish the section with a summary and discussion subsection 2.3.

2.1 The Model

Information Flows and Representations. The basic "data" items found in ICS are *representations*. This term embraces all forms of mental codes,

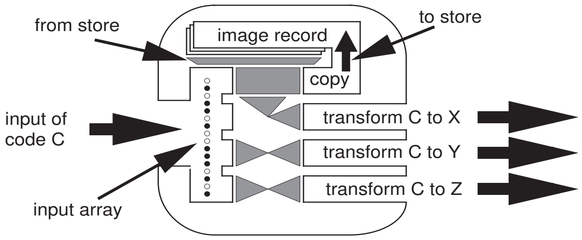

Figure 2: General Subsystem Format

from "patterns of shapes and colour" as found in visual sensory systems; to encodings of "force, target positions and of articulatory musculatures" as found in articulatory effectors; to "descriptions of entities and relationships in semantic space" as found in a semantic subsystem [2].

These representations are past amongst the components of the architecture, being transformed from one code to another in each component. Thus, the architecture can be seen as an *information flow* model, where multiple flows of representations (e.g. one from the acoustic sensory system and one from the visual sensory system) are relayed through the architecture and compete for resources. Thus, we have a model containing *data* - representations and *control* - paths by which representations can flow around the architecture.

Subsystems. The components of the architecture are called *subsystems* and all subsystems have the same general format, which is shown in figure 2. Subsystems are representation transformers - they input representations in a particular code, apply transformations to them and output them in a different code. In addition to this transformation function, they copy the representations that they input into a local record. This record can be accessed when applying future transformations.

Each component (subsystem) itself contains components:-

- *Input Array.* Representations received by a subsystem are placed into an input array. The input array must be able to handle multiple representations, since a subsystem can accept representations from multiple sources at any one time. We will return to this issue shortly.

- *Image Record.* The image record preserves the history of a subsystem. In order to do this, whenever the input array has new contents the subsystem copies them to the image record.

- *Transformations.* Each subsystem contains a set of independently evolving transformations[5]. Transformations take representations from the input

[5]In some formulations of the ICS model these transformations are refered to as processes. However, we will avoid this term, since it has a very specific meaning in process calculi.

array, apply some transformational operations to them and then relay a new (transformed) representation to a target subsystem. Transformations will typically change the code of the representation from the code of the current subsystem to the code of the target subsystem, i.e. from C to X, Y or Z in figure 2.

The Overall Architecture. The complete ICS architecture, which is shown in figure 3, is composed of nine subsystems, each of which is a specialization of the general subsystem format just highlighted. These subsystems evolve independently and concurrently to one another, subject to communication of representations when firing transformations. This communication takes place over a conceptual *data network*.

Subsystems fall into one of three categories: *sensory subsystems*: acoustic, visual and body state; *effector subsystems*: articulatory and limb; and *central subsystems*: morphonolexical, propositional, implicational and object. Sensory subsystems receive input from sensory systems, such as the auditory system; effector subsystems transmit outputs that control effector systems such as the limbs; while central subsystems perform internal cognitive processing. We consider each subsystem in turn (our presentation here is particularly closely derived from that to be found in [2][6].

- **Sensory**
  - *Acoustic* (`AC`) - receives representations from the auditory system encoding sound frequency (pitch), rhythm, intensity, etc;
  - *Visual* (`VIS`) - receives representations from the eyes encoding "patterns of shapes and colour", i.e. light wavelength (hue) and brightness;
  - *Body State* (`BS`) - receives representations encoding body sensations of pressure, pain, positions of parts of the body etc.
- **Effector**
  - *Articulatory* (`ART`) - outputs representations that control the force, target positions and timing of articulatory muscalatures, i.e. performs subvocal speech rehearsal;
  - *Limb* (`LIM`) - outputs representations that control the force, target positions and timing of skeletal muscalatures, i.e. initiates physical movement;
- **Central**
  - *Morphonolexical* (`MPL`) - works with an abstract structural description of entities and relationships in sound space, i.e. lexical identities of words, their status and order;

[6]Two additional subsystems: `SOM` and `VISC` are also considered in [2], but these will play no role in our analysis, so we will not consider them here.

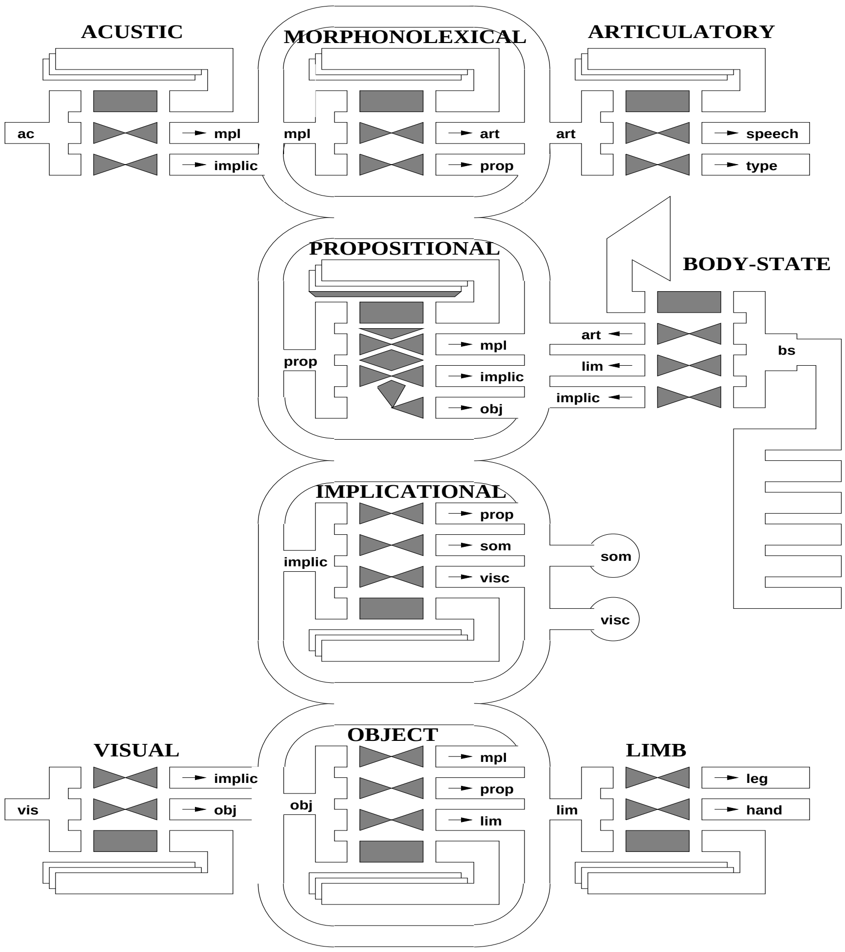

Figure 3: The ICS Architecture

– *Object* (OBJ) - works with an abstract structural description of entities and relationships in visual space, e.g. attributes of objects: shape and relative position;
– *Propositional* (PROP) - works with descriptions of entities and relationships in semantic space, i.e. gives semantic meaning to entities and highlights the semantic relationships between entities;
– *Implicational* (IMPLIC) - works with abstract descriptions of human existential space and holistic meaning;

The possible transformations between subsystems are shown in figure 3.

Configurations. Not all transformations will be involved in every cognitive task. For example, language understanding and production requires a different set of transformations to be active than say a perceptual-motor tracking task [2]. Thus, the concept of a (mental) configuration is introduced. Each configuration is associated with a particular cognitive task and is defined to be the set of subsystems and transformations involved in that task. Although, transformations that are not included in the configuration may still be passing representations, what they transmit is not relevant to the cognitive task at hand.

Multiple Flows and Blending. The "information flow" nature of the model should be becoming clear. Sources of flows are typically sensory subsystems, which receive sequences of representations from the external environment. Each representation is then relayed within the architecture by the firing of transformations. Thus, each sequence of representations received at each sensory subsystem generates a flow of representations around the architecture. Clearly, multiple flows can exist in the architecture at the same time. Thus, competition between flows for resources becomes a major element of the functioning of the system. This is one reason why the architecture has been viewed as so useful in the HCI setting where multi-modal human computer interfaces, which naturally generate multiple flows of input at multiple sensory subsystems, are being considered [44].

The existence of multiple flows also prompts the question of what happens if a subsystem is taking input from two different flows at the same time. This possibility arises in a number of subsystems, e.g. the implicational subsystem can receive from all the effector subsystems - acoustic, visual and body state. In fact, the architecture accommodates a number of different outcomes when multiple flows are received. In some configurations different flows may be kept quite separate, being placed into the same input array, but then being relayed via separate transformations.

However, the more interesting outcome is if a configuration requires an output transformation to use a representation which is a combination of two (or more) "competing" input representations. This possibility leads to the concept of *blending*.

Representations from different flows can be blended to create a composite representation. However, this can only happen if the two representations are *consistent*. Consistency between representations is not a precisely defined concept, although an indepth discussion of it is given in [2]. We can illustrate the concept with a simple example. Representations of a scene from visual subsystem (or in fact from any subsystem) can be thought of as having a *psychological subject*, informally, the object we are looking at in the scene, and a *psychological predicate*. Informally, the relationship between other basic units in the scene. Now if two representations are received from different flows and they reference different psychological subjects then they might be viewed as inconsistent and hence, would not be blendable [2].

Much of the richness of the architecture arises through this possibility to blend competing flows. In particular, constraints on allowed cognitive processing can be formulated in terms of blending, e.g. that representations are only blendable if they all exhibit the same psychological subject. We will see an example of such a constraint in section 4.

Stability. A further important concept in the architecture is that of stability. This is a property of information flows which gives one characterisation of the "quality" of the flow. It characterises the level of variability over time of representations in the flow. For example, in a constant visual environment it is likely that the flow of representations through the visual subsystem is very constant and thus, stable, i.e. there is little variability betwen adjacent and closely proximate representations. In contrast, in a rapidly changing visual environment there may be extreme variability between closely proximate representations and thus, the resulting flow is likely to be unstable. We treate stability as a derived concept which can be applied to any subsystem in a configuration. It is determined by observing the variability of data representations entering or leaving that subsystem over a period of time.

Buffering. Another aspect of the architecture is that in certain circumstances a subsystem can enter, so called *buffered mode*. In this mode a transformation in the subsystem switches from working directly on representations in the input array, to taking representations from the image record. This corresponds to "focal awareness" of an information flow [35]. The transformation to code X in figure 2 is in buffered mode.

In buffered mode copying from the image record and applying the buffered transformation become serial activities (in normal mode they are concurrent activities), allowing the transformation to access a representation that is "extended" in time [35]. This, in particular, enables the processing of representations from past experience.

When a transformation is buffered it *selects* representations from the image record. This mechanism of selection can take a number of forms, for example, it may make a comparison between what is in the input array and items in the image record using some aspect of the input array as a "key", e.g. the psychological subject of a representation.

The architecture imposes an important constraint on the process of buffering: only one transformation in a subsystem is allowed to be buffered at any given time. Furthermore, only one subsystem is allowed to be buffered.

Synchronous Evolution. In this paper we take a particular view of the concurrent behaviour of ICS. There are really two assumptions, the first of which is a necessary prerequisite for the second:

1. *Discreteness.* We assume the system takes discrete steps. This follows from the view that an information flow is a sequence of (discrete) representations.

2. *Synchrony.* We assume that subsystems takes steps together synchronously. Consequently, we can distinguish between two kinds of steps:

    - *Primitive Steps.* These are steps by ICS subsystems.
    - *Global Steps.* These are steps by the entire (or global) system. Such global steps comprise a set of primitive steps, typically, one (or more) for each subsystem and the overall system behaviour is a sequence of global steps.

For ICS, the firing of a transformation represents a primitive step and a global step collects together the firings of all transformations in the current configuration. Thus, all possible firings must complete before a new global step is started.

Although, such a synchrony assumption is not discussed in the standard introductions to ICS, e.g. [2], we believe that the resulting external behaviour of ICS faithfully reflects the spirit of the model.

2.2 ICS Scenarios

We consider how three cognitive tasks would be performed in ICS:-

- reading;
- speaking/pronounciation; and
- pointing.

each of which yields a particular configuration of the ICS system. Then we build a configuration which combines all these subconfigurations. This is the ICS configuration for deictic reference which will be used in section 6 and 7. Our discussion in this section mirrors that given in [21].

2.2.1 The Reading Configuration

The reading configuration is called $conf_{read}$ and is shown in figure 4. It inputs a flow of sensory information (which is assumed to be amenable to a reading interpretation, i.e. it can be decoded in to lexical items) at the `VIS` subsystem and then interprets it. The goal of this interpretation process is to associate semantic meaning to the lexical items contained in the representations received at `VIS`. In order to do this a number of subsystems have be be used:

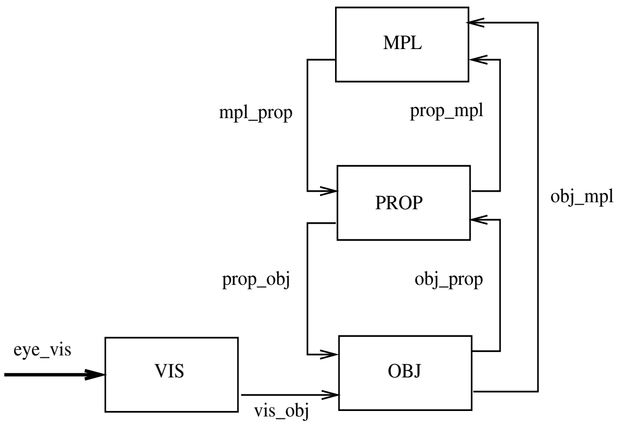

Figure 4: The Reading Configuration

- **VIS** is obviously used, it acts on representations received from the eyes, and generates object representations on `vis_obj`. These structure the sensory representations into a collection of visual objects and their relationships.
- **OBJ** interprets representations in object code using two output transformations: `obj_prop` and `obj_mpl`. The first of which associates semantic meaning to objects and relationships in the object representation. The second transforms the object code into a "structured representation of sound" which is input by the morphonolexical subsystem. This transformation is required in order that the items being viewed can be given a lexical interpretation.
- **PROP** receives representations in propositional code which express the semantic meaning of objects and relationships between objects. It transforms these representations using two output transformations: `prop_obj` and `prop_mpl` which feed semantic information back to the object and morphonolexical subsystems respectively. These feedback transformations enable the flows passing through `OBJ` (respectively `MPL`) to be interpreted in a more refined way using the semantic information determined at `PROP`.

Three of the subsystems in $conf_{reading}$ input from multiple sources: `OBJ`, `PROP` and `MPL`, each of which blends the two input flows it receives. The blendings at `MPL` and at `OBJ` are conceptually similar: they both blend the representations received from `PROP` in order to enrich the other flow received with extra semantic information, e.g. `prop_obj` enriches the flow from `vis_obj`.

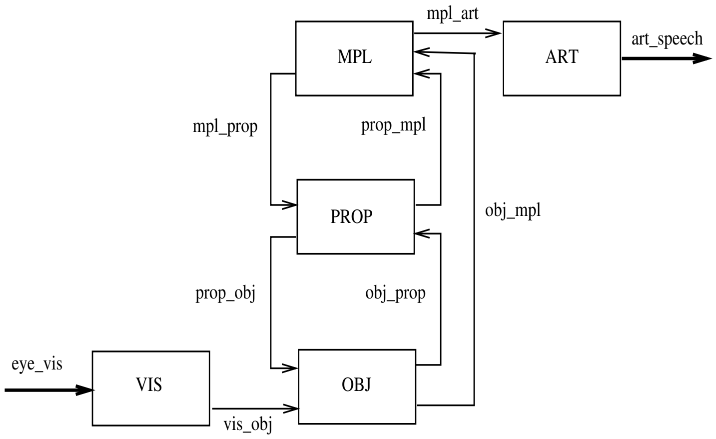

Figure 5: The Pronounciation Configuration

2.2.2 The Pronounciation Configuration

The speech/pronounciation configuration, see figure 5, called $conf_{pron}$, is obtained from $conf_{read}$ by adding the transformations `mpl_art` and `art_speech`. This enhancement enables the morphonolexical code received in the reading configuration by `MPL` to be interpreted as articulatory codes and output at `art_speech`.

2.2.3 The Pointing Configuration

The activity of pointing requires the involvement of a number of subsystems:

- **VIS** to observe the object to be pointed at;
- **OBJ** and **PROP** to interpret the representation received at `VIS`;
- **BS** to give information about the position of (for example) the hand;
- **LIM** to generate code for moving the hand in the required direction.

The configuration that results is shown in figure 6.

2.2.4 The Deixis Configuration

The configuration for deictic reference combines the configurations just presented. It is called $conf_{deixis}$ and is shown in figure 7. We incorporate body state feedback into the configuration. This will play an important role in the goals that we formulate in section 5. Intuitively, the `bs_lim` transformation provides feedback on the current position of the limbs, which enables future

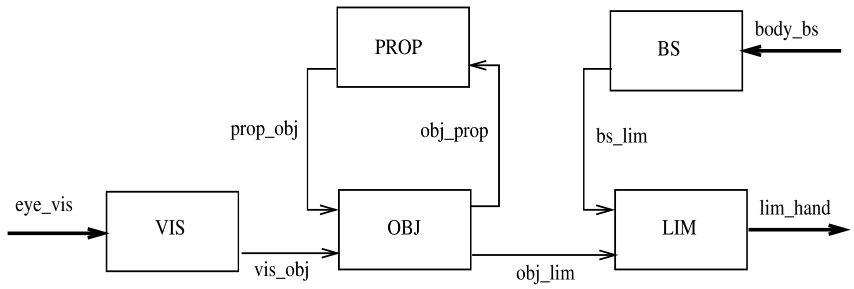

Figure 6: The Pointing Configuration

`lim_hand` transformations to be formulated correctly relative to the current hand position. The `bs_art` transformation behaves similarly, enabling speech to be formulated according to the current state of the mouth and vocal chords.

2.3 Discussion

This completes our exposition of ICS. It will be clear to the reader that the model is both extensive and general. Consequently, when "realizing" the architecture, as we will do in the next sections, we will not be able to capture its full generality. A good example of this is in the modelling of representations which we will do in a rather primitive manner. This is partially because formalizing such representations is a data modelling problem and here we wish to concentrate on issues of concurrent behaviour. However, although we do make simplifications, we believe that the realization we present faithfully captures the spirit and essential elements of ICS. We document simplifications whenever they arise.

3 Concurrency Theory

A complete theory of concurrency is quite complex with a number of components, e.g.,

- *A Specification Language* - or perhaps we should, more broadly, say a specification *notation* since graphical (non-language notations) can also be found, e.g. StateCharts [23]. However, what we will discuss here is a language in the classical sense, i.e. a syntax for writing descriptions of systems.
- *A Semantics* - Specification languages are at the user level: users describe their systems with them. However, it is typically more straightforward to formulate the mathematical properties of specifications at a lower level, e.g. in terms of the execution traces of a specification. Such lower level

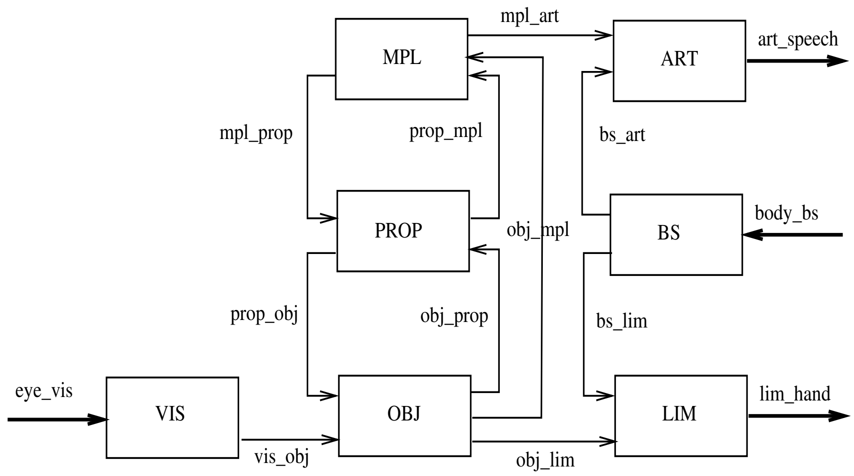

Figure 7: The Deixis Configuration

mathematical models are called *semantic models* and a mapping, called a *semantic map* defines the semantic model that corresponds to a particular specification [53].

- *Relations* - Specifications can be related by comparing their semantic interpretations; many such relations exist. In particular, equivalence relations can be identified [38, 52], which play the role of identity in formal theories, in the same way that = does in the theory of numbers.

Since we will not be using any heavy weight theory, we will not generally be concerned with the last two of these, rather almost all of what we present will be at the specification language level; the interested reader is refered to [6, 38, 47, 52] for further information on the other levels.

We will use a process calculus specification language, called LOTOS: Language of Temporal Ordering Specification [5] (we will also consider a specification logic in section 5). The choice of this method over, say, CSP [27] or CCS [38] is largely pragmatic: we have more experience with LOTOS and there are a number of tools available which are well suited to the analysis we will perform.

LOTOS is really two languages - a language for describing concurrent behaviour and a data language. The former is used to specify the order in which steps are made and the latter is used to describe data types associated with these steps.

The data language is an algebraic specificaton language; ACT-ONE [15][7], which allows data types, such as natural numbers, booleans, queues, tuples, etc to be defined. In the body of the paper we will restrict ourselves to informal descriptions of these data types; the appendix contains full definitions.

We will not use all the LOTOS language and thus we will only introduce the parts we need. In addition, we simplify some of the LOTOS syntax. The sublanguage that we use is formally presented in the appendix (section 9.1). For a full introduction the interested reader is refered to [5, 6]. In the following subsections, we slowly build up the language using fragments of our ICS specification for illustration.

3.1 The Nature of LOTOS Specification

Avoidance of overspecification, as discussed in section 1, is at the heart of process calculi. In particular, it is important that the correct interpretation is imposed on LOTOS descriptions: they express the "possible external behaviour" of a system. Specifications should be viewed as *black boxes*; they describe the order of possible external interaction, but do not prescribe how that interaction order is internally realised. Any physical system that realises the external behaviour is a satisfactory implementation.

The concept of the *environment* that a specification evolves in is crucial in obtaining this interpretation. The term environment refers to the behaviour that the *external observer* of a system wishes to perform. Note that this external observer could be either human or mechanical. Conceptually, a LOTOS specification only defines "possibilities" for evolution of a system and it is through interaction with a particular environment that these possibilities are resolved and realised. For example, if an environment cannot offer an action that a specification *must* perform deadlock will ensue. A deadlock is a state from which the system is unable to evolve.

As an illustration, we might view a LOTOS specification of ICS in the form depicted in figure 8, i.e. as a black box with interaction points, `eye_vis`, `lim_leg`, `lim_hand`, `body_bs`, `art_speech`, `art_hand` and `ear_ac`. Such interaction points are called *gates* (the term *port* is also sometimes used). It is only through these gates that an external observer can interact with the system.

Gates reference locations at which interactions can take place. At such gates *actions* are performed. These can be thought of as interaction activities, e.g. passing a value, sending a message or pressing a button[8]. In fact, the latter of

[7] In fact, the choice of ACT-ONE as the LOTOS data language has not proved completely successful [13] and in the current revision of the language [30] an alternative data language is being proposed.

[8] An important theoretical aspect of actions is that they are *atomic*, i.e. they cannot be divided in time. Consequently no two actions can occur at the same time and, thus, the occurrence of actions cannot overlap. For example, performing an action at `vis_obj` and at `eye_vis` cannot happen at the same time. The atomicity of actions has important consequences for the modelling of concurrency, see for example [38, 6]. However, the restriction to atomic actions does not limit expressiveness, since non-atomic activities can be specified in terms of the actions that delimit the activity, i.e. rather than defining an action which has duration we

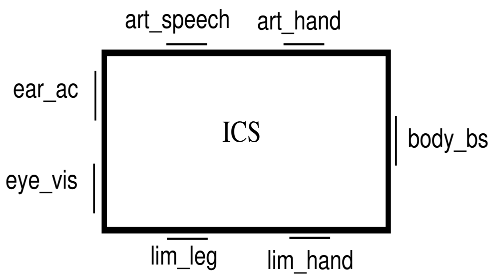

Figure 8: Black Box Interpretation of a LOTOS Specification

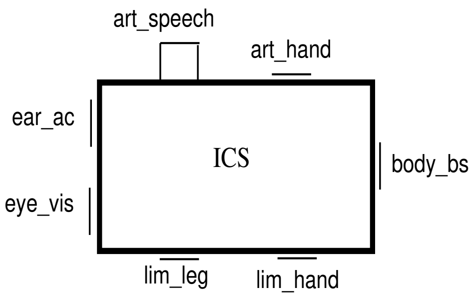

Figure 9: Action Offering as Buttons Popping Up

these yields a nice pictorial representation of interaction between environment and specification. LOTOS descriptions define the order in which actions can be offered at gates. Thus typically, actions are only offered intermitently at gates. We can view the offering of an action to the environment as the popping up of a button. For example, figure 9 depicts the situation in which an action is offered at `art_speech`, but not at any other gate. The environment can decide to push `art_speech` or to leave it unpushed. We could also have situations such as that depicted in figure 10 where both `art_speech` and `art_hand` are up and the external observer has a choice of actions to perform.

Actions come in two forms: *basic actions* and *data passing actions*. The former are unadorned gates. In ICS two such actions will be:

`tick` and `vis_buffered`

The first is used to denote a clock tick and the latter is offered to the environment when deciding whether a subsystem, here `VIS`, should enter buffered

can specify the atomic instant at which the activity starts and the atomic instant at which it stops.

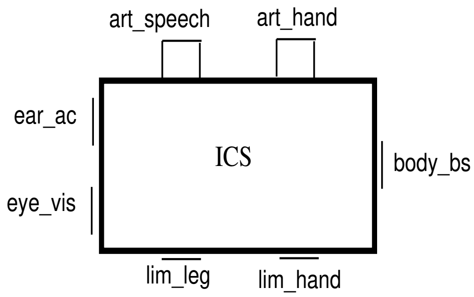

Figure 10: Choice of Action Offers

mode.

In contrast, data passing actions comprise a location for that interaction - a gate and a data passing attribute that is associated with performing the action. This attribute can either correspond to outputting or inputting a value. For example, we could have outputting actions:

`vis_obj!1` and `obj_mpl!2`

where `vis_obj` and `obj_mpl` are gates and `1` and `2` are representations (we will model representations as natural numbers, since a richer model, such as the super/subordinate mechanism described in [2] is not required for our purposes in this paper). Thus, an action of the general form:

```
g!E
```

denotes an output of the value of the (data) expression `E` on the gate `g`. Input actions can also be found in ICS, e.g.

`eye_vis?r1:Rep` and `vis_obj?r2:Rep`

where `eye_vis` and `vis_obj` are gates and `r1` and `r2` are variables of type `Rep`, i.e. representations. The effect of an input action is to receive a value on a gate and associate that value (more precisely bind it) to a variable, here `r1` (and `r2`). Thus, an action of the general form:

```
g?v:T
```

denotes an input of a value on gate `g`, which is bound to a variable `v` of type `T`.

Now importantly, two *complementary* actions can co-operate in performing an action (in precise terms they synchronise). For example, if,

```
vis_obj!1
```

is offered by one subsystem and,

```
vis_obj?r2:Rep
```

is offered by another subsystem, the two actions could be performed together. This is because they are complementary in the sense that they both take place at the same gate, `vis_obj`, and the first outputs a value, while the second inputs a value of the same type.

A special distinguished action, `i`, is also used; it denotes an *internal action*, i.e. an action that is hidden from the external observer. The occurrence of an internal action is not externally visible, thus, conceptually, no button is raised when it is offered or pushed when it is performed. It is important to note though that while an `i` action is not externally visible it may "indirectly" affect behaviour that is externally visible. Typically an `i` action will represent an internal decision, resolution of which, prescribes a particular visible behaviour. Internal actions play a central role in creating non-determinism, see section 3.6.

3.2 Behaviour Expressions

The basic syntactic units of LOTOS specification are *behaviours*. The operators that we introduce will characterise the possible behaviours that can be written in LOTOS.

There is one behaviour that we can highlight immediately, it is the null behaviour,

```
stop
```

which performs no actions and is synonymous with deadlock. `stop` is typically used to terminate a non-null behaviour, i.e. it indicates that a point has been reached at which no more behaviour can be performed.

3.3 Process Definition

As suggested a number of times before, concurrent systems contain components which evolve concurrently. Thus, in order to model such systems we clearly need a syntactic entity which corresponds to a component. In process calculi (hence their name) this syntactic entity is called a *process*.

We define processes using the syntax:

```
P := B
```

where `P` is a process variable (i.e. a name for a process) and `B` is an arbitrary behaviour. The effect of the definition is to associate (bind) the process variable `P` to the behaviour `B`. Thus, whenever we refer to `P`, `B` is executed.

In our ICS specification we will have processes for all the subsystems. These will have the obvious names:

`VIS`, `OBJ`, `LIM`, `PROP`, `BS`, `IMPLIC`, `MPL`, `AC` and `ART`

We will also have a clock process called:

```
CLOCK
```

To take a rather fatuous illustration, we could define the process `VIS` as:

```
VIS := stop
```

which states that `VIS` cannot do anything, it just behaves as a stopped system. In future sections we will give `VIS` a more interesting behaviour than this, you will be glad to know.

We use the convention that all process variables will be written in capitals.

3.4 Sequence

Basic sequencing of actions is defined in LOTOS using *action prefix* which has the general form,

```
a;B
```

where `a` is an action and `B` is a behaviour. `a;B` is a behaviour that will offer action `a` to the environment and if it is taken will behave as `B`.

We can also view `a;B` in terms of pushing buttons as a black box with a gate corresponding to `a` (and gates for all the external actions in `B`). The button `a` is initially the only button raised, if the environment pushes `a` then the black box behaves as `B` (e.g. new buttons will be raised).

As an illustration of action prefix,

```
eye_vis?r1:Rep ; stop
```

will offer an action at gate `eye_vis` (which binds a value to `r1`) and if it is performed, will deadlock. In addition, the behaviour,

```
vis_normal ; VIS_NORMAL
```

offers an action `vis_normal` and if it is taken will invoke the process `VIS_NORMAL`. This fragment of specification models the `VIS` subsystem offering the environment the chance to go into normal mode.

3.5 Choice

Choice is denoted,

$\texttt{B}_1$ `[]` $\texttt{B}_2$

and states that either behaviour $\texttt{B}_1$ or behaviour $\texttt{B}_2$ will be performed. The choice of which is determined by the first action of the two behaviours. Both actions will be offered to the environment which will choose which of the two to perform; this decision will resolve the choice.

The necessity to offer such choices largely arises because of the move to systems which contain concurrency. A behaviour offering a choice of a number of actions to perform is really offering a menu of possible interactions that concurrently executing components can select from. The behaviour is defining the set of actions it is willing to react to.

We can illustrate choice using the sequencing fragment highlighted at the end of the last subsection.

```
vis_normal; VIS_NORMAL  [] vis_buffered; VIS_BUFFMD
```

which will offer the environment the choice of performing `vis_normal` (and instantiating the process `VIS_NORMAL`) or performing `vis_buffered` (and instantiating the process `VIS_BUFFMD`). The broad structure of our subsystems will be:

```
VIS := vis_normal;VIS_NORMAL [] vis_buffered;VIS_BUFFMD
```

where `VIS_NORMAL` and `VIS_BUFFMD` are subprocesses of `VIS` which respectively implement normal behaviour and buffered behaviour.

We will also use a generalised choice operator, denoted,

```
choice x:T [] B(x)
```

where `x` is a variable, `T` is a type and `B` is a behaviour expression that is parameterised on the variable `x`. The operator allows choice over a set of parameterised behaviours. We will give an ICS illustration of its use in the next subsection.

3.6 Non-determinism

Non-determinism is defined in LOTOS as a special case of choice. Specific forms of choice yield a non-deterministic resolution of the alternatives [6]. We will consider just one such form:-

$\mathtt{i;\ B_1\ \ []\ \ i;\ B_2}$

The non-determinism arises because selection between the two initial actions of the choice is beyond the control of the environment, since the initial evolution of the behaviour is completely hidden from the external observer; in terms of button pushing, no buttons are raised. Thus, a wholely internal choice will be made to either evolve to behaviour $\mathtt{B_1}$ or to evolve to behaviour $\mathtt{B_2}$.

Non-determinism plays a number of roles in process calculi. In general it acts as an abstraction device. For example, non-determinism is often introduced when at a certain level of system development, we wish to abstract away from a particularly complex mechanism.

As an example from ICS, consider a transformation acting in buffered mode, say `vis_implic`, which rather than selecting from the `VIS` input array selects from `VIS`'s image record. The question this raises is which element of the image record does `vis_implic` select. Well there are many possible mechanisms and rather than delving into the intricacies of them, we simply abstract away from the issue and view selection from the image record as non-deterministic. We could describe the selection in the following way:

```
i; SELECT(0) [] i; SELECT(1) [] i; SELECT(2) ...
```

where `SELECT(j)` indicates selection of the `j`th entry in the image record.

Due to the infinite number of options here we would actually write this using generalised choice as:

```
choice x:Nat [] i; SELECT(x)
```

where `Nat` is the type of natural numbers.

What we are really doing here is abstracting away from the specific mechanism by which selection occurs. We are stating that some internal mechanism could occur and result in a selection being made, but at the particular level of abstraction we are considering, we are not interested in how this happens.

Non-determinism is also used in specification to allow implementation freedom. A non-deterministic choice between evolving to B_1 or to B_2 can be viewed as stating that implementations that behave in either way are satisfactory. In other terms, the specifier does not mind whether the system behaves as B_1 or as B_2. Such non-determinism may then be refined out during development.

3.7 Concurrency

3.7.1 Independent Parallelism

We begin with a special case of concurrency; this has the form,

B_1 ||| B_2

which states that the two behaviours B_1 and B_2 evolve independently in parallel (we will refer to B_1 and B_2 as *parallel threads*). Independent in this context means that there is no shared behaviour, which would arise if B_1 and B_2 performed actions together.

We will use this construct in order to describe the behaviour of subsystems when they are working in normal or buffered mode. For example, we can state, as follows,

```
vis_obj!1; stop ||| vis_implic!1; stop
```

that the output transformations of `VIS` are independent of one another.

Furthermore, input and output activity will be independent of each other. Thus, a basic structure for `VIS_NORMAL` could be:

```
VIS_NORMAL :=
            ( eye_vis?r1:Rep; stop )
            (* Input Ports *)
            |||
            ( vis_obj!1; stop ||| vis_implic!1; stop )
            (* Output Ports *)
```

stating that VIS_NORMAL's three transformations are performed independently in parallel of one another. Text within (* and *) are comments.

This process structure is general to all ICS subsystems, e.g. a description of OBJ_NORMAL would have the form:-

```
OBJ_NORMAL :=
            ( vis_obj?r1:Rep; stop ||| prop_obj?r2:Rep; stop )
            (* Input Ports *)
            |||
            ( obj_mpl!1; stop ||| obj_prop!1; stop
              ||| obj_lim!1; stop )
            (* Output Ports *)
```

which has the same form as VIS_NORMAL only we have different kinds and numbers of transformations.

As it stands, these descriptions of VIS_NORMAL and OBJ_NORMAL are very limited: they just perform a set of transformations and then deadlock. However, subsystems should clearly be able to perform their transformations repeatedly. This is something we will consider shortly.

3.7.2 General Form

As already stated, independent parallelism is a specific class of concurrent behaviour. Concurrency, in its most general form, is denoted,

$\mathtt{B}_1 \;|[\mathtt{x}_1, \ldots, \mathtt{x}_n]|\; \mathtt{B}_2$

which states that $\mathtt{B}_1$ and $\mathtt{B}_2$ evolve independently in parallel subject to the synchronisation of actions $\mathtt{x}_1, \ldots, \mathtt{x}_n$, i.e. an action $\mathtt{x}_i$ ($1 \leq i \leq n$) appearing in either $\mathtt{B}_1$ or $\mathtt{B}_2$ can only be executed if it synchronises with an $\mathtt{x}_i$ in the other behaviour.

As an example, we can compose the two processes, VIS_NORMAL and OBJ_NORMAL together in parallel, subject to synchronisation on the common action, vis_obj:

```
VIS_NORMAL |[vis_obj]| OBJ_NORMAL
```

which expresses that the processes VIS_NORMAL and OBJ_NORMAL will perform all actions separately, apart from vis_obj, which they will perfom together. Such synchronisation has two implications:-

1. *Synchronising processes wait for one another.* For example, if VIS_NORMAL reaches a point where it wishes to perform vis_obj, it must wait for OBJ_NORMAL to be ready before it can do it. Thus, when attempting to perform synchronised actions processes become blocked waiting for partner processes.

2. *Data attributes must match.* With VIS_NORMAL and OBJ_NORMAL this constraint is met since VIS_NORMAL outputs a representation on vis_obj, i.e. it performs,

```
vis_obj!1
```

while, `OBJ_NORMAL` inputs a representation on `vis_obj`, i.e. it performs,

```
vis_obj?r1:Rep
```

When synchronisation occurs, the value `1` is bound to the variable `r1`. The rules for matching of data attributes are a little subtle and we will not delve into their intricacies, see [5] for an explanation.

We call `|[...]|` generalised parallelism since independent parallelism can be derived from it, B_1 `|||` $B_2 = B_1$ `|[]|` B_2, i.e. general parallel composition with an empty synchronisation set (a further operator B_1 `||` B_2, fully synchronised parallelism can also be derived, but we will not need it here).

3.8 Recursion

As just discussed, specifications are not particularly interesting unless they contain repetitive behaviour. Process calculi use recursion in order to do this. As a very simple illustration we will use a clock process, which is defined as follows:-

```
CLOCK := tick; CLOCK
```

which will offer a tick action and then recur (by instantiating itself again). It will perform an infinite number of finite traces of the form[9]:

```
tick
tick tick
tick tick tick
....
....
```

3.9 Enabling

Action prefix defines sequencing for actions, however, we would also like to define sequencing of complete behaviours. This is supported by enabling,

B_1 `>>` B_2

which will evolve as B_1, then if B_1 terminates successfully, it will behave as B_2. The concept of successful termination is pivotal here. We do not wish B_1 `>>` B_2 to evolve to B_2 unless B_1 completes its evolution. In particular, if B_1 is in a deadlock state we would wish B_1 `>>` B_2 to also deadlock. Thus, we introduce a distinguished behaviour,

[9] In fact, we can give a number of different interpretations to such recursive behaviour, including infinite trace semantics [47]. However, the standard approach is to use finite traces since these reflect the role of the environment and the reactive nature of the model - we are interested in what can happen over time rather than the terminal behaviour. At a particular time the system will have performed some finite trace of the recursive behaviour.

```
exit
```

to denote successful termination.

Returning to our specification of VIS_NORMAL, we will replace the stop states with successful terminations in order that we can evolve through enabling to VIS_NORMAL again. The desired behaviour is:

```
VIS_NORMAL :=
            ( eye_vis?r1:Rep; exit)
            (* Input Ports *)
            |||
            ( vis_obj!1; exit ||| vis_implic!1; exit )
            (* Output Ports *)
            >> tick; VIS_NORMAL
```

This process will perform some interleaving of the actions: eye_vis, vis_obj and vis_implic; then a successful termination will take place, from which a tick is performed (this is used to control synchronous behaviour, see discussion in section 4.2.2) then VIS_NORMAL is called recursively. An important aspect of successful termination is that if a number of parallel threads exist, they must all terminate before the whole behaviour can terminate. Thus, all the three threads:

eye_vis?r1:Rep;exit , vis_obj!1;exit and vis_implic!1;exit

must terminate before the enabling operator >> can fire.

3.10 Hiding and Relabelling

These are the final operators of this section. The first, hiding, has the form,

hide x_1,....,x_n in B

where x_1,....,x_n are observable actions and B is an arbitrary behaviour. The operator behaves as B except that all actions in x_1,....,x_n are turned into internal actions.

Hiding enables *information hiding*: actions which are observable at one level of specification can be transformed into hidden actions at another level. Thus, behaviour that should not be visible, can be hidden. In effect, such hiding supports a form of abstraction, since the complexity of a part of the system is abstracted away from, by hiding it, when specifying another part.

As an illustraction, ICS contains a number of transformations which can be viewed as internal to the full system. Thus, we will naturally hide these transformations when we build the top level system. As an illustrative fragment, we might hide vis_obj, vis_implic, prop_obj, obj_mpl, obj_prop and obj_lim in our composition of VIS_NORMAL and OBJ_NORMAL,

```
hide vis_obj, vis_implic, prop_obj,
     obj_mpl, obj_prop, obj_lim in
                       VIS_NORMAL |[vis_obj]| OBJ_NORMAL
```

leaving only the sensory subsystem action `eye_vis` observable.

A relabelling operator is also provided. It has the form:

B \ [x_1/y_1,....,x_n/y_n]

where x_1,...,x_n, y_1,....,y_n are actions. It behaves as `B` apart from the fact that the occurrence of any action y_i is replaced by the action x_i.

As an ICS illustration, we will define processes to perform blending. These define blending mechanisms that are general, in the sense that they are defined over an arbitrary gate, `g` say. Then we specialise these general mechanisms using relabelling, e.g. if,

```
BLEND
```

is such a process, we might specialize it to act on the transformation `obj_prop` by invoking:

```
BLEND \ [obj_prop/g]
```

4 Specification of ICS

In this section we use the operators introduced in the last section to build a specification of ICS. The extra specification features we will need concern data types; these will be introduced in the next subsection (subsection 4.1). Then in subsection 4.2, we present the full ICS description.

4.1 Data Types in ICS

Although we will treate representations in a very simple way, we will use a number of other data structures in our specification. We introduce each of the main data types in turn:

Representations. We assume a type[10].

```
Rep
```

of representations, which we define as a type synonym for `Nat` (i.e. `Rep` is just a different name for `Nat`), the natural numbers. Thus, the elements of `Rep` will be:

```
0, 1, 2, 3, ....
```

[10]Note what we refer to as types are called sorts in ACT-ONE.

and we can use the natural number operations, e.g. $+$, $=$, $*$, with representations. By convention, `0` denotes a "null" representation, i.e. one that does not contain any information. We give a more indepth justification for using the natural numbers as representations at the start of subsection 5.4.

Input Array. We model the input array using a 4-tuple. This use of a static type, i.e. one with a fixed size, is only sufficient because no subsystem has more than 4 source subsystems The use of static types pervades all our data types. This is not an optimal or particularly elegant solution, in particular, the resulting data types are not easily extended, however, we have adopted it because it enables us to restrict ourselves to simple data types. Thus, we assume a type,

```
inArr
```

of 4-tuples of representations. In the sequel we will refer to elements of tuples as *slots*. Typical elements are:

```
#(2,3,0,4) , #(0,0,0,0) , #(1,2,1,3)
```

where the four elements of the tuple are indeed representations. Notice the second of these denotes a null input array.

The only operation defined over `inArr` is accessing elements of the array,

```
iaget :  indices, inArr -> Rep
```

which when given an index (we have four indice `j0`, `j1`, `j2` and `j3`, which index corresponding entries in an input array) and an input array, returns the representation that is located at that slot, e.g.

```
iaget( j0 , #(2,3,0,4) ) = 2  and
iaget( j1 , #(2,3,0,4) ) = 3
```

Each subsystem will have an input array, of type `inArr`, and each slot in the input array will receive input from a particular source subsystem. For example, `prop` has three input transformations:

`obj_prop`, `implic_prop`, `mpl_prop`

such that `obj_prop` maps into slot `0`, `implic_prop` maps into slot `1` and `mpl_prop` maps into slot `2`. Clearly, since `PROP` only has three input transformations, there will be an empty slot - the 4th slot. We adopt the convention that null representations are placed in empty slots.

This use of a static type, i.e. one with a fixed size, is only sufficient because no subsystem has more than 4 source subsystems The use of static types pervades all our data types. This is not an optimal or particularly elegant solution, in particular, the resulting data types are not easily extended, however, we have adopted it because it enables us to restrict ourselves to simple data types.

Image Record. We model image records as queues, where the elements of the queues are input arrays (in fact, many different varieties of dynamic data structure could be used here. We use queues because they are very easy to work with in ACT-ONE.). The type of image records is denoted

```
imRc
```

The full definition of the `imRc` type can be found in the appendix. Here we save the reader the details, but simply introduce the main operations, that we will need. Firstly, we assume a constant,

```
nil
```

which denotes an empty image record. Secondly, the operator,

```
add :  inArr, imRc -> imRc
```

adds an element, i.e. an input array, to an image record, returning a new image record. Thirdly, the operation,

```
select :  Nat, inArr, imRc -> inArr
```

gets an element from the image record. It takes a number (indicating which element in the image record is sought), an input array (which is not actually used in our current implementation[11]) and an image record and returns the item sought.

As an illustration, let `ir` denote the two element image record constructed as follows:

```
ir = add( #(1,2,1,3), add(#(2,3,0,4),nil) )
```

Then,

```
select( 0 , iA , ir ) = #(2,3,0,4),
select( 1 , iA , ir ) = #(1,2,1,3)
```

for any input array `iA` and by default,

```
select(x,iA,ir) = #(0,0,0,0)
```

for all `x`>1, i.e. selecting beyond the maximum element gives a null response.

Transformation Maps. This is an important data structure that we have not considered before. In fact, it does not appear in the basic formulation of ICS. We use it in order to record the slots in the input array from which a particular output transformation takes representations. Remember, that an output transformation can be a blend of a set of representations (where each representation is taken from a particular slot in the input array).

The transformation map is defined using two (4-tuple) types. Firstly, we assume a type:

[11] The input array is included in the selection processes as a place holder for a more sophisticated select operation which choses elements from the image record according to what is currently in the input array. However, this extra sophistication is not yet implemented.

```
slotmap
```

of 4-tuples of booleans, a typical element might be:

```
#(true,false,true,false)
```

which indicates the slots in the input array that are relevant for (i.e. need to be blended by) a particular output transformation. In the sequel we will use the term *alive* to describe the slots that are prescribed by a slot map.

As an illustration, if this slot map was associated with the `IMPLIC` output transformation:

```
implic_prop
```

it would indicate that the representation to be transmitted over `implic_prop` must be a blend of the `0`th and the `2`nd slots of `IMPLIC`s input array.

We assume an accessing function `smget`, e.g.

```
smget(j0,#(true,false,true,false) = true   and
smget(j3,#(true,false,true,false) = false
```

Then we build a data type of transformation maps, denoted,

```
Map
```

which is a 4-tuple of `slotmaps`. For example, if associated with `IMPLIC`, the transformation map:

```
#(  #( true, false, true, false ),
    #( false, true, false, false ),
    #( false, false, true, false ),
    #( false, false, false, false ) )
```

would indicate that output transformation,

`0` (i.e. `implic_prop`) uses the `0`th and `2`nd slots
`1` (i.e. `implic_visc`) uses the `1`st slot
etc,

We also have an accessing operator over maps, `mpget`, which, for example, if we let `m` denote the transformation map just shown, will behave as follows:

```
mpget(j0,m) = #( true, false, true, false )
mpget(j2,m) = #( false, false, true, false )
smget(j1,mpget(j2,m)) = false
```

Transformations. We assume a type, `Subsyst`, of constants:

`VISS`, `OBJJ`, `LIMM`, `IMPLICC`, `BSS`, `MPLL`, `ACC`, `ARTT` one for each subsystem. Then we define an operation:

```
tran :  Subsyst, Subsyst, Rep -> Rep
```

which when given a source and a target subsystem implements a particular transformation. For example,

```
tran(IMPLICC,PROPP,r)
```

applys the `implic` to `prop` transformation to a representation `r`.

Ultimately `tran` will perform transformations on representations, e.g. in the super/subordinate style outlined in [2]. However, for the moment we simply include a place holder for such mappings and view all transformation operations as the identity, i.e.

`tran(s,t,r) = r`
for all subsystems `s` and `t` and representations `r`

4.2 Complete Specification

4.2.1 General Subsystem Format

As a typical subsystem we build up the full `OBJ` specification.

Top Level Behaviour of Subsystems. Firstly, the `OBJ` process has the top-level behaviour:-

```
OBJ(iR:imRc,iA:inArr,m:Map) :=
      obj_normal; OBJ_NORMAL(iR,iA,m)
      []
      obj_buffered?b:indice; OBJ_BUFFMD(iR,iA,m,b)
```

which is in the same form as that presented in the previous subsection apart from some data aspects. `OBJ` is now a process with data parameters:

`iR` of type `imRc` , `iA` of type `inArr` and `m` of type `Map`

corresponding to the image record, input array and transformation map for `OBJ`.

The process offers the environment the choice of entering normal mode or buffered mode. The variable `b` is used to indicate which transformation will become buffered.

Blending. As suggested earlier, we use non-determinism to abstract away from particular mechanistic interpretations of blending. We can illustrate the general issue of blending as follows:-

> Consider the output transformation, `obj_mpl`; if this transformation uses representations from more than one slot in `OBJ`s input array (the transformation map will prescribe this), i.e. slots `0` and `1`, then blending determines which representation `obj_mpl` will act upon[12].

[12] We say "act upon" rather than "transmit" because if we consider the full ICS functionality, `obj_mpl` can apply an information transforming mapping to the representation it acts upon.

As discussed earlier, a central concept in blending is *consistency* of representations. Since we are using natural numbers to denote representations, an obvious way to model consistency is as (natural number) equality. This leads to an admittedly very coarse interpretation of consistency, but it will suffice for the analysis we have in mind.

We can consider a number of different possible forms of blending, which vary in their level of non-determinism. In our presentation we continue to use `obj_mpl` in `OBJ` as our example transformation. We assume that representation $\mathtt{r}_0$ is in slot `0` and $\mathtt{r}_1$ is in slot `1`.

1. *Fully Non-deterministic.* Under this approach one value from the set of all possible representations is non-deterministically selected for `obj_mpl` to act upon. Importantly, no attention is paid to the items in the input array. Selection is made completely non-deterministically over the set of *all* representations.

   This behaviour can be realised with the following LOTOS process:-

   ```
   BLEND1(X,Y:Subsyst) :=
     choice r:Rep [] i; g!tran(X,Y,r); exit
   ```

   which inputs two subsystem constants indicating the source and destination of the transformation to be performed and then offers a non-deterministic choice of the transformation acting on any possible representation.

   A particular instantiation of this process might be:

   ```
   BLEND1(OBJJ,PROPP)\[obj_prop/g]
   ```

   which, on invocation, will bind `OBJJ` to `X` as the source of the transformation, `PROPP` to `Y` as the destination of the transformation and will also relabel the gate `g` to `obj_prop`.

   Due to the looseness of specification involved, this approach yields some odd behaviour. For example, although `OBJ` might receive stable and consistent inputs in slots `0` and `1` if `BLEND1` is applied, `obj_mpl` might act upon an unstable flow which has no relation to the representations input.

   The reason for considering such a non-deterministic approach is that it provides an upper bound (in terms of loose specification) on blending. Importantly, all other solutions will, in computer science terms, be *refinements* of such a fully non-deterministic blending. The useful property that this yields is that anything we can prove about ICS with fully non-deterministic behaviour will also hold of any refinement. It might be that we cannot prove much about such an abstract specification, however, we know that what we can prove will hold of all ICS implementations. The subsection 9.6 in the appendix contains a formal justification of this statement.

2. *Non-deterministic Information Preservation.* According to this approach, we do consider the relevant slots in the input array. However, our strategy is very simple: we just make a non-deterministic choice between acting upon $\mathtt{r}_0$ and acting upon $\mathtt{r}_1$. This approach has some interesting characteristics:

   - If $\mathtt{r}_0 = \mathtt{r}_1$ (i.e. they are consistent) then the (common) consistent representation is automatically acted upon. Furthermore, if inputs to slots `0` and `1` are stable then the blended flow acted upon by `obj_mpl` will also be stable.
   - If $\mathtt{r}_0 \neq \mathtt{r}_1$ (i.e. they are inconsistent) then `obj_mpl` will act upon a flow that is made up of a mixture of $\mathtt{r}_0$s and $\mathtt{r}_1$s (the mixture arising from the non-determinstic choice). Consequently, inconsistency *over time* between $\mathtt{r}_0$ and $\mathtt{r}_1$ will yield a "randomly varying" flow at `obj_mpl`[13].

   Although still non-deterministic this approach is clearly more deterministic than approach 1. We can implement it as follows in LOTOS:

```
BLEND2(X,Y:Subsyst,id:indice,iA:inArr,m:Map) :=
    choice j:indice [] [smget(j,mpget(id,m))] ->
                         i; g!tran(X,Y,iaget(j,iA)); exit
```

   which has a similar structure to `BLEND1` except:

   - It has extra parameters, `id`, `iA`, `m`, which respectively reference the index of the transformation being applied; the input array and the transformation map currently being used.
   - The non-deterministic choice here is more restricted than that in `BLEND1`. The choice is parameterised on the four possible indice. Then the guard[14].

     ```
     [smget(j,mpget(id,m))] ->
     ```

     determines which index is "alive". It does this by accessing the relevant slot map (i.e. the `id`th) in the transformation map `m` and then seeing if the `j`th element in this slot map is prescribed.
   - If `j` is alive `iaget` accesses the entry it prescribes in the input array `iA`.

3. *Deterministic Information Preservation.* If $\mathtt{r}_0 \neq \mathtt{r}_1$ (i.e. they are inconsistent) then `obj_mpl` might act upon the null representation, `0`. The

[13]The term randomly varying here is somewhat loaded, by its nature non-deterministic selection will also allow an implementation in which just one of the two slots is always sampled from and thus, stability could be regenerated from inconsistent inputs.

[14]With `bc` a boolean condition, the syntax for a guard, `[bc]->B` will evolve to behaviour `B` if `bc` holds, otherwise it will deadlock.

intuition being that since r_0 and r_1 are not blendable this should be reflected in the representation acted upon. Alternatively, if $r_0 = r_1$ (i.e. they are consistent) this approach preserves the consistent representation and obj_mpl acts upon it.

The following process implements this behaviour:

```
BLEND3(X,Y:Subsyst,id:indice,iA:inArr,m:Map):=
             g!tran(X,Y,compare(mpget(id,m),iA)); exit
```

where,

```
   compare : slotmap, inArr -> Rep
```

takes a slotmap and an input array and relates the representations in all alive slots (as determined by the slotmap). It behaves as follows:

> If all alive slots are equal it returns that representation otherwise it returns 0.

The definition of compare can be found in the appendix.

4. *Consistent Information Generation (or Consistent Multiplicative Blending).* By this approach, when $r_0 = r_1$ we blend r_0 and r_1 and generate a new representation, which is conceptually a composite of r_0 and r_1. With natural numbers modelling representations we choose to multiply r_0 and r_1 together[15]. In addition, if $r_0 \neq r_1$ we return 0.

   We can implement consistent multiplicative blending as[16]:

```
BLEND4(X,Y:Subsyst,id:indice,iA:inArr,m:Map):=
 let sm:slotmap=mpget(id,m) in
    ( [equal(sm,iA)] -> g!tran(X,Y,mult(sm,iA)); exit
      []
      [not(equal(sm,iA))] -> g!0; exit )
```

   which with its first guard multiples representations in slots that are "alive". It uses mpget to access the relevant slot map in m and then it applies an operation,

```
    mult :  slotmap, inArr -> Rep
```

[15]In general, such an approach ensures that any representation blended with the null representation inherits its instability, i.e. $\forall x\ .\ x * 0 = 0$. In addition, it implies a special class of blending if one of the representations is 1, i.e. $\forall x\ .\ x * 1 = x$. Thus, we do not yield a new representation in this case. In general, we assume all non-null representations are bigger than 1, in order to avoid this situation.

[16]The let construct enables local definitions to be made, e.g. here we define a variable sm which is used in the body of the let construct.

which multiplies together representations in alive slots. The definition of `mult` can be found in the appendix.

In addition,

```
equal :  slotmap, inArr -> Bool
```

takes a `slotmap` and an input array and relates the representations in all alive slots (as determined by the `slotmap`) for equality. This definition is also in the appendix.

5. *Crude Information Generation (or Crude Multiplicative Blending).* In this approach multiplicative blending is crudely applied, whether or not the representations to be blended are consistent. It can be implemented as,

```
BLEND5(X,Y:Subsyst,id:indice,iA:inArr,m:Map):=
    g!tran(X,Y,mult(mpget(id,m),iA)); exit
```

Normal Mode. We can now present the full behaviour of `OBJ_NORMAL`, it is,

```
OBJ_NORMAL(iR:imRc,iA:inArr,m:Map) :=
  (( vis_obj?r1:Rep; exit(r1,any Rep) |||
     prop_obj?r2:Rep; exit(any Rep,r2) )
  (* Input Ports *)
  |||
  (   BLENDk(OBJJ,MPLL,j0,iA,m)\[obj_mpl/g] >> exit(any Rep,any Rep)
  ||| BLENDk(OBJJ,PROPP,j1,iA,m)\[obj_prop/g] >> exit(any Rep,any Rep)
  ||| BLENDk(OBJJ,LIMM,j2,iA,m)\[obj_lim/g] >> exit(any Rep,any Rep) )
  (* Output Ports *) )
  >> accept r1,r2:rep in
     tick; OBJ(add(#(r1,r2,0,0),iR),#(r1,r2,0,0),m)
```

The new aspects are:-

- In a similar way to `OBJ`, `OBJ_NORMAL` is now parameterised on the image record, input array and transformation map.
- We use parameterised successful termination. For example,

  ```
  exit(r1,any Rep)
  ```

  denotes a successful termination where two values are past through the termination. The first is the value of the variable `r1` and the second is unprescribed, i.e. it could be any representation; it is typically prescribed by the exit processes that terminate alternative parallel threads, e.g. `exit(any Rep,r2)` above.

- As suggested by our blending discussion, each output transformation is now applied through a process invocation, here `BLENDk` (the `k` determining which blending strategy to use).

- Companion to the parameterised termination is parameterised enabling. Thus, the enabling, `>>`, is followed by the behaviour:

```
accept r1,r2:rep in
       tick; OBJ(add(#(r1,r2,0,0),iR),#(r1,r2,0,0),m)
```

  which will accept two values through the enabling (which are exactly those yielded by the parameterised termination), bind them to the variables `r1` and `r2` and then evolve to:

```
tick; OBJ(add(#(r1,r2,0,0),iR),#(r1,r2,0,0),m)
```

- Finally, we reinstantiate `OBJ` with a new input array:

```
#(r1,r2,0,0)
```

  which is also added to the front of the image record. The transformation map is returned unchanged.

Buffered Mode. We can also show how buffered mode behaviour is realised. The key change here is to select from the image record when blending rather than from the input array. The following process gives a generic implementation of such a mechanism. The process is called `OUTPUTk`, where the `k` determines the blending strategy used. It encapsulates the behaviour of an arbitrary output transformation.

```
OUTPUTk(X,Y:Subsyst,b,j:indice,iR:imRc,iA:inArr,m:Map):=
 ([b eqq j] -> choice n:Nat [] BLENDk(X,Y,j,select(n,iA,iR),m)
 []
  [b neq j] -> BLENDk(X,Y,j,iA,m))
```

The process has the following parameters:

- `X`, `Y` determine the transformations being realised;

- `b` is a variable that states the indice that corresponds to the currently buffered transformation;

- `j` is the index of the current transformation.

- `iR`, `iA` and `m` are respectively the relevant image record, input array and transformation map.

As an illustration, we could instantiate the process as follows:-

```
OUTPUTk(OBJJ,PROPP,j1,j1,iR,iA,m)\[obj_prop/g]
```

which implements the output transformation `obj_prop` and specifies that it is in buffered mode.

The behaviour of `OUTPUTk` defines the necessary buffered behaviour. In particular, if the transformation being considered is buffered (as determined by the comparison between `b` and `j`) then the input array on which blending will be applied is non-deterministically selected from the image record using the operation `select`, which was described earlier. Thus, selection from the image record is completely non-deterministic. Other implementations could be considered but this simple implementation is satisfactory for this paper. Alternatively, if the current transformation is not buffered then blending is applied on the input array.

The overall behaviour of `OBJ_BUFFMD` is:

```
OBJ_BUFFMD(iR:imRc,iA:inArr,m:Map,b:indice) :=
( ( vis_obj?r1:Rep; exit(r1,any Rep)
||| prop_obj?r2:Rep; exit(any Rep,r2) )
(* Input Ports *)
 |||
( OUTPUTk(OBJJ,MPLL,b,j0,iR,iA,m)\[obj_mpl/g] >> exit(any Rep,any Rep)
  |||
  OUTPUTk(OBJJ,PROPP,b,j1,iR,iA,m)\[obj_prop/g] >> exit(any Rep,any Rep)
  |||
  OUTPUTk(OBJJ,LIMM,b,j2,iR,iA,m)\[obj_lim] >> exit(any Rep,any Rep) )
(* Output Ports *) )
>> accept r1,r2:rep in
        tick; OBJ(add(#(r1,r2,0,0),iR),#(r1,r2,0,0),m)
```

Notice that the behaviour of `OBJ_BUFFMD` is determined by the value of `b` past to it - if `b` equals `j0` then, `obj_mpl` goes into buffered mode, while if `b` equals `j1`, then, `obj_prop` goes into buffered mode and if `b` equals `j2`, then `obj_lim` goes into buffered mode.

4.2.2 Top Level behaviour

All subsystems have the same basic format as that just shown for `OBJ`. The only differences are that different transformations are fired in different subsystems and the number of transformations may be different. However, using the general format just outlined, the reader can determine the make-up of particular subsystems, by considering figure 3.

Using these subsystems we can define the top level behaviour of ICS as follows:

```
hide vis_implic, vis_obj, obj_mpl, obj_prop,
     obj_lim, implic_prop, implic_som, implic_visc,
     prop_mpl, prop_implic, prop_obj, bs_art, bs_lim,
```

```
     bs_implic, ac_mpl, ac_implic, mpl_art, mpl_prop
in
( clock
|[tick]|
( tick;
(((((((
VIS(nil,#(0,0,0,0),VISmap)
|[vis_obj,tick]|
OBJ(nil,#(0,0,0,0),OBJmap) )
|[obj_lim,tick]|
LIM(nil,#(0,0,0,0),LIMmap) )
|[vis_implic,tick]|
IMPLIC(nil,#(0,0,0,0),IMPLICmap) )
|[obj_prop,implic_prop,prop_obj,prop_implic,tick]|
PROP(nil,#(0,0,0,0),PROPmap) )
|[bs_implic,bs_lim,tick]|
BS(nil,#(0,0,0,0),BSmap) )
|[obj_mpl,prop_mpl,mpl_prop,tick]|
MPL(nil,#(0,0,0,0),MPLmap) )
|[ac_mpl,ac_implic,tick]|
AC(nil,#(0,0,0,0),ACmap) )
|[mpl_art,bs_art,tick]|
ART(nil,#(0,0,0,0),ARTmap) )
|[obj_buffered,vis_buffered,lim_buffered,implic_buffered,prop_buffered,
mpl_buffered,ac_buffered,art_buffered,bs_buffered,tick]|
tick; buffConstraint ) )
```

This construction follows the format of ICS given in figure 3. However, in addition we compose in parallel a clock process which forces the synchronous behaviour we discussed in section 2 and a buffer constraint which enforces the constraint that only one transformation can be buffered at any one time. In addition, we hide all transformation that are neither effector or sensor actions.

Subsystems are invoked with null image records and input arrays and with a particular transformation map. The make-up of each transformation map is defined before the particular subsystem is invoked, see the complete specificaton of ICS to be found in the appendix.

Buffer Constraint. The buffer constraint is defined as follows:

```
buffConstraint :=
  obj_buffered?b:indice; tick; buffConstraint
  []
  vis_buffered?b:indice; tick; buffConstraint
  []
  lim_buffered?b:indice; tick; buffConstraint
  []
  implic_buffered?b:indice; tick; buffConstraint
```

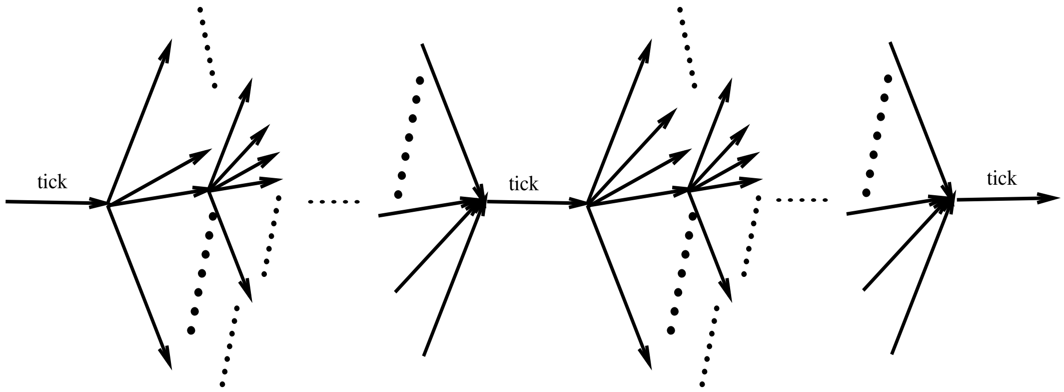

Figure 11: Synchronous Behaviour

```
[]
prop_buffered?b:indice; tick; buffConstraint
[]
mpl_buffered?b:indice; tick; buffConstraint
[]
ac_buffered?b:indice; tick; buffConstraint
[]
art_buffered?b:indice; tick; buffConstraint
[]
bs_buffered?b:indice; tick; buffConstraint
[]
tick; buffConstraint
```

which ensures that the environment cannot select more than one action of the form `Y_buffered` on each iteration of the system. This prevents more than one subsystem entering buffered mode. The possibility that no subsystem is buffered is accommodated by the behaviour,

```
tick; buffConstraint
```

Synchronous Behaviour. It is also worth recapping on the issue of the overall format of ICS behaviour. As discussed earlier, we interpret the behaviour of ICS in a synchronous manner. Evolution of our specification has the general format shown in figure 11 where each synchronous step appears as a fan out of actions. This fan out contains an arbitrary interleaving of all ICS transformations. Thus, all transformations must complete before the next tick takes place.

5 Goal Formulation in Interval Temporal Logic

5.1 Background

We now begin to consider what properties our ICS specification exhibits. We call these properties *goals*. They express *possible* global behaviours of ICS and we would like to verify whether our LOTOS specification of ICS can indeed exhibit these behaviours.

It is important to understand the nature of these goals. Specifically, we will be interested in showing,

> what behaviours ICS *can* and what behaviours it *cannot* exhibit.

however, we will at no stage verify that it *must* perform a particular task[17].

The goals we analyse are taken from previous syndetic modelling work. In particular, we consider two goals discussed in the work of Duke et al [18] and four goals discussed in the work of Faconti et al [21]. These goals concern the capability of ICS to perform particular forms of diectic reference and they are all formulated over the same ICS configuration - $conf_{deixis}$ which we introduced in subsection 2.2.4.

The next subsection 5.2 introduces the goal formulation notation that we will use, this is called interval temporal logic. Then subsection 5.3 introduces some useful operators that can be derived from the basic logic. Finally, subsection 5.4 formulates the ICS goals that we are interested in in the interval temporal logic. We also give a number of axioms for the logic in the appendix which are used in our proofs.

5.2 Interval Temporal Logic

We formulate our goals in a form of temporal logic called *Interval Temporal Logic* (ITL) [40]. The choice of temporal logic as a notation for formulating global properties/requirements of parallel systems, which is what our goals can broadly be interpreted as, is well accepted in concurrency theory [34]. The choice of ITL from the canon of temporal logics is perhaps less obvious. The main reason for this choice is that the type of goals we formulate seem to be elegantly expressed in ITL. This is mainly because stability can very naturally be expressed in such a logic.

The ITL we use is called Mexitl; it was developed with application in the multimedia field in mind [8, 9, 11], however, it seems to also be well suited to our goal formulation. Indepth introductions to the logic can be found elsewhere [8], here we restrict ourselves to a brief introduction.

Intervals. ITL's are defined over finite traces, called intervals, which represent runs/executions of a system (here ICS). Such intervals provide a semantic link between our LOTOS description of ICS and the goals that we formulate in

[17]This distinction is related to the distinction between *may* and *must* testing [24].

Mexitl. In particular, intervals can be derived from LOTOS specifications; we give such a semantic map in the appendix of this paper.

As an illustration of intervals consider the intervals generated from the execution of a single process, e.g. `OBJ`. A typical such interval would be:

```
obj_normal prop_obj_4 obj_mpl_2 vis_obj_0 obj_prop_2 obj_lim_0
i tick obj_normal obj_mpl_4 obj_lim_4 prop_obj_4 vis_obj_0 obj_prop_4
i tick
```

which represents the sequence of actions performed in a particular run of `OBJ`. The run contains two iterations of the process, i.e. the complete set of actions of the process are executed twice and these executions are divided by a `tick` action. Observe the following points:

- The `OBJ` transformations are arbitrarily interleaved[18] and thus occur in different orders in the two iterations.
- The data element of actions is flattened out. For example, we denote a particular instance of the action `vis_obj?r1:Rep` happening as `vis_obj_0` which indicates that the variable `r1` is instantiated by the null representation.
- The internal actions appearing in this interval arise from the firing of the enabling operator; see [5] for a discussion of the semantics of enabling.

It is also worth pointing out that in interval temporal logic terms, these are very simple interval models. In particular, the items in the traces are simply actions. More sophisticated intervals also include data state valuations in such items, but this sophistication will not be required here.

Mexitl Operators. We only consider a subset of the full Mexitl notation. It contains the following propositions, where P is an arbitrary proposition.

$$\begin{array}{rcl} P & ::= & a \mid p(E_1, ..., E_n) \mid E = E \mid \mathbf{False} \mid P \Rightarrow P \mid \\ & & \mathbf{len}(n) \mid P \ \mathbf{;}\ P \mid P \ \ \mathbf{proj}\ \ P \mid (\exists x \in T)P \end{array}$$

where $a \in \mathbf{Act}$; p is in a set of given predicates and E is an expression. We have the following operators:

- Firstly we assume a simple expression language, which enables us to write expressions such as $a + 5$.

[18] The most standard approach for modelling concurrent behaviour in process algebra is to use interleaving. Thus, the actions of two (independently) concurrent behaviours are arbitrarily interleaved. For example, `a b c`, `a c b` and `c a b` are all traces of the behaviour, `a; b; stop ||| c; stop`. The use of interleaving to model concurrency is justified by the assumption that all actions are atomic. Thus, no two actions can occur at overlapping instances - one must always get in before any of the others.

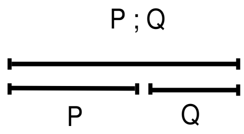

Figure 12: The Chop Operator

- $p(E_1, ..., E_n)$ denotes evaluation of a predicate according to n expressions. In standard fashion, we will often write binary predicates infix, e.g. $x < 10$.
- $E = E$ gives equality of expressions, e.g. $a + 5 = b$.
- **False** and $P \Rightarrow P$ are the familiar connectives of classical propositional logic.
- **len** is the length operator which measures the length of an interval. By convention the one item interval has length `0` and accordingly the `n` item interval has length `n-1`; i.e. **len** measures the number of transitions between items rather than the number of items themselves. **len**(15) holds over the `OBJ` interval that we highlighted above.
- **;** is the sequencing operator, *chop*, familiar from [40]. An interval satisfies P **;** Q if the interval can be divided into two contiguous sub-intervals (with the end-point of the first and the first-point of the second interval shared), such that P holds over the first sub-interval and Q holds over the second, see figure 12. In this depiction intervals are represented as line segments.
- **proj** is the projection operator, also described in [40]. An interval satisfies P **proj** Q if it can be sub-divided into a series of sub-intervals each of which satisfies P - we call P *the projection formula* - and a new interval formed from the end points of the sub-intervals satisfies Q, which we call *the projected formula*, see figure 13. In this depiction, Q holds over the interval formed by concatenating together the points shown.
- $(\exists x \in T)P$ gives existential quantification in the usual way.

5.3 Derived Operators

The primitive operators of Mexitl can be used to derive a large spectrum of further operators. In many circumstances these extra operators prove to be more usable than those of the core language. We derive the other connectives of classical propositional logic in the next subsection. Then we consider a number of different classes of derived temporal operator.

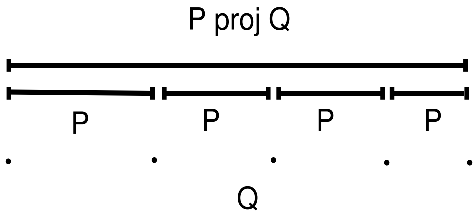

Figure 13: The Projection Operator

5.3.1 Logical Connectives

The remaining propositional logic connectives are derived in a standard fashion.

$$\neg P \ \equiv\ P \ \Rightarrow\ \textbf{False}$$

$$P \ \vee\ Q \ \equiv\ (\neg P) \Rightarrow Q$$
$$P \ \wedge\ Q \ \equiv\ \neg(P \Rightarrow \neg Q)$$
$$\textbf{True} \ \equiv\ \neg\textbf{False}$$
$$P \ \Leftrightarrow\ Q \ \equiv\ (P \Rightarrow Q) \ \wedge\ (Q \Rightarrow P)$$
$$\bigwedge_{x \,\in\, \{y_1,\, \ldots,\, y_n\}} P(x) \ \equiv\ P(y_1) \ \wedge\ \ldots\ \wedge\ P(y_n)$$

5.3.2 Basic Derived Temporal Operators

We present a small set of derived temporal operators which will all be used in our goal formulations and verifications. Additionally, we present a more comprehensive list of derived temporal operators in the appendix.

An interval is called empty if it contains one item:

$$\textbf{empty} \ \equiv\ \textbf{len}(0)$$

$\bigcirc P$ is the next operator, which is related to $\bigcirc$ in linear time temporal logic [34]:

$$\bigcirc P \ \equiv\ \textbf{len}(1) \ ; P$$

Eventually, $\Diamond P$, holds if there exists a *terminal* interval on which P holds:

$$\Diamond P \ \equiv\ \textbf{True} \ ; P$$

Henceforth, $\Box P$, is the dual of eventually; it holds if P holds over all terminal intervals:

$$\Box P \ \equiv\ \neg\Diamond\neg P$$

fin P requires that P holds at the last point in the interval:

$$\textbf{fin } P \;\equiv\; \Box(\textbf{empty} \Rightarrow P)$$

beg P requires that P holds at the first point in an interval:

$$\textbf{beg } P \;\equiv\; (\textbf{empty} \;\wedge\; P) \;\textbf{;}\; \textbf{True}$$

keep P ensures that P holds throughout an interval (apart from at the last point):

$$\textbf{keep } P \;\equiv\; \Box(\neg\textbf{empty} \Rightarrow P)$$

In addition, $\diamond P$ states that there exists an arbitrary interval on which P holds:

$$\diamond P \;\equiv\; \textbf{True} \;\textbf{;}\; P \;\textbf{;}\; \textbf{True}$$

5.3.3 ICS Derived Operators

We need some more derived operators - these are ICS specific. Firstly, we need a way of expressing stability. We do it as follows:

$$\begin{array}{rcl} \textbf{stable}(S,T) & \equiv & ((\textbf{beg}\,\texttt{tick} \;\wedge\; \bigcirc\;\textbf{keep}\,\neg\texttt{tick} \;\wedge\; \textbf{fin}\,\texttt{tick}) \\ & & \wedge \bigwedge_{\texttt{a}\,\in\, S} \Diamond\;\texttt{a}) \\ & & \;\textbf{proj}\;\;\textbf{len}(T) \end{array}$$

where $S \subseteq$ **Act** and $T \in \texttt{Nat}$. Also, we will write $\textbf{stable}(\{\texttt{a}\},T)$ as $\textbf{stable}(\texttt{a},T)$.

As an illustration,

$\textbf{stable}(\{\texttt{obj\_prop\_I}, \texttt{implic\_prop\_I}\}, 5)$

states that to view `obj_prop_I`[19] and `implic_prop_I` as simultaneously stable we must observe the two transformations repeating for five time units. In terms of our definition this means that we divide the interval over which the proposition holds into 5 contiguous subintervals in which both `obj_prop_I` and `implic_prop_I` occur. Each of these subintervals is bounded by `tick` actions, which ensure that the subinterval encodes the passage of one time unit. We measure the five time units by applying **len**(5) as the projected formula.

The following lemma encapsulates a very simple property of stability (it is a time continuity property).

Lemma 1
$\forall t, t' \in \texttt{Nat}\;.\;\textbf{stable}(S, t+t') \;\Leftrightarrow\; \textbf{stable}(S,t) \;\textbf{;}\; \textbf{stable}(S,t')$

[19]This is a LOTOS action with data flattened out. By way of illustration, the synchronisation of the two actions `g!2` and `g?n:Nat` would yield a flattened action `g_2`.

Proof
Assume $t, t' \in \texttt{Nat}$ and let,

$$\Phi = ((\mathbf{beg}\,\texttt{tick} \;\wedge\; \bigcirc\,\mathbf{keep}\,\neg\texttt{tick} \;\wedge\; \mathbf{fin}\,\texttt{tick}) \;\wedge\; \bigwedge_{a \,\in\, S} \Diamond\, a)$$

We can argue as follows (where we refer to specific laws of ITL, e.g. $[DistProj]$ which are listed in the appendix):

$$\begin{array}{ll}
 & \mathbf{stable}(S, t + t') \\
\equiv & \{ \text{ definition of stability } \} \\
 & \Phi \;\; \mathbf{proj} \;\; \mathbf{len}(t + t') \\
\equiv & \{ \; [AddLen] \; \} \\
 & \Phi \;\; \mathbf{proj} \;\; \mathbf{len}(t) \;\mathbf{;}\; \mathbf{len}(t') \\
\equiv & \{ \; [DistProj] \; \} \\
 & (\Phi \;\; \mathbf{proj} \;\; \mathbf{len}(t)) \;\mathbf{;}\; (\Phi \;\; \mathbf{proj} \;\; \mathbf{len}(t')) \\
\equiv & \{ \text{ definition of stability } \} \\
 & \mathbf{stable}(S, t) \;\mathbf{;}\; \mathbf{stable}(S, t')
\end{array}$$

$\bigcirc$

Using **stable** we can define an operator that determines the number of time units that will elapse in an interval:

$$\begin{array}{rcll}
\mathbf{elapsed}(0) & \equiv & \mathbf{empty} \;\wedge\; \texttt{tick} & \\
\mathbf{elapsed}(n) & \equiv & \mathbf{stable}(\emptyset, n) & \text{where } n \neq 0
\end{array}$$

Notice we count intervals between ticks rather than ticks themselves. This is in accordance with how intervals are treated in ITL, e.g. compare this with the interpretation of **len**. The following is a simple property which relates **stable** and **elapsed**.

Lemma 2
$\forall S \in \texttt{Act},\; t \in \texttt{Nat}\; (t \neq 0)\; .\; \mathbf{stable}(S, t) \implies \mathbf{elapsed}(t).$

Proof
Assume $t \neq 0$,

$$\begin{array}{ll}
 & \mathbf{stable}(S, t) \\
\equiv & \{ \text{ definition of stability } \} \\
 & ((\mathbf{beg}\,\texttt{tick} \;\wedge\; \bigcirc\,\mathbf{keep}\,\neg\texttt{tick} \;\wedge\; \mathbf{fin}\,\texttt{tick}) \;\wedge\; \bigwedge_{a \,\in\, S} \Diamond\, a) \;\; \mathbf{proj} \;\; \mathbf{len}(t) \\
\Rightarrow & \{ \; P \;\wedge\; Q \;\Rightarrow\; P \;;\; [MonoProj1] \; \} \\
 & (\mathbf{beg}\,\texttt{tick} \;\wedge\; \bigcirc\,\mathbf{keep}\,\neg\texttt{tick} \;\wedge\; \mathbf{fin}\,\texttt{tick}) \;\; \mathbf{proj} \;\; \mathbf{len}(t)
\end{array}$$

$$\equiv \quad \{ \text{ definition of elapsed } \}$$
$$\mathbf{elapsed}(t)$$

○

In addition, we can define a number of operators that we will use later. These characterise stable output at particular subsystems. Let `r` be an arbitrary representation. Then,

$$\begin{aligned}
\mathbf{lookat}(\mathtt{r}) &\equiv \mathbf{stable}(\mathtt{eye_vis_r}, T_{la}) \\
\mathbf{speak}(\mathtt{r}) &\equiv \mathbf{stable}(\mathtt{art_speech_r}, T_{sp}) \\
\mathbf{located}(\mathtt{r}) &\equiv \mathbf{stable}(\mathtt{lim_hand_r}, T_{lo}) \\
\mathbf{associate}(\mathtt{r}) &\equiv \mathbf{stable}(\mathtt{lim_hand_r}, T_{as})
\end{aligned}$$

Notice that for each of these stability operators we assume a natural number constant (T_{la}, T_{lo}, ..) which defines the length of time the particular action has to repeat for stability to have occured. For example, `art_speech_r` has to repeat for T_{sp} time units (actually $T_{sp} + 1$ ticks, because of the way **len** is defined) for stable speech (of representation `r`) to have occurred. As will be evident, `located` and `associate` are very similar properties, however, we distinguish them since they will arise at different places in our reasoning.

5.4 ICS Goals

As highlighted earlier, we are building upon two previous pieces of work on deixis in ICS: Faconti et al [21] and Duke et al [18]. The kind of deixis scenario they consider would be the capability of a system user to select from a list of items in a computer display while performing some other task, e.g. speaking. Central to both the Faconti et al and the Duke et al work is inconsistency arising due to attempted blending with representations denoting conflicting psychological subjects. So, the first issue to consider is, how will we model the notion of psychological subject.

Since it is the only aspect of representations that is relevant to our analysis we will view different natural number denotations as representing different psychological subjects. This means that comparing representations using natural number equality exactly corresponds to comparing whether the two representations have compatible psychological subject.

We must now consider the type of blending used at different subsystems. In particular, for the analysis that follows certain assumptions about blending and buffering will be needed in order that our analysis goes through. We consider these now:-

1. `lim_hand` acts upon a crude multiplicative blend of `bs_lim` and `obj_lim`. This blending is particularly important when we consider a mouse based interface. In this situation, the blend reflects that an association needs to

be set-up between the cursor in the visual world (which will arrive at `LIM` via `obj_lim`) and the current hand state which identifies the "zero" cursor position (which will arrive at `LIM` via `bs_lim`). Informally, the association states that "with my hand in the current position the cursor is located here".

2. `art_speech` acts on a crude multiplicative blend of `mpl_art` and `bs_art` for a similar reason to the previous point.

3. `mpl_art` acts on a deterministic information preserving blend of `obj_mpl` and `prop_mpl`. This ensures that these two input flows must be consistent for `mpl_art` to act on a non-null flow.

4. `obj_mpl` and `obj_lim` act on the same blend from `OBJ`'s input array.

5. We do not consider any buffering. However, the arguments would still be valid even in the presence of buffering, unless it was performed at either the `LIM` or the `ART` subsystems. Buffering at either of these subsystems would invalidate the reasoning we give in section 6.

Now let us highlight the goals that these previous workers have considered.

- Duke et al [18] consider the property:

  $$\text{[Duke et al 1]} \qquad (\forall \mathtt{I} \neq \mathtt{J}) \neg \diamondsuit (\mathbf{speak}(\mathtt{I}) \ \wedge \ \diamondsuit \, \mathbf{located}(\mathtt{J}))$$

  where `I` and `J` are representations. The property assumes that locating an item on the screen (i.e. pointing at it) does not take more time to stabilise than speech. In other words, $T_{lo} \leq T_{sp}$, which seems a reasonable assumption.

  In informal terms the property states that it is not possible to speak and point at "different" items on the screen at the same time, where different means, having different psychological subjects. Notice that no particular form of user interface, e.g. mouse or touch screen, is assumed.

  In addition, Duke et al consider the obvious related positive property:

  $$\text{[Duke et al 2]} \qquad (\forall \mathtt{I}) \diamondsuit (\mathbf{speak}(\mathtt{I}) \ \wedge \ \diamondsuit \, \mathbf{located}(\mathtt{I}))$$

  i.e. it is possible to speak and point simultaneously as long as it is the same item being considered in both cases.

- Faconti et al [21] relate the use of *mouse* based and *touch screen* interfaces. A number of properties come out of this work. First we consider the central negative property that they consider. It expresses that having read an item from the screen it is not possible to simultaneously pronounce/speak that item and point at it with a mouse based interface.

Assuming $T_{la} + T_{as} \leq T_{sp}$ we formalise the property using Mexitl as:

[Faconti et al 1]

$(\forall \mathtt{I}, \mathtt{B}, \mathtt{C}) \neg \diamondsuit(\mathbf{lookat}(\mathtt{I})\ ;\ (\mathbf{speak}(\mathtt{B} * \mathtt{I}) \wedge \diamondsuit(\mathbf{associate}(\mathtt{B} * \mathtt{C})\ ;\ \mathbf{lookat}(\mathtt{I}))))$

It has the following constituents:-

- I is a representation with psychological subject the desired item on the screen.
- C is a representation with psychological subject the cursor on the screen.
- B is an arbitrary representation (which will actually originate from the body state). It is used to indicate body feedback on the current state of the vocal chords and the hand.

The behaviour,

$$\mathbf{associate}(\mathtt{B} * \mathtt{C})\ ;\ \mathbf{lookat}(\mathtt{I})$$

is the most interesting part of the goal. It models that first body state feedback (position of hand) and the cursor are associated together. Then it denotes that the item is "seen". There is an assumption here that with an experienced user once he/she has the correct association between hand state and cursor and he/she finds the item he/she can move the cursor and select the item "automatically". In cognitive terms, the experienced user has a *proceduralisedised* understanding of the relationship between the distance and direction of movement of the mouse and the corresponding distance and direction of movement of the cursor.

Continuing with the mouse interface, [21] also consider two positive properties. These arise from sequentializing the actions involved in the deictic reference. We consider two such sequentializations[20]:

[Faconti et al 2]

$$(\forall \mathtt{I}, \mathtt{B}, \mathtt{C}) \diamondsuit(\mathbf{lookat}(\mathtt{I})\ ;\ \mathbf{speak}(\mathtt{B} * \mathtt{I})\ ;\ \mathbf{associate}(\mathtt{B} * \mathtt{C})\ ;\ \mathbf{lookat}(\mathtt{I}))$$

and,

[20] A third goal, in which the two cursor behaviours $\mathbf{associate}(\mathtt{B} * \mathtt{C})$ and $\mathbf{lookat}(\mathtt{I})$ are not viewed as atomic could also be considered. This goal would be:

$$(\forall \mathtt{I}, \mathtt{B}, \mathtt{I}) \diamondsuit(\mathbf{lookat}(\mathtt{I})\ ;\ \mathbf{associate}(\mathtt{B} * \mathtt{C})\ ;\ \mathbf{speak}(\mathtt{B} * \mathtt{I})\ ;\ \mathbf{lookat}(\mathtt{I}))$$

However, this goal is more difficult to analyse since it requires the item I to be retrieved from an image record when it is spoken. Consideration of this goal is left for further work.

[Faconti et al 3]

$$(\forall \mathtt{I},\mathtt{B},\mathtt{C}) \diamondsuit(\mathbf{lookat}(\mathtt{I})\ ;\ \diamondsuit(\mathbf{associate}(\mathtt{B} * \mathtt{C})\ ;\mathbf{lookat}(\mathtt{I}))\ ;\mathbf{speak}(\mathtt{B} * \mathtt{I}))$$

The second $\diamondsuit$ is needed here since we must allow some time to look at the cursor between the first **lookat** and the **associate**. This point will be clarified in section 7.

In addition, [21] consider deictic reference with a touchscreen interface. They argue that the simultaneous selection and speaking that ([Faconti et al 1] suggests) is not possible with a Mouse based interface, is possible with a touch screen interface. Due to the change of device, a different set of tasks is involved. The goal they consider is:

[Faconti et al 4]

$$(\forall \mathtt{I},\mathtt{B}) \diamondsuit(\mathbf{lookat}(\mathtt{I})\ ;(\mathbf{speak}(\mathtt{B} * \mathtt{I})\ \wedge\ \diamondsuit\, \mathbf{located}(\mathtt{B} * \mathtt{I})))$$

Importantly, in this goal it is not necessary to locate the cursor, thus, no change of psychological subject is required.

6 Verification

This section verifies that the two negative properties that we introduced in the last section are indeed satisfied by our ICS specification. They are [Duke et al 1]:

$$(\forall \mathtt{I} \neq \mathtt{J}) \neg \diamondsuit(\mathbf{speak}(\mathtt{I})\ \wedge\ \diamondsuit\, \mathbf{located}(\mathtt{J}))$$

and [Faconti et al 1]:

$$\neg \diamondsuit(\mathbf{lookat}(\mathtt{I})\ ;(\mathbf{speak}(\mathtt{B} * \mathtt{I})\ \wedge\ \diamondsuit(\mathbf{associate}(\mathtt{B} * \mathtt{C})\ ;\mathbf{lookat}(\mathtt{I}))))$$

Our strategy for verifying these properties is to show that they are both implied by a significantly simpler property and then show that this simpler property holds over ICS. We consider the simplification step in subsection 6.1 and the verification of the simpler property in subsection 6.2.

6.1 Simplification of Goals

Let us begin by considering [Faconti et al 1]. We work with the negation of [Faconti et al 1] (our final step will be to take the contrapositive of our argument to regain [Faconti et al 1]). Firstly,

$\neg$[Faconti et al 1]

$\equiv$ { De Morgan's laws }

$$(\exists \mathtt{I},\mathtt{B},\mathtt{C})\,\diamond(\mathbf{lookat}(\mathtt{I})\ \mathbf{;}\ (\mathbf{speak}(\mathtt{B}*\mathtt{I}) \wedge \diamond(\mathbf{associate}(\mathtt{B}*\mathtt{C})\ \mathbf{;}\ \mathbf{lookat}(\mathtt{I}))))$$

and we can further reason that,

$$
\begin{array}{ll}
 & \diamond(\mathbf{lookat}(\mathtt{I})\ \mathbf{;}\ (\mathbf{speak}(\mathtt{B}*\mathtt{I})\ \wedge\ \diamond(\mathbf{associate}(\mathtt{B}*\mathtt{C})\ \mathbf{;}\ \mathbf{lookat}(\mathtt{I})))) \\
\equiv & \{\ \text{definition of } \diamond\ \} \\
 & \mathbf{True}\ \mathbf{;}\ \mathbf{lookat}(\mathtt{I})\ \mathbf{;}\ (\mathbf{speak}(\mathtt{B}*\mathtt{I}) \wedge \\
 & \mathbf{True}\ \mathbf{;}\ \mathbf{associate}(\mathtt{B}*\mathtt{C})\ \mathbf{;}\ \mathbf{lookat}(\mathtt{I})\ \mathbf{;}\ \mathbf{True})\ \mathbf{;}\ \mathbf{True} \\
\Rightarrow & \{\ AssChop\ ;\ MonoChop2\ ;\ IdempChop\ ;\ P \Rightarrow \mathbf{True}\ \} \\
 & \mathbf{True}\ \mathbf{;}\ (\mathbf{speak}(\mathtt{B}*\mathtt{I}) \wedge \\
 & \mathbf{True}\ \mathbf{;}\ \mathbf{associate}(\mathtt{B}*\mathtt{C})\ \mathbf{;}\ \mathbf{lookat}(\mathtt{I})\ \mathbf{;}\ \mathbf{True})\ \mathbf{;}\ \mathbf{True} \\
\Rightarrow & \{\ AssChop\ ;\ MonoChop1\ ;\ IdempChop\ ;\ \text{monotonicity of } \wedge\ \} \\
 & \mathbf{True}\ \mathbf{;}\ (\mathbf{speak}(\mathtt{B}*\mathtt{I})\ \wedge\ \mathbf{True}\ \mathbf{;}\ \mathbf{associate}(\mathtt{B}*\mathtt{C})\ \mathbf{;}\ \mathbf{True})\ \mathbf{;}\ \mathbf{True} \\
\equiv & \{\ \text{definition of } \mathbf{associate} \text{ and } \mathbf{speak} \text{ [STEP *]}\ \} \\
 & \mathbf{True}\ \mathbf{;}\ (\mathbf{stable}(\mathtt{art\_speech\_B}*\mathtt{I},T_{sp}) \wedge \\
 & \mathbf{True}\ \mathbf{;}\ \mathbf{stable}(\mathtt{lim\_hand\_B}*\mathtt{C},T_{as})\ \mathbf{;}\ \mathbf{True})\ \mathbf{;}\ \mathbf{True} \\
\Rightarrow & \{\ \text{Can replace } \mathbf{True} \text{ by } (\exists x)\mathbf{elapsed}(x) \text{ since at least one tick} \\
 & \quad \text{must occur ; by comparing interval lengths}\ \} \\
 & (\exists x_1,x_2)\mathbf{True}\ \mathbf{;}\ (\mathbf{stable}(\mathtt{art\_speech\_B}*\mathtt{I},T_{sp}) \wedge \\
 & \mathbf{elapsed}(x_1)\ \mathbf{;}\ \mathbf{stable}(\mathtt{lim\_hand\_B}*\mathtt{C},T_{as})\ \mathbf{;}\ \mathbf{elapsed}(x_2))\ \mathbf{;} \\
 & \mathbf{True}\ \wedge\ x_1+T_{as}+x_2=T_{sp} \\
\equiv & \{\ \text{lemma 1}\ ;\ AssChop\ \} \\
 & (\exists x_1,x_2)\mathbf{True}\ \mathbf{;}\ (\\
 & (\mathbf{stable}(\mathtt{art\_speech\_B}*\mathtt{I},x_1)\ \mathbf{;} \\
 & \mathbf{stable}(\mathtt{art\_speech\_B}*\mathtt{I},T_{as})\ \mathbf{;} \\
 & \mathbf{stable}(\mathtt{art\_speech\_B}*\mathtt{I},x_2)) \wedge \\
 & (\mathbf{elapsed}(x_1)\ \mathbf{;}\ \mathbf{stable}(\mathtt{lim\_hand\_B}*\mathtt{C},T_{as})\ \mathbf{;}\ \mathbf{elapsed}(x_2)))\ \mathbf{;} \\
 & \mathbf{True}\ \wedge\ x_1+T_{as}+x_2=T_{sp} \\
\Rightarrow & \{\ \text{lemma 2}\ ;\ MonoChop1\ ;\ AssChop\ ;\ \text{monotonicity of } \wedge\ \} \\
 & (\exists x_1,x_2)\mathbf{True}\ \mathbf{;}\ (\\
 & (\mathbf{elapsed}(x_1)\ \mathbf{;}\ \mathbf{stable}(\mathtt{art\_speech\_B}*\mathtt{I},T_{as})\ \mathbf{;}\ \mathbf{elapsed}(x_2)) \wedge \\
 & (\mathbf{elapsed}(x_1)\ \mathbf{;}\ \mathbf{stable}(\mathtt{lim\_hand\_B}*\mathtt{C},T_{as})\ \mathbf{;}\ \mathbf{elapsed}(x_2))) \\
 & \mathbf{;}\ \mathbf{True}\ \wedge\ x_1+T_{as}+x_2=T_{sp} \\
\Rightarrow & \{\ \mathbf{elapsed}(t) \text{ is rigid and } DistChop\ \} \\
 & (\exists x_1,x_2)\mathbf{True}\ \mathbf{;}\ (\mathbf{elapsed}(x_1)\ \mathbf{;} \\
 & (\mathbf{stable}(\mathtt{art\_speech\_B}*\mathtt{I},T_{as})\ \wedge\ \mathbf{stable}(\mathtt{lim\_hand\_B}*\mathtt{C},T_{as}))\ \mathbf{;} \\
 & \mathbf{elapsed}(x_2))\ \mathbf{;}\ \mathbf{True}\ \wedge\ x_1+T_{as}+x_2=T_{sp} \\
\equiv & \{\ \text{definition of } \mathbf{stable}\ \} \\
 & (\exists x_1,x_2)\mathbf{True}\ \mathbf{;}\ (\mathbf{elapsed}(x_1)\ \mathbf{;}
\end{array}
$$

$\textbf{stable}(\{\texttt{art\_speech\_B} * \texttt{I}, \texttt{lim\_hand\_B} * \texttt{C}\}, T_{as})$ **;**
$\textbf{elapsed}(x_2))$ **; True** $\wedge\ x_1 + T_{as} + x_2 = T_{sp}$
$\Rightarrow$ { $P \Rightarrow$ **True** ; *IdempChop* ; *AssChop* }
True ; $\textbf{stable}(\{\texttt{art\_speech\_B} * \texttt{I}, \texttt{lim\_hand\_B} * \texttt{C}\}, T_{as})$ **; True**
$\equiv$ { definition of $\diamondsuit$ }
$\diamondsuit\,\textbf{stable}(\{\texttt{art\_speech\_B} * \texttt{I}, \texttt{lim\_hand\_B} * \texttt{C}\}, T_{as})$

Finally, putting together this argument and our first argument we can reason that:

$\neg$[Faconti et al 1]
$\Rightarrow$ { Monotonicity of $\exists$ }
$(\exists \texttt{I}, \texttt{B}, \texttt{C})\,\diamondsuit\,\textbf{stable}(\{\texttt{art\_speech\_B} * \texttt{I}, \texttt{lim\_hand\_B} * \texttt{C}\}, T_{as})$

○

So, this line of argument gives us that:

Property [a]

$\neg$[Faconti et al 1]
$\Rightarrow$ { Above argument }
$(\exists \texttt{I}, \texttt{B}, \texttt{C})\,\diamondsuit\,\textbf{stable}(\{\texttt{art\_speech\_B} * \texttt{I}, \texttt{lim\_hand\_B} * \texttt{C}\}, T_{as})$

Now let us additionally consider the property [Duke et al 1]. We can argue as follows:-

$\neg$[Duke et al 1]
$\equiv$ { De Morgan's laws ; clash avoiding variable renaming }
$(\exists \texttt{J} \neq \texttt{K})\,\diamondsuit(\textbf{speak}(\texttt{K})\ \wedge\ \diamondsuit\,\textbf{located}(\texttt{J}))$
$\Rightarrow$ { Weakening Conditions }
$(\exists \texttt{J}, \texttt{K})\,\diamondsuit(\textbf{speak}(\texttt{K})\ \wedge\ \diamondsuit\,\textbf{located}(\texttt{J}))$
$\equiv$ { $\{\texttt{Y} * \texttt{Z} \mid \texttt{Y}, \texttt{Z} \in \texttt{Rep}\} = \texttt{Rep}$ }
$(\exists \texttt{I}, \texttt{B}, \texttt{C})\,\diamondsuit(\textbf{speak}(\texttt{B} * \texttt{I})\ \wedge\ \diamondsuit\,\textbf{located}(\texttt{B} * \texttt{C}))$

Furthermore,

$\diamondsuit(\textbf{speak}(\texttt{B} * \texttt{I})\ \wedge\ \diamondsuit\,\textbf{located}(\texttt{B} * \texttt{C}))$
$\equiv$ { Assuming $T_{lo} = T_{as}$[21] }
[STEP *]

Now we can re-use the previous proof from [STEP *] to deduce that,

[21] This assumption that the time required to achieve stability of location and of association is the same, is an eminently reasonable assumption, since they are such related behaviours.

Property [b]

$$\begin{array}{ll} & \neg[\text{Duke et al 1}] \\ \Rightarrow & \{\ \text{Above argument}\ \} \\ & (\exists \mathtt{I}, \mathtt{B}, \mathtt{C}) \diamondsuit \mathbf{stable}(\{\mathtt{art\_speech\_B} * \mathtt{I}, \mathtt{lim\_hand\_B} * \mathtt{C}\}, T_{as}) \end{array}$$

We can now take the contrapositive of property [a] and [b] to obtain:-

$$\begin{array}{ll} & \neg(\exists \mathtt{I}, \mathtt{B}, \mathtt{C}) \diamondsuit \mathbf{stable}(\{\mathtt{art\_speech\_B} * \mathtt{I}, \mathtt{lim\_hand\_B} * \mathtt{C}\}, T_{as}) \\ \Rightarrow & \{\ \text{Above argument}\ \} \\ & [\text{Faconti et al 1}] \text{ and } [\text{Duke et al 1}] \end{array}$$

Thus, all we have to verify is,

$$\neg(\exists \mathtt{I}, \mathtt{B}, \mathtt{C}) \diamondsuit \mathbf{stable}(\{\mathtt{art\_speech\_B} * \mathtt{I}, \mathtt{lim\_hand\_B} * \mathtt{C}\}, T_{as})$$

and [Faconti et al 1] and [Duke et al 1] follow.

Even if the formal reasoning we have given here is complex in places, the intuition behind our argument should be straightforward. We can summarise it as follows:

> If we can show that a stable output for `art_speech_B*I` and `lim_hand_B*C` cannot be simultaneously generated for T_{as} time units, then it is certainly the case that the more complicated properties encoded in ¬[Faconti et al 1] and ¬[Duke et al 1] will also fail to hold.

6.2 Property Verification

Now we argue that the property,

$$\neg(\exists \mathtt{I}, \mathtt{B}, \mathtt{C}) \diamondsuit \mathbf{stable}(\{\mathtt{art\_speech\_B} * \mathtt{I}, \mathtt{lim\_hand\_B} * \mathtt{C}\}, T_{as})$$

holds over ICS. In standard fashion we denote satisfaction over ICS as $ICS \models P$, i.e. ICS satisfies the property P. We proceed by considering what the implications would be if,

$$ICS \models (\exists \mathtt{I}, \mathtt{B}, \mathtt{C}) \diamondsuit \mathbf{stable}(\{\mathtt{art\_speech\_B} * \mathtt{I}, \mathtt{lim\_hand\_B} * \mathtt{C}\}, T_{as})$$

held. We can argue as follows:

$$\begin{array}{ll} & ICS \ \models \ (\exists \mathtt{I}, \mathtt{B}, \mathtt{C}) \diamondsuit \mathbf{stable}(\{\mathtt{art\_speech\_B} * \mathtt{I}, \mathtt{lim\_hand\_B} * \mathtt{C}\}, T_{as}) \\ \equiv & \{\ \text{definition of } \diamondsuit \text{ ; from synchrony of ICS } (\exists n)\mathbf{elapsed}(n) \Leftrightarrow \mathbf{True}\ \} \\ & ICS \ \models \ (\exists \mathtt{I}, \mathtt{B}, \mathtt{C}, n)\mathbf{elapsed}(n) \ \mathbf{;} \\ & \mathbf{stable}(\{\mathtt{art\_speech\_B} * \mathtt{I}, \mathtt{lim\_hand\_B} * \mathtt{C}\}, T_{as}) \ \mathbf{;}\ \mathbf{True} \\ \Rightarrow & \{\ \mathtt{B*I} \text{ is a multiplicative blend of } \mathtt{mpl\_art} \text{ and } \mathtt{bs} \text{ ;} \end{array}$$

$\quad\quad$ `B*C` is a multiplicative blend of `obj_lim` and `bs` ;
$\quad\quad$ connectivity of $conf_{deixis}$ }

$$ICS \models (\exists \mathtt{I}, \mathtt{C}, n)\mathbf{elapsed}(n-1)\ ;$$
$$\mathbf{stable}(\{\mathtt{mpl\_art\_I}, \mathtt{obj\_lim\_C}\}, T_{as})\ ;\ \mathbf{True}$$

$\Rightarrow$ { `obj_mpl` and `obj_lim` act on same blend from `OBJ`'s input array }

$$ICS \models (\exists \mathtt{I}, \mathtt{C}, n)\mathbf{elapsed}(n-1)\ ;$$
$$\mathbf{stable}(\{\mathtt{mpl\_art\_I}, \mathtt{obj\_mpl\_C}\}, T_{as})\ ;\ \mathbf{True}$$

$\Rightarrow$ { `mpl_art` acts on deterministic information preserving blend
of `obj_mpl` (and `prop_mpl`) }

$$ICS \models (\exists \mathtt{I}, \mathtt{C}, n)\mathbf{elapsed}(n)\ ;$$
$$\mathbf{stable}(\{\mathtt{mpl\_art\_I}, \mathtt{obj\_mpl\_C}\}, T_{as}-1)\ ;\ \mathbf{True}$$
$$\wedge\ \mathtt{C} = \mathtt{I}$$

$\Rightarrow$ { $P \wedge Q \Rightarrow Q\ ; P \Rightarrow Q \implies (M \models P \Rightarrow M \models Q)$ }

$$ICS \models \mathtt{C} = \mathtt{I}$$

○

Thus, we have the following:

$$ICS \models (\exists \mathtt{I}, \mathtt{B}, \mathtt{C}) \diamond \mathbf{stable}(\{\mathtt{art\_speech\_B} * \mathtt{I}, \mathtt{lim\_hand\_B} * \mathtt{C}\}, T_{as})$$

$\Rightarrow$ { Above argument }

$$ICS \models \mathtt{C} = \mathtt{I}$$

from which we can take the contrapositive to obtain:

$$\neg(ICS \models \mathtt{C} = \mathtt{I})$$

$\Rightarrow$ { Above argument }

$$\neg(ICS \models (\exists \mathtt{I}, \mathtt{B}, \mathtt{C}) \diamond \mathbf{stable}(\{\mathtt{art\_speech\_B} * \mathtt{I}, \mathtt{lim\_hand\_B} * \mathtt{C}\}, T_{as}))$$

$\equiv$ { Logic }

$$ICS \models \neg((\exists \mathtt{I}, \mathtt{B}, \mathtt{C}) \diamond \mathbf{stable}(\{\mathtt{art\_speech\_B} * \mathtt{I}, \mathtt{lim\_hand\_B} * \mathtt{C}\}, T_{as}))$$

Now using the argument made in the previous subsection, we can deduce that:

$$ICS \models \text{[Faconti et al 1]} \quad \text{and} \quad ICS \models \text{[Duke et al 1]}$$

This line of reasoning is illustrated in table 1, which shows the representations acted on by particular transformations over time. The indice in this table do not indicate "semantically" different representations, e.g. $\mathtt{I}_i = \mathtt{I}_k$ for all $0 \leq j, k \leq n + T_{as}$, but rather simply allow instances of representations in different columns to be related, e.g. the `I` representation appearing on `mpl_art`

Time						
	lim_hand	obj_lim	obj_mpl	mpl_art		art_speech
.	..	..	..	..	..	..
.	..	..	..	..	..	..
$n-1$	..	C_0	C_0	..	I_0	..
n	$B*C_0$	C_1	C_1	C_0	I_1	$B*I_0$
.	$B*C_1$	C_2	C_2	C_1	I_2	$B*I_1$
.	$B*C_2$	C_3	C_3	C_2	I_3	$B*I_2$
.	..	..	..	C_3	..	..
.	$B*C_{T_{as}-1}$	$C_{T_{as}}$	$C_{T_{as}}$	..	$I_{T_{as}}$	$B*I_{T_{as}-1}$
$n+T_{as}$	$B*C_{T_{as}}$	.	.	$C_{T_{as}}$	.	$B*I_{T_{as}}$

Table 1: Illustration of Reasoning - $C_i = I_{i+1}$

at $n-1$ is the same instance of I that appears at art_speech at time n. The arguments given induce that $C_i = I_{i+1}$.

It is also worth pointing out that we have made a number of assumptions while making this argument. These are basic constraints which need to be imposed on ICS in order for [Faconti et al 1] and [Duke et al 1] to hold. They can be summarised as:

1. lim_hand acts upon a crude multiplicative blend of bs_lim and obj_lim.
2. art_speech acts upon a crude multiplicative blend of bs_art and mpl_art.
3. obj_mpl and obj_lim act on the same blend from OBJ's input array.
4. mpl_art acts on a deterministic information preserving blend of obj_mpl (and prop_mpl).
5. $T_{as} > 0$ (notice that if $T_{as} = 0$, i.e. for stability only one transformation needs to be observed, then I and C do not have to be related).
6. There is no buffering at LIM or ART.

However, these all seem reasonable assumptions, considering the nature of the cognitive task we are focusing on.

7 Simulation Analysis

Up to now we have restricted ourselves to verification of negative goals - what ICS cannot do. In this section we consider positive goals - what ICS is capable of doing. Importantly, we can only consider what it is capable of doing, what we do not verify is that it always performs a particular goal if it is set running. We would need a more prescriptive/mechanistic interpretation of ICS for this.

So, we are interested in possible behaviour, in testing theory terms - what *may* be performed rather than what *must* be performed.

There are a number of ways to perform such may verifications, one of which is indeed to use a testing approach. This would involve defining a tester process which exhibits the behaviour required by the goal, composing it in parallel with ICS and checking whether the test *may* succeed. Furthermore, there are tools available which when given a system and a testing process determine whether the test can succeed, e.g. the tool LOLA [16]. Although such testing is certainly very applicable to verification of positive goals, here we have taken a different approach. Our alternative is extremely simple and does not require as extensive an analysis of the system state space as a testing approach. The alternative is to simply explore by hand the state space of our ICS specification using a simulation engine in order to show that a particular trace (interval) can be performed by the specification. Exhibiting this interval will exactly show that ICS *can* satisfy the particular ITL goal being considered. In order to show that it *must* satisfy a particular goal would require us to look at all the intervals that can be generated from ICS.

We have used two tools in this simulation work - LOLA [16] and Smile [20] which are provided with the LOTOS tool kit - Lite [32]. Both tools allow the user of the system to step through the state space of a LOTOS specification. Whenever the specification reaches a choice point, the user decides how to resolve the choice. Thus conceptually, the user is providing the behaviour of the environment. Actually the user provides more than this, since the user also resolves any non-determinism in the specification.

The four positive goals that we highlighted in section 5 are:

[Duke et al 2]

$$(\forall \mathtt{I})\, \diamondsuit(\mathbf{speak}(\mathtt{I}) \;\wedge\; \diamondsuit\, \mathbf{located}(\mathtt{I}))$$

[Faconti et al 2]

$$(\forall \mathtt{I,B,C})\, \diamondsuit(\mathbf{lookat}(\mathtt{I})\;\mathbf{;}\;\mathbf{speak}(\mathtt{B}*\mathtt{I})\;\mathbf{;}\;\mathbf{associate}(\mathtt{B}*\mathtt{C})\;\mathbf{;}\;\mathbf{lookat}(\mathtt{I}))$$

[Faconti et al 3]

$$(\forall \mathtt{I,B,C})\, \diamondsuit(\mathbf{lookat}(\mathtt{I})\;\mathbf{;}\;\diamondsuit(\mathbf{associate}(\mathtt{B}*\mathtt{C})\;\mathbf{;}\;\mathbf{lookat}(\mathtt{I}))\;\mathbf{;}\;\mathbf{speak}(\mathtt{B}*\mathtt{I}))$$

[Faconti et al 4]

$$(\forall \mathtt{I,B})\, \diamondsuit(\mathbf{lookat}(\mathtt{I})\;\mathbf{;}\;(\mathbf{speak}(\mathtt{B}*\mathtt{I}) \;\wedge\; \diamondsuit\, \mathbf{located}(\mathtt{B}*\mathtt{I})))$$

We consider these in turn.

[Duke et al 2] and [Faconti et al 4]. We can show that [Faconti et al 4] implies [Duke et al 2] as follows:

$$(\forall \mathtt{I,B})\, \diamondsuit(\mathbf{lookat}(\mathtt{I})\;\mathbf{;}\;(\mathbf{speak}(\mathtt{B}*\mathtt{I}) \;\wedge\; \diamondsuit\, \mathbf{located}(\mathtt{B}*\mathtt{I})))$$

Time				
	eye_vis	art_speech	lim_hand	body_bs
0	# I_0	0	0	B_0
1	# I_1	0	0	B_1
2	# I_2	0	0	B_2
3	# I_3	0	I_0*B_1	B_3
4	I_4	+ I_0*B_2	\$ I_1*B_2	B_4
5	I_5	+ I_1*B_3	\$ I_2*B_3	B_5
6	I_6	+ I_2*B_4	\$ I_3*B_4	B_6
7	I_7	+ I_3*B_5	\$ I_4*B_5	B_7
8	I_8	+ I_4*B_6	I_5*B_6	B_8
9	I_9	+ I_5*B_7	I_6*B_7	B_9
10	I_{10}	+ I_6*B_8	I_7*B_8	B_{10}
11	I_{11}	+ I_7*B_9	I_8*B_9	B_{11}

Table 2: Interval for [Duke et al 2] and [Faconti et al 4]

$$
\begin{aligned}
\Rightarrow &\quad \{ \ P \Rightarrow \mathbf{True} \ ; MonoChop1 \ ; AssChop \ ; IdempChop \ \} \\
& (\forall \mathtt{I}, \mathtt{B}) \diamondsuit(\mathbf{speak}(\mathtt{B} * \mathtt{I}) \ \wedge \ \diamondsuit \, \mathbf{located}(\mathtt{B} * \mathtt{I})) \\
\equiv &\quad \{ \ \{ \mathtt{Z} * \mathtt{Y} \,|\, \mathtt{Z}, \mathtt{Y} \in \mathtt{Rep} \} = \mathtt{Rep} \ \} \\
& (\forall \mathtt{J}) \diamondsuit(\mathbf{speak}(\mathtt{J}) \ \wedge \ \diamondsuit \, \mathbf{located}(\mathtt{J}))
\end{aligned}
$$

Thus, a trace which validates [Faconti et al 4] will also validate [Duke et al 2]. This joint validation will be clear from inspection of the trace we exhibit. The full trace/interval is too big to present since during each time unit all the ICS transformations have to be performed. However, table 2 depicts how representations arise on the key transformations `eye_vis`, `art_speech`, `lim_hand` and `body_bs` in the fulfilling interval.

In order to perform this validation we have to assign values to T_{sp} and T_{lo}, i.e. we have to decide how many time units have to pass for stability of speech and location to have occured. We take the decision that[22]:

$$T_{sp} = 8 \quad \text{and} \quad T_{lo} = 4$$

however, it is also clear from our simulation runs that if T_{sp} and T_{lo} are set to any arbitrary values we could generate the necessary intervals to validate [Duke et al 2] and [Faconti et al 4]. We indicate the stable outputs that yield key satisfying components as follows:

lookat - # ; **speak** - + ; **located** - \$

[22]Factors influencing this choice of values are that we have accumulated the following constraints from earlier arguments, $T_{lo} = T_{as}$ and $T_{as} + T_{la} \leq T_{sp}$.

Time				
	eye_vis	art_sp	lim_hand	body_bs
0	# I_0	0	0	B_0
1	# I_1	0	0	B_1
2	# I_2	0	0	B_2
3	# I_3	0	$I_0 * B_1$	B_3
4	I_4	+ $I_0 * B_2$	$I_1 * B_2$	B_4
5	I_5	+ $I_1 * B_3$	$I_2 * B_3$	B_5
6	I_6	+ $I_2 * B_4$	$I_3 * B_4$	B_6
7	I_7	+ $I_3 * B_5$	$I_4 * B_5$	B_7
8	C_0	+ $I_4 * B_6$	$I_5 * B_6$	B_8
9	C_1	+ $I_5 * B_7$	$I_6 * B_7$	B_9
10	C_2	+ $I_6 * B_8$	$I_7 * B_8$	B_{10}
11	C_3	+ $I_7 * B_9$	\$ $C_0 * B_9$	B_{11}
12	# I_8	$C_0 * B_{10}$	\$ $C_1 * B_{10}$	B_{12}
13	# I_9	$C_1 * B_{11}$	\$ $C_2 * B_{11}$	B_{13}
14	# I_{10}	$C_2 * B_{12}$	\$ $C_3 * B_{12}$	B_{14}
15	# I_{11}	$C_3 * B_{13}$	$C_4 * B_{13}$	B_{15}

Table 3: Interval for [Faconti et al 2]

What the validation states is that if we have a stable input of `I` for a sufficient period of time at `eye_vis` eventually this input will feed through the system (blending with body state input on the way) to provide a stable (simultaneous) output at `art_speech` and `lim_hand` (consider the 8 time units, 4, 5, 6, 7, 8, 9, 10 and 11). Notice that meaningful output at `lim_hand` starts one time unit before it does at `art_speech`. This is due to the connectivity in $conf_{deixis}$.

[Faconti et al 2]. In a similar way we can validate this property. We can exhibit a trace/interval that exactly realises the property. However, we have rather presented an optimum trace/interval which exhibits the earliest time points at which a transformation can stabilise. Consequently, the subtasks of this goal, e.g. **lookat**(`I`) and **speak**(`B` $*$ `I`), are not completely sequentialised, rather they have some (but not complete) overlap. In addition, in the same way as previously we make the assumption:

$$T_{la} = T_{as} = 4 \text{ and } T_{sp} = 8$$

Table 3 illustrates the key aspects of the required interval. We indicate the stable outputs that yield components of [Faconti et al 2] as follows:

lookat - # ; **speak** - + ; **associate** - \$

[Faconti et al 3]. Once again we exhibit an "optimum" trace to validate this property, it is shown in table 4. Notice that as suggested by the second $\diamondplus$ in our goal:

Time				
	`eye_vis`	`art_sp`	`lim_hand`	`body_bs`
0	# I_0	.	.	B_0
1	# I_1	.	.	B_1
2	# I_2	.	.	B_2
3	# I_3	.	.	B_3
4	C_0	.	.	B_4
5	C_1	.	.	B_5
6	C_2	.	.	B_6
7	C_3	.	\$ C_0*B_5	B_7
8	I_4	.	\$ C_1*B_6	B_8
9	I_5	.	\$ C_2*B_7	B_9
10	I_6	.	\$ C_3*B_8	B_{10}
11	# I_7	.	.	B_{11}
12	# I_8	+ B_{10}*I_4	.	B_{12}
13	# I_9	+ B_{11}*I_5	.	B_{13}
14	# I_{10}	+ B_{12}*I_6	.	B_{14}
15	I_{11}	+ B_{13}*I_7	.	B_{15}
16	I_{12}	+ B_{14}*I_8	.	B_{16}
17	I_{13}	+ B_{15}*I_9	.	B_{17}
18	I_{14}	+ B_{16}*I_{10}	.	B_{18}
19	I_{15}	+ B_{17}*I_{11}	.	B_{19}

Table 4: Interval for [Faconti et al 3]

$$(\forall \mathtt{I},\mathtt{B},\mathtt{C})\,\Diamond(\mathbf{lookat}(\mathtt{I})\ ;\ \Diamond(\mathbf{associate}(\mathtt{B}*\mathtt{C})\ ;\mathbf{lookat}(\mathtt{I}))\ ;\mathbf{speak}(\mathtt{B}*\mathtt{I}))$$

the stable output of `B*C` at `lim_hand` cannot occur directly after **lookat**(`I`) since a stable input with psychological subject the cursor, i.e. `C`, must be received at `eye_vis` between the two behaviours. We indicate component stable outputs in the same way as in the previous table.

8 Conclusions

8.1 Discussion

We have applied techniques from concurrency theory to cognitive modelling. Our strategy has been to take an existing cognitive model, ICS, and interpret it in a standard concurrency theory notation - the process calculus, LOTOS. In addition, we have introduced an interval temporal logic, Mexitl, in which we have formulated a number of goals for ICS. Finally, we verified these goals using logical deduction and simulation techniques.

We have given a number arguments in the introduction to this paper for applying concurrency theory techniques to cognitive modelling. We will not reiterate these here. However, it is worth reconsidering the nature of the LOTOS specification of ICS and how our concurrency theory notations relate to more standard techniques for cognitive modelling.

Firstly, there is a clear spectrum of available modelling techniques, with the two extremes being programming based approaches, such as those typically used in cognitive modelling and techniques based on mathematical logic. A weakness of the former approaches is that they are often too prescriptive, forcing a particular "mechanistic" interpretation on the cognitive model. In contrast, a weakness of modelling based on the latter approaches is that logical descriptions often express global properties across the entire system. Consequently, such approaches typically fail to reflect the underlying component structure of the system being modelled. This can, for example, be seen in our interval temporal logic goals which express desired "overall" behaviour of ICS, but do not describe the system componentwise in any way.

Process calculi can be seen to sit between these two extremes. Firstly, the LOTOS specification we have given certainly reflects the component structure of the ICS model, e.g. we have a LOTOS process for each ICS subsystem. This makes the specification easier to understand and to maintain. Previous Modal Action Logic [21] based descriptions of ICS have not so directly reflected the component structure of ICS.

Secondly, process calculi provide tools for avoiding overprescriptive description of systems. In particular, they facilitate loose specification by allowing descriptions to contain non-determinism.

Another feature that prevents overspecification is the role of the environment. Often when describing systems it is unclear how to prescribe a certain behaviour. In process calculi, rather than forcing a particular mechanistic interpretation we can leave the decision open and let the environment make it. A good example of the use of such a strategy arises in our modelling of buffered mode behaviour. Specifically, the mechanism by which buffered mode is entered is still a matter for debate. Thus, rather than forcing a particular interpretation, we allow the observer of the system to control which subsystem enters buffered mode. This is done by offering actions such as:

```
obj_buffered
```

to the environment.

Another sense in which LOTOS specifications have a logical character is that they enable "conjunction" of global constraints. Such constraints can be composed in parallel with the system with the effect that the composite system reflects both the properties of the system and the added constraint[23]. For example, the process,

[23] In fact, it can be shown that parallel composition does not always behave in a truly conjunctive manner [7, 10], however, this subtlety is not important in the context of this paper.

```
buffConstraint
```

(globally) constrains the number of subsystems that can enter buffered mode.

Finally, we believe that the work presented here has made a valuable first step in a new area of research. However, clearly the techniques considered are not mature and there are many avenues for future research, which we consider now.

8.2 Further Work

We list some of the many topics for future research:-

- **LOTOS Specification Refinements.** There are a number of ways in which our LOTOS specification of ICS could be refined.
  - *More Generic Description* The description of ICS is not as simple or elegant as one might like. This is largely due to limitations in the expressiveness of LOTOS. One issue, for example, is that even though all subsystems have a very similar structure, we have to give a complete "specialized" description of each subsystem. Thus, we are not able to capture the generality of ICS subsystems. A better solution would be to define a single process which models a generic subsystem and then specialize it through parameter instantiation for each particular subsystem. An extended version of the LOTOS notation, to be called E-LOTOS, is currently being defined [30]. It adds a number of features that enhance the expressiveness of the language. It would be very interesting to see if using these enhancements would lead to a simpler and more elegant description of ICS.
  - *Alternative Top Level Structure.* In order to, perhaps, even more fully capture the basic ICS structure, we could give a top level structure to our description that has the following form:

    ```
    Subsystem_1 ||| Subsystem_2 ||| .... ||| Subsystem_n
    |[ ..... synch. gates .... ]|
    Network
    ```

    where the `Network` process would receive outputed representations from `Subsystems` and relay them to target `Subsystems`. Clearly, each of our current inter subsystem actions would have to be subdivided into two actions - a subsystem to network output action and a network to subsystem input action.
  - *Interactive Choice of Configurations.* We could allow the system user to choose the particular configuration he/she is interested in when he/she starts simulating with ICS. Depending upon the user input the system would evolve to a different top level composition of processes.

  - *A Set-up Constraint.* It might also aid usage of the system if we included a constraint process which could enforce user preferences on a particular simulation. For example, we could leave all decisions, such as which subsystem enters buffered mode or what forms of blending to use, for the user. Perhaps he/she could select a different constraint depending upon his/her particular preferences.

- **Redundant Outputs.** A technical issue which arises is that our ICS simulations typically generate "redundant" as well as meaningful outputs. For example, table 3 shows a fulfilling interval for the property [Faconti et al 2] and the trace is indeed satisfactory for this purpose. However, as a by product of fulfilling the goal, a stable output of the representation `C*B` is also generated at `art_speech` between time units `12` and `15`. It is somewhat difficult to assign a sensible meaning to such "speaking of a representation of the cursor". The existence of such *noise* in our system does not prevent us from analysing *may* capabilities, however, our verifications would be more justifiable if we could give an intuitive explanation for such redundant output. Alternatively of course, we could try to rework our LOTOS specification in order to eradicate such noise.

- **More Expressive Process Calculi.** In many respects the process calculus that we have used, LOTOS, is rather primitive. It is a product of the "first generation" of process calculi. However, there are now richer techniques (indeed such as E-LOTOS) which incorporate more advanced modelling capabilities, e.g. *real-time* process calculi [43], *probabilistic* and *stochastic* notations [25] and *mobile* calculi [39]. All of these added features are in one way or another relevant to the modlling of cognitive systems. For example, a full description of ICS would clearly need to explain how to move between configurations. Such dynamic reconfiguration of systems suggests that a mobile calculus should be used.

- **Executable Description.** A further limitation of the work presented here is that we do not generate an executable (in programming language terms) description of ICS. However, tools exist for generating executable code from LOTOS specifications, e.g. [33]. Applying these tools in the ICS context is an important topic for future research.

- **Alternative Formal Paradigms.** The concurrency theory field is now very rich and in addition to LOTOS and, process calculi in general, there are many alternative techniques. These each have different flavours and different relative benefits. Describing and analysing ICS in these alternative approaches is an obvious topic for future work. Three approaches that we are particularly interested to investigate are:

  1. *Complete Mexitl Description.* Describing ICS in Mexitl. This would enable us to reason directly (in the same formalism) between our ICS goals and ICS specification.

2. *Model Checking.* One of the most mature concurrency theory approaches is model checking, where by, an automata based description of a system is checked for satisfiability against a temporal logic property. Such verification could clearly be applied to ICS. The obvious technique to use would be Holzmann's SPIN/PROMELA formalism [29].
3. *Other Process Calculi.* Finally, we could use a different process calculus approach, say CSP (or even CCS). With CSP we could additionally formulate our ICS goals as CSP processes and check refinement between the system description and the goal description using the FDR tool [47].

Acknowledgements

The work presented here has been performed in the context of the TMR TACIT project and thus, I must thank all the members of the project. In particular, I would like to thank David Duke, David Duce, Meike Massink and John May with whom I have had valuable discussions about ICS. In addition, Meike Massink provided valuable comments on a draft of this paper. However, my greatest thanks go to Giorgio Faconti who has championed this work at CNR Istituto-CNUCE and has frequently over a Capuccino and a Ciocolatina (or is it a Ciocalatino, I can never remember) put me straight with regard to ICS.

References

[1] P.J. Barnard. Interacting cognitive subsystems: A psycholinguistic approach to short-term memory. In *Progress in the Psychology of Language*, volume 2. Lawrence Erlbaum Associates, 1985.

[2] P.J. Barnard. Interactive cognitive subsystems: Modelling working memory phenomena with a multi-processor architecture. In *Models of Working Memory.* Cambridge University Press, 1998. NEED TO CHECK THIS REFERENCE.

[3] P.J. Barnard and J. May. Interactions with advanced graphical interfaces and the deployment of latent human knowledge. In *Eurogrpahics Workshop on Design, Specification and Verification of Interactive Systes*, pages 15–49. Springer, June 1995.

[4] P.J Barnard and J.D. Teasdale. Interacting cognitive subsystems: A systemic approach to cognitive-affective interaction and change. *Cognition and Emotion*, 5:1–39, 1991.

[5] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, 1988.

[6] H. Bowman. An introduction to formal models of concurrency using LOTOS. Technical Report 15-96, University of Kent at Canterbury, 1996.

[7] H. Bowman, E. A. Boiten, J. Derrick, and M. W. A. Steen. Strategies for consistency checking based on unification. *Science of Computer Programming*, December 1998. To Appear.

[8] H. Bowman, H. Cameron, P. King, and S. Thompson. Mexitl: Multimedia in Executable Interval Temporal Logic. Technical Report 3-97, Computing Laboratory, University of Kent at Canterbury, May 1997.

[9] H. Bowman, H. Cameron, P. King, and S. Thompson. Specification and Prototyping of Structured Multimedia Documents using Interval Temporal Logic. In *International Conference on Temporal Logic*, Applied Logic Series. Kluwer, July 1997.

[10] H. Bowman, M.W.A. Steen, E.A. Boiten, and J. Derrick. A formal framework for viewpoint consistency. Technical report, Submitted for Publication, 1996.

[11] H. Bowman and S. J. Thompson. A tableaux method for interval temporal logic with projection. In *TABLEAUX'98, International Conference on Analytic Tableaux and Related Methods*, volume 1397 of *Lecture Notes in AI*, pages 108–123. Springer-Verlag, May 1998.

[12] D.E. Broadbent. *Perception and Communication*. Pergamon, 1958.

[13] N. Charles, H. Bowman, and S. Thompson. From ACT-ONE to Miranda, a Translation Experiment. *Computer Standards and Interfaces*, 19(1), May 1997.

[14] A.M. Collins and E.F. Loftus. A spreading activation theory of semantic processing. *Psychological Review*, 82:407–428, 1975.

[15] J. de Meer, R. Roth, and S. Vuong. Introduction to algebraic specifications based on the language ACT ONE. *Computer Networks and ISDN Systems*, 23:363–392, 1992.

[16] DIT. *LOLA: LOtos LAboratory*. Departamento de Ingenieria Telematica, Universidad Politecnica de Madrid, 1988. WWW : http://selva.dit.upm.es/ lotos/tools/lola.html.

[17] D.J. Duke. Reasoning about gestural interaction. *Eurographics'95*, 14(3), 1995.

[18] D.J. Duke, P.J. Barnard, J. May, and D.A. Duce. Systematic development of the human interface. In *Proceedings of APSEC'95, Second Asia Pacific Software Engineering Conference, Brisbane*. IEEE Computer Society Press, December 1995.

[19] D.J. Duke and D.A. Duce. The formalisation of a cognitive architecture and its application to reasoning about human computer interaction. *Formal Aspects of Computing*, 3, 1996.

[20] H. Eertink. Executing lotos specifications: the smile tool. In *LOTOSphere: Software development with LOTOS*. Kluwer Academic Publishers, 1994.

[21] G.P. Faconti and M. Massink. A formal account of deixis in multimodal interaction. In *Submitted for Publication*, 1998.

[22] A. Giacolone, C. Jou, and S.A. Smolka. Probabilities in processes: an algebraic/operational framework. Technical Report 88/20, Department of Computer Science, SUNY at Stony Brook, 1988.

[23] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[24] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[25] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996. Distinguished Dissertations in Computer Science.

[26] M.G. Hinchey and J.P. Bowen, editors. *Applications of Formal Methods*. Prentice-Hall, 1995.

[27] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[28] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[29] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[30] ISO. *Working Draft on Enhancements to LOTOS ISO/IEC JTC1/SC21/WG7/E-LOTOS*, January 1997.

[31] T.S. Kuhn. *The Structure of Scientific Evolutions*. Chicago University Press, 1970.

[32] LOTOSPHERE. *LOTOS Integrated Tool Environment*. LOTOSPHERE Project, 1988. WWW : http://wwwtios.cs.utwente.nl/lotos/lite/.

[33] J.A. Manas and T. de Miguel. From LOTOS to C. In *Formal Description Techniques*. Elsevier Science Publishers (North-Holland), 1989.

[34] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[35] The Medical Research Councils, Applied Psychology Unit, http://www.mrc-apu.cam.ac.uk/personal/phil.barnard/ics/index.html. *ICS Home Page*, 1998.

[36] R. Milner. Calculi for synchrony and asynchrony. *Journal of Theoretical Computer Science*, 25:267–310, 1985.

[37] R. Milner. Process constructors and interpretations. In *Information Processing 86*. Elsevier Publishers, 1986.

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

[39] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.

[40] B. Moskowski. *Executing Temporal Logic*. Cambridge University Press, 1986.

[41] A. Newell. *Unified Theories of Cognition*. Harvard University Press, 1990.

[42] A. Newell and H.A. Simon. *Human Problem Solving*. Prentice-Hall, 1972.

[43] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebra. In *Real-time Theory in Practice, LNCS 600*, pages 549–572. Springer-Verlag, June 1991.

[44] L. Nigay and J. Coutaz. A generic platform for addressing the multimodal challenge. In *Proceedings of ACM CHI'95*, pages 98–105. ACM Press, 1995.

[45] W. Reisig. *Petri Nets, An Introduction*. Springer-Verlag, 1982.

[46] W. Reitman. *Cognition and Thought*. Wiley, 1965.

[47] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall International Series in Computer Science, 1997.

[48] D.E. Rumelhart, J.L. McClelland, and The PDP Research Group, editors. *Parallel Distributed Processing, Volume 1. Foundations*. MIT Press, 1986.

[49] D.E. Rumelhart, J.L. McClelland, and The PDP Research Group, editors. *Parallel Distributed Processing, Volume 2. Psychological and Biological Models*. MIT Press, 1986.

[50] J.D. Teasdale and P.J. Barnard. *Affect, Cognition and Change: Re-modelling Depressive Thought.* Lawrence Erlbaum Associates, 1993.

[51] Chris Tofts. Describing social insect behaviour using process algebra. Technical report, University College Swansea, 1996. WWW - http://www.scs.leeds.ac.uk/chris/papers.html.

[52] R.J. van Glabbeek. The linear time - branching time spectrum (I and II). In *Concur'90 and Concur'93, LNCS 458 and LNCS 715.* Springer-Verlag, 1990 and 1993.

[53] G. Winskel. *The Formal Semantics of Programming Languages.* MIT, 1993.

9 Appendix

9.1 LOTOS

We give a more formal treatment of the LOTOS sublanguage that we use and we discuss semantic models for the sublanguage.

The Sublanguage. The set of all possible LOTOS behaviour expressions is denoted `Beh`; the variables $\mathtt{B}$, $\mathtt{B}_1$, $\mathtt{B}_2$, ... range over the set `Beh` and we assume a set `GATE` of gate names. Behaviours can take the following forms:-

$$\begin{array}{lll} \mathtt{B} & \equiv & \mathtt{stop} \mid \mathtt{exit(e^*)} \mid \mathtt{g\ d^*;\ B} \mid \mathtt{i;\ B} \mid \mathtt{[bc]->B} \mid \mathtt{B_1\ []\ B_2} \mid \\ & & \mathtt{choice\ x : T\ []\ B(x)} \mid \mathtt{B_1\ |[G]|\ B_2} \mid \mathtt{B_1 >>\ accept\ b^*\ in\ B_2} \mid \\ & & \mathtt{hide\ G\ in\ B} \mid \mathtt{B\backslash[g_1/h_1, ..., g_n/h_n]} \mid \mathtt{let\ c^*\ in\ B} \end{array}$$

where $\mathtt{f}^*$ denotes zero or more occurrences of `f`; `e` is an expression; $\mathtt{g}$, $\mathtt{g}_1$, $\mathtt{g}_2$, .., $\mathtt{h}_1$, $\mathtt{h}_2$, ... are in the set `GATE`; `d` is a data attribute of the form `!e` or `?x:T`; `bc` is a boolean condition; `x` is a data variable; `T` is an ACT-ONE type; `G` is a subset of `GATE`; `b` is a declaration of the form `x:T` and `c` is a definition of the form `x:T=e`.

Process definitions have the form `P := B`.

Labelled Transition Systems. The standard semantics for LOTOS are expressed in a structured operational semantic style and they map LOTOS behaviour expressions to labelled transition systems. Such a semantic definition can be found in [6]. Here we assume that such a mapping can be defined and we work with labelled transition systems.

Labelled transition systems, are labelled directed graphs with the following form:

$$(S, A, T, s_0)$$

where S is a set of states (the nodes of the graph), A is a set of actions (the labels of the graph); T is a transition relation (the edges of the graph) and s_0 is a start state. Elements of T are triples, e.g. (s, a, t), where $s, t \in S$ and $a \in A$, which states that there is a transition (edge) from state s to state t and it is labelled with the action a. In standard fashion we write $(s, a, t) \in T$ as $s - a \rightarrow t$.

Intervals. Mexitl is defined over finite state sequences. Each sequence is called an interval and $\mathcal{I}$ denotes the set of all possible intervals; $\sigma \in \mathcal{I}$ has the form:

$$\sigma_0, \sigma_1, ..., \sigma_{|\sigma|}$$

where $|\sigma|$ denotes the length of an interval and σ_i denotes the ith state in an interval. By convention the length of an interval is the number of states minus one and all intervals must have at least one item. We use $[\sigma]^i$ to denote the ith prefix of an interval and $(\sigma)^i$ to denote the ith suffix of an interval. Formally,

$$[\sigma]^i = \sigma_0, ..., \sigma_i$$

$$(\sigma)^i = \sigma_i, ..., \sigma_{|\sigma|}$$

We define intervals from behaviour expressions (via labelled transition systems) in two steps. First we define action intervals, then we build intervals. Action intervals are traces of actions and the set of action intervals of an arbitrary $\mathtt{B}_0 \in$`Beh` is denoted $\Pi(\mathtt{B}_0)$, which is defined:

$$\Pi(\mathtt{B}_0) = \{\mathtt{a}_0, ..., \mathtt{a}_n \mid \exists \mathtt{B}_1, .., \mathtt{B}_{n+1} \ . \ \mathtt{B}_0 - \mathtt{a}_0 \rightarrow \mathtt{B}_1 \ \wedge \ ... \ \wedge \ \mathtt{B}_n - \mathtt{a}_n \rightarrow \mathtt{B}_{n+1}\}$$

Now all states in an interval contain a distinguished entry, which indicates the action performed at that state. For a state σ_j the entry is indicated by the syntax, σ_j^{ac}.

The set of all intervals of $\mathtt{B}_0$ is denoted $\Omega(\mathtt{B}_0)$, which is defined as:

$$\Omega(\mathtt{B}_0) = \{\sigma_0, ..., \sigma_n \mid \exists \mathtt{a}_0, ..., \mathtt{a}_n \in \Pi(\mathtt{B}_0) \ . \ \forall i (0 \leq i \leq n) \ . \ \sigma_i^{ac} = a_i\}$$

9.2 Mexitl

As stated earlier Mexitl is interpreted over intervals and satisfaction is defined over an arbitrary interval as follows (the notation $\sigma \models P$ states that the trace σ satisfies the proposition P):-

$$\sigma \models a \ \text{ iff } \ a = \sigma_0^{ac}$$

$$\sigma \models p(E_1, ..., E_n) \ \text{ iff } \ [\![p]\!]([\![E_1]\!]_{\sigma_0}, ..., [\![E_n]\!]_{\sigma_0})$$

$$\sigma \models E_1 = E_2 \ \text{ iff } \ [\![E_1]\!]_{\sigma_0} = [\![E_2]\!]_{\sigma_0}$$

where $[\![\ ]\!]$ maps predicates to their semantic interpretation and $[\![\ ]\!]_{\sigma_i}$ evaluates expressions (in the obvious way) according to the bindings at state σ_i.

$$\sigma \not\models \textbf{False}$$

$$\sigma \models P_1 \implies P_2 \;\text{ iff }\; \sigma \models P_1 \text{ implies } \sigma \models P_2$$

$$\sigma \models \mathbf{len}(n) \;\text{ iff }\; |\sigma| = n$$

$$\sigma \models P_1 ; P_2 \;\text{ iff }\; \exists k \in \mathcal{N} \; (k \leq |\sigma| \text{ and } [\sigma]^k \models P_1 \text{ and } (\sigma)^k \models P_2)$$

$$\begin{array}{ll} \sigma \models P_1 \;\; \mathbf{proj} \;\; P_2 \;\text{ iff} & \exists m \in \mathcal{N} \text{ and } \exists \tau_0, \tau_1, ..., \tau_m \in \mathcal{N} \\ & (0 = \tau_0 < \tau_1 < ... < \tau_m = |\sigma| \text{ and} \\ & \forall j < m \; ([\sigma]^{\tau_{j+1}})^{\tau_j} \models P_1 \text{ and} \\ & \sigma_{\tau_0} \sigma_{\tau_1} ... \sigma_{\tau_m} \models P_2) \end{array}$$

$$\sigma \models (\exists x : T) P \;\text{ iff }\; \sigma' \models x \wedge P \;\text{ for some } \sigma' \simeq_x \sigma$$

We say that $\sigma' \simeq_x \sigma$ if σ and σ' are the same length and they have the same value on all actions and variables except (perhaps) x.

9.3 Linking LOTOS and Mexitl

Now we can link specifications in LOTOS and formulae in Mexitl in the obvious way. We define that a process satisfies, denoted $\rhd$, a formula as follows:

$$\forall B \in \texttt{Beh},\; Q \in \texttt{Mexitl} \;.\; B \rhd Q \textit{ iff } \forall \sigma \in \Omega(B) \;.\; \sigma \models Q$$

Thus, a LOTOS process B satisfies a Mexitl formula Q if and only if every trace of B satisfies/is a model of Q.

9.4 Derived Temporal Operators

The following derived operators have all been considered previously in the interval temporal logic literature. See for example, [40], [8].

An interval is called empty if it contains one item:

$$\mathbf{empty} \;\equiv\; \mathbf{len}(0)$$

more holds over non-empty intervals:

$$\mathbf{more} \;\equiv\; \neg\mathbf{empty}$$

$\bigcirc P$ is the next operator, which is related to $\bigcirc$ in linear time temporal logic [34]:

$$\bigcirc P \;\equiv\; \mathbf{len}(1) \;\mathbf{;}\; P$$

The weak next operator also holds over **empty** intervals:

$$\textcircled{w} P \;\equiv\; \bigcirc P \;\vee\; \mathbf{empty}$$

Eventually, $\diamond P$, holds if there exists a *terminal* interval on which P holds:

$$\diamond P \;\equiv\; \mathbf{True} \;\mathbf{;}\; P$$

Henceforth, $\Box P$, is the dual of eventually; it holds if P holds over all terminal intervals:

$$\Box P \ \equiv\ \neg\Diamond\neg P$$

skip holds over intervals of length `1`:

$$\textbf{skip} \ \equiv\ \bigcirc\textbf{empty}$$

fin P requires that P holds at the last point in the interval:

$$\textbf{fin}\ P \ \equiv\ \Box(\textbf{empty} \Rightarrow P)$$

halt P states that P *only* holds at the final point in an interval:

$$\textbf{halt}\ P \ \equiv\ \Box(P \Leftrightarrow \textbf{empty})$$

beg P requires that P holds at the first point in an interval:

$$\textbf{beg}\ P \ \equiv\ (\textbf{empty}\ \wedge\ P)\ \textbf{;}\ \textbf{True}$$

keep P ensures that P holds throughout an interval (apart from at the last point):

$$\textbf{keep}\ P \ \equiv\ \Box(\neg\textbf{empty} \Rightarrow P)$$

Double chop is a strong chop which requires that P does not hold over the one point interval:

$$P\ \textbf{;;}\ Q \ \equiv\ (P\ \wedge\ \neg\textbf{empty})\ \textbf{;}\ Q$$

We can also define alternative eventually and henceforth operators. Such as $\Diamond_{\mathrm{i}}\, P$ which requires that there exists an initial interval on which P holds:

$$\Diamond_{\mathrm{i}}\, P \ \equiv\ P\ \textbf{;}\ \textbf{True}$$

and its dual $\boxed{\scriptstyle \mathrm{i}}\, P$ which requires that for all initial intervals P holds:

$$\boxed{\scriptstyle \mathrm{i}}\, P \ \equiv\ \neg\,\Diamond_{\mathrm{i}}\,\neg P$$

In addition, $\Diamond_{\mathrm{a}}\, P$ states that there exists an arbitrary interval on which P holds:

$$\Diamond_{\mathrm{a}}\, P \ \equiv\ \textbf{True}\ \textbf{;}\ P\ \textbf{;}\ \textbf{True}$$

and $\boxed{\scriptstyle \mathrm{a}}\, P$ states that for all arbitrary intervals P holds:

$$\boxed{\scriptstyle \mathrm{a}}\, P \ \equiv\ \neg\,\Diamond_{\mathrm{a}}\,\neg P$$

9.5 Axiomatization of *Mexitl*

Axiomatization of *Mexitl* has been considered elsewhere [8, 11]. Here we just pick out some rules that we will use. Justification for the rules can be found in [8].

$$[AssChop] \quad P \,;(Q \,; R) \iff (P \,; Q) \,; R$$

$$[IdepmChop] \quad \mathbf{True} \,; \mathbf{True} \iff \mathbf{True}$$

$$[MonoChop1] \quad P \Rightarrow Q \;\vdash\; P \,; R \Rightarrow Q \,; R$$

$$[MonoChop2] \quad P \Rightarrow Q \;\vdash\; R \,; P \Rightarrow R \,; Q$$

$$[DistChop] \quad \text{for } R \text{ rigid} \quad (R \,; P) \wedge (R \,; Q) \implies (R \,; (P \wedge Q))$$

$$[AddLen] \quad \mathbf{len}(x+y) \iff \mathbf{len}(x) \,; \mathbf{len}(y)$$

$$[DistProj] \quad P \;\mathbf{proj}\; (Q \,; R) \iff P \;\mathbf{proj}\; Q \,; P \;\mathbf{proj}\; R$$

$$[MonoProj1] \quad P \Rightarrow Q \;\vdash\; P \;\mathbf{proj}\; R \Rightarrow Q \;\mathbf{proj}\; R$$

9.6 Formal Properties of Non-determinism

We justify the statement that,

> Whatever property holds over a process will also hold over its (non-deterministic) refinements.

In the context of this paper, this statement can be reformulated as,

> Whatever ICS formula holds over a LOTOS process will also hold over any reduction of the process,

where reduction, denoted *red*, is the LOTOS refinement relation. Its definition, see [6], ensures that,

> B *red* C *iff* B is more deterministic than C.

A standard property of reduction is,

$$B \; red \; C \implies (\forall \sigma \in I \,.\, \sigma \in \Omega(B) \;\Rightarrow\; \sigma \in \Omega(C))$$

i.e. if B is a reduction of C then any trace of B is also a trace of C. We use this property to derive the result that we require. Our argument is as follows:

$$
\begin{array}{ll}
 & C \ \triangleright\ Q \ \wedge\ B\ red\ C \\
\equiv & \{ \text{ Definition of } \triangleright\text{; above property of } red \ \} \\
 & \forall \sigma \in \mathcal{I} \,.\, \sigma \in \Omega(C) \ \Rightarrow\ \sigma \models Q \wedge \sigma \in \Omega(B) \ \Rightarrow\ \sigma \in \Omega(C) \\
\Rightarrow & \{ \text{ Transitivity of Implication } \} \\
 & \forall \sigma \in \mathcal{I} \,.\, \sigma \in \Omega(B) \ \Rightarrow\ \sigma \models Q \\
\equiv & \{ \text{ Definition of } \triangleright \ \} \\
 & B \ \triangleright\ Q
\end{array}
$$

9.7 ICS Specification

The following is the full LOTOS specification of ICS. The LOTOS notation used is slightly different from that used in the paper. For a complete introduction to this syntax see [5].

```
(* This is the full ICS specification. *)

specification ics [tick,vis_obj,prop_obj,obj_mpl,obj_prop,obj_lim,
obj_normal,obj_buffered,eye_vis,vis_implic,vis_normal,vis_buffered,bs_lim,
lim_leg,lim_arm,lim_hand,lim_normal,lim_buffered,prop_implic,bs_implic,
ac_implic,implic_prop,implic_visc,implic_som,implic_normal,implic_buffered,
mpl_prop,prop_mpl,prop_normal,prop_buffered,ac_mpl,mpl_art,mpl_normal,
mpl_buffered,ear_ac,ac_normal,ac_buffered,bs_art,art_sp,art_wr,art_normal,
art_buffered,vocal_bs,hand_bs,bs_normal,bs_buffered]: noexit

library Boolean, NaturalNumber endlib

type Rep is NaturalNumber renamedby
sortnames Rep for Nat
endtype
(* 0 is reserved for null representations *)

type indice is Boolean
sorts indice
opns j0, j1, j2, j3 : -> indice
     _eqq_, _neq_ : indice, indice -> Bool
eqns forall k,r: indice
ofsort Bool
     k=r =>
     k eqq r = true;

     k neq r = not(k eqq r)
```

```
endtype

type tuple is indice
formalsorts item
sorts tuple
opns # : item, item, item, item -> tuple
     get : indice, tuple -> item
eqns forall v,x,y,z: item
ofsort item
     get(j0,#(v,x,y,z)) = v;
     get(j1,#(v,x,y,z)) = x;
     get(j2,#(v,x,y,z)) = y;
     get(j3,#(v,x,y,z)) = z;
endtype

type inArr is tuple actualizedby Rep using
sortnames inArr for tuple
          Rep for item
opnnames iaget for get
endtype
(* We assume a maximum size for input Array's of 4. Consequently,
smaller arrays will have empty slots at the end. These will be set
to the null representation 0. *)

type imRc is inArr, NaturalNumber
sorts imRc
opns nil : -> imRc
     add : inArr, imRc -> imRc
     first : imRc -> inArr
     select : Nat, inArr, imRc -> inArr  (* Selects from the image array
                                         during buffered mode according
                                         to the first parameter. *)
     remove : imRc -> imRc
eqns forall x,y : inArr, z : imRc, n : Nat
ofsort inArr
  first(nil) = #(0,0,0,0);
  first(add(x,nil)) = x;
  first(add(x,add(y,z))) = first(add(y,z));

  select(0,x,z) = first(z);
  select(succ(n),x,z) = select(n,x,remove(z))
  (* This definition is a placeholder for something more
     sophisticated. It allows selection from the image record
     to be made by referencing a location in the image record. *)

ofsort imRc
```

```
  remove(nil) = nil;
  remove(add(x,nil)) = nil;
  remove(add(x,add(y,z))) = add(x,remove(add(y,z)));
endtype

type slotmap is tuple actualizedby Boolean using
sortnames slotmap for tuple
          Bool for item
opnnames smget for get
endtype

type Map is tuple actualizedby slotmap using
sortnames Map for tuple
          slotmap for item
opnnames mpget for get
endtype
(* The ith entry in the Map tuple indicates the set of slots in the
input array blendable at the ith output transformation. This set of
slots is indicated by a slotmap - true in the jth
location indicates that the jth slot is required. Thus,
in a similar way to with the input array, we assume no more
than 4 output transformations. *)

type subsyst is
sorts subsyst
opns VISS, OBJJ, LIMM, LEGG, ARMM, HANDD, VISCC, SOMM, SPP, WRR,
IMPLICC, BSS, PROPP, MPLL, ACC, ARTT : -> subsyst
endtype

type trans is subsyst, Rep
sorts trans
opns tran : subsyst, subsyst, Rep -> Rep
eqns forall s,t: subsyst, r : Rep
ofsort Rep
   tran(s,t,r)=r
endtype
(* For the moment all transformations between all subsystems are defined
as the identity operation *)

type blending is slotmap, inArr, Rep, Boolean
sorts blending
opns compare : slotmap, inArr -> Rep
     eval : Bool, Rep -> Rep
     _@_ : Rep, Rep -> Rep
     mult : slotmap, inArr -> Rep
eqns forall x0,x1,x2,x3:Bool, r,r0,r1,r2,r3:Rep
```

```
ofsort Rep
compare(#(x0,x1,x2,x3),#(r0,r1,r2,r3)) =
 (((eval(x0,r0) @ eval(x1,r1)) @ eval(x2,r2)) @ eval(x3,r3));

eval(true,r) = r;
eval(false,r) = succ(0);

r1 eq r2 => r1 @ r2 = r1;
r1 ne r2 and ((r1 eq succ(0)) or (r2 eq succ(0))) => r1 @ r2 = r1 * r2;
r1 ne r2 and ((r1 ne succ(0)) and (r2 ne succ(0))) => r1 @ r2 = 0;

mult(#(x0,x1,x2,x3),#(r0,r1,r2,r3)) =
   (((eval(x0,r0) * eval(x1,r1)) * eval(x2,r2)) * eval(x3,r3));
endtype

behaviour

(* Initalize the transformation maps for each of the subsystems. *)

let
VISmap:Map = #( #(true,false,false,false) ,
                #(true,false,false,false) ,
                #(false,false,false,false) , (* Redundant entry *)
                #(false,false,false,false) ), (* Redundant entry *)

OBJmap:Map = #( #(true,true,false,false) ,
                #(true,true,false,false) ,
                #(true,true,false,false) ,
                #(false,false,false,false) ), (* Redundant entry *)

LIMmap:Map = #( #(true,true,false,false) ,
                #(true,true,false,false) ,
                #(true,true,false,false) ,
                #(false,false,false,false) ), (* Redundant entry *)

IMPLICmap:Map = #( #(true,true,true,true) ,
                   #(true,true,true,true) ,
                   #(true,true,true,true) ,
                   #(false,false,false,false) ), (* Redundant entry *)

PROPmap:Map = #( #(true,true,true,false) ,
                 #(true,true,true,false) ,
                 #(true,true,true,false) ,
                 #(false,false,false,false) ), (* Redundant entry *)

BSmap:Map = #( #(true,false,false,false) ,
```

```
               #(true,true,false,false) ,
               #(false,true,false,false) ,
               #(false,false,false,false) ), (* Redundant entry *)

MPLmap:Map = #( #(true,true,true,false) ,
                #(true,true,true,false) ,
                #(false,false,false,false) , (* Redundant entry *)
                #(false,false,false,false) ), (* Redundant entry *)

ACmap:Map = #( #(true,false,false,false) ,
               #(true,false,false,false) ,
               #(false,false,false,false) , (* Redundant entry *)
               #(false,false,false,false) ), (* Redundant entry *)

ARTmap:Map = #( #(true,true,false,false) ,
                #(true,true,false,false) ,
                #(false,false,false,false) , (* Redundant entry *)
                #(false,false,false,false) ) (* Redundant entry *)
in
(
(* Top Level Behaviour of ICS. *)
clock[tick]
|[tick]|
(
tick;
(((((((
VIS[tick,eye_vis,vis_obj,vis_implic,vis_normal,vis_buffered]
(nil,#(0,0,0,0),VISmap)
|[vis_obj,tick]|
OBJ[tick,vis_obj,prop_obj,obj_mpl,obj_prop,obj_lim,obj_normal,obj_buffered]
(nil,#(0,0,0,0),OBJmap) )
|[obj_lim,tick]|
LIM[tick,obj_lim,bs_lim,lim_leg,lim_arm,lim_hand,lim_normal,lim_buffered]
(nil,#(0,0,0,0),LIMmap) )
|[vis_implic,tick]|
IMPLIC[tick,vis_implic,prop_implic,bs_implic,ac_implic,implic_prop,
implic_visc,implic_som,implic_normal,implic_buffered]
(nil,#(0,0,0,0),IMPLICmap) )
|[obj_prop,implic_prop,prop_obj,prop_implic,tick]|
PROP[tick,obj_prop,implic_prop,mpl_prop,prop_mpl,prop_obj,prop_implic,
prop_normal,prop_buffered]
(nil,#(0,0,0,0),PROPmap) )
|[bs_implic,bs_lim,tick]|
BS[tick,vocal_bs,hand_bs,bs_art,bs_implic,bs_lim,bs_normal,bs_buffered]
(nil,#(0,0,0,0),BSmap) )
|[obj_mpl,prop_mpl,mpl_prop,tick]|
```

```
MPL[tick,obj_mpl,prop_mpl,ac_mpl,mpl_art,mpl_prop,mpl_normal,
mpl_buffered]
(nil,#(0,0,0,0),MPLmap) )
|[ac_mpl,ac_implic,tick]|
AC[tick,ear_ac,ac_mpl,ac_implic,ac_normal,ac_buffered]
(nil,#(0,0,0,0),ACmap) )
|[mpl_art,bs_art,tick]|
ART[tick,bs_art,mpl_art,art_sp,art_wr,art_normal,art_buffered]
(nil,#(0,0,0,0),ARTmap)
)
|[obj_buffered,vis_buffered,lim_buffered,implic_buffered,prop_buffered,
mpl_buffered,ac_buffered,art_buffered,bs_buffered,tick]|
tick;
buffConstraint[obj_buffered,vis_buffered,lim_buffered,implic_buffered,
prop_buffered,mpl_buffered,ac_buffered,art_buffered,bs_buffered,tick]
) )

where

(************************ CLOCK process ***************************)

process clock[tick]:noexit:=
tick; clock[tick]
endproc (* clock *)

(************************ Buffer Constraint ****************************)

(* This process defines a constraint on the number of subsystems
that can be in buffered mode. The constraint is that at most one
subsystem may be in buffered mode (and the subsystem definitions
ensure that only one transformation within a subsystem can be
buffered). This is a basic constraint in the ICS model. *)

process buffConstraint[obj_buffered,vis_buffered,lim_buffered,implic_buffered,
prop_buffered,mpl_buffered,ac_buffered,art_buffered,bs_buffered,tick]:noexit:=

obj_buffered?b:indice; tick; buffConstraint[obj_buffered,vis_buffered,
                                 lim_buffered,implic_buffered,prop_buffered,
                                 mpl_buffered,ac_buffered,art_buffered,
                                 bs_buffered,tick]
[]
vis_buffered?b:indice; tick; buffConstraint[obj_buffered,vis_buffered,
                                 lim_buffered,implic_buffered,prop_buffered,
                                 mpl_buffered,ac_buffered,art_buffered,
                                 bs_buffered,tick]
[]
```

```
lim_buffered?b:indice; tick; buffConstraint[obj_buffered,vis_buffered,
                                 lim_buffered,implic_buffered,prop_buffered,
                                 mpl_buffered,ac_buffered,art_buffered,
                                 bs_buffered,tick]
[]
implic_buffered?b:indice; tick; buffConstraint[obj_buffered,vis_buffered,
                                 lim_buffered,implic_buffered,prop_buffered,
                                 mpl_buffered,ac_buffered,art_buffered,
                                 bs_buffered,tick]
[]
prop_buffered?b:indice; tick; buffConstraint[obj_buffered,vis_buffered,
                                 lim_buffered,implic_buffered,prop_buffered,
                                 mpl_buffered,ac_buffered,art_buffered,
                                 bs_buffered,tick]
[]
mpl_buffered?b:indice; tick; buffConstraint[obj_buffered,vis_buffered,
                                 lim_buffered,implic_buffered,prop_buffered,
                                 mpl_buffered,ac_buffered,art_buffered,
                                 bs_buffered,tick]
[]
ac_buffered?b:indice; tick; buffConstraint[obj_buffered,vis_buffered,
                                 lim_buffered,implic_buffered,prop_buffered,
                                 mpl_buffered,ac_buffered,art_buffered,
                                 bs_buffered,tick]
[]
art_buffered?b:indice; tick; buffConstraint[obj_buffered,vis_buffered,
                                 lim_buffered,implic_buffered,prop_buffered,
                                 mpl_buffered,ac_buffered,art_buffered,
                                 bs_buffered,tick]
[]
bs_buffered?b:indice; tick; buffConstraint[obj_buffered,vis_buffered,
                                 lim_buffered,implic_buffered,prop_buffered,
                                 mpl_buffered,ac_buffered,art_buffered,
                                 bs_buffered,tick]
[]
tick; buffConstraint[obj_buffered,vis_buffered,
                                 lim_buffered,implic_buffered,prop_buffered,
                                 mpl_buffered,ac_buffered,art_buffered,
                                 bs_buffered,tick]

endproc (* buffConstraint *)

(************************* The VIS process ******************************)

process VIS[tick,eye_vis,vis_obj,vis_implic,vis_normal,vis_buffered]
(iR:imRc,iA:inArr,m:Map): noexit :=
```

```
( (vis_normal;
VIS_NORMAL[tick,eye_vis,vis_obj,vis_implic,vis_normal,vis_buffered](iR,iA,m))
[]
(vis_buffered?b:indice;  (* b indicates which transformation is buffered *)
VIS_BUFFMD[tick,eye_vis,vis_obj,vis_implic,vis_normal,vis_buffered](iR,iA,m,b)) )

(* The choice here is between normal mode and buffered mode. The actual
mechanism used is not currently clear. Thus, we allow the environment to
choose through actions vis_normal and vis_buffered. *)

where
process VIS_NORMAL[tick,eye_vis,vis_obj,vis_implic,vis_normal,vis_buffered]
(iR:imRc,iA:inArr,m:Map) : noexit :=

  (( eye_vis?r1:Rep ; exit(r1) )
  (* Input Ports *)
  |||
  ( ( BLEND2[vis_obj](VISS,OBJJ,j0,iA,m) >> exit(any Rep) )
  ||| ( BLEND2[vis_implic](VISS,IMPLICC,j1,iA,m) >> exit(any Rep) ))
  (* Output Ports *) )
  >> accept r1:rep in
     tick; VIS[tick,eye_vis,vis_obj,vis_implic,vis_normal,vis_buffered]
           (add(#(r1,0,0,0),iR),#(r1,0,0,0),m)

endproc (* VIS_NORMAL *)

process VIS_BUFFMD[tick,eye_vis,vis_obj,vis_implic,vis_normal,vis_buffered]
(iR:imRc,iA:inArr,m:Map,b:indice): noexit :=

( ( eye_vis?r1:Rep ; exit(r1) )
(* Input Ports *)
|||
( ( OUTPUT2[vis_obj](VISS,OBJJ,b,j0,iR,iA,m) >> exit(any Rep) )
|||
( OUTPUT2[vis_implic](VISS,IMPLICC,b,j1,iR,iA,m) >> exit(any Rep) ) )
(* Output Ports *) )
>> accept r1:rep in
   tick; VIS[tick,eye_vis,vis_obj,vis_implic,vis_normal,vis_buffered]
         (add(#(r1,0,0,0),iR),#(r1,0,0,0),m)

endproc (* VIS_BUFFMD *)

endproc (* VIS *)
```

```
(************************ The OBJ process ******************************)

process OBJ[tick,vis_obj,prop_obj,obj_mpl,obj_prop,obj_lim,obj_normal,
obj_buffered](iR:imRc,iA:inArr,m:Map): noexit :=

( (obj_normal;
OBJ_NORMAL[tick,vis_obj,prop_obj,obj_mpl,obj_prop,obj_lim,obj_normal,
obj_buffered](iR,iA,m))
[]
(obj_buffered?b:indice;  (* b indicates which transformation is buffered *)
OBJ_BUFFMD[tick,vis_obj,prop_obj,obj_mpl,obj_prop,obj_lim,obj_normal,
obj_buffered](iR,iA,m,b)) )

(* The choice here is between normal mode and buffered mode. The actual
mechanism used is not currently clear. Thus, we allow the environment to
choose through actions obj_normal and obj_buffered. *)

where
process OBJ_NORMAL[tick,vis_obj,prop_obj,obj_mpl,obj_prop,obj_lim,obj_normal,
obj_buffered]
(iR:imRc,iA:inArr,m:Map) : noexit :=

  (( vis_obj?r1:Rep ; exit(r1,any Rep) ||| prop_obj?r2:Rep ; exit(any Rep,r2) )
  (* Input Ports *)
  |||
  (   ( BLEND2[obj_mpl](OBJJ,MPLL,j0,iA,m) >> exit(any Rep,any Rep) )
  ||| ( BLEND2[obj_prop](OBJJ,PROPP,j1,iA,m) >> exit(any Rep,any Rep) )
  ||| ( BLEND2[obj_lim](OBJJ,LIMM,j2,iA,m) >> exit(any Rep,any Rep) ) )
  (* Output Ports *) )
  >> accept r1,r2:rep in
     tick; OBJ[tick,vis_obj,prop_obj,obj_mpl,obj_prop,obj_lim,obj_normal,
               obj_buffered](add(#(r1,r2,0,0),iR),#(r1,r2,0,0),m)

endproc (* OBJ_NORMAL *)

process OBJ_BUFFMD[tick,vis_obj,prop_obj,obj_mpl,obj_prop,obj_lim,obj_normal,
obj_buffered]
(iR:imRc,iA:inArr,m:Map,b:indice): noexit :=

( ( vis_obj?r1:Rep ; exit(r1,any Rep) ||| prop_obj?r2:Rep ; exit(any Rep,r2) )
(* Input Ports *)
 |||
( ( OUTPUT2[obj_mpl](OBJJ,MPLL,b,j0,iR,iA,m) >> exit(any Rep,any Rep) )
|||
( OUTPUT2[obj_prop](OBJJ,PROPP,b,j1,iR,iA,m) >> exit(any Rep,any Rep) )
|||
```

```
( OUTPUT2[obj_lim](OBJJ,LIMM,b,j2,iR,iA,m) >> exit(any Rep,any Rep) ) )
(* Output Ports *) )
>> accept r1,r2:rep in
   tick; OBJ[tick,vis_obj,prop_obj,obj_mpl,obj_prop,obj_lim,obj_normal,
             obj_buffered](add(#(r1,r2,0,0),iR),#(r1,r2,0,0),m)

endproc (* OBJ_BUFFMD *)

endproc (* OBJ *)


(************************ The LIM process ******************************)

process LIM[tick,obj_lim,bs_lim,lim_leg,lim_arm,lim_hand,lim_normal,
lim_buffered](iR:imRc,iA:inArr,m:Map): noexit :=

( (lim_normal;
LIM_NORMAL[tick,obj_lim,bs_lim,lim_leg,lim_arm,lim_hand,lim_normal,
lim_buffered](iR,iA,m))
[]
(lim_buffered?b:indice;  (* b indicates which transformation is buffered *)
LIM_BUFFMD[tick,obj_lim,bs_lim,lim_leg,lim_arm,lim_hand,lim_normal,
lim_buffered](iR,iA,m,b)) )

(* The choice here is between normal mode and buffered mode. The actual
mechanism used is not currently clear. Thus, we allow the environment to
choose through actions lim_normal and lim_buffered. *)

where
process LIM_NORMAL[tick,obj_lim,bs_lim,lim_leg,lim_arm,lim_hand,lim_normal,
lim_buffered]
(iR:imRc,iA:inArr,m:Map) : noexit :=

  (( obj_lim?r1:Rep ; exit(r1,any Rep) ||| bs_lim?r2:Rep ; exit(any Rep,r2) )
  (* Input Ports *)
  |||
  ( ( BLEND2[lim_leg](LIMM,LEGG,j0,iA,m) >> exit(any Rep,any Rep) )
  ||| ( BLEND2[lim_arm](LIMM,ARMM,j1,iA,m) >> exit(any Rep,any Rep) )
  ||| ( BLEND2[lim_hand](LIMM,HANDD,j2,iA,m) >> exit(any Rep,any Rep) ) )
  (* Output Ports *) )
  >> accept r1,r2:rep in
     tick; LIM[tick,obj_lim,bs_lim,lim_leg,lim_arm,lim_hand,lim_normal,
               lim_buffered](add(#(r1,r2,0,0),iR),#(r1,r2,0,0),m)

endproc (* LIM_NORMAL *)
```

```
process LIM_BUFFMD[tick,obj_lim,bs_lim,lim_leg,lim_arm,lim_hand,lim_normal,
lim_buffered]
(iR:imRc,iA:inArr,m:Map,b:indice): noexit :=

( ( obj_lim?r1:Rep ; exit(r1,any Rep) ||| bs_lim?r2:Rep ; exit(any Rep,r2) )
(* Input Ports *)
|||
( ( OUTPUT2[lim_leg](LIMM,LEGG,b,j0,iR,iA,m) >> exit(any Rep,any Rep) )
|||
( OUTPUT2[lim_arm](LIMM,ARMM,b,j1,iR,iA,m) >> exit(any Rep,any Rep) )
|||
( OUTPUT2[lim_hand](LIMM,HANDD,b,j2,iR,iA,m) >> exit(any Rep,any Rep) ) )
(* Output Ports *) )
>> accept r1,r2:rep in
   tick; LIM[tick,obj_lim,bs_lim,lim_leg,lim_arm,lim_hand,lim_normal,
             lim_buffered](add(#(r1,r2,0,0),iR),#(r1,r2,0,0),m)

endproc (* LIM_BUFFMD *)

endproc (* LIM *)


(************************* SOM_VISC ******************************)

(* These subsystems are not currently implemented. *)


(************************* The IMPLIC process ******************************)

process IMPLIC[tick,vis_implic,prop_implic,bs_implic,ac_implic,implic_prop,
implic_visc,implic_som,implic_normal,implic_buffered]
(iR:imRc,iA:inArr,m:Map): noexit :=

(
(implic_normal;
IMPLIC_NORMAL[tick,vis_implic,prop_implic,bs_implic,ac_implic,implic_prop,
implic_visc,implic_som,implic_normal,implic_buffered](iR,iA,m))
[]
(implic_buffered?b:indice;  (* b indicates which transformation is buffered *)
IMPLIC_BUFFMD[tick,vis_implic,prop_implic,bs_implic,ac_implic,implic_prop,
implic_visc,implic_som,implic_normal,implic_buffered](iR,iA,m,b)) )

(* The choice here is between normal mode and buffered mode. The actual
mechanism used is not currently clear. Thus, we allow the environment to
choose through actions implic_normal and implic_buffered. *)
```

```
where
process IMPLIC_NORMAL[tick,vis_implic,prop_implic,bs_implic,ac_implic,
implic_prop,implic_visc,implic_som,implic_normal,implic_buffered]
(iR:imRc,iA:inArr,m:Map) : noexit :=

  (( vis_implic?r1:Rep ; exit(r1,any Rep,any Rep,any Rep)
     ||| prop_implic?r2:Rep ; exit(any Rep,r2,any Rep,any Rep)
     ||| bs_implic?r3:Rep ; exit(any Rep,any Rep,r3,any Rep)
     ||| ac_implic?r4:Rep ; exit(any Rep,any Rep,any Rep,r4) )
  (* Input Ports *)
  |||
  ( ( BLEND2[implic_prop](IMPLICC,PROPP,j0,iA,m) >>
                                  exit(any Rep,any Rep,any Rep,any Rep) )
  ||| ( BLEND2[implic_visc](IMPLICC,VISCC,j1,iA,m) >>
                                  exit(any Rep,any Rep,any Rep,any Rep) )
  ||| ( BLEND2[implic_som](IMPLICC,SOMM,j2,iA,m) >>
                                  exit(any Rep,any Rep,any Rep,any Rep) ) )
  (* Output Ports *) )
  >> accept r1,r2,r3,r4:rep in
     tick; IMPLIC[tick,vis_implic,prop_implic,bs_implic,ac_implic,implic_prop,
                  implic_visc,implic_som,implic_normal,implic_buffered]
                  (add(#(r1,r2,r3,r4),iR),#(r1,r2,r3,r4),m)

endproc (* IMPLIC_NORMAL *)

process IMPLIC_BUFFMD[tick,vis_implic,prop_implic,bs_implic,ac_implic,
implic_prop,implic_visc,implic_som,implic_normal,implic_buffered]
(iR:imRc,iA:inArr,m:Map,b:indice): noexit :=

(( vis_implic?r1:Rep ; exit(r1,any Rep,any Rep,any Rep)
   ||| prop_implic?r2:Rep ; exit(any Rep,r2,any Rep,any Rep)
   ||| bs_implic?r3:Rep ; exit(any Rep,any Rep,r3,any Rep)
   ||| ac_implic?r4:Rep ; exit(any Rep,any Rep,any Rep,r4) )
(* Input Ports *)
|||
( ( OUTPUT2[implic_prop](IMPLICC,PROPP,b,j0,iR,iA,m) >>
                                  exit(any Rep,any Rep,any Rep,any Rep) )
|||
( OUTPUT2[implic_visc](IMPLICC,VISCC,b,j1,iR,iA,m) >>
                                  exit(any Rep,any Rep,any Rep,any Rep) )
|||
( OUTPUT2[implic_som](IMPLICC,SOMM,b,j2,iR,iA,m) >>
                                  exit(any Rep,any Rep,any Rep,any Rep) ) )
(* Output Ports *) )
>> accept r1,r2,r3,r4:rep in
   tick; IMPLIC[tick,vis_implic,prop_implic,bs_implic,ac_implic,implic_prop,
```

```
                implic_visc,implic_som,implic_normal,implic_buffered]
                (add(#(r1,r2,r3,r4),iR),#(r1,r2,r3,r4),m)

endproc (* IMPLIC_BUFFMD *)

endproc (* IMPLIC *)


(************************ The PROP process ******************************)

process PROP[tick,obj_prop,implic_prop,mpl_prop,prop_mpl,prop_obj,prop_implic,
prop_normal,prop_buffered](iR:imRc,iA:inArr,m:Map): noexit :=

(
(prop_normal;
PROP_NORMAL[tick,obj_prop,implic_prop,mpl_prop,prop_mpl,prop_obj,prop_implic,
prop_normal,prop_buffered]
(iR,iA,m))
[]
(prop_buffered?b:indice;  (* b indicates which transformation is buffered *)
PROP_BUFFMD[tick,obj_prop,implic_prop,mpl_prop,prop_mpl,prop_obj,prop_implic,
prop_normal,prop_buffered]
(iR,iA,m,b)) )

(* The choice here is between normal mode and buffered mode. The actual
mechanism used is not currently clear. Thus, we allow the environment to
choose through actions prop_normal and prop_buffered. *)

where
process PROP_NORMAL[tick,obj_prop,implic_prop,mpl_prop,prop_mpl,prop_obj,
prop_implic,prop_normal,prop_buffered](iR:imRc,iA:inArr,m:Map) : noexit :=

((  obj_prop?r1:Rep ; exit(r1,any Rep,any Rep)
||| implic_prop?r2:Rep ; exit(any Rep,r2,any Rep)
||| mpl_prop?r3:Rep ; exit(any Rep,any Rep,r3) )
(* Input Ports *)
|||
( ( BLEND2[prop_mpl](PROPP,MPLL,j0,iA,m) >> exit(any Rep,any Rep,any Rep) )
||| ( BLEND2[prop_obj](PROPP,OBJJ,j1,iA,m) >> exit(any Rep,any Rep,any Rep) )
||| ( BLEND2[prop_implic](PROPP,IMPLICC,j2,iA,m) >>
                                         exit(any Rep,any Rep,any Rep) ) )
(* Output Ports *) )
>> accept r1,r2,r3:rep in
   tick; PROP[tick,obj_prop,implic_prop,mpl_prop,prop_mpl,prop_obj,prop_implic,
              prop_normal,prop_buffered](add(#(r1,r2,r3,0),iR),#(r1,r2,r3,0),m)
```

```
endproc (* PROP_NORMAL *)

process PROP_BUFFMD[tick,obj_prop,implic_prop,mpl_prop,prop_mpl,prop_obj,
prop_implic,prop_normal,prop_buffered](iR:imRc,iA:inArr,m:Map,b:indice)
:noexit :=

((  obj_prop?r1:Rep ; exit(r1,any Rep,any Rep)
||| implic_prop?r2:Rep ; exit(any Rep,r2,any Rep)
||| mpl_prop?r3:Rep ; exit(any Rep,any Rep,r3) )
(* Input Ports *)
|||
(
( OUTPUT2[prop_mpl](PROPP,MPLL,b,j0,iR,iA,m) >> exit(any Rep,any Rep,any Rep) )
|||
( OUTPUT2[prop_obj](PROPP,OBJJ,b,j1,iR,iA,m) >> exit(any Rep,any Rep,any Rep) )
|||
( OUTPUT2[prop_implic](PROPP,IMPLICC,b,j2,iR,iA,m) >>
                                              exit(any Rep,any Rep,any Rep) ) )
(* Output Ports *) )
>> accept r1,r2,r3:rep in
   tick; PROP[tick,obj_prop,implic_prop,mpl_prop,prop_mpl,prop_obj,prop_implic,
              prop_normal,prop_buffered](add(#(r1,r2,r3,0),iR),#(r1,r2,r3,0),m)

endproc (* PROP_BUFFMD *)

endproc (* PROP *)


(************************* The BS process *******************************)

process BS[tick,vocal_bs,hand_bs,bs_art,bs_implic,bs_lim,bs_normal,bs_buffered]
(iR:imRc,iA:inArr,m:Map): noexit :=

(
(bs_normal;
BS_NORMAL[tick,vocal_bs,hand_bs,bs_art,bs_implic,bs_lim,bs_normal,bs_buffered]
(iR,iA,m))
[]
(bs_buffered?b:indice;  (* b indicates which transformation is buffered *)
BS_BUFFMD[tick,vocal_bs,hand_bs,bs_art,bs_implic,bs_lim,bs_normal,bs_buffered]
(iR,iA,m,b)) )

(* The choice here is between normal mode and buffered mode. The actual
mechanism used is not currently clear. Thus, we allow the environment to
choose through actions bs_normal and bs_buffered. *)
```

```
where
process BS_NORMAL[tick,vocal_bs,hand_bs,bs_art,bs_implic,bs_lim,bs_normal,
bs_buffered](iR:imRc,iA:inArr,m:Map) : noexit :=

  (( vocal_bs?r1:Rep; exit(r1,any Rep) ||| hand_bs?r2:Rep; exit(any Rep,r2))
  (* Input Ports *)
  |||
  (
  ( BLEND2[bs_art](BSS,ARTT,j0,iA,m) >> exit(any Rep,any Rep) )
  ||| ( BLEND2[bs_implic](BSS,IMPLICC,j1,iA,m) >> exit(any Rep,any Rep) )
  ||| ( BLEND2[bs_lim](BSS,LIMM,j2,iA,m) >> exit(any Rep,any Rep) ) )
  (* Output Ports *) )
  >> accept r1,r2:rep in
     tick; BS[tick,vocal_bs,hand_bs,bs_art,bs_implic,bs_lim,bs_normal,
              bs_buffered](add(#(r1,r2,0,0),iR),#(r1,r2,0,0),m)

endproc (* BS_NORMAL *)

process BS_BUFFMD[tick,vocal_bs,hand_bs,bs_art,bs_implic,bs_lim,bs_normal,
bs_buffered](iR:imRc,iA:inArr,m:Map,b:indice): noexit :=

(( vocal_bs?r1:Rep; exit(r1,any Rep) ||| hand_bs?r2:Rep; exit(any Rep,r2))
(* Input Ports *)
|||
( ( OUTPUT2[bs_art](BSS,ARTT,b,j0,iR,iA,m) >> exit(any Rep,any Rep) )
|||
( OUTPUT2[bs_implic](BSS,IMPLICC,b,j1,iR,iA,m) >> exit(any Rep,any Rep) )
|||
( OUTPUT2[bs_lim](BSS,LIMM,b,j2,iR,iA,m) >> exit(any Rep,any Rep) ) )
(* Output Ports *) )
>> accept r1,r2:rep in
   tick; BS[tick,vocal_bs,hand_bs,bs_art,bs_implic,bs_lim,bs_normal,
            bs_buffered](add(#(r1,r2,0,0),iR),#(r1,r2,0,0),m)

endproc (* BS_BUFFMD *)

endproc (* BS *)


(************************* The MPL process ******************************)

process MPL[tick,obj_mpl,prop_mpl,ac_mpl,mpl_art,mpl_prop,mpl_normal,
mpl_buffered](iR:imRc,iA:inArr,m:Map): noexit :=

(
(mpl_normal;
```

```
MPL_NORMAL[tick,obj_mpl,prop_mpl,ac_mpl,mpl_art,mpl_prop,mpl_normal,
mpl_buffered](iR,iA,m))
[]
(mpl_buffered?b:indice;  (* b indicates which transformation is buffered *)
MPL_BUFFMD[tick,obj_mpl,prop_mpl,ac_mpl,mpl_art,mpl_prop,mpl_normal,
mpl_buffered](iR,iA,m,b)) )

(* The choice here is between normal mode and buffered mode. The actual
mechanism used is not currently clear. Thus, we allow the environment to
choose through actions mpl_normal and mpl_buffered. *)

where
process MPL_NORMAL[tick,obj_mpl,prop_mpl,ac_mpl,mpl_art,mpl_prop,mpl_normal,
mpl_buffered](iR:imRc,iA:inArr,m:Map) : noexit :=

(( obj_mpl?r1:Rep ; exit(r1,any Rep,any Rep)
||| prop_mpl?r2:Rep ; exit(any Rep,r2,any Rep)
||| ac_mpl?r3:Rep ; exit(any Rep,any Rep,r3) )
(* Input Ports *)
|||
( ( BLEND2[mpl_art](MPLL,ARTT,j0,iA,m) >> exit(any Rep,any Rep,any Rep) )
||| ( BLEND2[mpl_prop](MPLL,PROPP,j1,iA,m) >> exit(any Rep,any Rep,any Rep) ) )
(* Output Ports *) )
>> accept r1,r2,r3:rep in
   tick; MPL[tick,obj_mpl,prop_mpl,ac_mpl,mpl_art,mpl_prop,mpl_normal,
             mpl_buffered](add(#(r1,r2,r3,0),iR),#(r1,r2,r3,0),m)

endproc (* MPL_NORMAL *)

process MPL_BUFFMD[tick,obj_mpl,prop_mpl,ac_mpl,mpl_art,mpl_prop,mpl_normal,
mpl_buffered]
(iR:imRc,iA:inArr,m:Map,b:indice): noexit :=

(( obj_mpl?r1:Rep ; exit(r1,any Rep,any Rep)
||| prop_mpl?r2:Rep ; exit(any Rep,r2,any Rep)
||| ac_mpl?r3:Rep ; exit(any Rep,any Rep,r3) )
(* Input Ports *)
|||
( ( OUTPUT2[mpl_art](MPLL,ARTT,b,j0,iR,iA,m) >> exit(any Rep,any Rep,any Rep) )
|||
( OUTPUT2[mpl_prop](MPLL,PROPP,b,j1,iR,iA,m) >>
                                            exit(any Rep,any Rep,any Rep) ) )
(* Output Ports *) )
>> accept r1,r2,r3:rep in
   tick; MPL[tick,obj_mpl,prop_mpl,ac_mpl,mpl_art,mpl_prop,mpl_normal,
             mpl_buffered](add(#(r1,r2,r3,0),iR),#(r1,r2,r3,0),m)
```

```
endproc (* MPL_BUFFMD *)

endproc (* MPL *)


(************************ The AC process *****************************)

process AC[tick,ear_ac,ac_mpl,ac_implic,ac_normal,ac_buffered]
(iR:imRc,iA:inArr,m:Map): noexit :=

(
(ac_normal;
AC_NORMAL[tick,ear_ac,ac_mpl,ac_implic,ac_normal,ac_buffered](iR,iA,m))
[]
(ac_buffered?b:indice;  (* b indicates which transformation is buffered *)
AC_BUFFMD[tick,ear_ac,ac_mpl,ac_implic,ac_normal,ac_buffered](iR,iA,m,b)) )

(* The choice here is between normal mode and buffered mode. The actual
mechanism used is not currently clear. Thus, we allow the environment to
choose through actions ac_normal and ac_buffered. *)

where
process AC_NORMAL[tick,ear_ac,ac_mpl,ac_implic,ac_normal,ac_buffered]
(iR:imRc,iA:inArr,m:Map) : noexit :=

  (( ear_ac?r1:Rep ; exit(r1) )
  (* Input Ports *)
  |||
  ( ( BLEND2[ac_mpl](ACC,MPLL,j0,iA,m) >> exit(any Rep) )
  ||| ( BLEND2[ac_implic](ACC,IMPLICC,j1,iA,m) >> exit(any Rep) ) )
  (* Output Ports *) )
  >> accept r1:rep in
     tick; AC[tick,ear_ac,ac_mpl,ac_implic,ac_normal,ac_buffered]
              (add(#(r1,0,0,0),iR),#(r1,0,0,0),m)

endproc (* AC_NORMAL *)

process AC_BUFFMD[tick,ear_ac,ac_mpl,ac_implic,ac_normal,ac_buffered]
(iR:imRc,iA:inArr,m:Map,b:indice): noexit :=

( ( ear_ac?r1:Rep ; exit(r1) )
(* Input Ports *)
|||
( ( OUTPUT2[ac_mpl](ACC,MPLL,b,j0,iR,iA,m) >> exit(any Rep) )
|||
```

```
( OUTPUT2[ac_implic](ACC,IMPLICC,b,j1,iR,iA,m) >> exit(any Rep) ) )
(* Output Ports *) )
>> accept r1:rep in
   tick; AC[tick,ear_ac,ac_mpl,ac_implic,ac_normal,ac_buffered]
            (add(#(r1,0,0,0),iR),#(r1,0,0,0),m)

endproc (* AC_BUFFMD *)

endproc (* AC *)


(************************* The ART process ******************************)

process ART[tick,bs_art,mpl_art,art_sp,art_wr,art_normal,art_buffered]
(iR:imRc,iA:inArr,m:Map): noexit :=

(
(art_normal;
ART_NORMAL[tick,bs_art,mpl_art,art_sp,art_wr,art_normal,art_buffered](iR,iA,m))
[]
(art_buffered?b:indice;  (* b indicates which transformation is buffered *)
ART_BUFFMD[tick,bs_art,mpl_art,art_sp,art_wr,art_normal,art_buffered]
(iR,iA,m,b)) )

(* The choice here is between normal mode and buffered mode. The actual
mechanism used is not currently clear. Thus, we allow the environment to
choose through actions art_normal and art_buffered. *)

where
process ART_NORMAL[tick,bs_art,mpl_art,art_sp,art_wr,art_normal,art_buffered]
(iR:imRc,iA:inArr,m:Map) : noexit :=

  (( bs_art?r1:Rep ; exit(r1,any Rep) ||| mpl_art?r2:Rep ; exit(any Rep,r2) )
  (* Input Ports *)
  |||
  ( ( BLEND2[art_sp](ARTT,SPP,j0,iA,m) >> exit(any Rep,any Rep) )
  ||| ( BLEND2[art_wr](ARTT,WRR,j1,iA,m) >> exit(any Rep,any Rep) ) )
  (* Output Ports *) )
  >> accept r1,r2:rep in
     tick; ART[tick,bs_art,mpl_art,art_sp,art_wr,art_normal,art_buffered]
               (add(#(r1,r2,0,0),iR),#(r1,r2,0,0),m)

endproc (* ART_NORMAL *)

process ART_BUFFMD[tick,bs_art,mpl_art,art_sp,art_wr,art_normal,art_buffered]
(iR:imRc,iA:inArr,m:Map,b:indice): noexit :=
```

```
( ( bs_art?r1:Rep ; exit(r1,any Rep) ||| mpl_art?r2:Rep ; exit(any Rep,r2) )
(* Input Ports *)
|||
( ( OUTPUT2[art_sp](ARTT,SPP,b,j0,iR,iA,m) >> exit(any Rep,any Rep) )
|||
( OUTPUT2[art_wr](ARTT,WRR,b,j1,iR,iA,m) >> exit(any Rep,any Rep) ) )
(* Output Ports *) )
>> accept r1,r2:rep in
   tick; ART[tick,bs_art,mpl_art,art_sp,art_wr,art_normal,art_buffered]
             (add(#(r1,r2,0,0),iR),#(r1,r2,0,0),m)

endproc (* ART_BUFFMD *)

endproc (* ART *)

(************************* BLENDING Functions *******************)

process BLEND1[g](X,Y:Subsyst,id:indice,iA:inArr,m:Map): exit :=
    choice r:Rep [] i; g!tran(X,Y,r); exit
endproc (* BLEND1 *)

process BLEND2[g](X,Y:Subsyst,id:indice,iA:inArr,m:Map): exit :=
    choice j:indice [] [smget(j,mpget(id,m))] ->
                i; g!tran(X,Y,iaget(j,iA)); exit
endproc (* BLEND2 *)

process BLEND3[g](X,Y:Subsyst,id:indice,iA:inArr,m:Map): exit :=
    g!tran(X,Y,compare(mpget(id,m),iA)); exit
endproc (* BLEND3 *)

process BLEND4[g](X,Y:Subsyst,id:indice,iA:inArr,m:Map): exit :=
    let sm:slotmap=mpget(id,m) in
      ( [compare(sm,iA) gt succ(0)] -> g!tran(X,Y,mult(sm,iA)); exit
        []
        [compare(sm,iA) le succ(0)] -> g!tran(X,Y,0); exit )
endproc (* BLEND4 *)

process BLEND5[g](X,Y:Subsyst,id:indice,iA:inArr,m:Map): exit :=
    g!tran(X,Y,mult(mpget(id,m),iA)); exit
endproc (* BLEND4 *)

(************************* OUTPUTING Functions ********************)
(* Used in buffered mode *)

process OUTPUT1[g](X,Y:Subsyst,b,j:indice,iR:imRc,iA:inArr,m:Map): exit :=
```

```
  [b eqq j] -> ( choice n:Nat [] BLEND1[g](X,Y,j,select(n,iA,iR),m) )
  []
  [b neq j] -> ( BLEND1[g](X,Y,j,iA,m) )
endproc (* OUTPUT1 *)

process OUTPUT2[g](X,Y:Subsyst,b,j:indice,iR:imRc,iA:inArr,m:Map): exit :=
  [b eqq j] -> ( choice n:Nat [] BLEND2[g](X,Y,j,select(n,iA,iR),m) )
  []
  [b neq j] -> ( BLEND2[g](X,Y,j,iA,m) )
endproc (* OUTPUT2 *)

process OUTPUT3[g](X,Y:Subsyst,b,j:indice,iR:imRc,iA:inArr,m:Map): exit :=
  [b eqq j] -> ( choice n:Nat [] BLEND3[g](X,Y,j,select(n,iA,iR),m) )
  []
  [b neq j] -> ( BLEND3[g](X,Y,j,iA,m) )
endproc (* OUTPUT3 *)

process OUTPUT4[g](X,Y:Subsyst,b,j:indice,iR:imRc,iA:inArr,m:Map): exit :=
  [b eqq j] -> ( choice n:Nat [] BLEND4[g](X,Y,j,select(n,iA,iR),m) )
  []
  [b neq j] -> ( BLEND4[g](X,Y,j,iA,m) )
endproc (* OUTPUT4 *)

process OUTPUT5[g](X,Y:Subsyst,b,j:indice,iR:imRc,iA:inArr,m:Map): exit :=
  [b eqq j] -> ( choice n:Nat [] BLEND5[g](X,Y,j,select(n,iA,iR),m) )
  []
  [b neq j] -> ( BLEND5[g](X,Y,j,iA,m) )
endproc (* OUTPUT5 *)

endspec
```

9.8 Notes

There are some remaining isues and points:-

- Notio of system hitting equilibrium - where outputs have stabilised and will nto change unless the inputs change.
- Distinction between implicit and explicit blending of body state with `LIM/SPEECH`. We only apply a multiplicative blend when the body state has to be considered explicitly, i.e. if the cursor needs to be associated with the body state. Normal speech and pointing does not require such a multiplicative blend. It is an implicit action.
- `id` and `X`, `Y` represent the same information - should combine the two into a single concept.