Behavioural Types for Synchronous Software Composition [1]

Barry Norton [2]

Knowledge Media Institute, Open University, Milton Keynes, UK

Abstract

Digital signal–processing (DSP) development tools such as Ptolemy, LabView and iConnect allow application developers to assemble reactive systems by connecting predefined components in generalised dataflow graphs and by hierarchically building new components by encapsulating sub–graphs. We follow the literature in calling this approach *dataflow-oriented* development. Previous work has shown how a novel process calculus, CaSE, can provide a model for this form of software, and how this can be used as the basis for a system of behavioural types. Well-typedness in this system implies reactivity (non-termination) in terms of a generalisation to the dataflow principle of consistency, which was previously unable to handle, in general, statefulness and non-determinism. In the previous presentation the typing rules were parameterised in a semantic behavioural equivalence, temporal observation congruence, which specialises CCS's notion of weak bisimulation to this setting. In this work, we show how a complete axiom system for CaSE allows these equivalences to be reduced to a syntactic check, which is more fitting to a type system.

Key words: Behavioural Types, Process Algebra, Software Composition, Component-Based Development

1 Introduction

The dataflow style provides a well understood basic formalism for designers of embedded digital signal processing systems. For this reason it is no surprise that tools presenting system design in this style became very popular and have retained their dominance. In applications involving measurement/monitoring and control there is a need for rapid application design and for the agility to make rapid design changes and variations.

Systems such as LabView [6] and iConnect [18] include huge libraries of components to interface with measurement hardware and must cope with this drive for rapid development since they are used to create *ad hoc* measurement systems and bespoke embedded controllers respectively. In order to deal with dynamic aspects such as user interaction, LabView includes control primitives, familiar from imperative languages, as first class and graphically represented entities within a therefore heterogeneous design language [6]. iConnect, on the other hand, like the dataflow models in the Ptolemy framework, chooses rather to generalise over the dataflow model to allow control signals and the behaviour of actors to be both non-deterministic and stateful [18].

Previous work has shown how the process algebra CaSE can be used to model these latter systems [17] and to check their 'consistency' [10], *i.e.* the absence of buffer overflow during scheduling [16]. Previous models, in particular the token flow model [4], had been unable to model non-determinism and statefulness generally, reducing both to their simple cases via probabilistic assumptions unlikely to be met by general components. The CaSE model, on the other hand, provides a compositional account for such general components.

The approach is also similar to another branch of research in Ptolemy where interface automata [7] are used to describe the I/O behaviour of components. In Ptolemy these have been used to build a behavioural type system [11], but this is point-to-point checking whereas the CaSE model allows type inference to be built compositionally. Furthermore, the behavioural type system in Ptolemy does not guarantee consistency, whereas this is provided for by type checking (which is behavioural equivalence with respect to inferred type) in our model. Finally, the behavioural type system in Ptolemy has necessitated an extension to the interface automata model, to represent 'transient states', but this has not been given a full formal treatment. In CaSE the principle of maximal progress allows the consistent treatment of patient and insistent states, and a congruence is provided that allows instantaneous behaviours to be abstracted over, courtesy of the hiding operator which turns local timing conditions into silent behaviour which is therefore unobservable.

The drawback in our previous model is that types for encapsulation are parameterised in the behavioural, *i.e.* semantic, version of the congruence. This is somewhat out of character with the notion of a type system, whose intention may be perceived as abstracting from the behaviour of a program to allow syntactic conditions to constrain the behaviours to only those which are correct. By providing a complete axiomatisation of the behavioural equivalance we allow this check to be made on syntactic terms and make our system for static validation more persuasively presented as a type system.

The paper is organised as follows. In Sections 2, 3 and 4 we review the setting, and in Section 5 the CaSE process calculus. In Section 6 we present for the first time the algebraic characterisation of this model via an axiom system. In Section 7 we update our behavioural type system to use this syntactic version of equivalence. Finally in Section 8 we conclude.
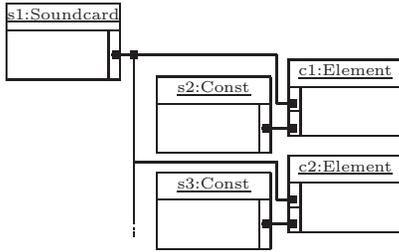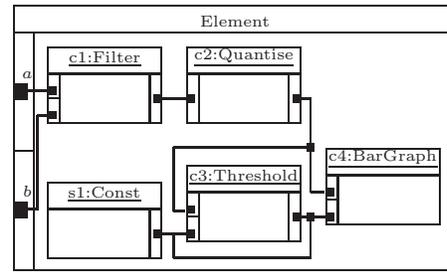
2

Fig. 1. Example Application

Fig. 2. Example Encapsulation

2 Dataflow-Oriented Design

We shall exemplify dataflow-oriented systems with a digital spectrum analyser
built from the components shown in Figures 3 through 8. The overall design
is shown in Figure 1, which relies also on a further component defined by
encapsulation as shown in Figure 2.

We observe that the user-oriented representations shown for component
semantics are finite automata where the labels are divided into:

- *output channels*, labelled with a '!' and named for the output ports of the
  component (shown on the right) and the channel 'r' which signals 'readiness
  for execution' to the scheduler;

- *input channels*, labelled with a '?' and named for the input ports (shown
  on the right of the component) as well as the channel 'e', via which the
  scheduler signals permission to execute;

- unlabelled *internal* steps, which we understand as the execution of the al-
  gorithm that the component encapsulates.

We observe, but do not formalise, here that well-formed descriptions must be
labelled only from these alternatives any path must strictly follow the cycle:
input, readiness, scheduling, execution, output.

The intuition for the example in Figure 1 is that it will compute on input
from a 'sound card' device, introduced via the instance $c1$ of the *Soundcard*
component and ultimately display the results via instances of the *BarGraph*
component, as encapsulated within the *Element* component. The point of
the remainder of the *Element* component is to prepare the raw values for
display. This is parameterised not only in the raw value, but also in the range
by which to filter so that each instance will effect a different element in the
overall spectrum.

We see from its diagram that *Soundcard* is initially ready for execution
(emitting $r!$) and when allowed always produces a value (*i.e.* an output $c!$)
and becomes ready again. Similarly as soon as *Quantise* receives a value at
its input port ($a?$) it becomes ready to execute, which involves computation
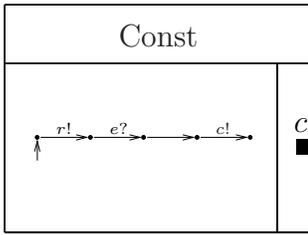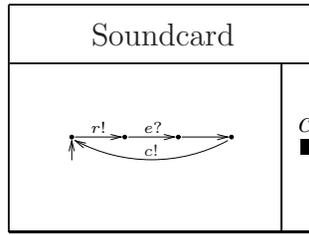and the output of a value, and the cycle begins again.
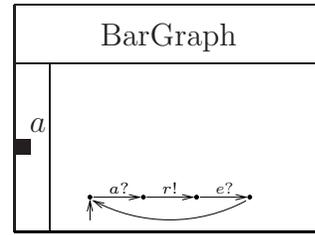
3

Fig. 3. Const

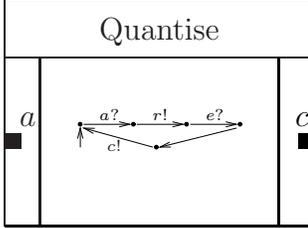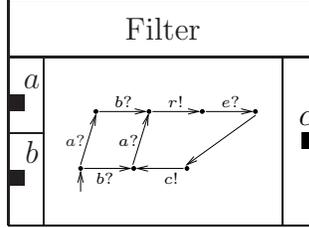Fig. 4. Soundcard

Fig. 5. BarGraph
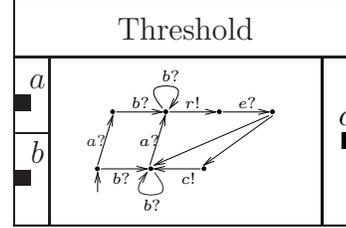

Fig. 6. Quantise

Fig. 7. Filter

Fig. 8. Threshold

To set the frequency range of each instance of the *Element* component, an instance of *Const* is used. This is initially ready to execute and provide such a value but thereafter never becomes ready again. This is a simple form of statefulness; a more advanced form is present in the behaviour of *Filter*, which needs one value on each port to become ready but thereafter only accepts input on port a. Our intuition for this behaviour is that port b is a filter range and port a a signal to filter; being a digital filter, the functional behaviour is implemented using calculation over previous values (of a under b) and therefore the filtering parameters cannot be dynamically changed.

The *Bargraph* component shows how this behaviour can be extended with the use of signal absence. Input port a is used to carry a signal to display on the bar and port b is used for a peak value 'needle', which need not change just because the signal does. As a consequence one value is required of each, for an initial complete display, in the first cycle. Thereafter the component is just as able to proceed with or without a new value on b, once a value has been received at a.

The *Threshold* component shows a variation on this behaviour and how non-determinism can also play a part. The implementation of this component will accept a threshold value at port b and a 'signal' value at port a; it will *only* propagate values received on a if they exceed the threshold. Unlike the bar graph component this allows the 'over-writing' of the threshold value multiple times before execution. By including a feedback loop, possible due to precisely this non-determinism, back to the threshold input we force this component to store the 'global' peak value.

3 A Dataflow-Oriented ADL

We now relate, in Table 1, the architecture description language (ADL) for dataflow-oriented systems defined in [16]. This establishes that a given system composes a positive number of instances of 'source components' (fixed by the language) — like Const and Soundcard, we see the reason for distinguishing these in the following section — introduced via the **insts** keyword, with a positive number of computation components (a core set of which are contained in the language, with more formed by encapsulation) introduced via the **instc** keyword, and some number of wires to define their communication.

The network of wires that connect outputs of named component instances to inputs of similar, formed with the **wire** keyword, are interpreted as forming a system of forks connected to each output and joins connected to each input.

As well as the core computation components provided by the framework, the **instc** keyword may be applied to encapsulated subsystems of computation components formed with the **enc** keyword. There are also special wires, made available via the **iwire** and **owire** keywords, to expose inputs and outputs without naming their connected parties that are used to expose ports during encapsulation.

$$
\begin{aligned}
System ::=\ &\mathbf{sys}(SourceSystem, CompSystem) \\
SourceSystem ::=\ &SourceInstance\ | \\
&SourceInstance; SourceSystem \\
SourceInstance ::=\ &\mathbf{insts}(Source, \text{name}) \\
Source ::=\ &\mathbf{Const}\ | \\
&\mathbf{Soundcard} \\
CompSystem ::=\ &CompInstance\ | \\
&CompInstance; CompSystem\ | \\
&CompSystem; Wire \\
CompInstance ::=\ &\mathbf{instc}(Comp, \text{name}) \\
Comp ::=\ &\mathbf{Filter}\ | \\
&\mathbf{Quantise}\ | \\
&\mathbf{BarGraph}\ | \\
&\mathbf{enc}(CompSystem, \text{name}, I, O) \\
Wire ::=\ &\mathbf{wire}(\text{name}, \text{port}, \text{name}, \text{port})\ | \\
&\mathbf{iwire}(\text{port}, \text{name}, \text{port})\ | \\
&\mathbf{owire}(\text{name}, \text{port}, \text{port})
\end{aligned}
$$

Table 1
Architectural Description Language

4 Synchronisation Principles

4.1 Synchrony

The *synchrony hypothesis* embodies the principle that reactive systems should be considered as forming a complete *instantaneous* reaction to stimuli, *i.e.* one formed infinitely faster than the environment can react back [2]. In this way, system behaviour may be broken up into a discrete series of synchronous steps. This principle is implemented in iConnect on two levels. Firstly, the provision of *source components* allows the internalisation of environmental sampling into the component model. Source components are those without input ports, *i.e. Soundcard* and *Const* in the example, all others are called *computation components* and are involved in the computation of a synchronous reaction to data from source components. We can thus see how *iConnect* schedules systems in a synchronous manner by allowing instances of source components to be executed and thereafter executing all instances of computation components that become 'ready' (*i.e. ready for execution*) until the list of these is exhausted, at which point the source components are considered again *etc.*

The means by which we compositionally achieve this scheduling style in a coordination model is via the composition of the agents (the first with source components, the second computation) shown in Figures 9 and 10, these effect serialised execution via a 'token passing game': owing the t_e (environment) token allows source components to execute; the t_c token allows computation components to execute. The timing of the game is arranged via the global synchronous clock, σ, and a clock σ_n local to each participant based on the unique instance name n. The agent patiently waits for a request to execute (the doubled circle implies that any clock not explicitly drawn 'idles', *i.e.* labels an implicit self-transition) and responds by waiting for the appropriate token then, having received it, signalling permission to execute e. When the σ_n clock ticks, the execution is complete and the token is returned, either (preferentially to a peer) or, if the σ clock signals that this is not possible, to the other kind of component. The difference between the source and computation scheduler is that the former must wait for the global synchronous clock before running again.
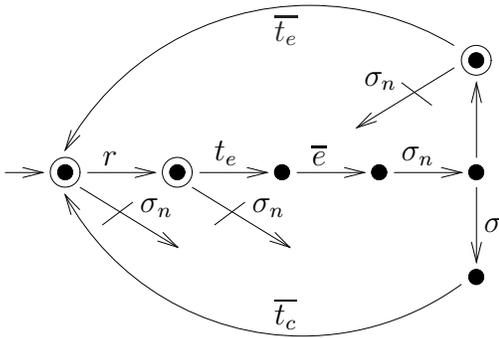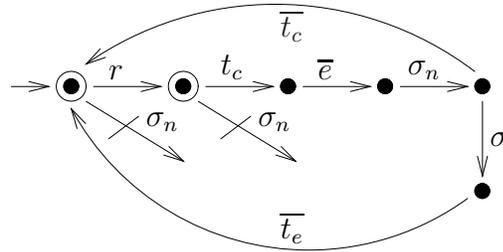

Fig. 9. Basic Source Scheduler   Fig. 10. Basic Computation Scheduler

6

4.2 Isochrony

'Joins' in our dataflow designs behave in the manner referred to in the literature as non-deterministic merge, wherein the arrival of a value at either side of the join triggers a communication to the recipient and does not pre–determine or prejudice from which side the next value will be propagated. A join is shown at the second input port of $c3$ in Figure 2.

'Forks', on the other hand, (as shown connected to the output of $s1$ in Figure 1) have a broadcast behaviour, meaning that all recipients will receive the value. In order to sensibly reason about designs, especially those involving loops, it is an established principle of hardware design — just as the synchrony hypothesis is an established principle of software design — that forks should behave in a manner consistent with the *isochronic* view. Isochronic forks not only broadcast exactly one copy of each value arriving to each recipient, but communicate these values *at the same time* so that it is not possible for an earlier recipient to prejudice the handling of the data by another recipient.

The means by which these forks are achieved is the composition of the agents shown in Figures 11 and 12. One copy of the 'broadcast' agent is composed for each output c of the instance n. On receipt of the output, a uniquely-named copy c_n is broadcast to each recipient until there are no more such as measured by the associated clock σ_{c_n} bound by maximal progress. The recipients are all instances of the 'wire' component, one copy of this being composed for each wire in the system from this output to the input a of a component m. The 'right hand' behaviour of the wire agent shows that the value is patiently waited for and then insistently delivered before synchronising on the associated clock.

The 'left hand' behaviour is novel to this presentation and involves a clock uniquely named for the input, σ_{a_m}. This clock is prevented in the initial state (and in all states associated with a positive value), but if the output clock fires without a value being received this will signal that all computation that might lead to the output is complete and allow the input clock to tick. This clock will only tick when all such agents can synchronise, having been through clocks signalling that their associated outputs will not appear, and so encodes the absence of the input signal and then the communication of this. An occurence of the global synchronous clock σ means that this behaviour is reset.
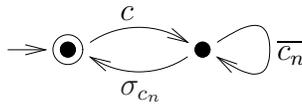
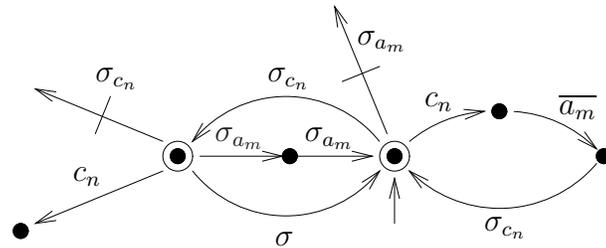

Fig. 11. Isochronous Broadcast   Fig. 12. Isochronous Wire

5 CaSE: Calculus for Synchrony and Encapsulation

CaSE is derived from CCS [13] via TPL [8], which includes (i) a single abstract clock σ, which is interpreted not quantitatively as some number encoding an exact time but qualitatively as a recurrent, global and abstract synchronisation event; (ii) a timeout operator $\lfloor E \rfloor \sigma(F)$ similar to ATP [14], where the occurrence of σ deactivates process E and activates F; (iii) the concept of maximal progress, which imposes that a clock can only tick if no process can engage in any further internal activity τ.

CaSE further extends TPL by (i) allowing for *multiple clocks* $\sigma, \rho, \ldots$ as in PMC [1] and CSA [5] while, in contrast to PMC and CSA, maintaining the global interpretation of maximal progress; (ii) two generalisations of the time-out operator to multiple clocks — in $\lfloor E \rfloor \sigma(F)$ another clock ρ can chose for E, in $\lceil E \rceil \sigma(F)$ only an action transition from E will remove the timeout; (iii) explicit 'stalling' operators Δ and Δ_σ that *locally* prohibit the ticking of all clocks and of clock σ, respectively (iv) *clock–hiding* operators $/\sigma$ such that all clock ticks of process P are internalised in process P/σ.

$$a, \overline{a}, b, \overline{b}, \cdots \in \Lambda \cup \overline{\Lambda}$$

$$\rho, \sigma, \cdots \in \mathcal{T}$$

$$L \subseteq \Lambda$$

$$T \subseteq \mathcal{T}$$

$$\alpha, \beta, \cdots \in \Lambda \cup \overline{\Lambda} \cup \{\tau\}$$

$$\gamma, \delta \cdots \in \Lambda \cup \overline{\Lambda} \cup \{\tau\} \cup \mathcal{T}$$

$$\mathcal{E} ::= \mathbf{0} \mid \Delta \mid \Delta_\sigma \mid \alpha.\mathcal{E} \mid \mathcal{E} + \mathcal{E} \mid$$
$$\lfloor \mathcal{E} \rfloor \sigma(\mathcal{E}) \mid \lceil \mathcal{E} \rceil \sigma(\mathcal{E}) \mid \mathcal{E}|\mathcal{E} \mid$$
$$\mu X.\mathcal{E} \mid X \mid \mathcal{E} \setminus a \mid \mathcal{E}/\sigma$$

$$\mathcal{F} ::= \mathbf{0} \mid \Delta \mid \Delta_T \mid$$
$$\gamma.\mathcal{F} \mid \underline{\gamma}.\mathcal{F} \mid \underline{\gamma}_T.\mathcal{F} \mid \mathcal{F} + \mathcal{F} \mid$$
$$\lfloor \mathcal{F} \rfloor \sigma(\mathcal{F}) \mid \lfloor \mathcal{F} \rfloor \vec{\sigma}(\vec{\mathcal{F}}) \mid \lfloor \mathcal{F} \rfloor \vec{\sigma}(\mathcal{F}) \mid$$
$$\lceil \mathcal{F} \rceil \sigma(\mathcal{F}) \mid \lceil \mathcal{F} \rceil \vec{\sigma}(\vec{\mathcal{F}}) \mid \lceil \mathcal{F} \rceil \vec{\sigma}(\mathcal{F}) \mid$$
$$\mathcal{F}|\mathcal{F} \mid \mu X.\mathcal{F} \mid X \mid \mathcal{F} \setminus L \mid \mathcal{F}/\vec{\sigma}$$

$$\Delta_T \overset{\text{def}}{=} \sum_{\sigma \in T} \Delta_\sigma$$

$$\underline{\alpha}.E \overset{\text{def}}{=} \alpha.E + \Delta$$

$$\underline{a}_T.E \overset{\text{def}}{=} a.E + \Delta_T$$

$$\underline{\sigma}.E \overset{\text{def}}{=} \lfloor \Delta \rfloor \sigma(E)$$

$$\sigma.E \overset{\text{def}}{=} \lceil \mathbf{0} \rceil \sigma(E)$$

$$\underline{\sigma}_T.E \overset{\text{def}}{=} \lceil \Delta_T \rceil \sigma(E)$$

$$\lfloor E \rfloor \rho(F)\sigma(G) \overset{\text{def}}{=} \lfloor \lfloor E \rfloor \rho(F) \rfloor \sigma(G)$$

$$\lfloor E \rfloor \vec{\sigma}(F) \overset{\text{def}}{=} \lfloor E \rfloor \sigma_1(F) \cdots \sigma_n(F)$$

$$\lfloor E \rfloor \vec{\sigma}(\vec{F}) \overset{\text{def}}{=} \lfloor E \rfloor \sigma_1(F_1) \cdots \sigma_n(F_n)$$

$$\text{likewise } \lceil E \rceil \cdots$$

$$E \setminus L \overset{\text{def}}{=} E \setminus a_1 \cdots \setminus a_n$$

$$\text{where } L = \{a_1 \cdots a_n\}$$

$$E/\vec{\sigma} \overset{\text{def}}{=} E/\sigma_1 \cdots /\sigma_n$$

$$\text{where } \vec{\sigma} = \sigma_1 \cdots \sigma_n$$

Table 2
Core ($\mathcal{E}$) and Derived ($\mathcal{F}$) CaSE Syntax

The syntax of CaSE is defined, as shown in Table 2, with respect to a countable set of names, Λ, and a countable set of clocks, $\mathcal{T}$. For convenience a derived syntax is also defined and used throughout this paper. The semantics of the core expressions is defined by a labelled transition system $\langle \mathcal{E}, \mathcal{A} \cup \mathcal{T}, \longrightarrow, \mathcal{E} \rangle$ according to the operational semantics in Table 3.

$$\overline{\mathbf{0} \xrightarrow{\sigma} \mathbf{0}} \qquad \overline{\alpha.E \xrightarrow{\alpha} E} \qquad \overline{a.E \xrightarrow{\sigma} a.E} \qquad \overline{\Delta_\sigma \xrightarrow{\rho} \Delta_\sigma}1$$

$$\frac{E \xrightarrow{\alpha} E'}{E + F \xrightarrow{\alpha} E'} \qquad \frac{F \xrightarrow{\alpha} F'}{E + F \xrightarrow{\alpha} F'} \qquad \frac{E \xrightarrow{\sigma} E' \quad F \xrightarrow{\sigma} F'}{E + F \xrightarrow{\sigma} E' + F'}$$

$$\frac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F} \qquad \frac{F \xrightarrow{\alpha} F'}{E \mid F \xrightarrow{\alpha} E \mid F'} \qquad \frac{E \xrightarrow{a} E' \quad F \xrightarrow{\overline{a}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$$

$$\frac{E \xrightarrow{\sigma} E' \quad F \xrightarrow{\sigma} F' \quad E \mid F \xrightarrow{\tau}}{E \mid F \xrightarrow{\sigma} E' \mid F'} \qquad \frac{E \xrightarrow{\gamma} E'}{E \setminus a \xrightarrow{\gamma} E' \setminus a}2$$

$$\frac{E \xrightarrow{\tau}}{\lfloor E \rfloor \sigma(F) \xrightarrow{\sigma} F} \qquad \frac{E \xrightarrow{\gamma} E'}{\lfloor E \rfloor \sigma(F) \xrightarrow{\gamma} E'}3 \qquad \frac{E \xrightarrow{\gamma} E'}{\mu X.E \xrightarrow{\gamma} E'\{\mu X.E/X\}}$$

$$\frac{E \xrightarrow{\tau}}{\lceil E \rceil \sigma(F) \xrightarrow{\sigma} F} \qquad \frac{E \xrightarrow{\alpha} E'}{\lceil E \rceil \sigma(F) \xrightarrow{\alpha} E'}3 \qquad \frac{E \xrightarrow{\rho} E'}{\lceil E \rceil \sigma(F) \xrightarrow{\rho} \lceil E' \rceil \sigma(F)}1$$

$$\frac{E \xrightarrow{\sigma} E'}{E/\sigma \xrightarrow{\tau} E'/\sigma} \qquad \frac{E \xrightarrow{\alpha} E'}{E/\sigma \xrightarrow{\alpha} E'/\sigma} \qquad \frac{E \xrightarrow{\rho} E' \quad E \xrightarrow{\sigma}}{E/\sigma \xrightarrow{\rho} E'/\sigma}1$$

where: $^1\rho \neq \sigma$, $^2\gamma \neq a$, $^3\gamma \neq \sigma$

Table 3
CaSE Semantics

The basic form of communication, formed as $a.E$ is 'patient', *i.e.* the process will wait through any number of clocks until the communication is matched (at which point clocks are prevented by maximal progress). In the derived syntax we create also an 'insistent' form of communication $\underline{a}.E$, which allows no clocks to tick until the communication, and a 'relatively insistent' form $\underline{a}_T.E$, which is insistent for a closed set of clocks; we can interpret this as 'turning off' the clock, like a stopwatch, while waiting for the communication.

6 Algebraic Theory of CaSE

The basic equational axioms for CCS, the so-called summation and tau axioms, remain sound for CaSE (and so are prefixed 'M' for Milner in our naming). For the recursion axioms we have to introduce an extra guard for the first (and so this axiom is prefixed 'T' for timed).

Definition 6.1 Summation Axioms

MS1	$E + F = F + E$	MS2	$E + (F + G) = (E + F) + G$
MS3	$E + E = E$	MS4	$E + \mathbf{0} = E$

Definition 6.2 Tau Axioms

MT1	$\alpha.\tau.E = \alpha.E$	MT2	$E + \tau.E = \tau.E$
MT3	$\alpha.(E + \tau.F) + \alpha.F = \alpha.(E + \tau.F)$		

Definition 6.3 Recursion Axioms

TR1 $\mu X.E = E\{\mu X.E/X\}$, provided X is guarded in E

MR2   If $F = E\{F/X\}$ then $F = \mu X.E$, provided X is guarded in E

Further to this there are additional tau axioms and new axioms account for preemption, hiding and time-out.

Definition 6.4 (Additional) Tau Axioms

TT1	$\tau.E = \tau.E + \Delta$	TT2	$\lfloor \tau.E + F \rfloor \sigma(G) = \tau.E + F$
TT3	$\alpha.\lfloor E \rfloor \sigma(F) = \alpha.\lfloor E \rfloor \sigma(\tau.F)$		

Definition 6.5 Preemption Axioms

TD1 $\Delta = \Delta + \Delta_\sigma$ TD2 $\lfloor \Delta_\sigma \rfloor \vec{\rho}(\vec{E}) + \lfloor F \rfloor \sigma(G) = \lfloor \Delta_\sigma \rfloor \vec{\rho}(\vec{E}) + F$

(where $\sigma \notin \vec{\rho}$)

Definition 6.6 Time-out Axioms

TO1 $\lfloor \lfloor E \rfloor \sigma(F) \rfloor \sigma(G) = \lfloor E \rfloor \sigma(G)$

TO2 $\lfloor E \rfloor \rho(F) \sigma(G) = \lfloor E \rfloor \sigma(G) \rho(F)$ (where $\rho \neq \sigma$)

TO3 $\lfloor E \rfloor \sigma(F) + \lfloor G \rfloor \sigma(H) = \lfloor E + G \rfloor \sigma(F + H)$

TO4 $\mathbf{0} = \lfloor \mathbf{0} \rfloor \sigma(\mathbf{0})$ TO5 $a.E = \lfloor a.E \rfloor \sigma(a.E)$

TO6 $\Delta_\rho = \lfloor \Delta_\rho \rfloor \sigma(\Delta_\rho)$ (where $\rho \neq \sigma$)

The significant departure from the CCS axioms, though, is in the so-called 'unguardedness axioms'. In CCS we can remove a silent loop with 'branches', such as $\mu X.(E + \tau.(F + \tau.X))$, in terms of a direct non-deterministic choice $\mu X.(\tau.E + F)$. In CaSE this is unsound, since clocks shared by E and F can proceed in the latter, whereas they were blocked by maximal progress in the former. The use of Δ allows these silent loops to be removed soundly.

Definition 6.7 Unguardedness Axioms

TU1 $\mu X.(X + E) = \mu X.(E + \Delta)$ TU2 $\mu X.(\tau.X + E) = \mu X.\tau.(E + \Delta)$

MU3 $\mu X.(\tau.(X + E) + F) = \mu X.(\tau.X + E + F)$

The first necessary proof is that all of these axioms are sound with respect to the previous behavioural equivalence for closed terms.

Definition 6.8 Temporal Weak Bisimulation (t.w.b.)

A symmetric relation $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$ is a t.w.b. if whenever $\langle E, F \rangle \in \mathcal{R}$:

- If $E \xrightarrow{\gamma} E'$ then $\exists F' \cdot F \xRightarrow{\hat{\gamma}} F'$ and $\langle E', F' \rangle \in \mathcal{R}$ ($\hat{\tau} = \epsilon$, $\hat{\gamma} = \gamma$ otherwise)

We write $E \approx F$, if $\langle E, F \rangle \in \mathcal{R}$ for some t.w.b. $\mathcal{R}$.

Definition 6.9 Temporal Observation Congruence (t.o.c.)

A symmetric relation $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$ is a t.o.c. if whenever $\langle E, F \rangle \in \mathcal{R}$:

  (i) $E \xrightarrow{\alpha} E'$ implies $\exists F'. F \xRightarrow{\alpha} F'$ and $E' \approx F'$.
  (ii) $E \xrightarrow{\sigma} E'$ implies $\exists F'. F \xrightarrow{\sigma} F'$ and $\langle E', F' \rangle \in \mathcal{R}$.

We write $E \approx^c F$ if $\langle E, F \rangle \in \mathcal{R}$ for some temporal observation congruence $\mathcal{R}$.

Theorem 6.10 *Soundness*

The axiom system $Ax_T \overset{def}{=} \{$ MS1, MS2, MS3, MS4, TR1, MR2, MT1, MT2, MT3, TT1, TT2, TT3, TO1, TO2, TO3, TO4, TO5, TO6, TU1, TU2, MU3, TD1, TD2 $\}$ *is sound w.r.t. temporal observation congruence, i.e:*

$$\text{If } Ax_T \vdash E = F \text{ then } E \approx^c F$$

The only significant departure from Milner's original soundness proofs in the timed case is the use of a sound version of 'weak bisimulation up to', the original having been mistakenly claimed a weak bisimulation [12].

Although an extended version of the expansion law is also needed, which space prevents us from including, the core of the theory is to show that these axioms are complete for temporal observation congruence over closed terms in the sub-language $\mathcal{F} ::= \mathbf{0} \mid \Delta \mid \Delta_\sigma \mid \alpha.\mathcal{F} \mid \mathcal{E} \vert \mathcal{E} \mid \lfloor \mathcal{F} \rfloor \sigma(\mathcal{F}) \mid \mu X.\mathcal{F} \mid X$.

Theorem 6.11 *Completeness*

The axiom system Ax_T *completely characterises temporal observation congruence, i.e: If $E \approx^c F$ then $Ax_T \vdash E = F$*

The proof broadly follow's Milner scheme in using the weak bisimulation that must exist between two congruent processes to nominate pairs of equivalent states then represented in a combined normal form as equations (understood as states) in an equation system $S : \widetilde{X} = \widetilde{H}$. In particular, each $H_i \in \mathcal{F}$ in a $(\vec{\sigma}, \widetilde{W})$-standard timed equation system has free variables in $\widetilde{X} \cup \widetilde{W}$, and is of the form (where $T_i = \mathcal{T}$ implies $\Delta_{T_i} = \Delta$):

$$\left\lfloor \sum_{k}^{K_i} \alpha_{ik}.X_{f(i,k)} + \Delta_{T_i} \right\rfloor \sigma_{g(i,1)} \left(X_{h(i,1)} \right) \sigma_{g(i,2)} \left(X_{h(i,2)} \right) \cdots \sigma_{g(i,c_i)} \left(X_{h(i,c_i)} \right)$$

$$+ \sum_{w=1}^{w_i} \left\lfloor W_{d(i,w)} \right\rfloor \sigma_{g(i,e(i,w))} \left(\mathbf{0} \right) \sigma_{g(i,e(i,w)+1)} \left(\mathbf{0} \right) \cdots \sigma_{g(i,c_i)} \left(\mathbf{0} \right)$$

Our approach to completeness is informed by [3], but we note that the approach of a 'pruning' operator in intermediate syntax is not useful where deterministic time and idling is included, since there is no process by which a pruned **0** process can be represented. In this sense we unite different clock-free processes in [9] (which also captures an equivalence finer than weak bisimulation and is not (time-)deterministic), [8] and [1] (where the axiomatisation is only complete for finite behaviours), each of which have no communication behaviour, like **0**, but affect other processes' synchronisation. In our context, though, the process Δ has already been shown to have useful properties as a first class member of the syntax, especially unlike [3]'s operator which only exists in an 'intermediate syntax', and has therefore been generalised to Δ_σ.

Consequently, the conversion of unguarded terms to congruent guarded ones is much simpler in our case and follows Milner's style directly, using the additional Δ summands to mark the extra preemptive power. Preemption is actually effected by Δ_σ, axioms TD1 and TD2 taken together showing how this is derived from the introduced Δ processes. While we should like a simpler axiom $\lfloor E \rfloor \sigma(F) + \Delta_\sigma = E + \Delta_\sigma$, to combine normal forms, which we must do first in equational characterisation of individual process and then in the formation of the common equation system, we need to preempt through the linear syntax for timeout operators and so an axiom scheme is needed. We can see that the simpler form is a derived axiom (of the trivial case).

To complete the proof, we build on an omission from Milner's construction — that the distinguished variable X_0, the 'start state' in a standard equation system, should not appear in any defining expression — and impose that all variables representing states reachable by clock transitions from X_0 must be distinct and not appear in the defining expressions of the other equations. In this way, the extra gap between equivalence and congruence in the timed case is represented. Thereafter although, as pointed out in [1], 'Hennessy's Lemma' (that $E \approx F$ but $E \not\approx^c F$ implies either $\tau.E \approx^c F$ or $E \approx^c \tau.F$) does not extend in general to the timed case, this does apply when $\nexists \sigma \cdot E \xrightarrow{\sigma}$. Thus we can drop the process of 'saturation' from clocks ([3] mistakenly claims that an equation system can be both standard and saturated for low-priority actions), and 'patch these up' in place to show provable satisfaction of the combined equation system, by the congruent expressions, and so provable equality.

7   Behavioural Types

As shown in Table 4, the interface automata introduced in Section 2 have a simple representation in the CCS fragment of the calculus. The context used for typing judgements carries the local synchronous clock, ranged over by σ, and collects together inputs and outputs introduced. Table 5 shows that the basic notion of composition is directly based on parallel composition. The isochronous wire agent introduced in Section 4.2, and its simplifications for connecting across hierarchy, are encoded in Table 6.

Table 7 shows how the agents discussed in Section 4.1, together with the broadcast agent from Section 4.2, are composed with component types to make an instantiated type. Finally in Table 8 it is shown how hiding — as well as restriction, as is normal in CCS — is used to achieve encapsulation. Both of these rules are expressed now in terms of the equality expressed by the axiom system over CaSE detailed in Section 6. In the first rule, we are essentially checking that the system-level synchronous clock remains live (checking congruence to **0** for this clock and all actions, since $E =_{\vec{\sigma}} F$ iff $(E \mid \mu X.\lfloor \Delta \rfloor \vec{\sigma}(X)) = (E \mid \mu X.\lfloor \Delta \rfloor \vec{\sigma}(X)))$. The second rule establishes that the assigned type Q accurately represents, from as observational point of view, the behaviour of the encapsulated subsystem.

$$\frac{\quad \text{—} \quad}{\sigma, \{\}, \{c\} \vdash \textbf{Const} : \overline{r}.e.\tau.\overline{c}.\textbf{0}}$$

$$\frac{\quad \text{—} \quad}{\sigma, \{\}, \{c\} \vdash \textbf{Soundcard} : \mu X.\overline{r}.e.\tau.\overline{c}.X}$$

$$\frac{\quad \text{—} \quad}{\sigma, \{a, b\}, \{c\} \vdash \textbf{Filter} : \quad \begin{aligned} &a.b.\overline{r}.e.\tau.c.\mu X.b.\overline{r}.e.\tau.c.X+ \\ &b.a.\overline{r}.e.\tau.c.\mu X.b.\overline{r}.e.\tau.c.X \end{aligned}}$$

$$\frac{\quad \text{—} \quad}{\sigma, \{a\}, \{c\} \vdash \textbf{Quantise} : \mu X.a.\overline{r}.e.\tau.\overline{c}.X}$$

$$\frac{\quad \text{—} \quad}{\sigma, \{a\}, \{\} \vdash \textbf{BarGraph} : \mu X.a.\overline{r}.e.\tau.X}$$

Table 4

Example Ground Types

$$\frac{\sigma, I_E, O_E \vdash S : E \quad I_E \cap I_F = \emptyset \qquad \sigma, I_F, O_F \vdash T : F \quad O_E \cap O_F = \emptyset}{\sigma, I_E \cup I_F, O_E \cup O_F \vdash S; T : E|F}$$

Table 5

Composition Type

$$\frac{\quad}{\sigma, \{\}, \{\} \vdash \textbf{wire}(n, c, m, a) : \mu X.c_n.\overline{a_m}.\sigma_{c_n}.X}$$

$$\frac{\quad}{\sigma, \{\}, \{\} \vdash \textbf{iwire}(a, m, b) : \mu X.a.\overline{b_m}.\sigma_a.X}$$

$$\frac{\quad}{\sigma, \{\}, \{\} \vdash \textbf{owire}(n, c, d) : \mu X.c_n.\overline{\overline{d}}.\sigma_{c_n}.X}$$

Table 6

Wire Types

13

$$\sigma, \{\}, O \vdash C : E$$

$$\overline{\sigma, \{\}, \{c_n \mid c \in O\} \vdash \mathbf{insts}(C, n) : \quad (E \mid}$$

$$\Pi_{c \in O} \, \mu X. \underline{c}_{\sigma_{c_n}} . \, \mu Y. \lfloor \overline{c_n}.Y \rfloor \sigma_{c_n}(X) \mid$$

$$\mu X.r.t_{e_{\sigma_n}}.\overline{e}.\underline{\sigma_n}.\lfloor \overline{t_e}.\sigma.X \rfloor \sigma(\overline{t_c}.X))$$

$$) \setminus (O \cup \{r, e\}) \,/\, \sigma_n$$

$$\sigma, I, O \vdash C : E$$

$$\overline{\sigma, \{a_n \mid a \in I\}, \{c_n \mid c \in O\} \vdash \mathbf{instc}(C, n) : \quad (E\{a \to a_n \mid a \in I\} \mid}$$

$$\Pi_{c \in O} \, \mu X. \underline{c}_{\sigma_{c_n}} . \, \mu Y. \lfloor \overline{c_n}.Y \rfloor \sigma_{c_n}(X) \mid$$

$$\mu X.\underline{r}_{\sigma_n}.(\overline{r_e}.t_{c_{\sigma_n}}.\overline{e}.\underline{\sigma_n}.\lfloor \overline{t_c}.X \rfloor \sigma(\overline{t_e}.X) + t_{c_{\sigma_n}}.\overline{e}.\underline{\sigma_n}.\lfloor \overline{t_c}.X \rfloor \sigma(\overline{t_e}.X))$$

$$) \setminus (O \cup \{r, e\}) \,/\, \sigma_n$$

Table 7
Instantiation Types

8 Conclusions

We have shown how the standing of our behavioural type system for dataflow-oriented software composition can be increased via a syntactic account for the observational equivalence between processes. We have also sketched the results from [15] showing how this syntactic account is sound and complete.

The axiom system presented is the first for a weak bisimulation-based 'observational' equivalence in the presence of maximal progress-bound timing. While TPL was given a complete axiom system [8], this captured a testing-based equivalence which is divergence sensitive, whereas in order to deal with loops in dataflow graphs we need to abstract from silent loops with exits. The fact that in general, as pointed out in [1], this can create effectively nondeterministic weak clock transitions is obviated via global preemption.

$$\sigma_s, \{\}, O_1 \vdash S : E \qquad \sigma_s, I, O_2 \vdash C : F$$

$$\frac{(E \mid F \mid t_e.\mathbf{0}) \setminus (I \cup O_1 \cup O_2)/\{\sigma_o \mid o \in O_1 \cup O_2\} =_{\sigma_s} \mathbf{0}}{\sigma_s, \{\}, \{\} \vdash \mathbf{system}(S, C) : \mathbf{0}}$$

$$\sigma_n, I_1, O_1 \vdash C : E$$

$$\frac{(E \mid \mu X.\underline{r_e.\overline{r}.e.\overline{t_c}}_{\sigma_n}.\underline{\sigma_n}.X) \setminus (I_1 \cup O_1 \cup \{r_e, t_c\})/\{\sigma_o \mid o \in O_1\}/\sigma_n =_\sigma F}{\sigma, I_2, O_2 \vdash \mathbf{enc}(C, n, I_2, O_2) : F}$$

Table 8
Encapsulation Types

References

[1] H. Andersen and M. Mendler. A multi-clock asynchronous process algebra. In *Proc. 5th Euro. Symp. on Prog. (ESOP '94)*, LNCS 788, pages 58–73, 1994.

[2] G. Berry and G. Gonthier. The Esterel synchronous programming language. *Science of Computer Programming*, 19(2):87–152, 1992.

[3] M. Bravetti and R. Gorrieri. A complete axiomatization for observational congruence of prioritized finite-state behaviours. In *Proc. 27th Intl. Clqm. on Automata, Lang. and Prog. (ICALP 2000)*, LNCS 1853, pages 744–755, 2000.

[4] J. T. Buck and Edward A. Lee. *Advanced Topics in Dataflow Computing and Multi-threading*, chapter The Token Flow Model. IEEE Computer Society, 1993.

[5] R. Cleaveland, G. Lüttgen, and M. Mendler. An algebraic theory of multiple clocks. In *8th Intl. Conference on Concurreny Theory (CONCUR '97)*, LNCS 1243, pages 166–180. Springer-Verlag, 1997.

[6] Jon Conway and Steve Watts. *Software Engineering Approach to LabVIEW*. Prentice Hall, 2003.

[7] L. de Alfaro and T.A. Henzinger. Interface automata. In *Proc. 8th Euro. Soft. Eng. Conf. & 9th ACM Symp. on Foundations of Soft. Eng. (ESEC/FSE 2001)*, volume 26, 5 of *ACM Soft. Eng. Notes*, pages 109–120, 2001.

[8] M. Hennessy and T. Regan. A process algebra for timed systems. *Information and Computation*, 117(2):221–239, March 1995.

[9] H. Hermanns and M. Lohrey. Priority and maximal progress are completely axiomatisable. In *9th Intl. Conf. on Concurrency Theory (CONCUR'98)*, LNCS 1446, pages 237–252, 1998.

[10] E. Lee. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):223–235, April 1991.

[11] E. A. Lee and Y. Xiong. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing*, 16(1):210–237, Jan 2004.

[12] A. J. R. G. Milner and D. Sangiorgi. Techniques of weak bisimulation up-to. In *Proc. 3rd Intl. Conf. on Concurrency Theory (CONCUR'92)*, LNCS 630, 1992.

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

[14] X. Nicollin and J. Sifakis. The algebra of timed processes, ATP: Theory and application. *Information and Computation*, 114:131–178, 1994.

[15] B. Norton. A complete axiomatisation of observation congruence for deterministic time under maximal progress. Technical Report KMI-05-6, Open University, 2005.

[16] B. Norton and M. Fairtlough. Reactive types for dataflow-oriented software architectures. In *Proc. 4th IEEE/IFIP Conf. on Soft. Architecture (WICSA2004)*, volume P2172, pages 211–220. IEEE CS Press, 2004.

[17] B. Norton, G. Lüttgen, and M. Mendler. A compositional semantic theory for synchronous component-based design. In *14th Intl. Conference on Concurreny Theory (CONCUR '03)*, LNCS 2761, 2003.

[18] A. Sicheneder *et al.* Tool-supported design and program execution for signal processing applications using modular software components. In *Intl. Wrks. on Soft. Tools for Tech. Transfer STTT'98*. BRICS Tech. Rep. NS-98-4, 1998.