A Compositional Semantic Theory for Synchronous Component–based Design*

Barry Norton[1], Gerald Lüttgen[2], and Michael Mendler[3]

[1] Department of Computer Science, University of Sheffield, UK.
e–mail: b.norton@dcs.shef.ac.uk
[2] Department of Computer Science, University of York, UK.
e–mail: gerald.luettgen@cs.york.ac.uk
[3] Informatics Theory Group, University of Bamberg, Germany.
e–mail: michael.mendler@wiai.uni-bamberg.de

Abstract. Digital signal processing and control (DSPC) tools, such as LabView and iConnect, allow application developers to assemble systems by connecting predefined components in signal–flow graphs and by hierarchically building new components via encapsulating sub–graphs. Run–time environments then dynamically schedule components for execution on some embedded processor, typically in a synchronous cycle–based fashion, and check whether one component jams another by producing outputs faster than can be consumed. Currently, there do not exist formal models of DSPC schedulers that would enable compositional static verification of real–time constraints, such as jam–freeness.
This paper develops a process–algebraic coordination model for synchronous component–based design, which directly lends itself to compositionally formalising the monolithic semantics of DSPC tools. By uniformly combining the well–known concepts of abstract clocks, maximal progress and clock–hiding, it is shown how the DSPC principles of dynamic synchronous scheduling, isochrony and encapsulation may be captured faithfully and compositionally in process algebra, and how observation equivalence may facilitate compositional jam checks. These results provide a foundation for enhancing existing DSPC tools by allowing behavioural validations to be conducted automatically at compile–time.

Keywords: Coordination models, process algebra, digital signal processing, scheduling, jam analysis.

Correspondence: Gerald Lüttgen
Address: Dept. of Computer Science, The University of York
Heslington, York, YO10 5DD, UK
E–mail: gerald.luettgen@cs.york.ac.uk
Phone: +44 190 443–4774
Fax: +44 190 443–2767

* Research supported by EPSRC grant GR/M99637 and British Council grant PRO1163/CH.

B. Norton, G. Luettgen, M. Mendler: A compositional semantic theory for synchronous component-based design.
In R. Amadio, D. Lugiez (eds.), Int'l Conference on Concurrency Theory (CONCUR'03), Springer LNCS 2761, 2003, pp. 461-476.

1 Introduction

One important domain for embedded–systems designers are *digital signal processing and control applications* (DSPC). These involve dedicated software for control and monitoring problems in industrial production plants, or software embedded in engineering products. The underlying programming style within this domain relies on component–based design. Over many years, engineers have built rich repositories of pre–compiled and well–tested software components (PID–controllers, FIR–filters, FFT–transforms, etc.) that encapsulate technological know–how and hide design complexity behind clear interfaces. Applications can then be programmed efficiently by simply interconnecting components, which frees the application engineer from most of the error–prone low–level programming tasks. Design efficiency is further aided by the fact that DSPC programming tools, including LabView [15], iConnect [25] and Ptolemy [16], typically provide a graphical user interface that supports hierarchical abstraction. Hierarchical extensions of signal–flow graphs permit the *encapsulation* of sub–systems into single components, thus facilitating reuse of system designs.

While the visual signal–flow formalism facilitates mainly the structural design of DSPC applications, the overall behaviour of a component–based system manifests itself only once its components are scheduled and executed on an embedded processor. This *scheduling* is often handled dynamically by run–time environments, as is the case in LabView and iConnect, in order to achieve more efficient and adaptive real–time behaviour as well as some form of control flow. The scheduling typically follows a natural cycle–based synchronous execution model with the phases *collect input* (I), *compute reaction* (R) and *deliver output* (O). This IRO scheduling model is uniformly applied to composite signal–flow graphs as well as their individual components, which may themselves be built hierarchically from smaller entities (cf. Sec. 2). At the top level, the scheduler continuously iterates between executing the *source components* that produce new inputs, e.g., by reading sensor values, and one executing *computation components* that transform input values into output values, which are then delivered to the system environment, e.g., via actuators. Each phase obeys the *synchrony principle* [11], i.e., in (I) all source components are given a chance to collect input from the environment before any computation component is executed, in (R) every computation component whose inputs are available will be scheduled for execution, and in (O) all generated outputs will be delivered before the current cycle ends. The constraint in phase (O), which is known as *isochrony* [12, 18], implies that each output signal will be 'instantaneously' received at each connected input. This synchronous scheme can naturally be applied in a hierarchically nested fashion, abstracting a causal sequence of RO–steps produced by a sub–system into a single RO–step.

Like in synchronous programming, the implicit synchrony hypothesis of IRO scheduling assumes that the reaction of a (sub–)system is always faster than its environment issues execution requests. If a component cannot consume its input signals at the pace at which they arrive, a *jam* occurs [25]. In practice, jams usually indicate serious real–time problems (cf. Sec. 2). Unfortunately, in

B. Norton, G. Luettgen, M. Mendler: A compositional semantic theory for synchronous component-based design.
In R. Amadio, D. Lugiez (eds.), Int'l Conference on Concurrency Theory (CONCUR'03), Springer LNCS 2761, 2003, pp. 461-476.

existing tools, such as iConnect, there are neither compile–time nor run–time checks for detecting jams, thereby forcing engineers to rely on extensive simulations for validating their applications. Moreover, there is no formal model of IRO scheduling for DSPC programming systems that could be used for the compositional analysis of jams, and the question of how to distribute the monolithic IRO scheduler into a uniform coordination model has not been addressed in the literature. Such a model would be extremely useful given that a good deal of real–time validation of DSPC applications could be reduced to jam analysis, in the form of static verification of jam–freeness.

The objective of this paper is to show that a relatively small number of standard concepts studied in concurrency theory provides the key to naturally and *compositionally* formalising the semantics of component–based DSPC designs, and thus to enable static compositional jam checks. The most important concepts from the process–algebra tool–box are *handshake* synchronisation from CCS [19], and *abstract clocks* in combination with *maximal progress* as investigated in temporal process algebras [2], specifically TPL [13], PMC [1] and CSA [6]. We use handshake synchronisation for achieving serialisation, and atomicity and maximal progress clocks for reflecting synchrony. Finally, given maximal progress, synchronous encapsulation may be naturally captured in terms of *clock–hiding*, similar to hiding in CSP [14]. We will uniformly integrate all three concepts into a single process language (cf. Sec. 3), to which we refer as *Calculus for Synchrony and Encapsulation* (CaSE). This calculus conservatively extends CCS in being equipped with a behavioural theory based on observation equivalence [19].

As main contribution, we will formally establish that CaSE is expressive enough for faithfully modelling the principles of IRO scheduling and for reasoning about jams (cf. Sec. 4). First, using a single clock and maximal progress we will show how one may derive a decentralised description of the synchronous scheduler. Second, we prove that isochrony across connections can be modelled via multiple clocks and maximal progress. Third, the subsystems–as–components principle is captured by the clock–hiding operator. Moreover, we will argue that observation equivalence lends itself for statically detecting jams by reducing jam checking to timelock checking.

In the light of these results, our modelling in CaSE yields a *coordination model* for synchronous component–based design, whose virtue is its compositional style for specifying and reasoning about DSPC systems. In particular, our results disprove the perception of designers of DSPC tools that the presence of a global run–time environment and a centralised scheduler precludes the compositional, static capture of semantic properties of DSPC programs, including jam–freeness. Thus, CaSE provides a foundation for developing future–generation DSPC tools that offer the compositional, static analysis techniques desired by engineers.

2 An Example of DSPC Design

Our motivating example is a *digital spectrum analyser*, which is sketched in the signal–flow graph of Fig. 1. The task is to analyse an audio signal and continually

B. Norton, G. Luettgen, M. Mendler: A compositional semantic theory for synchronous component-based design.
In R. Amadio, D. Lugiez (eds.), Int'l Conference on Concurrency Theory (CONCUR'03), Springer LNCS 2761, 2003, pp. 461-476.

show an array of bar–graphs representing the intensity of the signal in disjoint sections of the frequency range. Our spectrum analyser is designed with help of components Soundcard, Const, Element and BarGraph. Each instance c1, c2, ... of Element, written as ck:Element or simply ck for $k = 1, 2, \ldots$, is responsible for displaying one bar–graph. ck:Element is connected to the single instance s0 of component Soundcard, s0:Soundcard, which generates the audio signal and provides exactly one audio value each time it is scheduled. It is also connected to instance sk:Const of component Const, which initialises ck:Element by providing filter parameters when it is first scheduled. In contrast to components Soundcard and Const, Element is not a basic but a hierarchical component. Indeed, every ck encapsulates one instance of Filter, Quantise and BarGraph, respectively, namely ck1:Filter, ck2:Quantise and ck3:BarGraph as shown in Fig. 2.

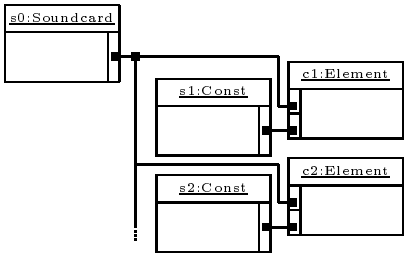

Fig. 1. Example Application

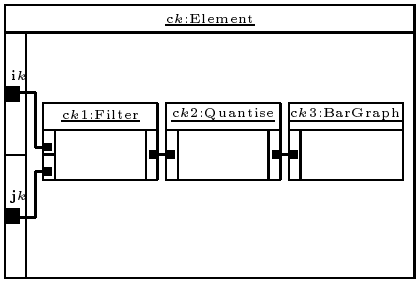

Fig. 2. An instance of Element

Scheduling. According to IRO scheduling, our example application will be serialised as follows within each IRO–cycle. First, each source component instance gets the chance to execute. In the first cycle, this will be s0:Soundcard and all sk:Const, which will be interleaved in some arbitrary order. In all subsequent cycles, only s0:Soundcard will request to be scheduled, since sk:Const can only produce a value once. Each produced value will be instantaneously propagated to each ck:Element, for all $k \geq 1$, according to the isochronic broadcast. The scheduler then switches to scheduling computation components. Since all necessary inputs of each ck are available in each IRO–cycle, every ck will request to be scheduled. The scheduler will serialise these requests, each ck will execute accordingly, and the current IRO–cycle ends as no outputs generated within ck are to be propagated to its environment. However, since each ck encapsulates further component instances, its execution is non–trivial and involves a sub–scheduler that will schedule ck1:Filter, ck2:Quantise and ck3:BarGraph in such a way that an RO–cycle of these instances will appear atomic to ck. This ensures that the scheduling of the inner ck1, ck2 and ck3 will be not be interleaved with executing any of the sibling instances cl, for $l \neq k$, of ck.

Isochronic output. Whenever s0:Soundcard is scheduled in our example system, it generates an audio signal whose value is propagated via a wire that forks to port ik of each instance ck:Element, for $k \geq 1$. In order for the array of bar–graphs to display a consistent state synchronous with the environment, all ck

must have received the new value from s0:Soundcard before any cl:Element may be scheduled. Thus, s0:Soundcard and all ck:Element, for $k \geq 1$, must synchronise to transmit sound values instantaneously. This form of synchronisation is called *isochrony* [12] in hardware, where it is the weakest known synchronisation principle from which non–trivial sequential behaviour can be implemented safely without internal real–time glitches [18].

Jams. Let us now consider what happens if instances s0:Soundcard and s1:Const are accidently connected the wrong way around, i.e., s0:Soundcard is connected to port j1 and s1:Const to port i1 of c1:Element. Recall that c11:Filter within c1:Element will only read a value, an initialisation value, from port j1 in the first IRO–cycle and never again afterwards. Thus, when the value of s0:Soundcard produced in the third cycle is propagated to port j1, the system *jams*. This is because the value that has been produced in the second IRO–cycle and stored at this port, has not yet been read by c11:Filter. Observe that a jam is different from a deadlock; indeed our example system does not deadlock since all instances of Element other than c1:Element continue to operate properly.

3 CaSE: Calculus for Synchrony and Encapsulation

This section presents our process calculus CaSE, which serves as a framework for deriving our formal coordination model for DSPC design in Sec. 4. CaSE is inspired by Hennessy and Regan's TPL [13], which is an extension of Milner's CCS [19] with regard to syntax and operational semantics. In addition to CCS, TPL includes (i) a *single abstract clock* σ that is interpreted not quantitatively as some number but qualitatively as a recurrent global synchronisation event; (ii) a *timeout operator* $\lfloor P \rfloor \sigma(Q)$ similar to ATP [20], where the occurrence of σ deactivates process P and activates Q; (iii) the concept of *maximal progress* [27] that implements the synchrony hypothesis by demanding that a clock can only tick within a process, if the process cannot engage in any internal activity τ.

CaSE further extends TPL by (i) allowing for *multiple clocks* $\sigma, \rho, \ldots$ as in PMC [1] and CSA [6] while, in contrast to PMC and CSA, maintaining the global interpretation of maximal progress; (ii) explicit *timelock* operators Δ and Δ_σ that prohibit the ticking of all clocks and of clock σ, respectively; (iii) *clock–hiding* operators P/σ that internalise all clock ticks of process P. Clock hiding is basically hiding as in CSP [14], i.e., hidden actions are made non–observable. In combination with maximal progress this has the important effect —so far unexplored in the process–algebra community— that all inner clock ticks become included within the synchronous cycle of an outer clock. This is the essence of synchronous encapsulation, as is required by the subsystems–as–components principle in DSPC design. Finally, in contrast to TPL and similar to CCS and CSA, we will equip CaSE with a bisimulation–based semantic theory [19].

Syntax and operational semantics. We let $\Lambda = \{a, b, \ldots\}$ be a countable set of *input actions* and $\overline{\Lambda} = \{\overline{a}, \overline{b}, \ldots\}$ be the set of complementing *output actions*. As in CCS [19], an action a communicates with its complement $\overline{a}$ to produce

B. Norton, G. Luettgen, M. Mendler: A compositional semantic theory for synchronous component-based design.
In R. Amadio, D. Lugiez (eds.), Int'l Conference on Concurrency Theory (CONCUR'03), Springer LNCS 2761, 2003, pp. 461-476.

the *internal action* τ. The symbol $\mathcal{A}$ denotes the set of all actions $\Lambda \cup \overline{\Lambda} \cup \{\tau\}$. Moreover, CaSE is parameterised in a set $\mathcal{T} = \{\sigma, \rho, \ldots\}$ of *abstract clocks*, or clocks for brief. The syntax of CaSE is defined by the following BNF:

$$P \ ::= \ \mathbf{0} \mid \Delta \mid \Delta_\sigma \mid x \mid \alpha.P \mid P+P \mid P|P \mid P \setminus L \mid P/\sigma \mid \lfloor P \rfloor \sigma(P) \mid \mu x.P \ ,$$

where x is a *variable* taken from some countably infinite set, and $L \subseteq \mathcal{A} \setminus \{\tau\}$ is a *restriction set*. Further, we use the standard definitions for *static* and *dynamic* operators, *free* and *bound* variables, *open* and *closed* terms, and *guarded* terms. We refer to closed and guarded terms as *processes*, collected in the set $\mathcal{P}$. For convenience, we write $\overline{L}$ for the set $\{\overline{a} \mid a \in L\}$ and extend the timeout operator to sequences of clocks by defining $\lfloor P \rfloor =_{\mathrm{df}} P$ and $\lfloor P \rfloor \sigma_1(Q_1) \ldots \sigma_n(Q_n) =_{\mathrm{df}} \lfloor \lfloor P \rfloor \sigma_1(Q_1) \ldots \sigma_{n-1}(Q_{n-1}) \rfloor \sigma_n(Q_n)$. Finally, if P contains actions $a_1, a_2, \ldots, a_n$ and the free variable x only, we write $x(a_1, a_2, \ldots, a_n) \stackrel{\mathrm{def}}{=} P$ for the process $\mu x.P$. Then, $x(b_1, b_2, \ldots, b_n)$ denotes the process $\mu x.P'$, where P' results from P by simultaneously substituting actions a_i by b_i, for $1 \leq i \leq n$.

Table 1. Operational semantics for CaSE

$$\text{Act} \ \frac{}{\alpha.P \stackrel{\alpha}{\to} P} \qquad \text{tAct} \ \frac{}{\alpha.P \stackrel{\sigma}{\to} \alpha.P} \ \alpha \neq \tau$$

$$\text{Sum1} \ \frac{P \stackrel{\alpha}{\to} P'}{P+Q \stackrel{\alpha}{\to} P'} \qquad \text{tNil} \ \frac{}{\mathbf{0} \stackrel{\sigma}{\to} \mathbf{0}} \qquad \text{tStall} \ \frac{}{\Delta_\sigma \stackrel{\rho}{\to} \Delta_\sigma} \ \sigma \neq \rho$$

$$\text{Sum2} \ \frac{Q \stackrel{\alpha}{\to} Q'}{P+Q \stackrel{\alpha}{\to} Q'} \qquad \text{tSum} \ \frac{P \stackrel{\sigma}{\to} P' \quad Q \stackrel{\sigma}{\to} Q'}{P+Q \stackrel{\sigma}{\to} P'+Q'}$$

$$\text{Res} \ \frac{P \stackrel{\alpha}{\to} P'}{P \setminus L \stackrel{\alpha}{\to} P' \setminus L} \ \alpha \notin L \cup \overline{L} \qquad \text{tRes} \ \frac{P \stackrel{\sigma}{\to} P'}{P \setminus L \stackrel{\sigma}{\to} P' \setminus L}$$

$$\text{Par1} \ \frac{P \stackrel{\alpha}{\to} P'}{P|Q \stackrel{\alpha}{\to} P'|Q} \qquad \text{tPar} \ \frac{P \stackrel{\sigma}{\to} P' \quad Q \stackrel{\sigma}{\to} Q'}{P|Q \stackrel{\sigma}{\to} P'|Q'} \ P|Q \stackrel{\tau}{\not\to}$$

$$\text{Par2} \ \frac{Q \stackrel{\alpha}{\to} Q'}{P|Q \stackrel{\alpha}{\to} P|Q'} \qquad \text{tHid1} \ \frac{P \stackrel{\sigma}{\to} P'}{P/\sigma \stackrel{\tau}{\to} P/\sigma}$$

$$\text{Par3} \ \frac{P \stackrel{a}{\to} P' \quad Q \stackrel{\overline{a}}{\to} Q'}{P|Q \stackrel{\tau}{\to} P'|Q'} \qquad \text{tHid2} \ \frac{P \stackrel{\rho}{\to} P'}{P/\sigma \stackrel{\rho}{\to} P'/\sigma} \ \sigma \neq \rho, \ P \stackrel{\sigma}{\not\to}$$

$$\text{Hid} \ \frac{P \stackrel{\alpha}{\to} P'}{P/\sigma \stackrel{\alpha}{\to} P'/\sigma} \qquad \text{tTO1} \ \frac{}{\lfloor P \rfloor \sigma(Q) \stackrel{\sigma}{\to} Q} \ P \stackrel{\tau}{\not\to}$$

$$\text{TO} \ \frac{P \stackrel{\alpha}{\to} P'}{\lfloor P \rfloor \sigma(Q) \stackrel{\alpha}{\to} P'} \qquad \text{tTO2} \ \frac{P \stackrel{\rho}{\to} P'}{\lfloor P \rfloor \sigma(Q) \stackrel{\rho}{\to} P'} \ \sigma \neq \rho$$

$$\text{Rec} \ \frac{P[\mu x.P/x] \stackrel{\alpha}{\to} P'}{\mu x.P \stackrel{\alpha}{\to} P'} \qquad \text{tRec} \ \frac{P[\mu x.P/x] \stackrel{\sigma}{\to} P'}{\mu x.P \stackrel{\sigma}{\to} P'}$$

The *operational semantics* of a CaSE process P is given by a labelled transition system $\langle \mathcal{P}, \mathcal{A} \cup \mathcal{T}, \longrightarrow, P \rangle$, where $\mathcal{P}$ is the set of states, $\mathcal{A} \cup \mathcal{T}$ the alphabet, $\longrightarrow$ the transition relation, and P the start state. We refer to transitions with labels in $\mathcal{A}$ as *action transitions* and to those with labels in $\mathcal{T}$ as *clock transitions*. The transition relation $\longrightarrow \subseteq \mathcal{P} \times (\mathcal{A} \cup \mathcal{T}) \times \mathcal{P}$ is defined in Table 1 using

B. Norton, G. Luettgen, M. Mendler: A compositional semantic theory for synchronous component-based design.
In R. Amadio, D. Lugiez (eds.), Int'l Conference on Concurrency Theory (CONCUR'03), Springer LNCS 2761, 2003, pp. 461-476.

operational rules. For the sake of simplicity, we also write γ for a representative of $\mathcal{A} \cup \mathcal{T}$, as well as $P \xrightarrow{\gamma} P'$ for $\langle P, \gamma, P' \rangle \in \longrightarrow$ and $P \xrightarrow{\gamma}$ for $\exists P' \in \mathcal{P}.\, P \xrightarrow{\gamma} P'$. Our semantics is set up such that it enjoys a couple of important properties, for all clocks $\sigma \in \mathcal{T}$: (i) *maximal progress*, i.e., $P \xrightarrow{\sigma}$ implies $P \not\xrightarrow{\tau}$; (ii) *time determinacy*, i.e., $P \xrightarrow{\sigma} P'$ and $P \xrightarrow{\sigma} P''$ implies $P' = P''$. It is time determinacy that distinguishes clock ticks from CSP broadcast communication.

Intuitively, the *nil* process $\mathbf{0}$ permits all clocks to tick, while the *timelock* operators Δ and Δ_σ prohibit the ticking of any clock and of all clocks except σ, respectively. Process $\alpha.P$ may engage in action α and then behave like P. If $\alpha \neq \tau$, it may also idle for each clock σ; otherwise, all clocks are stopped, thus respecting maximal progress. The *summation operator* $+$ denotes nondeterministic choice, i.e., process $P+Q$ may behave like P or Q. According to time determinacy, time has to proceed equally on both sides of summation, i.e., $P+Q$ can engage in a clock transition and thus delay the nondeterministic choice if and only if both P and Q can. Process $P|Q$ stands for the *parallel composition* of P and Q according to an interleaving semantics with synchronised communication on complementary actions resulting in the internal action τ. Again, time has to proceed equally on both sides of the operator, and the side condition of Rule (tPar) ensures maximal progress. The *restriction operator* $\backslash L$ prohibits the execution of actions in $L \cup \overline{L}$ and thus permits the scoping of actions. The *clock–hiding operator* $/\sigma$ within a process P/σ turns every tick of clock σ in P into the internal action τ. This not only hides clock σ but also pre–empts all other clocks ticking at the same states as σ, according to Rule (tHid2). Process $\lfloor P \rfloor \sigma(Q)$ behaves as process P, and it can perform a σ–transition to Q, provided P cannot engage in an internal action as is reflected in the side condition of Rule (tTO1). The timeout operator disappears as soon as P engages in some transition labelled differently from σ. Finally, $\mu x.\, P$ denotes *recursion*, i.e., $\mu x.\, P$ behaves as a distinguished solution of the equation $x = P$.

Our interpretation of prefixes $\alpha.P$ adopted above, for $\alpha \neq \tau$, is *relaxed* [13], i.e., we allow the process to idle on clock ticks. In the remainder, *insistent prefixes* $\underline{\alpha}.P$ [1], which do not allow clocks to tick, will prove convenient as well. These can be expressed in CaSE by $\underline{\alpha}.P =_{\mathrm{df}} \alpha.P + \Delta$. Similarly, one may define a prefix that only lets clocks not in T tick, for $T \in \mathcal{T}$, by $\underline{\alpha}_T.P =_{\mathrm{df}} \alpha.P + \Delta_T$, where $\Delta_T =_{\mathrm{df}} \sum_{\sigma \in T} \Delta_\sigma$. Finally, we abbreviate $\lfloor \Delta \rfloor \sigma(P)$ by $\underline{\sigma}.P$.

Temporal observation equivalence and congruence. This section equips CaSE with a bisimulation–based semantics [19]. For the purposes of this paper we will concentrate on *observation equivalence* and *congruence*. The straightforward adaptation of strong bisimulation to our calculus immediately leads to a behavioural congruence, as can easily be verified by inspecting the format of our operational rules and by applying well–known results for structured operational semantics with negative premises [26]. Observational equivalence is a notion of bisimulation in which any sequence of internal τ's may be skipped. For $\gamma \in \mathcal{A} \cup \mathcal{T}$ we define $\hat{\gamma} =_{\mathrm{df}} \epsilon$ if $\gamma = \tau$ and $\hat{\gamma} =_{\mathrm{df}} \gamma$, otherwise. Further, let $\stackrel{\epsilon}{\Rightarrow} =_{\mathrm{df}} \xrightarrow{\tau}{}^*$ and $P \stackrel{\gamma}{\Rightarrow} P'$ if there exist processes P'' and P''' such that $P \stackrel{\epsilon}{\Rightarrow} P'' \xrightarrow{\gamma} P''' \stackrel{\epsilon}{\Rightarrow} P'$. Carrying over weak bisimulation [19] to CaSE leads to the following definition.

Definition 1. *A symmetric relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a* temporal weak bisimulation *if $P \xrightarrow{\gamma} P'$ implies $\exists Q'.\, Q \stackrel{\hat{\gamma}}{\Rightarrow} Q'$ and $\langle P', Q' \rangle \in \mathcal{R}$, for every $\langle P, Q \rangle \in \mathcal{R}$ and $\gamma \in \mathcal{A} \cup \mathcal{T}$. We write $P \approx Q$ if $\langle P, Q \rangle \in \mathcal{R}$ for some temporal weak bisimulation $\mathcal{R}$.*

Temporal observation equivalence $\approx$ is compositional for all operators except summation and timeout. However, for proving compositionality regarding parallel composition and hiding, both of which are defined by operational rules involving negative side conditions, the following proposition is central.

Proposition 1. *If $P \approx Q$ and $P \xrightarrow{\sigma} P'$ then $\exists Q', Q''.\, Q \stackrel{\epsilon}{\Rightarrow} Q'' \xrightarrow{\sigma} \stackrel{\epsilon}{\Rightarrow} Q'$, $P \approx Q''$, $P' \approx Q'$ and $\{\gamma \in \mathcal{A} \cup \mathcal{T} \mid P \xrightarrow{\gamma}\} = \{\gamma \in \mathcal{A} \cup \mathcal{T} \mid Q'' \xrightarrow{\gamma}\}$.*

The validity of this proposition is due to the maximal–progress property in CaSE. This is also why we do not need to equip temporal observation equivalence with complex conditions on initial action sets, as is necessary in calculi incorporating a weaker notion of maximal progress [6]. As usual, observation equivalence is not compositional for the choice operators summation and timeout. To identify the largest equivalence contained in $\approx$, the summation fix of CCS is not sufficient. As in other work in temporal process algebras [27], the deterministic nature of clocks implies the following definition of *temporal observation congruence*.

Definition 2. *A symmetric relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a temporal observation congruence if for every $\langle P, Q \rangle \in \mathcal{R}$, $\alpha \in \mathcal{A}$ and $\sigma \in \mathcal{T}$:*

1. *$P \xrightarrow{\alpha} P'$ implies $\exists Q'.\, Q \stackrel{\alpha}{\Rightarrow} Q'$ and $P' \approx Q'$.*
2. *$P \xrightarrow{\sigma} P'$ implies $\exists Q'.\, Q \xrightarrow{\sigma} Q'$ and $\langle P', Q' \rangle \in \mathcal{R}$.*

We write $P \cong Q$ if $\langle P, Q \rangle \in \mathcal{R}$ for some temporal observational congruence $\mathcal{R}$.

As desired, we obtain the following result, whose proof is standard [6, 19].

Thm 1. *The equivalence $\cong$ is the largest congruence contained in $\approx$.*

It is not difficult to see that CaSE is a conservative extension of CCS [19]. Indeed, CCS can be identified in terms of syntax, operational semantics and bisimulation semantics as the sub–calculus of CaSE which is obtained by defining $\mathcal{T} = \emptyset$. For finite–state systems, temporal observational equivalence can be computed efficiently, using standard partition–refinement algorithms as implemented in existing verification tools, such as the CWB–NC [7].

4 A Synchronous Coordination Model with Encapsulation

This section introduces our coordination model for DSCP applications on the basis of our process calculus CaSE. Particular emphasis is given regarding the issues of component instantiation, synchronous scheduling, isochronic forks, and jam analysis. We illustrate our modelling using the digital–spectrum–analyser example introduced in Sec. 2.

We start off with attaching behavioural descriptions to basic components, which describe their interface behaviour to a scheduler. Informally, we say that the states of component descriptions must each belong to one of the classes '*input*', '*ready*', '*waiting*', '*internal*', '*output*' and '*finished*'. Input states may be non–deterministic and are source of transitions labelled from a set $I \subset \Lambda$ to either input states or ready states. Ready states must have one transition labelled $\overline{r}$, a request to execute, to a waiting state which must have one transition labelled g, which is a signal that the request is granted, leading to an internal state. Internal states again have non–deterministic transitions, but labelled only in the silent action τ, representing the progress and resulting state of the internal computation; that this may vary, being dependent on the values of the input data, is modelled by non–determinism. The destination of internal states must be output states from which a deterministic sequence $\overline{o_1}, \ldots, \overline{o_n}$ of outputs, i.e., members of the set $O \subset \Lambda$, is produced by transitions to output states or a finished state, at which point all output is delivered. The sequence $\vec{\overline{o}} = [\overline{o_1}, \ldots, \overline{o_n}]$ is chosen non–deterministically by the internal τ–computation that leads into the output region. Finished states have one transition labelled $\overline{g}$, signifying that the thread of control is being handed back, to either an input state or $\mathbf{0}$, meaning that the component is finished. Observe the scheduling sequence $r \cdots g \cdots \overline{g}$, in which the component requests execution by r, obtains the execution grant through g and finally signals completion with $\overline{g}$.

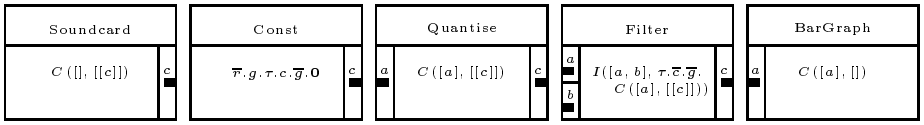

Fig. 3. **Fig. 4.** **Fig. 5.** **Fig. 6.** **Fig. 7.**

For any model, I, O and $\{r, g\}$ must be disjoint and in the following we use only a, b and decorations thereof to name input channels, and range over these with i, and c, d and decorations thereof to name outputs, ranged over by o. Having defined as usual the 'syntactic sort' of a process $\mathcal{S}(P) \subset \Lambda$, we define also the 'input sort' and 'output sort', $\mathcal{I}(P) \subset I, \mathcal{O}(P) \subset O$, in the natural way. Note that the request and grant ports $\{r, g\}$, which connect a component with its scheduling environment, are also part of the component sort. Moreover, we may use the following parameterised definitions to define the typical component behaviour that is consistent with the above interface description, such as those seen in Figs. 3–7 for our example application.

$$I(\vec{i}, P) \stackrel{\text{def}}{=} \begin{cases} \sum_{1 \le j \le n = |\vec{i}|} i_j \,.\, I([i_1, i_2, \cdots, i_{j-1}, i_{j+1}, \cdots a_n], P) & \text{if } |\vec{i}| > 0 \\ \overline{r} \,.\, g \,.\, P & \text{otherwise} \end{cases}$$

$$O(\vec{o}, P) \stackrel{\text{def}}{=} \begin{cases} \overline{o_1} \,.\, O([o_2, \cdots, o_{|\vec{o}|}], P) & \text{if } |\vec{o}| > 0 \\ \overline{g} \,.\, P & \text{otherwise} \end{cases}$$

$$C(\vec{i}, \vec{\overline{o}}) \stackrel{\text{def}}{=} I(\vec{i}, \sum_{1 \le j \le |\vec{\overline{o}}|} \tau \,.\, O(\vec{o}_j, C(\vec{i}, \vec{\overline{o}})))$$

B. Norton, G. Luettgen, M. Mendler: A compositional semantic theory for synchronous component-based design.
In R. Amadio, D. Lugiez (eds.), Int'l Conference on Concurrency Theory (CONCUR'03), Springer LNCS 2761, 2003, pp. 461-476.

First, $I(\vec{i}, P)$ defines an input phase in the vector of inputs $\vec{i}$ which continues, when scheduled, in the internal state P; $O(\vec{o}, P)$ defines an output phase with fixed sequence of outputs $\vec{o} = [\overline{o_1}, \ldots, \overline{o_n}]$, continuing in input state P having handed back the thread of control; $C(\vec{i}, \vec{\vec{o}})$ defines a typical cyclic component in its input state with input ports $\vec{i}$ and a choice of possible output sequences represented as a (finite) sequence of sequences $\vec{\vec{o}}$ of individual output actions. Each component $\vec{o}_j$, for $1 \leq j \leq |\vec{\vec{o}}|$, of $\vec{\vec{o}}$ may be an arbitrary (finite) sequence $\vec{o}_j = [\overline{o}_{j1}, \ldots, \overline{o}_{jn_j}]$ of output actions, $o_{ji} \in O$. In this way we can model arbitrary output behaviour. Each execution cycle of $C(\vec{i}, \vec{\vec{o}})$ involves all inputs from $\vec{i}$ and non–deterministic output behaviour defined by the members of $\vec{\vec{o}}$.

Returning to our example, observe that the CaSE process of Filter given in Fig. 5, $I(\{a, b\}, \tau.\overline{c}.\overline{g}.C([a], [[c]]))$, features the desired behaviour as described in Sec. 2. In particular, it reads from *both* inputs a, b only in the first scheduling cycle. In all subsequent ones, i.e., after passing once through $\overline{r}.g.\tau.\overline{c}.\overline{g}$, it behaves as the cyclic process $C([a], [[c]])$ that only ever consumes from input a.

4.1 Component instantiation. A pure component as introduced above uses the $\{r, g\}$ interface to negotiate its execution with its environment. From the point of view of the component, it does not matter whether it communicates with a centralised or a distributed scheduler. In this section, we develop a concept of *wrappers* for harnessing components with enough local control so they participate coherently in a global IRO scheduling scheme, without presence of a global scheduler. Indeed all wrappers added together represent a distributed version of an imagined central IRO scheduler. Wrapping a component is an instantiation into a communication *and* scheduling discipline, as in Ptolemy [16]. Our general model for a 'wrapper' is the following:

$$\begin{aligned} &\mathrm{InstWrapper}(\mathrm{SchedIdiom}, \mathrm{CommIdiom}, \sigma_s)[\_] \stackrel{\mathrm{def}}{=} \\ &\qquad (\ (\_ \mid \mathrm{SchedIdiom}(\sigma_s, \_)) \setminus \mathcal{O}(\_) \cup \{r, g\} \mid \\ &\qquad\quad \Pi_{o \in \mathcal{O}(\_)} \mathrm{CommIdiom}(o)\ ) \setminus \{b_o, f_o \mid o \in \mathcal{O}(\_)\}\ , \end{aligned}$$

where SchedIdiom and CommIdiom are scheduling and communication idiom, respectively. To implement IRO scheduling we consider two variants of SchedIdiom, namely CompInst and SourceInst to be discussed below.

Synchronous scheduling. Given a basic component C, we can instantiate it either as a computation component, using idiom $\mathrm{CompInst}(\sigma_s, C)$, or as a source component with $\mathrm{SourceInst}(\sigma_s, C)$, where σ_s represents the phase clock. This clock organises strict alternation between source and computation phases and, by way of maximal progress, implements run–to–completion within each phase. To achieve serialisation on a single thread of control, a token–passing style is used, where a component may only execute if it possesses the execution token and surrenders this when the execution is complete.

The token is passed on label t_c between computation components, and on label t_e between source components. Furthermore, each output event o is split systematically into a request–acknowledge pair $\overline{b}_o \,.\, f_o$, to prepare for isochronic

B. Norton, G. Luettgen, M. Mendler: A compositional semantic theory for synchronous component-based design.
In R. Amadio, D. Lugiez (eds.), Int'l Conference on Concurrency Theory (CONCUR'03), Springer LNCS 2761, 2003, pp. 461-476.

output distribution:

$$\begin{aligned}
\mathrm{CompInst}(\sigma_s, _) &\stackrel{\mathrm{def}}{=} r \,.\, (\overline{r_e} \,.\, t_c \,.\, \overline{g} \,.\, \mathrm{Inst}(t_c, t_e, \mathrm{CompInst}(\sigma_s, _), \sigma_s, _) + \\
&\qquad\qquad t_c \,.\, \overline{g} \,.\, \mathrm{Inst}(t_c, t_e, \mathrm{CompInst}(\sigma_s, _), \sigma_s, _))) \\
\mathrm{SourceInst}(\sigma_s, _) &\stackrel{\mathrm{def}}{=} \sigma_s \,.\, r \,.\, t_e \,.\, \overline{g} \,.\, \mathrm{Inst}(t_e, t_c, \mathrm{SourceInst}(\sigma_s, _), \sigma_s, _)) \\
\mathrm{Inst}(t_1, t_2, P, \sigma_s, _) &\stackrel{\mathrm{def}}{=} \sum_{o \in \mathcal{O}(_)} o \,.\, \overline{b_o} \,.\, f_o \,.\, \mathrm{Inst}(t_1, t_2, P, \sigma_s, _) + g \,.\, \lfloor \overline{t_1} \,.\, P \rfloor \sigma_s (\overline{t_2} \,.\, P)
\end{aligned}$$

The idiom $\mathrm{CompInst}(\sigma_s, C)$ will accept the request signal r from component C and wait for an execution token t_c from some other instantiated peer component. When there is no execution token present at the same hierarchy level, $\mathrm{CompInst}(\sigma_s, C)$ instead will issue a request $\overline{r}_e$ to its environment and then wait for t_c. Once t_c has arrived, $\mathrm{CompInst}(\sigma_s, C)$ will grant C its original request with signal g. The wrapper then behaves as $\mathrm{Inst}(t_c, t_e, \mathrm{CompInst}(\sigma_s, C), \sigma_s, C)$ while component C will execute. In this state, whenever C finishes and has data to distribute, the scheduling idiom passes on each output signal o to the communication idiom as b_o (understood as a 'broadcast o' signal) and waits for a signal f_o (understood as 'finished broadcasting o') in return. Once the component cycle is finished, signalled by a return of grant g, an attempt is made to pass on the token $t_1 = t_c$ to one of the peers. If, however, even in the presence of this offer, clock σ_s ticks thereby signalling the end of the phase, then the other token $t_2 = t_e$ is passed out, which will start the source phase. The source idiom $\mathrm{SourceInst}(\sigma_s, C)$ is analogous to $\mathrm{CompInst}(\sigma_s, C)$, except that t_e and t_c are interchanged. Also, since sources must execute at most once per cycle, their idiom $\mathrm{SourceInst}(\sigma_s, C)$ starts with an initial phase clock. In this way the clock has to tick between any two executions of the same source component.

Consider the Filter component of our example system of Fig. 2, whose behavioural definition in Fig. 5 we also abbreviate as Filter. According to our wrapper definition, the first step in instantiating this computation component is as follows:

$$\mathrm{Filter}' \stackrel{\mathrm{def}}{=} (\mathrm{Filter} \mid \mathrm{SchedIdiom}(\sigma_s, \mathrm{Filter})) \setminus \{c, r, g\}\,,$$

with the new 'external' interface sort consisting of $\{t_c, t_e, r_e, \sigma_s\}$ for the scheduling and $\{a, b, b_c, f_c\}$ for data input and output.

Isochronic Broadcast. There are two parts to form compositional isochronic forks in our coordination model. The first is a 'broadcast agent' $\mathrm{IsoBroad}(o)$, which has been referred to as $\mathrm{CommIdiom}(o)$ above:

$$\begin{aligned}
\mathrm{IsoBroad}(o) &\stackrel{\mathrm{def}}{=} \underline{b_o}_{\sigma_o} \,.\, \mathrm{IsoBroad}'(o) \\
\mathrm{IsoBroad}'(o) &\stackrel{\mathrm{def}}{=} \lfloor \overline{o} \,.\, \mathrm{IsoBroad}'(o) \rfloor \sigma_o (\overline{f_o} \,.\, \mathrm{IsoBroad}(o))
\end{aligned}$$

For each broadcast request b_o of the component wrapped, an arbitrary number of 'copies' of each signal will be communicated on $\overline{o}$ until the clock σ_o defining

the isochronous instant in which the communication occurs ticks and ends that instant. Because of maximal progress σ_o can only proceed when there are no further receivers listening on o. In this way the signal o obtains maximal distribution. Only when all receivers are saturated will $\overline{f}_o$ occur, thereby signalling the run–to–completion of the broadcast back to the component. Note that isochrony cannot be modelled faithfully in Hoare's CSP [14] or Prasad's CBS [23]. While the broadcasting primitive in CSP does not distinguish between senders and receivers and thus ignores the direction in which information is propagated, the one in CBS does not force receivers to synchronise with senders.

We can now complete our instantiation of the Filter component, extending Filter$'$ by bolting on one IsoBroad(c) for the output line c. This gives

$$\text{Filter}'' \stackrel{\text{def}}{=} (\text{Filter}' \mid \text{IsoBroad}(c)) \setminus \{b_c, f_c\}$$

which is the same as InstWrapper(CompInst, IsoBroad, σ_s)[Filter]. Note that the scheduling interface of Filter now is $\{t_c, t_e, r_e, \sigma_s, \sigma_c\}$, which is extended by the isochrony clock σ_c and for data signalling $\{a, b, c\}$. The output broadcast which was controlled in Filter$'$ by the pair b_c, f_c is now handled by the single line c *together* with the clock σ_c. The fact that control is now via a clock is what will make the wiring up of Filter$''$ with an arbitrary number of broadcast receivers compositional.

For the following, we assume that we have obtained computation component instances Quantise$''$ and BarGraph$''$ in a similar way. These can then be used to assemble the sub–systems Element, as well as source components Soundcard$''$, Const$''$ required for the example application of Fig.1. All these are InstWrapper instantiations of the corresponding basic components seen in Figs. 3–7.

4.2 Isochronic wiring. Now that we have instantiations of our components we need to wire them up. In our setting wires are specific agents that actively participate in broadcasts along them. Our 'wire agents'

$$\text{IsoWire}(o, \sigma_s, i) \stackrel{\text{def}}{=} o \,.\, \underline{\overline{i}}_{\{\sigma_s, \sigma_o\}} \,.\, \underline{\sigma_o} \,.\, \text{IsoWire}(o, \sigma_s, i)$$

connect a producer on output o with a consumer on input i. Whenever this agent IsoWire(o, σ_s, i) picks up a signal o it blocks clock σ_o until it has successfully delivered with $\overline{i}$. Then, σ_o must tick for the wire to cycle back and be ready again. In this way, a single transmission along the wire is sandwiched between o and σ_o. Since σ_o is a global event, we can compositionally join together and synchronise an arbitrary number of parallel wire transmissions. To formalise this, we introduce *closed forks*, defined in the CCS sub–calculus of CaSE:

$$\text{Fork}(o, \vec{i}) \stackrel{\text{def}}{=} b_o \,.\, \text{Fork}'(o, \vec{i}, \vec{i})$$

$$\text{Fork}'(o, \vec{i}, \vec{j}) \stackrel{\text{def}}{=} \begin{cases} \sum_{l \le n = |\vec{j}|} \overline{j_l} \,.\, \text{Fork}'(o, \vec{i}, [j_1, j_2, \cdots, j_{l-1}, j_{l+1}, \cdots j_n]) & \text{if } |\vec{j}| > 1 \\ \overline{j_1} \,.\, \overline{f_o} \,.\, \text{Fork}(o, \vec{i}) & \text{if } |\vec{j}| = 1 \end{cases}$$

The following theorem shows that our isochronic forks behave equivalently to closed forks, for any number of processes attached to the isochronic broadcast.

B. Norton, G. Luettgen, M. Mendler: A compositional semantic theory for synchronous component-based design.
In R. Amadio, D. Lugiez (eds.), Int'l Conference on Concurrency Theory (CONCUR'03), Springer LNCS 2761, 2003, pp. 461-476.

Thm 2. $(IsoBroad(o) \,|\, \Pi_{j \leq |\vec{i}|} IsoWire(o, \sigma_s, i_j))/\sigma_o \,|\, \Delta_{\sigma_s} \cong Fork(o, \vec{i}) \,|\, \Delta_{\{\sigma_s, \sigma_o\}}$

Note that in our modelling, the synchronous phase clock σ_s is admitted only in the initial state, i.e., as long as the wire is 'empty'. The wire's ability to pass on its previous value and thus to become empty, however, depends on the input–readiness of the component instances to which $\underline{i}$ connects forward. If any such instance is not ready, clock σ_s will never again tick, turning this local 'jam' condition into a timing flaw, which is a global condition.

Returning to our example, we could build an instantiated sub–system for Element, which overall behaves like a computation component, as follows:

$$\begin{aligned} \text{Subsystem}'' \stackrel{\text{def}}{=} (&\text{Filter}''(c, d, o_1) \mid \text{Isowire}(o_1, \sigma_s, i_1) \mid \text{Quantise}''(i_1, o_2) \mid \\ &\text{Isowire}(o_2, \sigma_s, i_2) \mid \text{BarGraph}''(i_2) \\ &) \setminus \{o_1, i_1, o_2, i_2\}/\{\sigma_{o_1}, \sigma_{o_2}\}\,, \end{aligned}$$

where c, d, o_1, o_2, i_1, i_2 are some arbitrarily chosen names for the input and output ports of the components. Note that we have restricted away the internal channels $\{o_1, i_1, o_2, i_2\}$ and hidden the internal isochrony clocks $\{\sigma_{o_1}, \sigma_{o_2}\}$. For the top–level system of Fig. 1 a similar composition, say Application$''$, can be formed from one source instance Soundcard$''$ and a suitable number of source instances Const$''$ and component instances Element$''$. We discuss next how one can obtain Element$''$ from a composite sub–system such as Subsystem$''$ by encapsulation.

4.3 Encapsulation. To encapsulate a signal–flow graph such as Subsystem$''$ as a basic component in itself, we finally define another 'wrapper', which is in some sense the inverse of the instantiation wrapper:

$$\begin{aligned} \text{EncWrapper}(\sigma_s)[\_] &\stackrel{\text{def}}{=} (\_ \mid \text{Enc}(\sigma_s)) \setminus \{t_c, t_e, r_e\}/\sigma_s \\ \text{Enc}(\sigma_s) &\stackrel{\text{def}}{=} \underline{r_e \,.\, \overline{r} \,.\, g \,.\, \overline{t_c}}_{\sigma_s} \,.\, t_e \,.\, \underline{\overline{g}}_{\sigma_s} \,.\, \text{Enc} \quad . \end{aligned}$$

The EncWrapper translates back the scheduling interface $\{t_c, t_e, r_e, \sigma_s\}$ into $\{r, g\}$, which is the interface of a pure component, while keeping the signal input and output ports intact. In the parallel composition Subsystem$''$ | Enc(σ_s), the process Enc(σ_s) picks up any request for token r_e from the computation components inside Subsystem$''$, passes it out as a request r, then waits for the grant signal g, upon which gives the execution token down into Subsystem$''$ via $\overline{t_c}$. At that point, it waits patiently for signal t_e from Subsystem$''$, which indicates that Subsystem$''$ has finished one phase cycle. Then, Enc(σ_s) finishes off by handing out the $\overline{g}$ signal to its environment, whence signalling completion of one computation run. Thus, seen from the outside, Subsystem$''$ | Enc(σ_s) behaves like a basic component, while Enc(σ_s) emulates a token–passing environment to the inner Subsystem$''$.

EncWrapper is obtained from Subsystem$''$ | Enc(σ_s) by restricting away all token passing signals $\{t_c, t_e, r_e\}$ and hiding the internal synchronous phase clock σ_s. All ticks of σ_s are turned into τ's, which from the point of view of the environment now count as proper internal computation steps. The resulting encap-

sulated system Element $\stackrel{\text{def}}{=}$ EncWrapper(σ_s)[Subsystem$''$] is a fresh basic component. It may be instantiated again as a computation component using InstWrapper to give Element$''$ $\stackrel{\text{def}}{=}$ InstWrapper(CompInst, IsoBroad, σ_s)[Element], which may be assembled into the complete system Application$''$, as suggested above.

Note that a signal–flow graph to be encapsulated may only sensibly be made up of instances of computation components and the new component will be a computation component in any inputs not restricted away. The values communicated by all wires should also first be restricted, as should the values offered by broadcast outputs and the clocks that bound them should be hidden. In order to form an output from the new component, a wire should be connected to supply the value at a desired port whose name is not then restricted. Bearing this in mind, we can now formally state the desired encapsulation property, namely that the encapsulation of a component instance should behave equivalently to the original component.

Thm 3. *Let Comp be an arbitrary component. Then*

$$\begin{aligned} &EncWrapper(\sigma_s)[\ (InstWrapper(CompInst, IsoBroad, \sigma_s)[Comp] \\ &\qquad\qquad \mid \Pi_{o\in\mathcal{O}(Comp)} IsoWire(o, \sigma_s, o))/\vec{\sigma}\ ] \\ &\quad \approxeq\ Comp \mid \Delta_{\{\sigma_s\}\cup\vec{\sigma}}\,, \end{aligned}$$

where $\vec{\sigma}$ consists of all isochronic clocks σ_o, for $o \in \mathcal{O}(Comp)$.

4.4 Jams analysis. As suggested above, a jam is said to occur when a 'wire' is unable to pass on the value it is holding. In our model this produces a path to a state where $\overline{i}\,.\,P + \Delta_{\{\sigma_s,\sigma_{o_{\mathsf{name}}}\}}$ is a *timelocked* parallel component in the global state. Since the isochronic broadcast agent cannot complete until its clock ticks, it will not signal completion and so the relevant instantiation wrapper will not release the token and the system is deadlocked. Thus, a jam manifests itself as a timelock within our compositional coordination model. This is made explicit in the following theorem.

Thm 4. *If System possesses only τ– and σ_s–transitions and no infinite τ–computations, then the following holds, where Check $\stackrel{def}{=} \mu x.\lfloor\Delta\rfloor\sigma_s(x)$:*

$$System \approx Check \quad iff \quad \nexists s \in \{\tau,\sigma_s\}^*.\, System \stackrel{s}{\rightarrow} P \stackrel{\sigma_s}{\not\Longrightarrow}\,.$$

We conclude this section with two observations on how time–lock and thus jams may be escaped. Further parallel composition of a jammed system may lead to the ability to escape the insistent states causing the jams via communication, which is exactly how an engineer is supposed to resolve the problem. Alternative, the scheduling clock may be hidden, but this may only be done in our coordination model via encapsulation. It is undesirable for jams to be hidden by encapsulation, as they are design faults that should be revealed. Indeed, in our planned integration of our coordination model within a type system for DSPC programming environments, which will be expanded upon in the next section, encapsulating faulty components will be disallowed.

B. Norton, G. Luettgen, M. Mendler: A compositional semantic theory for synchronous component-based design.
In R. Amadio, D. Lugiez (eds.), Int'l Conference on Concurrency Theory (CONCUR'03), Springer LNCS 2761, 2003, pp. 461-476.

5 Related Work

To the best of our knowledge, our coordination model is the first formal model of the synchronous and hierarchical scheduling discipline behind DSPC programming systems. Our process–algebraic approach complements existing work in *distributed object–oriented systems* [21] and in *architectural description languages* [3]. There, the focus is on distributed software rather than embedded centralised systems, and consequently on asynchronous rather than synchronous component behaviour. However, in both application domains, object–oriented systems and software architectures, formalisms have been investigated for describing and reasoning about the interface behaviour of components, too.

In object–oriented systems, simple automata–based frameworks have been studied, where finite automata model the life–cycle of objects [21]. Within these frameworks, one may then reason at compile–time whether each invocation of an object's method at run–time is permissable. This semantic analysis is different from jam analysis in DSPC applications, but similar to the *compatibility analysis* of interface automata employed in Ptolemy [9], which we will discuss below. A process–algebraic model of this theory for object–oriented design has been developed as well and is presented in [24]. In architectural description languages, the formalism of process algebra has been studied by Bernardo et al. [3]. Their approach rests on the use of CSP–style broadcast communication together with asynchronous parallel composition. Like in our application domain of DSPC design, the intention is to identify communication problems, but these are diagnosed in terms of deadlock behaviour [4].

As illustrated earlier, deadlock is a more specific property than the jam property investigated by us: a jam in one component jams the whole system, but a deadlock in one component does not necessary result in a system deadlock. To observe 'local' deadlocks in single components, a theory of *location equivalence* [5] has been developed in the literature, which refines Milner's theory of observation equivalence [19] and observes the location, or system component, from which an action is performed. Observation equivalence is unnecessarily expressive and complicated for the purposes of static jam analysis. Indeed, we have shown that there is no need to refer to locations when analysing jams, for which one may simply check via temporal observation equivalence.

From a practical point of view, we envision our coordination model based on the process calculus CaSE to play the role of a *reactive–types* language [22]. This would enable designers to specify the intended interactions between a given component and its environment as a type, and permit tool implementations to reduce type checking to temporal observation–equivalence checking. This idea is somewhat similar to the one of *behavioural types* in the Ptolemy community [17]. Behavioural types in Ptolemy are based on the formalism of *interface automata* [8] and employed for checking the so–called *compatibility* property between components [9]. However, interface automata are not expressive enough to reason about jams, which Ptolemy, for the restricted class of synchronous data–flow (SDF) models, handles by linear–algebra techniques. In contrast, CaSE's semantic theory is more general than SDF and lends itself to compositionally

B. Norton, G. Luettgen, M. Mendler: A compositional semantic theory for synchronous component-based design.
In R. Amadio, D. Lugiez (eds.), Int'l Conference on Concurrency Theory (CONCUR'03), Springer LNCS 2761, 2003, pp. 461-476.

checking jams at compile–time. Note in this context that our proposed notion of reactive type is different from the one studied for the synchronous language Signal [10]. Behavioural types in Signal are defined in terms of Signal's clock calculus, which focuses on data–flow rather than signal–flow.

6 Conclusions and Future Work

This paper presented a novel compositional coordination model for the synchronous component–based design of and reasoning about DSPC applications, thus complementing work in distributed object–oriented systems and architectural description languages. Our coordination model benefited from several ideas that have been investigated in the field of concurrency theory. In particular, we demonstrated that the semantic concepts underlying the IRO principle of DSPC tools, namely dynamic synchronous scheduling, isochrony and encapsulation, can be captured by uniformly combining the process–algebraic concepts of abstract clocks, maximal progress and clock hiding. To do so, we defined the process calculus CaSE and developed its behavioural theory based on temporal observation equivalence. This equivalence was then used to prove that clocks and maximal progress are indeed sufficient for compositionally describing DSP schedulers, including the isochronic propagation of output signals, and that clock hiding reflects the encapsulation process in hierarchical design. In addition, CaSE facilitates the static, compositional reasoning about jams in DSPC applications, via observation–equivalence checks against timelocked processes.

Future work should proceed along two orthogonal directions. First, CaSE should be integrated in DSPC tools in the form of a reactive–types system. Second, the semantic theory of CaSE should be completed by providing an axiomatisation of temporal observation congruence for regular processes.

References

1. H.R. Andersen and M. Mendler. An asynchronous process algebra with multiple clocks. In *ESOP '94*, vol. 788 of *LNCS*, pp. 58–73, 1994.
2. J.C.M. Baeten and C.A. Middelburg. *Process Algebra with Timing*. EATCS Monograph. Springer-Verlag, 2002.
3. M. Bernardo, P. Ciancarini, and L. Donatiello. On the formalisation of architectural types with process algebras. In *MASCOTS 2000*, pp. 140–148. ACM Press, 2000.
4. M. Bernardo, P. Ciancarini, and L. Donatiello. Detecting architectural mismatches in process algebraic descriptions of software systems. In *WICSA 2001*, pp. 77–86. IEEE Comp. Soc. Press, 2001.
5. G. Boudol, I. Castellani, M. Hennessy, and A. Kiehn. Observing localities. *TCS*, 114(1):31–61, 1993.
6. R. Cleaveland, G. Lüttgen, and M. Mendler. An algebraic theory of multiple clocks. In *CONCUR '97*, vol. 1243 of *LNCS*, pp. 166–180, 1997.
7. R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In *CAV '96*, vol. 1102 of *LNCS*, pp. 394–397, 1996.

B. Norton, G. Luettgen, M. Mendler: A compositional semantic theory for synchronous component-based design.
In R. Amadio, D. Lugiez (eds.), Int'l Conference on Concurrency Theory (CONCUR'03), Springer LNCS 2761, 2003, pp. 461-476.

8. L. de Alfaro and T.A. Henzinger. Interface automata. In *ESEC/FSE 2001*, vol. 26, 5 of *Softw. Eng. Notes*, pp. 109–120. ACM Press, 2001.
9. A. Chakrabarti et al. Interface compatibility checking for software modules. In *CAV 2002*, vol. 2404 of *LNCS*, pp. 428–441, 2002.
10. P. le Guernic, T. Gautier, M. le Borgne, and C. le Maire. Programming real-time applications with Signal. *Proc. of the IEEE*, 79(9):1305–1320, 1991.
11. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1992.
12. S. Hauck. Asynchronous design methodologies: An overview. *Proc. of the IEEE*, 83(1):69–93, 1995.
13. M. Hennessy and T. Regan. A process algebra for timed systems. *Inform. and Comp.*, 117:221–239, 1995.
14. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
15. G.W. Johnson and R. Jennings. *LabVIEW Graphical Programming*. McGraw-Hill, 2001.
16. E.A. Lee. Overview of the Ptolemy project. Techn. Rep. UCB/ERL M01/11, Univ. of California at Berkeley, 2001.
17. E.A. Lee and Y. Xiong. Behavioral types for component-based design. Techn. Rep. UCB/ERL M02/29, Univ. of California at Berkeley, 2002.
18. A.J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *6th MIT Conf. on Advanced Research in VLSI*, pp. 263–287. MIT Press, 1990.
19. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
20. X. Nicollin and J. Sifakis. The algebra of timed processes, ATP: Theory and application. *Inform. and Comp.*, 114:131–178, 1994.
21. O. Nierstrasz. Regular types for active objects. In *OOPSLA '93*, ACM SIGPLAN Notices, pp. 1–15, 1993.
22. B. Norton. Reactive types for dataflow-oriented component-based development. In *AVoCS 2002*, vol. CSR-02-6 of *Techn. Report Series*. Univ. of Birmingham, 2002.
23. K.V.S. Prasad. Programming with broadcasts. In *CONCUR 93*, vol. 715 of *LNCS*, pp. 173–187, 1993.
24. F. Puntigam. Type specifications with processes. In *FORTE '95*, vol. 43 of *IFIP Conf. Proc.* Chapman & Hall, 1995.
25. A. Sicheneder et al. Tool-supported software design and program execution for signal processing applications using modular software components. In *STTT '98*, BRICS Notes Series NS-98-4, pp. 61–70, 1998.
26. C. Verhoef. A congruence theorem for structured operational semantics with predicates and negative premises. *Nordic J. of Computing*, 2(2):274–302, 1995.
27. W. Yi. CCS + time = An interleaving model for real time systems. In *ICALP '91*, vol. 510 of *LNCS*, pp. 217–228, 1991.

B. Norton, G. Luettgen, M. Mendler: A compositional semantic theory for synchronous component-based design.
In R. Amadio, D. Lugiez (eds.), Int'l Conference on Concurrency Theory (CONCUR'03), Springer LNCS 2761, 2003, pp. 461-476.

APPENDIX: PROOF SKETCHES

Given the space constraints, we can only sketch the proofs of our main theorems. The full proofs will be made available in a technical report.

For the proofs of our main theorems of Sec. 4, it will be convenient to first formally specify the class of processes, typified by $Comp$, to which our allowable components will belong.

$$\begin{aligned}
Comp \stackrel{\text{def}}{=} \{P \mid\ & \forall Q, s \in \mathcal{A}^* \cdot (P \stackrel{s}{\rightarrow} Q) \Rightarrow \\
& \exists R \cdot Q \cong R \in (Inputs(P) \cup Ready(P) \cup Waiting(P) \cup \\
& \qquad Internal(P) \cup Outputs(P) \cup Finished(P))
\end{aligned}$$

$$\begin{aligned}
Inputs(P) \stackrel{\text{def}}{=} \{Q \mid\ & Q \in Deriv(P) \wedge \exists i \in I \cdot Q \stackrel{i}{\rightarrow} \wedge \\
& \forall \alpha \in \mathcal{A}, R \in \mathcal{P} \cdot Q \stackrel{\alpha}{\rightarrow} R) \Rightarrow \alpha \in I \wedge \\
& \exists S \in \mathcal{P} \cdot R \cong S \in (Inputs(P) \cup Ready(P))\}
\end{aligned}$$

$$\begin{aligned}
Ready(P) \stackrel{\text{def}}{=} \{Q \mid\ & Q \in Deriv(P) \wedge Q \stackrel{\overline{r}}{\rightarrow} \wedge \\
& \forall \alpha \in, R, R' \in \mathcal{P} \cdot (Q \stackrel{\alpha}{\rightarrow} R \wedge Q \stackrel{\alpha}{\rightarrow} R') \Rightarrow \alpha = \overline{r} \wedge \\
& \exists S \in \mathcal{P} \cdot R \cong R' \cong S \in Waiting(P)\}
\end{aligned}$$

$$\begin{aligned}
Waiting(P) \stackrel{\text{def}}{=} \{Q \mid\ & Q \in Deriv(P) \wedge Q \stackrel{g}{\rightarrow} \wedge \\
& \forall \alpha \in \mathcal{A}, R, R' \in \mathcal{P} \cdot (Q \stackrel{\alpha}{\rightarrow} R \wedge Q \stackrel{\alpha}{\rightarrow} R') \Rightarrow \alpha = g \wedge \\
& \exists S \in \mathcal{P} \cdot R \cong R' \cong S \in Internal(P)\}
\end{aligned}$$

$$\begin{aligned}
Internal(P) \stackrel{\text{def}}{=} \{Q \mid\ & Q \in Deriv(P) \wedge Q \stackrel{\tau}{\rightarrow} \wedge \\
& \forall \alpha \in \mathcal{A}, R \in \mathcal{P} \cdot (Q \stackrel{\alpha}{\rightarrow} R) \Rightarrow \alpha = \tau \wedge \\
& \exists S \in \mathcal{P} \cdot R \cong S \in Outputs(P)\}
\end{aligned}$$

$$\begin{aligned}
Outputs(P) \stackrel{\text{def}}{=} \{Q \mid\ & Q \in Deriv(P) \wedge \exists o \in O \cdot Q \stackrel{o}{\rightarrow} \wedge \\
& \forall \alpha, \alpha' \in \mathcal{A}, R, R' \in \mathcal{P} \cdot (Q \stackrel{\alpha}{\rightarrow} R \wedge Q \stackrel{\alpha'}{\rightarrow} R') \Rightarrow \alpha = \alpha' \wedge \\
& \exists S \cdot R \cong R' \cong S \in (Outputs(P) \cup Finished(P)\}
\end{aligned}$$

$$\begin{aligned}
Finished(P) \stackrel{\text{def}}{=} \{Q \mid\ & Q \in Deriv(P) \wedge Q \stackrel{\overline{g}}{\rightarrow} \wedge \\
& \forall \alpha \in \mathcal{A}, R, R' \in \mathcal{P} \cdot (Q \stackrel{\alpha}{\rightarrow} R \wedge Q \stackrel{\alpha}{\rightarrow} R') \Rightarrow \alpha = \overline{g} \wedge \\
& \exists S \cdot R \cong R' \cong S \in (Inputs(P) \cup \mathbf{0})\}
\end{aligned}$$

Where we define each of the allowable classes of states using the following relation, intended to find the first representative for each equivalence class:

$$\begin{aligned}
Deriv(P) \stackrel{\text{def}}{=} \{Q \mid\ & \exists s \in \mathcal{A}^* \cdot P \stackrel{s}{\rightarrow} Q \wedge \\
& \nexists t, u \in \mathcal{A}^+, Q \in \mathcal{P} \cdot t : u = s \wedge P \stackrel{t}{\rightarrow} Q' \wedge Q \cong Q'\}
\end{aligned}$$

We furthermore observe that, due to the statement that all components behaviours are finite state, each of these above sets will be finite for an acceptable description of component behaviour. We also remind ourselves that the

behaviours must be expressed as CCS terms, so we may infer loops of all clocks at every state not a member of $Internal(\text{Comp})$ for a given Comp.

Proof Sketch for Theorem 3 ('Encapsulation Transparency')

Proof. We first define:

$$\begin{aligned}
\mathcal{O} &\stackrel{\text{def}}{=} \mathcal{O}(Comp)\\
\text{Broads} &\stackrel{\text{def}}{=} \Pi_{o\in\mathcal{O}}\text{IsoBroad}(o_{\mathsf{inst}})\\
\text{Wires} &\stackrel{\text{def}}{=} \Pi_{o\in\mathcal{O}}\text{IsoWire}(o_{\mathsf{inst}},\sigma_s,o)\\
P^* &\stackrel{\text{def}}{=} P \mid \Delta_{\{\sigma_s\}\cup\vec{\sigma}}\\
&\quad \vec{\sigma} \text{ as a vector of clocks such that } \forall o\in\mathcal{O}(Comp)\cdot\exists j\cdot\sigma_j=\sigma_{o_{\mathsf{inst}}}\\
&\qquad\qquad \text{and } \nexists j\cdot\sigma_j\notin\mathcal{O}(\text{Comp})
\end{aligned}$$

We allow ourselves to drop implicit parameters, e.g.:

$$\text{InstWrapper}[\_] \stackrel{\text{def}}{=} \text{InstWrapper}(\text{CompInst},\text{IsoBroad},\sigma_s)[\text{Comp}]$$

Then one may re–state the theorem as:

$$\text{EncWrapper}[(\text{InstWrapper}[\text{Comp}] \mid \text{Wires})/\vec{\sigma} \approxeq \text{Comp}^*$$

To show this we construct first a (weak) bisimulation:
$\mathcal{R} \stackrel{\text{def}}{=} \mathcal{R}_{Input}\cup\mathcal{R}_{Ready}\cup\mathcal{R}_{Waiting}\cup\mathcal{R}_{Internal}\cup\mathcal{R}_{Output}\cup\mathcal{R}_{Finished}$, where:

$$\begin{aligned}
\mathcal{R}_{Input} \stackrel{\text{def}}{=}\ & \{< \text{EncWrapper}[(\text{InstWrapper}[P] \mid \text{Wires})/\vec{\sigma}]\ ,\ P^* > \mid\\
&\qquad P\in Inputs(\text{Comp})\}\\
\mathcal{R}_{Ready} \stackrel{\text{def}}{=}\ & \{< \text{EncWrapper}[(\text{InstWrapper}[P] \mid \text{Wires})/\vec{\sigma}],\quad P^* >,\\
& \quad < \text{EncWrapper}[((Q \mid r_e\,.\,t_c\,.\,\overline{g}\,.\,\text{Inst} + t_c\,.\,\overline{g}\,.\,\text{Inst} \mid\\
&\qquad \text{Broads})\setminus\mathcal{O}\cup\{r,g\} \mid \text{Wires})/\vec{\sigma}], P^* >,\\
& \quad < (((Q \mid t_c\,.\,\overline{g}\,.\,\text{Inst} \mid \text{Broads})\setminus\mathcal{O}\cup\{r,g\} \mid \text{Wires})/\vec{\sigma}) \mid\\
&\qquad \underline{\overline{r}\,.\,g\,.\,\overline{t_c}}_{\sigma_s}\,.\,t_e\,.\,\underline{\overline{g}}_{\sigma_s}\,.\,\text{Enc})\setminus\{t_c,r_e\}/\sigma_s\,, P^* > \mid\\
&\qquad P\in Ready(\text{Comp}), P\stackrel{\overline{r}}{\to}Q\}\\
\mathcal{R}_{Waiting} \stackrel{\text{def}}{=}\ & \{< (((P \mid t_c\,.\,\overline{g}\,.\,\text{Inst} \mid \text{Broads})\setminus\mathcal{O}\cup\{r,g\} \mid \text{Wires})/\vec{\sigma}) \mid\\
& \underline{g\,.\,\overline{t_c}}_{\sigma_s}\,.\,t_e\,.\,\underline{\overline{g}}_{\sigma_s}\,.\,\text{Enc})\setminus\{t_c,r_e\}/\sigma_s, P^* > \mid P\in Waiting(\text{Comp})\}\\
\mathcal{R}_{Internal} \stackrel{\text{def}}{=}\ & \{< (((P \mid t_c\,.\,\overline{g}\,.\,\text{Inst} \mid \text{Broads})\setminus\mathcal{O}\cup\{r,g\} \mid \text{Wires})/\vec{\sigma}) \mid\\
&\qquad \underline{\overline{t_c}}_{\sigma_s}\,.\,t_e\,.\,\underline{\overline{g}}_{\sigma_s}\,.\,\text{Enc})\setminus\{t_c,r_e\}/\sigma_s\,,\ Q^* >,\\
& \quad < (((P \mid \overline{g}\,.\,\text{Inst} \mid \text{Broads})\setminus\mathcal{O}\cup\{r,g\} \mid \text{Wires})/\vec{\sigma})\\
&\qquad \mid t_e\,.\,\underline{\overline{g}}_{\sigma_s}\,.\,\text{Enc})\setminus\{t_c,r_e\}/\sigma_s\,,\ Q^* >,\\
& \quad < (((Q \mid \text{Inst} \mid \text{Broads})\setminus\mathcal{O}\cup\{r,g\} \mid \text{Wires})/\vec{\sigma})\\
&\qquad \mid t_e\,.\,\underline{\overline{g}}_{\sigma_s}\,.\,\text{Enc})\setminus\{t_c,r_e\}/\sigma_s\,,\ Q^* > \mid\\
&\qquad\qquad P\in Waiting(\text{Comp}), P\stackrel{g}{\to}Q\}
\end{aligned}$$

$$
\begin{aligned}
\mathcal{R}_{Outputs} \stackrel{\text{def}}{=} \{ &< (((P \mid \text{Inst} \mid \text{Broads}) \setminus \mathcal{O} \cup \{r,g\} \mid \text{Wires})/\vec{\sigma}) \\
&\qquad \mid t_e \,.\, \underline{\overline{g}}_{\sigma_s} \,.\, \text{Enc}) \setminus \{t_c, r_e\}/\sigma_s \,,\ P^* >, \\
&< (((Q \mid \overline{b_c} \,.\, f_c \,.\, \text{Inst} \mid \text{Broads}) \setminus \mathcal{O} \cup \{r,g\} \mid \text{Wires})/\vec{\sigma}) \\
&\qquad \mid t_e \,.\, \underline{\overline{g}}_{\sigma_s} \,.\, \text{Enc}) \setminus \{t_c, r_e\}/\sigma_s \,,\ P^* >, \\
&< (((Q \mid f_c \,.\, \text{Inst} \mid \Pi_{o \in \mathcal{O} \setminus c} \text{IsoBroad}(o_{\mathsf{inst}}) \\
&\qquad \mid \text{IsoBroad}'(c_{\mathsf{inst}})) \setminus \mathcal{O} \cup \{r,g\} \\
&\qquad \mid \text{Wires})/\vec{\sigma}) \mid t_e \,.\, \underline{\overline{g}}_{\sigma_s} \,.\, \text{Enc}) \setminus \{t_c, r_e\}/\sigma_s \,,\ P^* >, \\
&< (((Q \mid f_c \,.\, \text{Inst} \mid \Pi_{o \in \mathcal{O} \setminus c} \text{IsoBroad}(o_{\mathsf{inst}}) \\
&\qquad \mid \text{IsoBroad}'(c_{\mathsf{inst}})) \setminus \mathcal{O} \cup \{r,g\} \\
&\qquad \mid \Pi_{o \in \mathcal{O} \setminus c} \text{IsoWire}(o_{\mathsf{inst}}, \sigma_s, o) \\
&\qquad \mid \underline{\overline{c}}_{\{\sigma_s, \sigma_{c_{\mathsf{inst}}}\}} \,.\, \underline{\sigma_{c_{\mathsf{inst}}}} \,.\, \text{IsoWire}(c_{\mathsf{inst}}, \sigma_s, c))/\vec{\sigma}) \\
&\qquad \mid t_e \,.\, \underline{\overline{g}}_{\sigma_s} \,.\, \text{Enc}) \setminus \{t_c, r_e\}/\sigma_s \,,\ P^* >, \\
&< (((Q \mid f_c \,.\, \text{Inst} \mid \Pi_{o \in \mathcal{O} \setminus c} \text{IsoBroad}(o_{\mathsf{inst}}) \\
&\qquad \mid \text{IsoBroad}'(c_{\mathsf{inst}})) \setminus \mathcal{O} \cup \{r,g\} \\
&\mid \Pi_{o \in \mathcal{O} \setminus c} \text{IsoWire}(o_{\mathsf{inst}}, \sigma_s, o) \mid \underline{\sigma_{c_{\mathsf{inst}}}} \,.\, \text{IsoWire}(c_{\mathsf{inst}}, \sigma_s, c))/\vec{\sigma}) \\
&\qquad \mid t_e \,.\, \underline{\overline{g}}_{\sigma_s} \,.\, \text{Enc}) \setminus \{t_c, r_e\}/\sigma_s \,,\ Q^* >, \\
&< (((Q \mid f_c \,.\, \text{Inst} \mid \Pi_{o \in \mathcal{O} \setminus c} \text{IsoBroad}(o_{\mathsf{inst}}) \\
&\qquad \mid \overline{f_c} \,.\, \text{IsoBroad}(c_{\mathsf{inst}})) \setminus \mathcal{O} \cup \{r,g\} \mid \text{Wires} \\
&\qquad \mid t_e \,.\, \underline{\overline{g}}_{\sigma_s} \,.\, \text{Enc}) \setminus \{t_c, r_e\}/\sigma_s \,,\ Q^* > \mid \\
&\qquad P \in \mathit{Output}(\text{Comp}), P \xrightarrow{\overline{c}} Q\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{R}_{Finished} \stackrel{\text{def}}{=} \{ &< (((P \mid \text{Inst} \mid \text{Broads}) \setminus \mathcal{O} \cup \{r,g\} \mid \text{Wires} \\
&\qquad \mid t_e \,.\, \underline{\overline{g}}_{\sigma_s} \,.\, \text{Enc}) \setminus \{t_c, r_e\}/\sigma_s \,,\ P^* >, \\
&< (((Q \mid \lfloor \overline{t_c} \,.\, \text{CompInst} \rfloor \sigma_s (\overline{t_e} \,.\, \text{CompInst}) \\
&\qquad \mid \text{Broads}) \setminus \mathcal{O} \cup \{r,g\} \mid \text{Wires} \\
&\qquad \mid t_e \,.\, \underline{\overline{g}}_{\sigma_s} \,.\, \text{Enc}) \setminus \{t_c, r_e\}/\sigma_s \,,\ P^* >, \\
&< (((Q \mid \overline{t_e} \,.\, \text{CompInst} \mid \text{Broads}) \setminus \mathcal{O} \cup \{r,g\} \mid \text{Wires} \\
&\qquad \mid t_e \,.\, \underline{\overline{g}}_{\sigma_s} \,.\, \text{Enc}) \setminus \{t_c, r_e\}/\sigma_s \,,\ P^* >, \\
&< (((Q \mid \text{CompInst} \mid \text{Broads}) \setminus \mathcal{O} \cup \{r,g\} \mid \text{Wires} \\
&\qquad \mid \underline{\overline{g}}_{\sigma_s} \,.\, \text{Enc}) \setminus \{t_c, r_e\}/\sigma_s \,,\ P^* >, \\
&< (((\mathbf{0} \mid \text{CompInst} \mid \text{Broads}) \setminus \mathcal{O} \cup \{r,g\} \mid \text{Wires} \\
&\qquad \mid \text{Enc}) \setminus \{t_c, r_e\}/\sigma_s \,,\ \mathbf{0}^* > \mid \\
&\qquad P \in \mathit{Finished}(\text{Comp}), P \xrightarrow{\overline{g}} Q\}
\end{aligned}
$$

From this bisimulation we show congruence since all clocks but those in the set $\{\sigma_s\} \cup \vec{\sigma}$ idle on both sides of the equivalence until an input is received at which point it is only necessary, from the definition of the congruence, to show observation equivalence.

Proof Sketch for Theorem 2 ('Isochronic Forks')

Proof. We first construct a weak bisimulation as follows:

$$\{< (\text{IsoBroad}(o) \mid \Pi_{n \leq |\vec{i}|} \text{IsoWire}(o, \sigma_s, i_n))/\sigma_o \mid \Delta_{\sigma_s}, \text{Fork}(o, \vec{i}) \mid \Delta_{\{\sigma_s, \sigma_o\}} >\}$$

$$\cup$$

$$\{< (\mathrm{IsoBroad}'(o) \mid \mathit{\Pi}_{n \leq |\vec{j}|}\, \underline{\sigma_o}\,.\,\mathrm{IsoWire}(o, \sigma_s, j_n)$$

$$\mid \mathit{\Pi}_{n \leq |\vec{k}|}\, \underline{\overline{k_n}}_{\{\sigma_s, \sigma_o\}} \cdot \underline{\sigma_o}\,.\,\mathrm{IsoWire}(o, \sigma_s, k_n)$$

$$\mid \mathit{\Pi}_{n \leq |\vec{l}|}\, \mathrm{IsoWire}(o, \sigma_s, l_n))/\sigma_o \mid \mathit{\Delta}_{\sigma_s}, \mathrm{Fork}'(o, \vec{i}, (\vec{i} \setminus \vec{j})) \mid \mathit{\Delta}_{\{\sigma_s, \sigma_o\}} > \mid$$

$$|\vec{j}| + |\vec{k}| + |\vec{l}| = |\vec{i}| \wedge \forall n \leq |\vec{i}| \cdot (\exists m \cdot j_m = i_n) \vee (\exists m \cdot k_m = i_n) \vee (\exists m \cdot l_m = i_n)\}$$

(Abusing the notation to form $\vec{i} \setminus \vec{j}$ meaning all those members of $\vec{i}$ not already included in $\vec{j}$, i.e., not having been already communicated.)

$$\cup$$

$$\{< (\overline{f_o}\,.\,\mathrm{IsoBroad}(o) \mid \mathit{\Pi}_{n \leq |\vec{i}|}\mathrm{IsoWire}(o, \sigma_s, i_n))/\sigma_o \mid \mathit{\Delta}_{\sigma_s}, \overline{f_o}\,.\,\mathrm{Fork}(o, \vec{i}) \mid \mathit{\Delta}_{\{\sigma_s, \sigma_o\}} >\}$$

We then observe that both agents idle on all clocks except σ_s and σ_o until the a–transition is taken and are therefore congruent.

Proof Sketch for Theorem 4 ('Jam Check')

Proof. From the definition of Rule (tTO1), we see that $\mathrm{Check} \stackrel{\mathrm{def}}{=} \mu x.\lfloor \mathit{\Delta} \rfloor \sigma_s(x)$ defines a state with a self–transition in σ_s only. Under the assumptions of the theorem, System produces transitions in labels σ_s and τ only. Since all σ_s transitions of Check may be matched weakly under observation equivalence, any system in which σ_s is live can therefore be shown equivalent. On the other hand, a system where there exists a path to a state where σ_s is indefinitely stalled clearly cannot match successive ticks of σ_s and therefore will not be equivalent.