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 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. This paper develops a process–algebraic model of coordination 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 jam checks at compile–time.

1 Introduction

One important domain for embedded–systems designers are *digital signal processing and control* (DSPC) applications. 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*, based on the rich repositories of pre–compiled and well–tested software components (PID–controllers, FIR–filters, FFT–transforms, etc.) built by engineers over many years. Applications are simply programmed by interconnecting components, which frees engineers from most of the error–prone low–level programming tasks. Design efficiency is further aided by the fact that DSPC programming tools, including LabView [9], iConnect [15] and Ptolemy [10], provide a graphical user interface that supports hierarchical extensions of signal–flow graphs. These permit the *encapsulation* of sub–systems into single components, thus enabling the reuse of system designs.

* Research supported by EPSRC grant GR/M99637.

While the visual signal–flow formalism facilitates the structural design of DSPC applications, the behaviour of a component–based system manifests itself only once its components are scheduled 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. The scheduling typically follows a cycle–based execution model with the phases *collect input* (I), *compute reaction* (R) and *deliver output* (O). At the top level, the scheduler continuously iterates between executing the *source components* that produce new inputs, e.g., by reading sensor values, and 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*, 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* [6], implies that each output signal will be 'simultaneously' and 'instantaneously' received at each connected input. This synchronous scheme can be applied in a hierarchical fashion, abstracting a sequence of RO–steps produced by a sub–system into a single RO–step (cf. Sec. 2).

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 [15], indicating a serious real–time problem (cf. Sec. 2). Unfortunately, in existing tools, there are no compile–time checks for detecting jams, thereby forcing engineers to rely on extensive simulations for validating their applications before delivery. Moreover, there is no formal model of IRO scheduling for DSPC programming systems that can be used for the static analysis of jams, and the question of how to distribute the monolithic IRO scheduler into a uniform model of coordination has not been addressed in the literature either.

The objective of this paper is to show that a relatively small number of standard concepts studied in concurrency theory provides the key to *compositionally* formalising the semantics of component–based DSPC designs, and to enabling static jam checks. The most important concepts from the process–algebra tool–box are *handshake* synchronisation from CCS [12] and *abstract clocks* in combination with *maximal progress* as investigated in temporal process algebras, specifically TPL [7], PMC [1] and CSA [3]. We use handshake synchronisation for achieving serialisation and maximal–progress clocks for enforcing synchrony. Finally, given maximal progress, synchronous encapsulation may be captured naturally in terms of *clock–hiding*, similar to hiding in CSP [8]. 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) and which conservatively extends CCS in being equipped with a behavioural theory based on *observation equivalence* [12].

As our main contribution we will formally establish that CaSE is expressive enough for faithfully modelling the principles of IRO scheduling and for capturing 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 this way, our modelling in CaSE yields a *model of coordination* for synchronous component–based design, whose virtue is its compositional style for specifying and reasoning about DSPC systems and its support for the static capture of semantic properties of DSPC programs. Thus, CaSE provides a foundation for developing new–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* whose hierarchical signal–flow graph is sketched in Fig. 1. The task is to analyse an audio signal and continually 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 the 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 assessing the intensity of one frequency range, which is then displayed by component instance dk:BarGraph. The first input port ei_{k1} of ck:Element is connected to the output port $\overline{\mathrm{so}}$ of the single instance s0:Soundcard, which generates the audio signal and provides exactly one audio value each time it is scheduled. As can be seen by the wire stretching from output port $\overline{\mathrm{co}}_k$ to input port ei_{k2}, ck:Element 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, ck1:Filter, and one of Quantise, ck2:Quantise, as shown in Fig. 1 on the right–hand side.

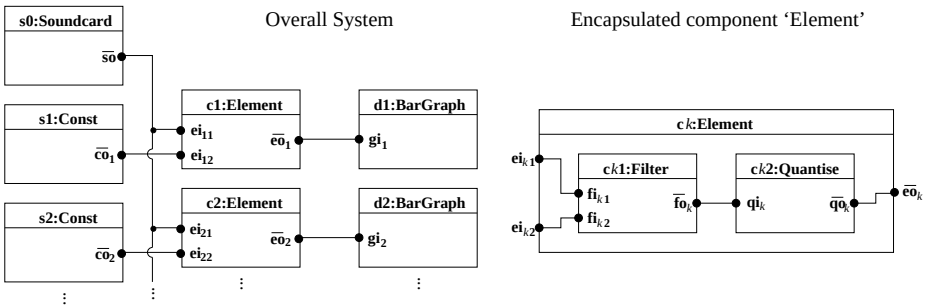

Fig. 1. Example: Digital spectrum analyser

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 sound value will be instantaneously propagated from output port $\overline{\text{so}}$ of s0 to the input port ei_{k1} of each ck:Element, for all $k \geq 1$, according to the principle of isochronic broadcast discussed below. 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 synthesised frequency–strength signal will be emitted by component ck2:Quantise via port $\overline{\text{qo}}_k$ and propagated by ck through port $\overline{\text{eo}}_k$. Upon reception of this signal by dk:BarGraph at port gi_k, this computation component instance will also request to be scheduled and, according to the synchrony hypothesis, granted execution within the same IRO–cycle. When all components dk have executed, the current IRO–cycle ends since these do not generate outputs that need to be propagated to the system environment.

It is important to note that, since each ck encapsulates further computation component instances, its execution is non–trivial and involves a sub–scheduler that will schedule ck1:Filter and ck2:Quantise in such a way that an RO–cycle of these instances will appear atomic outside of ck. This ensures that the scheduling of the inner ck1 and ck2 will not be interleaved with the execution of any sibling instance cl of ck, for $l \neq k$, or any component instance dk.

Isochronic output. Whenever s0:Soundcard is scheduled in our example system, it generates an audio signal whose value is propagated via a wire from port $\overline{\text{so}}$, which forks to port ei$k$1 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* [6] 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.

Jams. Let us now consider what happens if instances s0:Soundcard and s1:Const are accidently connected the wrong way around, i.e., output port $\overline{\text{so}}$ is connected to input port ei_{12}, and output port $\overline{\text{co}}_1$ of s1:Const to input port ei_{11} of c1:Element. Recall that c11:Filter within c1:Element will only read a value, an initialisation value, from port ei_{12} in the first IRO–cycle and never again afterwards. Thus, when the value of s0:Soundcard produced in the second cycle is propagated to port ei_{12} and further to fi_{12}, the system *jams*. This is because the value that has been produced in the second IRO–cycle and stored at this latter 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 model of coordination for DSPC design in Sec. 4. The purpose here is not to develop yet another process algebra, but to tailor several well–studied semantic concepts for addressing a specific application domain. CaSE is inspired by Hennessy and Regan's TPL [7], which is an extension of Milner's CCS [12] 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* $\lfloor P \rfloor \sigma(Q)$, where the occurrence of σ deactivates process P and activates Q; (iii) the concept of *maximal progress* 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 [3], 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 σ of process P. Clock hiding is basically hiding as in CSP [8], i.e., hidden actions are made non–observable. In combination with maximal progress, this has the important effect 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 for modelling isochronous broadcast and the subsystems–as–components principle. Finally, in contrast to TPL and similar to CCS and CSA, we will equip CaSE with a bisimulation–based semantic theory [12].

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 [12], an action a communicates with its complement $\overline{a}$ to produce 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:

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

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 $\{\overline{a} \mid a \in L\}$, where $\overline{\overline{a}} =_{\mathrm{df}} a$, and $x \stackrel{\mathrm{def}}{=} P$ for the process $\mu x.P$.

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

Table 1. Operational semantics of CaSE

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

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

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

$$\text{Res}\ \frac{P \xrightarrow{\alpha} P'}{P\backslash L \xrightarrow{\alpha} P'\backslash L}\ \alpha \notin L \cup \overline{L} \qquad \text{tRes}\ \frac{P \xrightarrow{\sigma} P'}{P\backslash L \xrightarrow{\sigma} P'\backslash L}$$

$$\text{Par1}\ \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \text{tPar}\ \frac{P \xrightarrow{\sigma} P' \quad Q \xrightarrow{\sigma} Q'}{P|Q \xrightarrow{\sigma} P'|Q'}\ P|Q \xrightarrow{\tau}\!\!\!\!\!\!/\ \ $$

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

$$\text{Par3}\ \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad \text{tHid2}\ \frac{P \xrightarrow{\rho} P'}{P/\sigma \xrightarrow{\rho} P'/\sigma}\ \sigma \neq \rho,\ P \xrightarrow{\sigma}\!\!\!\!\!\!/\ \ $$

$$\text{Hid}\ \frac{P \xrightarrow{\alpha} P'}{P/\sigma \xrightarrow{\alpha} P'/\sigma} \qquad \text{tTO1}\ \frac{}{\lfloor P \rfloor \sigma(Q) \xrightarrow{\sigma} Q}\ P \xrightarrow{\tau}\!\!\!\!\!\!/\ \ $$

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

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

operational rules. We 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 $\exists P' \in \mathcal{P}.\, P \xrightarrow{\gamma} P'$. Note that, despite the negative side conditions of some rules, the transition relation is well–defined for guarded processes. Our semantics obeys the following properties, for all clocks $\sigma \in \mathcal{T}$: (i) *maximal progress*, i.e., $P \xrightarrow{\sigma}$ implies $P \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 broadcasting [8].

Intuitively, the *nil* process $\mathbf{0}$ permits all clocks to tick, while the *timelock* processes Δ and Δ_σ prohibit the ticking of any clock and of clock 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. Because of time determinacy, time has to proceed equally on both sides of summation. 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 in P at the same states as σ, by 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 an action transition, but persists along clock transitions. Finally, $\mu x.\, P$ denotes *recursion* and behaves as a distinguished solution of the equation $x = P$.

Our interpretation of prefixes $\alpha.P$ adopted above, for $\alpha \neq \tau$, is *relaxed* [7], i.e., we allow this 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 $T \subseteq \mathcal{T}$, by $\underline{\alpha}_T.P =_{\mathrm{df}} \alpha.P + \Delta_T$, where $\Delta_T =_{\mathrm{df}} \sum_{\sigma \in T} \Delta_\sigma$. As usual, $\sum$ denotes the indexed version of operator $+$, with the empty summation understood to be process $\mathbf{0}$. For convenience, we abbreviate $\lfloor \mathbf{0} \rfloor \sigma(P)$ by $\sigma.P$, and $\lfloor \Delta \rfloor \sigma(P)$ by $\underline{\sigma}.P$. We also write $P/\{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ for $P/\sigma_1/\sigma_2 \cdots /\sigma_k$, if the order in which clocks are hidden is inessential. Moreover, for finite $A \subseteq \mathcal{A} \backslash \{\tau\}$ and process P, we let $A.P$ stand for the recursively defined process $\sum_{a \in A} a.(A \backslash \{a\}).P$, if $A \neq \emptyset$ and P, otherwise. Finally, instead of relabelling as in CCS [12] we use syntactic substitution, e.g., $P[a'/a, b'/b]$ relabels all occurrences of actions a, $\overline{a}$, b, $\overline{b}$ in P by a', $\overline{a}'$, b', $\overline{b}'$, respectively.

Temporal observation equivalence and congruence. This section equips CaSE with a bisimulation–based semantics [12]. 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 [16]. Observation equivalence is a notion of bisimulation in which any sequence of 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 $\stackrel{\epsilon}{\Rightarrow} =_{\mathrm{df}} \stackrel{\tau}{\rightarrow}^*$ and $P \stackrel{\gamma}{\Rightarrow} P'$ if there exist processes P'' and P''' such that $P \stackrel{\epsilon}{\Rightarrow} P'' \stackrel{\gamma}{\rightarrow} P''' \stackrel{\epsilon}{\Rightarrow} P'$.

Definition 1. *A symmetric relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a* temporal weak bisimulation *if $P \stackrel{\gamma}{\rightarrow} P'$ 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 for $\gamma \in \mathcal{A} \cup \mathcal{T}$. 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, the following proposition is central.

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

The validity of this proposition is due to the maximal–progress property in CaSE. To identify the largest equivalence contained in $\approx$, the summation fix of CCS is not sufficient. As in other work in temporal process algebras [3], the deterministic nature of clocks implies the following definition.

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* $\exists Q'.\, Q \overset{\alpha}{\Rightarrow} Q'$ *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 observation congruence* $\mathcal{R}$.

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

CCS [12] can be identified in terms of syntax, operational semantics and bisimulation semantics as the sub–calculus of CaSE that is obtained by setting $\mathcal{T} = \emptyset$.

4 A Synchronous Coordination Model with Encapsulation

This section presents our model of coordination for DSPC applications on the basis of our process calculus CaSE. As illustrated in Fig. 2 we will successively model the key ingredients of a DSPC application: the behaviour of its source and computation components towards its environment (Figs. 2(a) and (b)), a compositional version of the centralised scheduler which is distributed to 'wrap' each component instance (Figs. 2(c) and (d)), the application's isochronous forks connecting output and input ports (Fig. 2(e)), and the facility to encapsulate several computation components (Fig. 2(f)). Having these ingredients at hand, a CaSE model of a DSPC application can then be built systematically along the structure of hierarchical signal–flow graphs, which we will illustrate by way of the digital–spectrum–analyser example introduced in Sec. 2. A particular emphasis will be given on showing how our modelling may facilitate static jam analysis.

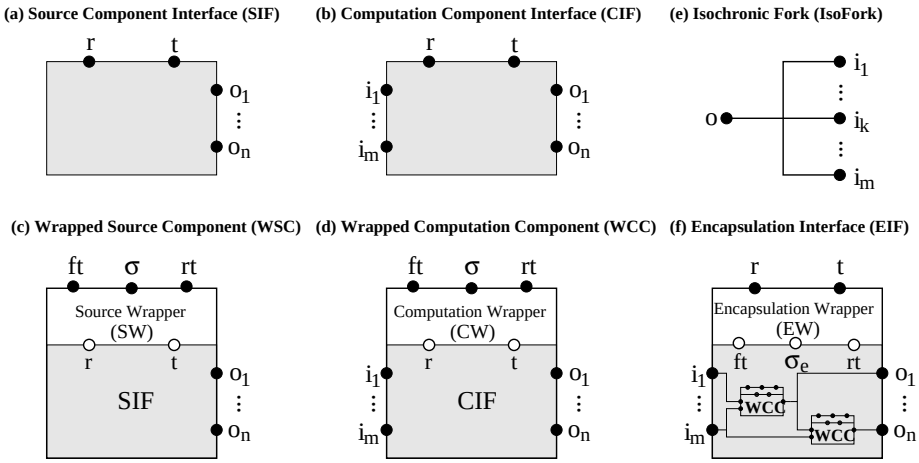

Fig. 2. Illustration of our modelling toolbox

4.1 Component interfaces. A *component interface* describes the interaction of a source component or a basic computation component, which does not encapsulate a subsystem, with its environment via its ports (cf. Figs. 2(a) and (b)). These ports include a component's *output* ports, $O = \{o_1, \ldots, o_n\}$, for

$n \geq 0$, and, in case of a computation component, its *input* ports, $I = \{i_1, \ldots, i_m\}$, for $m \geq 1$. Note that we abstract from values carried by signals through ports. In addition, each component interfaces to the system scheduler via port r, over which a component sends *request–to–be–scheduled* messages, and port t via which a *token* is passed between the scheduler and the component, with the intention that a component can go ahead with its computation of output signals whenever it holds the token.

Formally, source and computation component interfaces are processes specified in the following CCS sub–languages of CaSE, where $i \in I$ and $o \in O$:

Source Component Interface	*Computation Component Interface*
$\mathrm{SIF} ::= \mathbf{0} \mid x \mid \mathrm{SR} \mid \mu x.\,\mathrm{SR}$	$\mathrm{CIF} ::= \mathbf{0} \mid x \mid \mathrm{CI} \mid \mu x.\,\mathrm{CI}$
	$\mathrm{CI} ::= i.\mathrm{CI} \mid \mathrm{CI} + \mathrm{CI} \mid i.\mathrm{CR}$
$\mathrm{SR} ::= \overline{r}.t.\tau.\mathrm{SO}$	$\mathrm{CR} ::= \overline{r}.t.\tau.\mathrm{CO}$
$\mathrm{SO} ::= \overline{o}.\mathrm{SO} \mid \mathrm{SO} + \mathrm{SO} \mid \overline{t}.\mathrm{SIF}$	$\mathrm{CO} ::= \overline{o}.\mathrm{CO} \mid \mathrm{CO} + \mathrm{CO} \mid \overline{t}.\mathrm{CIF}$

Intuitively, after reading its inputs in case of a computation component, a component instance (i) requests to be scheduled (action $\overline{r}$), (ii) waits for receiving the scheduler's token (action t), which indicates that the request has been granted and ensures serialisation on the underlying single processor, (iii) computes the output signal values (internal action τ), (iv) outputs these signal values over the corresponding output ports, and (v) returns the token to the scheduler (action $\overline{t}$). The interfaces of the source and basic computation component instances of our example system can then be specified as follows:

$$\mathrm{SIF}_{\mathrm{s}0} \stackrel{\mathrm{def}}{=} \overline{r}.t.\tau.\overline{so}.\overline{t}.\mathrm{SIF}_{\mathrm{s}0} \qquad \mathrm{SIF}_{\mathrm{s}k} \stackrel{\mathrm{def}}{=} \overline{r}.t.\tau.\overline{co}_k.\overline{t}.\mathbf{0}$$

$$\mathrm{CIF}_{\mathrm{c}k1} \stackrel{\mathrm{def}}{=} fi_{k1}.fi_{k2}.\overline{r}.t.\tau.\overline{fo}_k.\overline{t}.\mathrm{CIF}'_{\mathrm{c}k1} \qquad \mathrm{CIF}'_{\mathrm{c}k1} \stackrel{\mathrm{def}}{=} fi_{k1}.\overline{r}.t.\tau.\overline{fo}_k.\overline{t}.\mathrm{CIF}'_{\mathrm{c}k1}$$

$$\mathrm{CIF}_{\mathrm{c}k2} \stackrel{\mathrm{def}}{=} qi_k.\overline{r}.t.\tau.\overline{qo}_k.\overline{t}.\mathrm{CIF}_{\mathrm{c}k2} \qquad \mathrm{CIF}_{\mathrm{d}k} \stackrel{\mathrm{def}}{=} gi_k.\overline{r}.t.\tau.\overline{t}.\mathrm{CIF}_{\mathrm{d}k}$$

Note that sk:Const produces an output $\overline{\mathrm{so}}$ during the first cycle only, while ck:Element reads an input from port fi_{k1} during the first cycle only, as desired.

4.2 Component instances and scheduling. As seen above, a component uses its ports r and t to negotiate its execution with a scheduler. 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 component instances with enough local control so they participate coherently in a global IRO–scheduling scheme, without the presence of a global scheduler (cf. Figs. 2(c) and (d)). Indeed all wrappers added together will represent a distributed version of an imagined central IRO scheduler.

Before introducing our distributed scheduler we present, for reference, an abstract model of the global centralised scheduler, as employed in the DSPC tool iConnect [15]. It uses an abstract clock σ that reflects the phase clock inherent in IRO scheduling. This clock organises the strict alternation between source and computation phases and, by way of maximal progress, implements run–to–completion within each phase. The global scheduler is defined via the following two sets of process equations, namely CSC that models the computation phase and CSS that models the source phase. They are stated relative to the sets $\mathcal{S}$ of source component instances and $\mathcal{C}$ of computation component instances within

the signal–flow graph under consideration.

$$\mathrm{CSC}(W,\sigma) \stackrel{\text{def}}{=} \lfloor C(W,\sigma)\rfloor\sigma(\mathrm{CSS}(\emptyset,\emptyset,\sigma))$$
$$C(W,\sigma) \stackrel{\text{def}}{=} (\sum_{c\in\mathcal{C}\setminus W} r_c.\mathrm{CSC}(W\cup\{c\},\sigma)) + (\sum_{c\in W} \overline{t}_c.\underline{t_c}_\sigma.\mathrm{CSC}(W\setminus\{c\},\sigma))$$
$$\mathrm{CSS}(W,D,\sigma) \stackrel{\text{def}}{=} \lfloor S(W,D,\sigma)\rfloor\sigma(\mathrm{CSC}(\emptyset,\sigma))$$
$$S(W,D,\sigma) \stackrel{\text{def}}{=} (\sum_{s\in\mathcal{S}\setminus(W\cup D)} r_s.\mathrm{CSS}(W\cup\{s\},D,\sigma)) + (\sum_{s\in W} \overline{t}_s.\underline{t_s}_\sigma.\mathrm{CSS}(W\setminus\{s\},D\cup\{s\},\sigma))$$

The process equations are parameterised in the phase clock σ, as well as the set W of component instances that are waiting for their scheduling request to be granted and the set D of source component instances that have already executed during the current source phase. Recall that each source component instance can execute at most once during each source phase, while each computation component instance may execute several times during a computation phase. While there are component instances that request to be scheduled or wait for being scheduled, the scheduler remains in the current phase, as is enforced by maximal progress. Otherwise, the phase clock may tick and switch phases.

To distribute this centralised scheduler over each component instance, all we assume is that the single embedded processor, on which the DSPC application is scheduled, provides some facility to ensure mutual exclusion. This may be modelled via a single token that the processor passes on to the component instance that may execute next: $\mathrm{CPUtoken} \stackrel{\text{def}}{=} \overline{ft}.rt.\mathrm{CPUtoken}$, where ft stands for *fetch token* and rt for *release token*. Now, we may define the wrapping of computation and source component instances via meta–processes WCC and WSC, respectively. They are parameterised in the computation (source) component interface CIF_c (SIF_s) of a given computation (source) component instance c (s), as well as in the phase clock σ.

$$\mathrm{WCC}(\mathrm{CIF}_c,\sigma) \stackrel{\text{def}}{=} (\mathrm{CIF}_c \,|\, \mathrm{CW}(\sigma))\setminus\{r,t\} \quad \mathrm{CW}(\sigma) \stackrel{\text{def}}{=} \lfloor r.ft.\overline{t}.\underline{t}_\sigma.\overline{rt}.\mathrm{CW}(\sigma)\rfloor\sigma(\sigma.\mathrm{CW}(\sigma))$$
$$\mathrm{WSC}(\mathrm{SIF}_s,\sigma) \stackrel{\text{def}}{=} (\mathrm{SIF}_s \,|\, \sigma.\mathrm{SW}(\sigma))\setminus\{r,t\} \; \mathrm{SW}(\sigma) \stackrel{\text{def}}{=} \lfloor r.ft.\overline{t}.\underline{t}_\sigma.\overline{rt}.\sigma.\sigma.\mathrm{SW}(\sigma)\rfloor\sigma(\sigma.\mathrm{SW}(\sigma))$$

Consider process $\mathrm{WCC}(\mathrm{CIF}_c,\sigma)$, which runs the wrapping process $\mathrm{CW}(\sigma)$ alongside the computation component interface CIF_c. Both synchronise via the now internalised channels r and t. If the component instance c signals its desire to be scheduled via a communication on channel r, the wrapping process $\mathrm{CW}(\sigma)$ waits until it may fetch the CPU token (action ft), passes on the token via the internal channel t, waits until the token has been passed back via the same channel, i.e., until the execution of c is complete, and then surrenders the token to the CPU (action $\overline{rt}$). If no computation component instance wishes to be scheduled, process $\mathrm{CW}(\sigma)$ may time out, thus allowing the overall system to switch to the source phase. In this state, component instance c must wait until the clock ticks again, i.e., until the scheduling has returned to the computation phase. The behaviour of $\mathrm{WSC}(\mathrm{SIF}_s,\sigma)$ wrapping source component instances is similar, except that those may only be scheduled once during each source phase. Thus, the source wrapper process $\mathrm{SW}(\sigma)$ makes sure that two clock ticks have to pass before a request of the wrapped source component instance is considered again.

Moreover, note that the initial σ–prefix in front of the wrapping process $\mathrm{SW}(\sigma)$ ensures that the first source phase begins with the first ticking of σ. The following theorem shows that our compositional approach to scheduling coincides with the centralised one, where $\Pi_{k\in K} P_k$ stands for the parallel composition of processes P_k, for a finite index set K.

Theorem 2. *Let $\mathcal{S}$ ($\mathcal{C}$) be a finite set of source (computation) components with interfaces SIF_s (CIF_c), for $s \in \mathcal{S}$ ($c \in \mathcal{C}$), let σ be the phase clock, and let $R =_{df} \{r_s, t_s \mid s \in \mathcal{S}\} \cup \{r_c, t_c \mid c \in \mathcal{C}\}$. Then*

$$(\Pi_{s\in\mathcal{S}}\, WSC(SIF_s, \sigma) \mid \Pi_{c\in\mathcal{C}}\, WCC(CIF_c, \sigma) \mid CPUtoken)\backslash\{ft, rt\} \approxeq (\Pi_{s\in\mathcal{S}} SIF_s[r_s/r, t_s/t] \mid \Pi_{c\in\mathcal{C}} CIF_c[r_c/r, t_c/t] \mid CSC(\emptyset, \sigma)))\backslash R\,.$$

4.3 Isochronic Forks. Before encoding isochronous forks in CaSE we present their naive modelling in CCS. To do so, we introduce a new output prefix $\overline{o}; P$ and assume that output port $\overline{o}$ shall be connected to input ports $I = \{i_1, i_2, \ldots, i_m\}$ via an isochronic fork, as sketched in Fig. 2(e). We need to ensure that the signal transmitted via $\overline{o}$ reaches all i_k, for $1 \leq i \leq m$, before process P executes. To model this, we define $\overline{o}; P =_{\mathrm{df}} \overline{o}.f_o.P$ and $\mathrm{ForkWire}(\overline{o}, I) =_{\mathrm{df}} o.\overline{I}.\overline{f}_o.\mathrm{ForkWire}(\overline{o}, I)$. Here, $\mathrm{ForkWire}(\overline{o}, I)$ models the forking wire between port $\overline{o}$ and the ports in I. This wire distributes messages from the output port to all input ports and, once finished, signals this via the distinguished action $\overline{f}_o$. The sending process $\overline{o}; P$ has to wait for synchronisation on f_o before it can proceed with P, whence ensuring isochrony. While this solution is feasible, it requires that the number of intended recipients of a broadcasted signal is fixed up front and cannot grow as components are added to a signal–flow graph.

To overcome this problem we employ isochronic wires that connect the output port with exactly one input port, and use a fresh clock σ_o under maximal progress for synchronisation between sender and receivers of a broadcast signal. In analogy to the above we define the new isochronic output prefix $\overline{o}{:}P =_{\mathrm{df}} C_{\overline{o},P}$ with $C_{\overline{o},P} \stackrel{\mathrm{def}}{=} \lfloor \underline{\overline{o}}.C_{\overline{o},P} \rfloor \sigma_o(P)$ and an isochronic wire connecting $\overline{o}$ to input port i by $\mathrm{IsoWire}(\overline{o}, i) =_{\mathrm{df}} \underline{o}_{\sigma_o}.\underline{\overline{\imath}}_{\sigma_o}.\underline{\sigma}_o.\mathrm{IsoWire}(\overline{o}, i)$. Thus, for a broadcast request $\overline{o}$, an arbitrary number of copies of the signal will be communicated on $\overline{o}$ until clock σ_o, which defines the isochronous instant in which the communication occurs, ticks and ends that instant. Because of maximal progress, σ_o can only tick when there are no further receivers listening on o. In this way signal $\overline{o}$ obtains maximal distribution, and one can add further receiving ports j later on by simply including a new isochronic wire from $\overline{o}$ to j without having to change the existing model. The following theorem shows that our compositional approach to isochronic broadcast faithfully models isochronous forks.

Theorem 3. *Let $\overline{o} \in \overline{\Lambda}$, $I \subseteq_{fin} \Lambda$ and $P \in \mathcal{P}$. Then*

$$(\overline{o}{:}P \mid \Pi_{i\in I}\, IsoWire(\overline{o}, i))\backslash\{o\}/\sigma_o \;\approxeq\; (\overline{o};P \mid ForkWire(\overline{o}, I))\backslash\{o, f_o\} \mid \Delta_{\sigma_o}\,.$$

The parallel component Δ_{σ_o} caters for the fact that the clock hiding operator $/\sigma_o$ eliminates clock σ_o. From now on we assume that all action prefixes $\overline{o}.P$ referring to the output ports of our component interfaces are replaced by isochronic ones $\overline{o}{:}P$, e.g., $\mathrm{SIF_{s0}}$ becomes $\overline{r}.t.\tau.\overline{so}{:}\overline{t}.\mathrm{SIF_{s0}}$.

Note that isochronous wiring cannot be modelled faithfully and compositionally in Hoare's CSP [8] or Prasad's CBS [13]. While the broadcasting primitive in CSP ignores the direction in which information is propagated, the one in CBS does not force receivers to synchronise with the sender.

4.4 Encapsulation. Hierarchical signal–flow graphs allow system designers to encapsulate several interconnected computation components, i.e., a subsystem, into a single computation component. As depicted in Fig. 2(f), a subsystem is a tuple $\langle \mathcal{C}_e, W_e, I, O, W_I, W_O \rangle$ that consists of (i) a finite set $\mathcal{C}_e \subseteq \mathcal{C}$ of computation components, with disjoint sets of input ports I_e and sets of output ports O_e, (ii) a set of internal isochronic wires connecting output ports in O_e with input ports in I_e, (iii) a set of input ports $I = \{i_1, \ldots, i_m\}$, (iv) a set of output ports $O = \{o_1, \ldots, o_n\}$, (v) a set $W_I \subseteq I \times I_e$ of isochronic wires connecting the input ports of the subsystem with the input ports of the encapsulated components, and (vi) a set $W_O \subseteq O_e \times O$ of isochronic wires connecting the output ports of the encapsulated components with the output ports of the subsystem. In the example of Fig. 1 we have ck:Element $= \langle \{c k1\text{:Filter}, c k2\text{:Quantise}\}, \{\langle \overline{\text{fo}}_k, \text{qi}_k \rangle\}, \{\text{ei}_{k1}, \text{ei}_{k2}\}, \{\overline{\text{eo}}_k\}, \{\langle \text{ei}_{k1}, \text{fi}_{k1} \rangle, \langle \text{ei}_{k2}, \text{fi}_{k2} \rangle\}, \{\langle \overline{\text{qo}}_k, \overline{\text{eo}}_k \rangle\} \rangle$. The CaSE model of this subsystem is given by

$$\begin{aligned} \text{Element}_k(\sigma_e) \stackrel{\text{def}}{=} (\mathit{\Pi}_{c \in \mathcal{C}_e} \text{WCC}'(\text{CIF}_c, \sigma_e) \mid \mathit{\Pi}_{\langle \overline{o}_e, i_e \rangle \in W_e} \text{IsoWire}(\overline{o}_e, i_e) \mid \\ \mathit{\Pi}_{\langle i, i_e \rangle \in W_I} \text{IsoWire}(\overline{i}, i_e) \mid \mathit{\Pi}_{\langle \overline{o}_e, \overline{o} \rangle \in W_O} \text{IsoWire}(\overline{o}_e, o)) \backslash I_e \backslash O_e / \sigma_{O_e} \,, \end{aligned}$$

where $\sigma_{O_e} =_{\text{df}} \{\sigma_{o_e} \mid o_e \in O_e\}$ contain the clocks governing the encapsulated isochronic wires. Also, $\text{WCC}'(\text{CIF}_c, \sigma_e) \stackrel{\text{def}}{=} (\text{CIF}_c \mid \text{CW}'(\sigma_e)) \backslash \{r, t\}$ is an updated version of our instantiation wrapper given in Sec. 4.2, with $\text{CW}'(\sigma_e) \stackrel{\text{def}}{=} \lfloor r.(ft.\overline{t}.\underline{t}_{\sigma_e}.\overline{rt}.\text{CW}'(\sigma_e) + \overline{r}_e.ft.\overline{t}.\underline{t}_{\sigma_e}.\overline{rt}.\text{CW}'(\sigma_e)) \rfloor \sigma_e(\sigma_e.\text{CW}'(\sigma_e))$. As subsystems must be executed atomically, the first encapsulated computation component that is ready to execute needs to request the mutual–exclusion token from its environment (action $\overline{r}_e$), i.e., from the subsystem at the next higher hierarchy level. Our modelling of encapsulation must then ensure that the token is only passed up to the environment once all computation components within the subsystem, which are able to execute, have actually executed. This is achieved via an *encapsulation wrapper* $\text{EW}(\text{SS}, I, O, \sigma_e)$ that is parameterised in the CaSE model SS of the subsystem under consideration, with input ports I, output ports O and subsystem clock σ_e. The encapsulation wrapper essentially translates back the scheduling interface $\{ft, rt, \sigma_e\}$ into $\{r, t\}$, which is the scheduling interface of a basic component.

$$\begin{aligned} \text{EW}(\text{SS}, I, O, \sigma_e) &\stackrel{\text{def}}{=} (\text{SS}[i'_1/i_1, \ldots, i'_m/i_m, o'_1/o_1, \ldots, o'_n/o_n] \mid \text{EI}(I, \sigma_e) \mid \text{EO}(O, \sigma_e)) \\ &\quad \backslash \{i'_1, \ldots, i'_m, o'_1, \ldots, o'_n, r_e\} / \sigma_I / \sigma_e \\ \text{EI}(I, \sigma_e) &\stackrel{\text{def}}{=} \sum_{i \in I} i.\overline{i}'\text{:EI}(I, \sigma_e) \;+\; \underline{r_e}_{\sigma_e}.\overline{\underline{r}}_{\sigma_e}.\underline{t}_{\sigma_e}.\text{EI}'(I, \sigma_e) \\ \text{EI}'(I, \sigma_e) &\stackrel{\text{def}}{=} \lfloor \overline{ft}.rt.\text{EI}'(I, \sigma_e) \rfloor \sigma_e(\overline{\underline{t}}_{\sigma_e}.\text{EI}(I, \sigma_e)) \\ \text{EO}(O, \sigma_e) &\stackrel{\text{def}}{=} \sum_{\overline{o} \in O} o'.\overline{o}\text{:EO}(O, \sigma_e) \,, \end{aligned}$$

where all i', for $i \in I$, and $\overline{o}'$, for $\overline{o} \in O$, are fresh port names not used in SS, and where $\sigma_I =_{\text{df}} \{\sigma_i \mid i \in I\}$. The wrapper process $\text{EI}(I, \sigma_e)$ propagates all input

signals entering the subsystem to the desired receiving components, within the same cycle of the subsystem clock σ_e. Once an inner component requests to be scheduled (action r_e), the wrapper process forwards this request via port $\overline{r}$ to the next upper hierarchy level and waits for the token, indicating granted access to the embedded processor, to be passed down via port t. In this state, the encapsulation wrapper essentially behaves as process CPUtoken has done before, i.e., engaging in a communication cycle between ports $\overline{ft}$ and rt, until no further encapsulated component wishes to execute, i.e., until clock σ_e triggers a timeout and the token is passed back up (action $\overline{t}$). The outputs produced by components within the subsystem are instantaneously propagated to the subsystem's environment via the parallel process $\mathrm{EO}(O, \sigma_e)$, which is part of the encapsulation wrapper. Note that our encapsulation wrapper hides the inner clock σ_e, whose ticking thus appears like an internal, unobservable computation, from the point of view of components outside the subsystem under consideration. The following theorem puts the subsystems–as–components principle on a formal footing.

Theorem 4. *Let SS be the CaSE model of a subsystem* $\langle \mathcal{C}_e, W_e, I, O, W_I, W_O \rangle$ *using* σ_e *as subsystem clock. Then, there exists a computation component* c *with input ports* I*, output ports* O *and component interface* CIF_c *such that* $EW(SS, I, O, \sigma_e) \cong EW(WCC'(CIF_c, \sigma_e), I, O, \sigma_e) \,|\, \Delta_{\sigma_I \cup \sigma_{O_e}}$.

We now have all tools of our DSPC modelling toolbox to complete the overall CaSE model $\mathrm{DSA}(\rho)$ of the digital spectrum analyser of Fig. 1, under phase clock ρ and given the component interfaces provided in Sec. 4.1: $\mathrm{DSA}(\rho) \stackrel{\mathrm{def}}{=}$

$$
\begin{aligned}
&(\mathrm{WSC}(\mathrm{SIF}_{\mathrm{s}0}, \rho) \,| \\
&\quad \Pi_{k \geq 1}(\mathrm{WSC}(\mathrm{SIF}_{\mathrm{s}k}, \rho) \,|\, \mathrm{WCC}(\mathrm{EW}(\mathrm{Element}_k(\sigma_k), \{\mathrm{ei}_{k1}, \mathrm{ei}_{k2}\}, \{\overline{\mathrm{eo}}_k\}, \sigma_k), \rho) \,| \\
&\qquad\quad \mathrm{WCC}(\mathrm{CIF}_{\mathrm{d}k}, \rho) \,|\, \mathrm{IsoWire}(\overline{\mathrm{so}}, \mathrm{ei}_{k1}) \,|\, \mathrm{IsoWire}(\overline{\mathrm{co}}_k, \mathrm{ei}_{k2}) \,|\, \mathrm{IsoWire}(\overline{\mathrm{eo}}_k, \mathrm{gi}_k)) \\
&)\backslash\{\mathrm{co}_k, \mathrm{ei}_{k1}, \mathrm{ei}_{k2}, \mathrm{eo}_k, \mathrm{gi}_k \,|\, k \geq 1\}\backslash\{\mathrm{so}\}/\{\sigma_{\mathrm{co}_k}, \sigma_{\mathrm{eo}_k} \,|\, k \geq 1\}/\sigma_{\mathrm{so}}
\end{aligned}
$$

Observe that our modelling proceeds along the structure of the hierarchical signal–flow graph of Fig. 1.

4.5 Jam analysis. A jam occurs when an output signal value produced by one component cannot be consumed by an intended receiving component within the same IRO–cycle. In current DSPC tools, jams are detected by the run–time system; upon detection of a jam, a DSPC application is simply terminated.

In our model of coordination we will encode jams in such a way that a jam manifests itself as a timelock regarding the overall system clock ρ. Such a timelock will occur when an isochronic wire is unable to pass on the value it holds. This can be achieved by modifying processes $\mathrm{IsoWire}(\overline{o}, i)$ throughout, such that clock ρ is stopped when the wire already stores a signal value but has not yet been able to pass it on to port i; formally, $\mathrm{IsoWire}(\overline{o}, i) =_{\mathrm{df}} \underline{o}_{\sigma_o} \overline{\underline{i}}_{\{\rho, \sigma_o\}} . \underline{\sigma}_o . \mathrm{IsoWire}(\overline{o}, i)$. Consequently, the local 'jam' condition is turned into a timing flaw, which is a global condition that stops the complete system, as desired. The next theorem makes this mathematically precise; note that our model of coordination of a DSPC system does not possess any infinite τ–computations, as long as the system

does not contain some computation components that are wired–up in feedback loops in which these components continuously trigger themselves.

Theorem 5. *Let P be a process that possesses only τ– and ρ–transitions and no infinite τ–computations, and let $Check =_{df} \mu x.\lfloor\Delta\rfloor\rho(x)$. Then $P \approx Check$ if and only if $\nexists P' \nexists s \in \{\tau,\rho\}^*$. $P \stackrel{s}{\rightarrow} P' \stackrel{\rho}{\not\Rightarrow}$.*

Hence, when considering that our model of coordination for an arbitrary hierarchical signal–flow graph can be automatically constructed from the flow graph's given component interfaces, one may statically check for jams by employing well–known algorithms for computing temporal observation equivalence [4].

5 Related Work

To the best of our knowledge, our process–algebraic model of coordination is the first formal model of the synchronous and hierarchical scheduling discipline behind DSPC tools. It complements existing work in *distributed object–oriented systems* and in *architectural description languages*. There, the focus is on distributed software rather than on embedded centralised systems, and consequently on asynchronous rather than on synchronous component behaviour.

In object–oriented systems, process–algebraic frameworks have been studied, where processes model the life–cycle of objects [14]. Within these frameworks, one may reason at compile–time whether each invocation of an object's method at run–time is permissible. This semantic analysis is different from jam analysis in DSPC applications, but similar to the *compatibility analysis* of interface automata [5], which we will discuss below. In architectural description languages, the formalism of process algebra has been studied by Bernardo et al. [2]. 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. 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 necessarily result in a system deadlock.

From a practical point of view we envision our model of coordination based on the process calculus CaSE to play the role of a *reactive–types* language. 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 [11]. Behavioural types are based on the formalism of *interface automata* [5] and employed for checking the *compatibility* property between components. However, interface automata are not expressive enough to reason about jams, which Ptolemy handles by linear–algebra techniques for the restricted class of synchronous data–flow (SDF) models. In contrast, CaSE's semantic theory is more general than SDF and lends itself to checking jams at compile–time.

6 Conclusions and Future Work

This paper presented a novel compositional model of coordination for the synchronous component–based design of and reasoning about DSPC applications. 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, which have been studied in the concurrency–theory community. The standard notion of temporal observation equivalence then facilitates the desired static reasoning about jams in DSPC applications. Future work should integrate our work in DSPC tools in the form of a reactive–types system. A prototype written in Haskell is currently being implemented in Sheffield.

Acknowledgements. We thank the anonymous referees, as well as Rance Cleaveland and Matt Fairtlough for their valuable comments and suggestions.

References

1. H.R. Andersen and M. Mendler. An asynchronous process algebra with multiple clocks. In *ESOP '94*, volume 788 of *LNCS*, pages 58–73, 1994.
2. M. Bernardo, P. Ciancarini, and L. Donatiello. Detecting architectural mismatches in process algebraic descriptions of software systems. In *WICSA 2001*, pages 77–86. IEEE Comp. Soc. Press, 2001.
3. R. Cleaveland, G. Lüttgen, and M. Mendler. An algebraic theory of multiple clocks. In *CONCUR '97*, volume 1243 of *LNCS*, pages 166–180, 1997.
4. R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In *CAV '96*, volume 1102 of *LNCS*, pages 394–397, 1996.
5. L. de Alfaro and T.A. Henzinger. Interface automata. In *ESEC/FSE 2001*, volume 26, 5 of *Softw. Eng. Notes*, pages 109–120. ACM Press, 2001.
6. S. Hauck. Asynchronous design methodologies: An overview. *Proc. of the IEEE*, 83(1):69–93, 1995.
7. M. Hennessy and T. Regan. A process algebra for timed systems. *Inform. and Comp.*, 117:221–239, 1995.
8. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
9. G.W. Johnson and R. Jennings. *LabView Graphical Programming*. McGraw, 2001.
10. E.A. Lee. Overview of the Ptolemy project. Technical Report UCB/ERL M01/11, Univ. of California at Berkeley, 2001.
11. E.A. Lee and Y. Xiong. Behavioral types for component-based design. Technical Report UCB/ERL M02/29, Univ. of California at Berkeley, 2002.
12. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
13. K.V.S. Prasad. Programming with broadcasts. In *CONCUR '93*, volume 715 of *LNCS*, pages 173–187, 1993.
14. F. Puntigam. Type specifications with processes. In *FORTE '95*, volume 43 of *IFIP Conf. Proc.* Chapman & Hall, 1995.
15. 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, pages 61–70, 1998.
16. C. Verhoef. A congruence theorem for structured operational semantics with predicates and negative premises. *Nordic J. of Computing*, 2(2):274–302, 1995.