Electronic Notes in Theoretical Computer Science 68 No. 3 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume68.html 16 pages

SGCCS: A Graphical Language for Real-Time Coordination [1]

David Šafránek [2]

Faculty of Informatics
Masaryk University Brno
Czech Republic

Abstract

We present a graphical coordination language SGCCS as a language for modeling of coordination in discrete real-time. SGCCS can be viewed as a graphical version of Synchronous Calculus of Communicating Systems (SCCS). Specification in SGCCS consists of both graphical and visual components. We give an example of the visual syntax and basic concepts of SGCCS, then we formalize the syntax textually using special terms. Further, we define semantics of SGCCS via mapping of these terms into SCCS expressions.

1 Introduction

In [5], a graphical coordination language for synchronous coordination of heterogeneous components was presented. We implemented a simple editor for this language ([20]) with support for translation of a graphical specification of a coordination model into the Calculus of Communicating Systems (CCS [17]).

Due to our participation on a current project of a formal design of a hardware IPv6 router, we have been motivated to extend our graphical editor to support real-time design. It lead us to extend semantics of the coordination model supported by GCCS to deal with time. We adapted a notion of a synchronous extension of CCS (SCCS, [16]). We extended the GCCS language to its synchronous version SGCCS.

In this paper, we would like to present SGCCS as a graphical formalism for formal modeling of coordination of heterogeneous components, for which time plays a critical role. We assume time being discrete and global. All of the coordinated components are observed as performing their actions in so

[1] This work has been supported in part by the Grant Agency of Czech Republic grant No. 201/00/1023.
[2] Email: xsafran1@fi.muni.cz

called time-slots. Any time-slot can be seen as a list of actions to be performed simultaneously at a discrete tick of the global clock. In other words, actions of all coordinated components are non-atomic and are performed synchronously. In fact, this concept models non-interleaving behavior of components [9]. We can distinguish between simultaneous and interleaved firing of actions in this model.

It is worth noting that there are two different notions of synchrony in the concept of our language. First one is the concept of synchronous communication of components in the sense of instantaneous handshake or multicast interaction among them, and the second one is the concept of time-slots we have mentioned above.

The most of existing coordination models [4] are based on the idea of modeling of asynchronous coordination of various software components interacting with each other usually in a distributed environment [18]. Call for formal languages specialized to specification of coordination is high in present, i.e., because of the Internet and the increasing occurrence of distributed systems. In general, there are not many coordination languages, which handle real-time properties [14]. Unfortunately, there exist systems, for which the abstraction of time and non-interleaved actions is not appropriate. If we deal with large systems composed of some components for which time is critical, we need formal tools suitable for specification of coordination of their timed behavior. Being formally specified, this systems can be verified using formal methods [7] and automated tools such as The Concurrency Workbench [19], which supports model checking for SCCS. In addition, it is very helpful to support this methods with visual formal languages.

There exist some graphical formalisms for modeling of real-time systems. One of them is the language GCSR [6], which is based on the idea of asynchronous execution of interleaved actions (as in Timed Algebras [8]). There is also a large group of graphical formalisms based on the notion of Statecharts [10], e.g., its extension Timed Statecharts [13]. Another well-known graphical formalism are Petri-Nets and their extensions (e.g., [12]). In general, we cannot easily distinguish between the coordination and the behavioral level of specification in these formalisms. Coordination aspects are closely related with behavioral aspects and cannot be simply separated. In other words, these formalisms cannot be simply used as a coordination "umbrella" for components specified in different formalisms.

An example of a visual coordination language which is aimed to model communication among components with full abstraction from their behavior is Visifold [3]. This diagrammatic formalism is based on the Manifold language which adapts the notion of asynchronous communication [1]. The concept of ports we use in SGCCS is similar to that one used in Manifold.

Using the language SGCCS presented here we can model synchronous coordination in systems composed from heterogeneous components, for which discrete real-time properties and non-interleaving character of actions are nec-

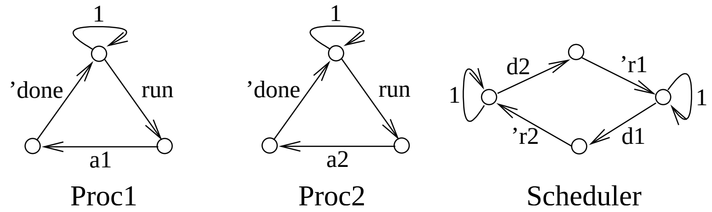


Fig. 1. The process level of SGCCS

essary to handle. SGCCS has the ability to be useful for modeling both synchrony and asynchrony "features" of systems together in one formalism. This strong expressiveness is taken from Milner's synchronous CCS. It is important to note, that SGCCS is not a programming language, as the most of coordination languages are (e.g. CORBA, Linda, JavaSpaces). SGCCS should be viewed in the same way as its asynchronous version GCCS, that is, it should be taken as a design language.

2 Overview of Synchronous GCCS

We present SGCCS here as an exogenous coordination language [2] supporting only signal interaction among components. The data-passing communication extension is the aim of our future work. Because our language is theoretically based on the calculus SCCS, it can be easily seen that the data-passing feature is quite a natural extension to the concept described in this paper.

As in GCCS, systems in SGCCS are graphically specified at two levels – the *process* level and the *network level* (the *hierarchy* of networks). At the process level the behavior of a component is specified using a transition diagram with input and output signals (so called actions in the CCS style). An example of a process level specification can be seen in figure 1. A special *1*-action denotes an internal action. The environment cannot interact with a particular component which performs this action. There is no restriction to use only this notion of modeling behavior of components. The network level can be taken as a coordination umbrella for components specified in various formalisms.

Coordination relations among components are specified using so called *nets*. Components are included in a net as boxes, which represent *interfaces* with places for communication – so called *ports*. Ports export the component actions of the same name to the environment. Unlike the concept of ports in Manifold, we consider ports as bi-directional. Thus, both input and output signals can go through them. The synchronous model allows to signal arbitrarily many actions simultaneously via ports of a particular component. Even any action can be instantaneously replicated to arbitrary many copies some in the input form and others in the output form and signaled in a particular time-slot through its port.

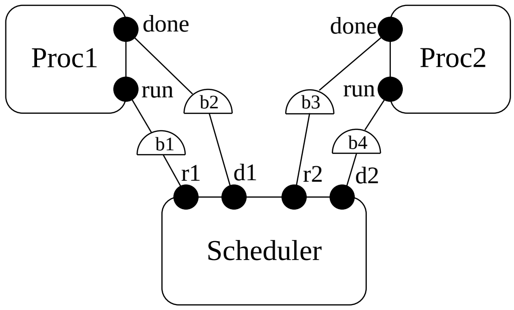


Fig. 2. The network level of SGCCS

As can be seen in figure 2, components are connected by ports to so called *buses*. Buses have two main meanings. At first, they export actions involved on ports interconnected by buses to be observed by the environment relabeled. So the actions run and $'r1$ of the components $Proc1$ and $Scheduler$ in figure 2 can be seen as actions $'b1$ or $b1$ by the environment of the net. The environment can be specified by another net, so called the higher level net. The current net is embedded into the higher level net as one of its components. This creates the hierarchy of nets, which can be viewed as a tree. In the root of the tree is the most abstract net.

Another meaning of buses is interaction among components. In SGCCS, we distinguish two types of interaction – synchronization and asynchronous communication. By the term *synchronization* we mean handshake between two components. Due to the concept of synchronous firing of actions, all desired handshakes are sensed to be performed in a particular time-slot. In general, we can distinguish between two kinds of synchronization. Firstly, we can strictly require a synchronization to be performed in the current time-slot, we call this *non-delayed synchronization*. Secondly, we can leave the components, which cannot synchronize in the current time-slot, to wait until the synchronization will be possible. We call this kind of synchronization the *delayed synchronization*.

We have two types of buses in SGCCS, we call them *synchronous buses* and *asynchronous buses*. Synchronous buses are represented by half-ellipses and asynchronous buses by ellipses. Components $Proc1$ and $Scheduler$ of a scheduler example in figure 2 are connected by two synchronous buses. It means they can participate in synchronization, if the actions to be currently fired on ports connected to bus $b1$ or $b2$ make an input-output pair (run, $'r1$ or $'done$, $d1$). In general, arbitrarily many components can be connected by their ports to a particular bus. All possible input-output synchronization pairs are performed in a time-slot leading to an internal *1*-action. If two components can synchronize with two other components, then the non-deterministic choice of particular simultaneous synchronization pairs is applied. It is important to note, that the maximal set of all possible synchronization pairs is satisfied in the current time-slot.

We can leave any bus unlabeled. This has the effect of forcing synchronization. Imagine we replaced the bus $b1$ in figure 2 with an unlabeled bus and we removed all the *1*-transitions in all diagrams in figure 1. If $Proc1$ is at the actual time-slot in the state of going to perform the action run and $Scheduler$ is just going to perform $'r2$ to start the process $Proc2$, then the whole system is deadlocked. The reason is, that the first process has to perform action run in the current time-slot, but it can be done only in a pair with action $'r1$ of the scheduler. Unfortunately, because of the actual state of the scheduler, this action cannot be performed. The solution for this problem is just the explicit reflexive *1*-transition of $Proc1$, which allows synchronization to be postponed till the time-slot in which the scheduler performs $'r1$ will come. Such a delayed synchronization cannot lead to deadlock, but it can fall to live-lock. The concept of delayed synchronization is equivalent to synchronization in CCS.

In contrast, asynchronous buses present different model of communication. More specifically, asynchronous buses represent a virtual mailboxes, which model non-blocking message passing. A sending process can leave in the bus a message for receiving process by performing an output action, and continue with firing next action. Its counterpart can take the message by an input action whenever in one of the following time-slots.

In conclusion, we can model both synchronous and asynchronous interaction among components. This property makes our coordination model universal and useful for specification of complex systems with heterogeneous concepts of interaction.

3 Definition of Synchronous GCCS

3.1 Syntax

To formalize syntax of the graphical objects we have presented in the previous section, we define their formal textual counterparts as two types of SGCCS *terms* – so called *nets* and *lists*. Following definition is based on the definition of GCCS terms which was published in [5].

At first, we have to set up a suitable notation and present some basic definitions.

3.1.1 Ports and interfaces

Let $\mathcal{A}$ denote a countable set of *names of ports*, so that $1 \notin \mathcal{A}$. It is worth noting that we do not need to distinguish between ports and their names because the port names occurring in an interface of a particular component are unambiguous. As we will see later, if we consider the scope of a particular net, each port will be given unambiguously by its interface. We denote $a, a_1, a_2, \ldots$ the members of $\mathcal{A}$.

We define *interface* of a component i as a finite subset of port names. Formally, interface $I_i \subseteq \mathcal{A}$ is a finite set of port names.

3.1.2 Buses

Let $\mathcal{B}$ be a countable set of *buses*. We use $b, b_1, b_2 \ldots$ as the notation for its members. To treat asynchronous and synchronous buses separately, we define two countable sets $\mathcal{B}_{sync}$ – representing *synchronous buses*, and $\mathcal{B}_{async}$ – representing *asynchronous buses*, so that $\mathcal{B} = \mathcal{B}_{sync} \cup \mathcal{B}_{async}$, and satisfying $\mathcal{B}_{sync} \cap \mathcal{B}_{async} = \emptyset$.

Unfortunately, each net may contain arbitrarily many buses sharing the same name. Each occurrence of a particular bus specifies different connections among subsystems in a net, therefore we need to distinguish between the set of buses and the set of bus names.

Taking this problem into account, we take a countable set of *bus names* $\mathcal{B}_n = \mathcal{A} \cup \{\epsilon_1, \epsilon_2, ...\}$, where for any $i \in \mathcal{N}$, $\epsilon_i \notin \mathcal{A}$ is an implicit name of an *internal bus*. Additionally, we define a mapping function $bus : \mathcal{B} \to \mathcal{B}_n$ which for any bus returns its name.

3.1.3 Actions

As in SCCS, we distinguish between *input* and *output particle actions*. Naturally, we derive particle action names from the set of port names. We use the notation $\Sigma = \mathcal{A}$ for a set of input particles and $\bar{\Sigma} = \{\bar{a} | a \in \mathcal{A}\}$ for a set of output particles.

We will use the notation $l, l_1, l_2, \ldots$ for members of the set $\Sigma \cup \bar{\Sigma}$. Further, we define an internal *1*-action, $\mathit{1} \notin \Sigma \cup \bar{\Sigma}$. Let generate the action structure *Act* from a set $\Sigma \cup \bar{\Sigma} \cup \{\mathit{1}\}$ using the operation '$\cdot$', satisfying:

- $\forall \alpha, \beta \in Act.\, \alpha \cdot \beta = \beta \cdot \alpha$
- $\forall \alpha, \beta, \gamma \in Act.\, \alpha \cdot (\beta \cdot \gamma) = (\alpha \cdot \beta) \cdot \gamma$
- $\forall \alpha \in Act.\, \alpha \cdot \mathit{1} = \mathit{1} \cdot \alpha = \mathit{1}$

We call members of *Act* *composite actions* and use the notation $\alpha, \beta, \gamma, \ldots$ for them.

- We denote any particle $\bar{l} \in \bar{\Sigma}$ as $l^{-1} \in Act$. For any particle $l \in \Sigma \cup \bar{\Sigma}$ we define its inversion in *Act* as $l^{-1} \in Act$, so that $l^{-1} = \bar{l} \in \bar{\Sigma}$ is a particle complementary to l and $(l^{-1})^{-1} = l$. We claim $l \cdot l^{-1} = \mathit{1}$.
- The inversion can be simply extended to deal with composite actions:

$$(\alpha \cdot \beta)^{-1} = \beta^{-1} \cdot \alpha^{-1} = \alpha^{-1} \cdot \beta^{-1} \qquad \alpha \cdot \alpha^{-1} = \alpha^{-1} \cdot \alpha = \mathit{1}$$

Let us note that now we can view the action structure $(Act, \cdot, ^{-1}, \mathit{1})$ as an Abelian group. It allows us to write composite actions in the form $\forall n \in \mathcal{N}.\, a_1^{z_1} \cdot a_2^{z_2} \cdot \cdots \cdot a_n^{z_n} \in Act$, where $a_i \neq a_j$ are mutually distinct particles for any $i \neq j$, $\forall i \in \{1, ..., n\} : z_i \in \mathcal{Z} \setminus \{0\}$ are non-zero integer powers, and for any $z < 0$, a^z denotes $(a^{-1})^{|z|} = \bar{a}^{|z|}$.

We define a homomorphism $ports : Act \to \mathcal{P}(\mathcal{A})$ mapping any composite action $\alpha \in Act$ to the set of relevant port names:

(i) for any particle $a \in \Sigma \cup \bar{\Sigma}$, $ports(a) = ports(a^{-1}) = \{a\}$

(ii) $ports(1) = \emptyset$

(iii) for any $\alpha, \beta \in Act$, $ports(\alpha \cdot \beta) = ports(\alpha) \cup ports(\beta)$

Definition 3.1 We define a set $\mathcal{G}$ of SGCCS *terms.*

- A tuple $\mathcal{P} = \langle Q, \rightarrow, q_0, q \rangle$, which represents a configuration of a nondeterministic finite state labeled transition system defined as tuple $\langle Q, q_0, Act_Q, \{\xrightarrow{\alpha} \mid \alpha \in Act_Q\}\rangle$, where
  - $\cdot$ $Act_Q \subseteq Act$ is a sub-group of Act,
  - $\cdot$ Q is a finite set of states, $\rightarrow \subseteq Q \times Act_Q \times Q$ is a finite transition relation,
  - $\cdot$ $q_0 \in Q$ is the initial state, and $q \in Q$ is the current state,

  is a term of the type list, $\mathcal{P} \in \mathcal{G}$. We say $\mathcal{P}$ is a *list-term.*
- A tuple $\mathcal{M} = \langle \bar{N}, B, L \rangle$, where
  - $\cdot$ $\bar{N} = \langle \langle S_1, I_1 \rangle, ..., \langle S_n, I_n \rangle \rangle$, $n \in \mathcal{N}$, is a finite sequence of n pairs, each of them represents the ith *subsystem* (S_i, $i \in \{1, ..., n\}$) embedded into its *interface* $I_i \subseteq \mathcal{A}$. Each of the subsystems $S_i \in \mathcal{G}$ is an arbitrary net-term or list-term.
  - $\cdot$ $B \subseteq \mathcal{B}$, $B = B_{sync} \cup B_{async}$ is a finite set of *buses*, where
    $B_{sync} \subseteq \mathcal{B}_{sync}$ is a finite set of *synchronous buses*
    $B_{async} \subseteq \mathcal{B}_{async}$ is a finite set of *asynchronous buses.*
    Note that $B_{sync} \cap B_{async} = \emptyset$.
  - $\cdot$ $L \subseteq \{1, ..., n\} \times \mathcal{A} \times \mathcal{B}$ is a finite set of *links*, satisfying
    $\forall \langle i, a, b \rangle \in L$: $a \in I_i$
    $\forall \langle i, a_1, b_1 \rangle$ and $\langle j, a_2, b_2 \rangle \in L : (i = j \wedge a_1 = a_2) \Rightarrow b_1 = b_2$,

is a term of the type net, $\mathcal{M} \in \mathcal{G}$. We say $\mathcal{M}$ is a *net-term.*

Remark 3.2 Let $\mathcal{M} = \langle \bar{N}, B, L \rangle \in \mathcal{G}$ be a term. We will use the notation $B(\mathcal{M}) = B$ for the set of all buses of the net $\mathcal{M}$. If $\mathcal{M}$ is a list-term, then we define $B(\mathcal{M}) = \emptyset$.

We defined the notion of terms. One can admit that the definition is not strict enough to include only terms for which the intended semantics could be sensed. We allow this freedom because we would like to support modular design with the opportunity of specifying reusable components. The possible problem can be seen in the following example.

Example 3.3 Assume we have a net with two free ports a_1 and a_2 not connected to any bus, and assume this net has no buses of the name a_1 nor a_2. Now imagine we have added this net as a component to a higher level net. Suppose the interface of this component includes ports a_1, a_2, a_3. Now what is the semantics of the port a_3? This anomaly can happen if we modify a component during the design phase, so that a component which had all the three ports is replaced with a two ports component without changing its interface, which is fixed in the higher level net.

We solve this problem by defining a set of all visible names that can be exported by a component and then we define a notion of the so called well-defined SGCCS term.

Definition 3.4 Let $\mathcal{M} = \langle\langle\langle S_1, I_1\rangle, ..., \langle S_n, I_n\rangle\rangle, B, L\rangle \in \mathcal{G}$ be a term. For any $i \in \{1, ..., n\}$ we define a set $\mathcal{I}(i) := \{x \in I_i \mid \forall b \in B(\mathcal{M}) : \langle i, x, b\rangle \notin L\}$ of all *free port names* included in the interface I_i.

Definition 3.5 Let $\mathcal{M} \in \mathcal{G}$ be a term. We define the set $\mathcal{A}(\mathcal{M})$ of all *visible names* in $\mathcal{M}$ as follows:

- If $\mathcal{M} \equiv \langle Q, \rightarrow, q_0, q\rangle$, assuming that $\mathcal{M}$ is defined as a configuration of a LTS with labels in $Act_Q \subseteq Act$, then

$$\mathcal{A}(\mathcal{M}) := \bigcup\{ports(\alpha) \mid \alpha \in Act_Q\}.$$

- If $\mathcal{M} \equiv \langle\langle\langle S_1, I_1\rangle, ..., \langle S_n, I_n\rangle\rangle, B, L\rangle$, then

$$\mathcal{A}(\mathcal{M}) := \bigcup_{i=1}^{n} \mathcal{I}(i) \cup \{bus(b) \mid b \in \bigcup_{i=1}^{n} B(S_i) \wedge bus(b) \notin \{\epsilon_1, \epsilon_2, ...\} \\ \wedge \exists j \in \{1, ..., n\}, a \in \mathcal{A} : \langle j, a, b\rangle \in L\}.$$

Definition 3.6 We say a *term* $\mathcal{M} \in \mathcal{G}$ is *well-defined*, if

(i) $\mathcal{M} \equiv \langle Q, \rightarrow, q_0, q\rangle$

(ii) or $\mathcal{M} \equiv \langle\langle\langle S_1, I_1\rangle, ..., \langle S_n, I_n\rangle\rangle, B, L\rangle$, satisfying
- $\forall i \in \{1, ..., n\} : I_i \subseteq \mathcal{A}(S_i)$
- all of the sub-terms $S_1, ..., S_n$ are well-defined terms.

The constraint presented by the last definition can be sensed as a base for consistency checking of the syntax of a graphical specification.

Remark 3.7 Let us note that from this moment we will assume the set of terms $\mathcal{G}$ to be restricted only to the subset of well-defined terms.

3.2 Semantics of SGCCS

We assume Λ is a set of SCCS expressions as defined in [16]. We define semantics of SGCCS terms indirectly by mapping them to SCCS agents. We denote this mapping as $\psi : \mathcal{G} \rightarrow \Lambda$. In other words, for any well-defined term $\mathcal{M} \in \mathcal{G}$ we are looking for a SCCS expression $\psi(\mathcal{M}) \in \Lambda$. The LTS semantics of this SCCS expression then will define the intended semantics of the term $\mathcal{M}$.

3.2.1 Semantics of list-terms

In the last section we defined list-terms of SGCCS as configurations of a finite state LTS $\langle Q, q_0, Act_Q, \{\xrightarrow{\alpha} \mid \alpha \in Act_Q\}\rangle$. The following lemma constructively transforms this LTS to a closed system of SCCS definitions.

Lemma 3.8 *For any finite state LTS* $\mathcal{S} = \langle Q, q_0, Act_Q, \{\stackrel{\alpha}{\rightarrow} \mid \alpha \in Act_Q\}\rangle$ *there exists a closed system of SCCS definitions* $\{q_i \stackrel{df}{=} E_i \mid i \in \{0, ..., n-1\}\}$*, where* $n \in \mathcal{N}$ *is a number of states in* $\mathcal{S}$*.*

Proof. The principle of the construction is based on a breadth-first search algorithm running on a graph representing the LTS $\mathcal{S}$. Starting in the initial state q_0 we transform each state to an SCCS agent according to the following scheme. Assume q is the current state.

- If $\mathcal{S}$ cannot perform any transition from q, then $\mathtt{q} \stackrel{df}{=} nil$.
- If $\mathcal{S}$ can perform transitions $q \stackrel{\alpha_1}{\rightarrow} q'_1$, $q \stackrel{\alpha_2}{\rightarrow} q'_2$, ..., $q \stackrel{\alpha_n}{\rightarrow} q'_n$, then we define an agent:

$$\mathtt{q} \stackrel{df}{=} \alpha_1 : \mathtt{q}'_1 + \alpha_2 : \mathtt{q}'_2 + \cdots + \alpha_n : \mathtt{q}'_\mathtt{n}$$

During each step we mark current state as a visited state. The fact that the state q_i was visited means q_i has been already assigned an expression $q_i \stackrel{df}{=} E_i$, so we can use that as an agent reference in the other definitions.

Finally, we have to ensure that the constructed system of definitions is closed. But this is implied by the fact that after visiting all the states there is a CCS definition for each of them. Additionally, no other agents are referenced in all expressions we used. □

Now it is straightforward to define the semantics of a list-term.

Definition 3.9 Let $\mathcal{M} = \langle Q, \rightarrow, q_0, q_i\rangle \in \mathcal{G}$ be a list-term defined as a configuration of a LTS $\mathcal{S}$. We define its semantics as the semantics of the SCCS agent q_i expressed as the system of SCCS definitions in the previous lemma. Assuming E_i is the expression from the definition $q_i \stackrel{df}{=} E_i$ we define the mapping $\psi(\mathcal{M}) = E_i$.

In general, we present SGCCS entirely as a language for synchronous coordination of processes, which may be specified using any formalism semantically "compatible" with a finite state transition system. That is, only we need to do before using the chosen process specification language is to define a transformation of its constructs to SCCS expressions, as we have done it for pure LTS.

To fulfill the formalization of the semantics of list-terms, it is important to note, that the sets of agent names representing list-terms of different processes must be mutually disjoint to avoid any conflicts.

3.2.2 Semantics of net-terms

Let $\mathcal{M} = \langle\langle\langle S_1, I_1\rangle, ..., \langle S_n, I_n\rangle\rangle, B, L\rangle \in \mathcal{G}$ be a well-defined term. We assume, with respect to the inductive definition of SGCCS terms, that any of sub-terms S_i for $i \in \{1, ..., n\}$ is represented as a SCCS agent $\mathtt{S_i}$.

We aim to set up a SCCS expression, whose semantics defines the semantics of the net $\mathcal{M}$. It is rather technical to present this construction fully formally, so we sketch it with help of figure 3.

In general, we define semantics of $\mathcal{M}$ as the semantics of a SCCS expression in the so called standard product form (the modification of the standard concurrent form defined in [17] replacing the "|" operator with "×"). More specifically, we deal with the product of all of the n agents representing the subsystems of $\mathcal{M}$. The intended semantics of coordination of all the components contained in the net is given using several morphisms and restrictions applied to the agents representing these components in the product. We consider morphisms and restrictions (pruning) defined for particles (see [9] for details).

Formally, assuming for any $i \in \{1, ..., n\}$ that $\mathtt{F}$ and $\mathtt{F_i} : Act \to Act$ are morphisms, and $\mathtt{R}$ and $\mathtt{R_i}$ sets of restricted particles, we map $\mathcal{M}$ to an agent in the following form.

$$\psi(\mathcal{M}) = ((\mathtt{S_1}\backslash\backslash\mathtt{R_1})[\mathtt{F_1}] \times \cdots \times (\mathtt{S_n}\backslash\backslash\mathtt{R_n})[\mathtt{F_n}])\backslash\backslash\mathtt{R}[\mathtt{F}]$$

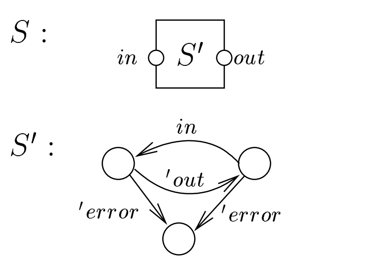
We have to treat sub-terms of the type list with special care (see figure 3a). The intended meaning is that the only allowed actions are those actions of a particular list-term, which has their ports in the interface of the list. In more detail, whenever a list-term $\mathcal{P}$ can perform an action $\alpha \in Act$, abstracting from any possible connections of ports to buses, it is performed only if all of the particles in α have their corresponding ports in the interface. Formally, for any list-term embedded into $\mathcal{M}$ as the ith sub-term, the condition $ports(\alpha) \subseteq I_i$ must hold to fire α. We express this behavior as a restriction $\backslash\backslash\mathtt{R_i}$ applied to each agent $\mathtt{S_i}$. The restricted particles are those not exported by any port in the interface the list-term is embedded into.

The next step is to treat subsystems interconnected together via buses. In particular, we aim to define semantics of interaction of components sharing a certain bus $b \in B$. We distinguish two types of buses, so let $b \in B_{sync}$ be a synchronous bus at first. Moreover, assume b is an internal bus ($bus(b) = \epsilon$). The example in figure 3b shows an instance of this situation. The intended semantics is to force synchronization of all the three components in the current tick of the global clock. More specifically, assuming that subsystems $\mathtt{S_1}, \mathtt{S_2}, \mathtt{S_3}$ are defined as $\alpha_1 : \mathtt{S}'_1, \alpha_2 : \mathtt{S}'_2, \alpha_3 : \mathtt{S}'_3$ and connected by ports a_1, a_2, a_3 to the internal synchronous bus b, the only allowed action is $\alpha_1\alpha_2\alpha_3 = \mathit{1}$. That is, if actions $\alpha_1, \alpha_2, \alpha_3$ form together a valid synchronization, an internal $\mathit{1}$-action is fired. The term "valid synchronization" is based on the fact, that each action to be performed on the port of any subsystem connected to b must have its inversion among the actions of other subsystems connected to the bus b.

The intended behavior can be formalized in SCCS using the product operator applied to agents representing the components to be synchronized. If we have k components ($1 \leq k \leq n$) connected by ports $a_1, ..., a_k$ to an internal synchronous bus $b \in B_{sync}$, $bus(b) = \epsilon$, we express that as the following agent:

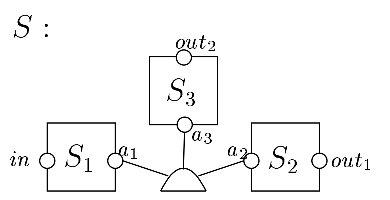
$$(\mathtt{S_1}[\mathtt{F_1}] \times \mathtt{S_2}[\mathtt{F_2}] \times \cdots \times \mathtt{S_k}[\mathtt{F_k}])\backslash\backslash\mathtt{R}$$

(a)



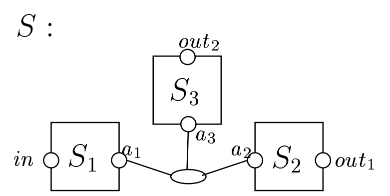
$$S \stackrel{df}{=} S'\backslash\backslash\{error\}$$

(b)



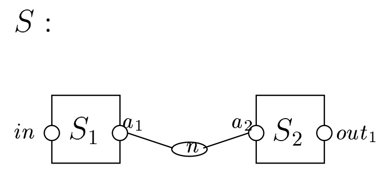
$$S \stackrel{df}{=} (S_1[\epsilon/a_1] \times S_2[\epsilon/a_2] \times S_3[\epsilon/a_3])\backslash\backslash\{\epsilon\}$$

(c)



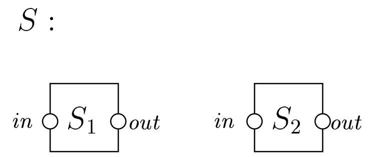
$$S \stackrel{df}{=} (S_1[tb_1/a_1] \times S_2[tb_2/a_2] \times S_3[tb_3/a_3] \times \mathtt{B}^3[tb_1/a_1, tb_2/a_2, tb_3/a_3, \epsilon/c])\backslash\backslash\{tb_1, tb_2, tb_3, \epsilon\}$$

(d)



$$S \stackrel{df}{=} (S_1[tb_1/a_1] \times S_2[tb_2/a_2] \times \mathtt{B}^2[tb_1/a_1, tb_2/a_2, tn/c])\backslash\backslash\{tb_1, tb_2\}[n/tn]$$

(e)



$$S \stackrel{df}{=} (S_1[\sigma_1/in, \sigma_2/out] \times S_2[\sigma_3/in, \sigma_4/out])[in/\sigma_1, out/\sigma_2, in/\sigma_3, out/\sigma_4]$$

Fig. 3. Principles of mapping of nets to SCCS agents.

Where each of the morphisms has the form $\mathtt{F_i} = [\epsilon/a_i]$, and $\mathtt{R} = \{\epsilon\}$.

Now assume the bus b is not an internal bus, let $bus(b) = n$, then the intended semantics is slightly different. Synchronization is not forcing in this case. That is, all the particles a_1, a_2, a_3 in actions $\alpha_1, \alpha_2, \alpha_3$ are relabeled to n in $\alpha_1\alpha_2\alpha_3$. Now synchronization can happen at the level of the higher net-term. The only difference in the resulting SCCS expression is in $\mathtt{R}$, which is now $\emptyset$. Another type of bus is an asynchronous bus. That is, let $b \in B_{async}$ and $bus(b) = n$. So we assume b is a labeled bus. Analogously to labeled synchronous buses, the intended semantics has two meanings. The first meaning is exporting actions relabeled to the bus name to the net-term one level higher in the SGCCS hierarchy. The second meaning is non-blocking message passing. We consider this as a non-deterministic choice between participating in communication at the current level or exporting actions higher in the net hierarchy. We express this behavior with help of a special SGCCS agent $\mathtt{B}^k$, which models an asynchronous bus with k ports. For two ports $\{a_1,a_2\}$, $\mathtt{B}^2$ is defined as follows:

$$\begin{aligned}\mathtt{B}^2 \stackrel{df}{=} & (\delta\bar{a}_2 \times a_1 : \mathtt{B}^2) + (\delta\bar{a}_1 \times a_2 : \mathtt{B}^2) \\ & + (\delta\bar{a}_1 \times c : \mathtt{B}^2) + (\delta\bar{a}_2 \times c : \mathtt{B}^2) \\ & + (\delta\bar{c} \times a_1 : \mathtt{B}^2) + (\delta\bar{c} \times a_2 : \mathtt{B}^2),\end{aligned}$$

where for any $i \in \{1,2\}$, $\delta a_i \stackrel{df}{=} \mathit{1} : \delta a_i + a_i : \mathbf{1}$. Recall that agent $\mathbf{1} \stackrel{df}{=} \mathit{1} : \mathbf{1}$, for more details see [17].

The first line of the definition of agent $\mathtt{B}^2$ is an agent which solves internal communication ($\mathtt{B}^2$ acts as an internal bus). The second and third lines define an agent allowing relabeling and inter-level communication.

To precise our construction, there is one problem we have to discuss. If any action α fired by an agent connected via a port a to a bus b has a particle a with the power z, then $|z| \geq 1$. To satisfy arbitrary composite action α, which could be of the form a^z, $\mathtt{B}^2$ should have in its sort a relevant complementary counterpart. The expansion of $\mathtt{B}^2$ to be done to fulfill this needs is rather technical. It is based on the idea of replacing each sub-agent of $\mathtt{B}^2$ (e.g., $\delta\bar{a}_1 \times a_2 : \mathtt{B}^2$) with a choice of agents fulfilling all the cases of what can happen on the port a_1. We deal with actions of the form $a_1^{z_1} \cdot \bar{a}_1{}^{z_2}$, where $z_1 \geq 0$ and $z_2 \geq 0$. There exists the upper bound max_z, so that $z_1 + z_2 \leq max_z$. The number max_z is given by a number of components of the relevant subsystem. Hence, the intended expansion of $\mathtt{B}^2$ is finite. We will describe this expansion formally later in this section.

The resulting agent $\mathtt{B}^2$ can be added to the product of agents interconnected by the bus b, as can be seen in figure $3d$. To avoid from any name conflicts communication actions are relabeled to new unambiguous names of the form $t*$.

Internal asynchronous buses can be viewed as special cases of labeled asynchronous buses we have defined above, assuming $bus(b) = \epsilon$. The intended meaning is forcing of internal non-blocking message passing between components. For example, one of systems S_1, S_2, S_3 in figure $3c$ can leave a message in the bus. In the same tick or in one of the following ticks, one of the remaining two systems can take this message. We express the behavior of an internal asynchronous bus also using the agent $\mathtt{B}^k$. The only difference is in embedding of this agent into the resulting product. Now we have to rename the action c to an internal unambiguous name of the bus ϵ. Then we apply a restriction of ϵ to the resulting product. Note that the effect of the sub-agents in the definition of $\mathtt{B}^k$, which contain relabeling action c, is annulled in this case. In figure 3c there is an example of a three-port internal asynchronous bus.

Finally, we have to set up formally semantics of the general bus with arbitrarily many ports. We aim to define agent $\mathtt{B}^k$, $k > 1$, which models a bus with k ports. The expanded agent should contain a choice of all possible combinations of all the k particles representing actions that can be performed on all ports of the bus. Each of these particles can appear in the tuple in the input or output form, and delayed or non-delayed. Formally, for $k > 1$ the agent $\mathtt{B}^k$ has the following form:

$$\begin{gathered}\mathtt{B}^k \stackrel{df}{=} \textstyle\sum_{i,j\in\{1,\ldots,k\},i\neq j} \quad \sum_{z\in Arr_{2,k}(\{1,\ldots,max_z\})}(E_i \times E'_j : \mathtt{B}^k) \\ + (E_i \times F'_j : \mathtt{B}^k) + \ (F_i \times E'_j : \mathtt{B}^k)\end{gathered}$$

where $Arr_{2,k}(\{1,...,max_z\})$ is a set of all arrays of the dimension $\langle 2,k\rangle$ containing natural numbers from the set $\{0,...,max_z\}$ as members, and satisfying the condition $\forall i \in \{1,...,k\}.\, 1 \leq z_{1,i} + z_{2,i} \leq max_z$. Expressions E_i, E'_j, F_i, and F'_j have the following form:

$$\begin{gathered}E_i \equiv \delta(\bar{a}_i^{z_{1,i}} a_i^{z_{2,i}}), \;\; E'_j \equiv a_j^{z_{1,j}} \bar{a}_j^{z_{2,j}} \\ F_i \equiv \delta(\bar{c}^{z_{1,i}} c^{z_{2,i}}), \;\; F'_j \equiv c^{z_{1,j}} \bar{c}^{z_{2,j}}\end{gathered}$$

Now we have to consider possible conflicts of port and bus names. See figure 3e. One problem can be caused by free ports of the same name. Implicitly, according to semantics of the SCCS product, this ports allow synchronization. We restrict this undesirable feature by relabeling this ports to brand new names to disallow synchronization, and applying the inverse morphisms to return back the exact names.

Another problem is in dealing with buses of the same name in a net-term. Note, that names of buses and names of free ports make together a set of visible names of a particular net-term. As we transformed the net-term into a SGCCS expression, we allowed synchronization of all agents connected to buses of the same name. To disable this, we analogously relabel buses names to new unambiguous names and then return back to their previous names via inverse morphisms. This mechanism also prevents buses from synchronizing with free ports of the same name. To achieve the correct behavior of internal buses, we defined a countable set of names for internal buses in the previous section.

Putting all together, we get an expression describing the net-term $\mathcal{M}$ at the current level of the net hierarchy. Considering the order of application of the principles showed above, it is necessary to ensure that it does not play any role. The main reasons for that are the associativity of the SCCS product and the fact that port names are unambiguous concerning the interface they belong to. Additionally, all the restrictions are applied locally (directly to sub-terms), or globally (directly to the whole product). If we consider relabeling, note that all the possible conflicts that could be caused by an incorrect composition of morphisms are avoided using unambiguous names.

Finally, branching through the whole hierarchy of SGCCS terms in the breadth-first search style recursively assigning expressions to subsystems, we get the complete SCCS description of the most abstract net-term.

4 Conclusion and future work

We have presented a graphical formalism SGCCS for synchronous exogenous coordination in discrete real-time. This language adds the notion of synchronous non-interleaved actions (so called composite actions) to the GCCS coordination language [5]. In contrast to asynchronous GCCS, two different types of buses are distinguished in SGCCS. Using this two constructs one can model both synchronous and asynchronous coordination. Thus, our language is suitable for modeling of coordination in systems, where actions are performed synchronously in time-slots, where the coordination is either synchronous or asynchronous, and where discrete time should be taken into account. We mean for example coordination of hardware components being measured by the frequency of a particular global clock. We believe that the universality of our coordination model could be also useful for more complex systems which combines, e.g., asynchronous software and synchronous hardware components.

The main advantage of SGCCS is that it is an exogenous coordination language, hence one can model the coordination architecture of components without any detail knowledge about their behavior. This allows abstraction and application of the top-down methodology during the design phase. Concepts such this one are common in the component-based design [11].

We are currently working on a synchronous extension of the graphical editor [20]. Thus, analogously to the editor for GCCS, we aim to transform a model represented graphically in SGCCS into SCCS according to the definition of SGCCS semantics we have presented in this paper. Hence, e.g., one will be able to apply the μ-calculus model checking and equivalence checking to that model using the Concurrency Workbench tool [19].

Considering future work, we are going to add the value-passing feature to our language, i.e., we aim to develop a type system for messages that can be sent through particular ports and buses. Another extension can be incorporating of some other graphical formalisms to the process level of our language, e.g., Petri-Nets, which allow modeling of non-interleaving [15], and Statecharts [10], which have the feature of hierarchical modeling at the behavioral level. Other types of buses could be also added to our formalism to support instantaneous multicast communication or other coordination mechanisms which are useful in hardware and software design and can be modeled in SCCS.

References

[1] Arbab, F., *The IWIM Model for Coordination of Concurrent Activities*, in: *Proceedings of 1st Int. Conf. on Coordination Models and Languages* (1996), pp. 34–56.

[2] Arbab, F., *What Do You Mean, Coordination?*, Bulletin of the Dutch Association for Theoretical Computer Science (1998).

[3] Bouvry, P. and F. Arbab, *Visifold: A Visual Environment for a Coordination Language*, in: *Coordination Languages and Models* (1996).

[4] Ciancarini, P., *Coordination Models and Languages as Software Integrators*, ACM Computing Surveys **28(2)** (1996), p. 300.

[5] Cleaveland, R., X. Du and S. A. Smolka, *GCCS: A Graphical Coordination Language for System Specification*, in: *Proceedings of Fourth International Conference on Coordination Models and Languages* (2000).

[6] D.Clarke, I. L., H. Ben-Abdallah and O. Sokolsky, *PARAGON: A Paradigm for the Specification, Verification, and Testing of Real-Time Systems*, in: *IEEE Aerospace Conference*, 1997.

[7] Edmund M. Clarke, O. G. and D. A. Peled, "Model Checking," Cambridge : MIT Press, 1999.

[8] Flavio Corradini, D. D. and P. Inverardi, *On the Relationschop among four Timed Process Algebras*, Fundamenta Informaticae **34** (1999).

[9] Gray, D., "Introduction to the Formal Design of Real-Time Systems," Springer, 1999.

[10] Harel, D., *Statecharts: A Visual Formalism for Complex Systems*, Technical report, The Weizmann Institute of Science (1987).

[11] Heineman, G. T. and W. T. Councill, "Component-Based Software Engineering: Putting the Pieces Together," Addison Wesley, 2001.

[12] J.Wang, "Timed Petri Nets, Theory and Application," Kluwer Academic Publishers, 1998.

[13] Kesten, Y. and A. Pnueli, *Timed and Hybrid Statecharts and their Textual Representation*, in: *Formal Techniques in Real Time and Fault Tolerant Systems* (1991).

[14] Lemniotes, T. A. and G. A. Papadopoulos, *Real-Time Coordination in Distributed Multimedia Systems*, in: *IPDPS 2000* (2000).

[15] Leucker, M., *On Model Checking Synchronised Hardware Circuits*, in: *Proceedings of 6th Asian Computing Science Conference* (2000), pp. 182–ff.

[16] Milner, R., *A Calculus for Synchrony and Asynchrony*, Theoretical Computer Science **25** (1983).

[17] Milner, R., "Communication and Concurrency," Prentice-Hall, 1989.

[18] Papadopoulos, G. A. and F. Arbab, *Modelling Activities in Information Systems using the Coordination Language MANIFOLD*, in: *Proceedings of the 1998 ACM symposium on Applied Computing* (1998).

[19] Stevens, P., "The Edinburgh Concurrency Workbench User Manual," (2001). URL `http://www.dcs.ed.ac.uk/home/cwb/`

[20] Šafránek, D., "Graphical Specification of Concurrent Systems (in Czech)," Master's thesis, Masaryk University, Brno (2001).