Coordination with Structured Composition for Cyber-physical Systems [1]

Simon MAURER [a,2], Raimund KIRNER [a]
[a] *School of Computer Science, University of Hertfordshire, Hatfield, UK*

Abstract. Structured programming has become a very successful programming paradigm as it provides locality of a program's control flow. Similar concepts of locality are desired for the specification and development of concurrent and parallel systems. In the domain of cyber-physical systems or embedded computing it is challenging to identify such structured compositions since control flow tends to be driven by concurrently acting reactive components with often circular dataflow relations.

In this paper we discuss foundations for a structured coordination language for cyber-physical systems. We study car platooning as a use case, exhibiting typical challenges of cyber-physical systems. Based on that use case we show a possible structured composition pattern and outline coordination network construction mechanisms.

Keywords. cyber-physical systems, coordination languages, structured programming

Introduction

The increasing system complexity of cyber-physical systems (CPS) drives a high pressure on software engineering methodologies to assure an effective and predictable control of resources. Coordination languages have been proposed as a solution to tackle this problem by decomposing application software into coordination and algorithmic programming [1]. In this paper we present coordination language constructs, based on streaming networks, suitable for CPS where complex interaction patterns pose a challenge for structured system composition.

As described by Arbab in [2], coordination models can be classified into endogenous and exogenous coordination, describing whether coordination is done from within a behavioural component or from the outside respectively. We further relate to the aspect of structured programming and focus on the ability of coordination languages to describe networks in a structured manner.

With respect to application aspects, we classify systems to either process data in transformative or reactive manner where CPS often fall into the latter

[1]The research leading to these results has received funding from the FP7 ARTEMIS-JU research project "ConstRaint and Application driven Framework for Tailoring Embedded Real-time Systems" (CRAFTERS).

[2]Corresponding Author: Simon Maurer, E-mail: s.maurer@herts.ac.uk

class. Additionally, we distinguish between systems relying on persistent state or systems following a functional behaviour.

Our aim is to provide exogenous language constructs that enforce structure, support reactive data processing, and allow persistent state within behavioural components. The proposed constructs are based on streaming networks where the topology of the network imposes dependencies between components and where message streams are coordinated within the network.

The reminder of this paper is structured as follows: In Section 1 we discuss classification aspects of coordination languages and relate them to the challenges of cyber-physical systems (CPS). In Section 2 we introduce the underlying coordination model and in Section 3 we describe network operators. Section 4 presents an example of a CPS where we apply the presented coordination concepts. Section 5 discusses related work and Section 6 concludes the paper.

1. Coordination of Cyber-physical Systems

The underlying model of the language constructs presented in this paper consists of three conceptual elements: the *computational components*, the *routing network*, and the *extra-functional requirements layer*. A computational component contains a part of the behavioural description of the application. Multiple computational components are connected and coordinated by the routing network which itself is composed of coordination components. The extra-functional requirements layer serves to annotate boxes and network elements with extra-functional requirements such as location information for distributed systems or real-time requirements in CPS.

The following Subsections discuss properties of coordination languages and requirements of languages targeting CPS.

1.1. Transformative vs. Reactive Data Processing

In contrast to a transformative system that takes inputs, performs computation, and produces outputs, CPS are often reactive systems where inputs are coupled to outputs via the environment. The output of a tansformative system can be formulated as a function of the state of the system and its input: $out = f(state, in)$. Reactive systems, on the other hand, additionally, have a relation where the input of the system is a function of its output and an unknown variable imposed by the environment: $out = f(state, in) \ \wedge \ in = f'(x, out)$.

The implications of a reactive system with respect to the presented coordination concepts are: support for bidirectional communication between computational components, a data-flow direction that can not be defined clearly on an application level, and computational components with a persistent internal state.

1.2. Exogenous vs. Endogenous Coordination Model

An exogenous coordination model assures a clear separation between coordination and behaviour by leaving the behavioural components oblivious of the coordination constructs that exert coordination upon them. An endogenous coordination

model, on the other hand, has no such separation and the coordination is done from within a behavioural component which makes the component aware of the coordination exert on it.

An exogenous model assures clear separation of concerns which is of great importance to simplify development, integration, and verification processes. In the domain of CPS where systems are often safety critical, this is a property we need to enforce.

1.3. Persistent State vs. Non-persistent State

The design of new applications in the domain of CPS often relies on legacy code. For this reason, state must be allowed in computational components and they cannot be restricted to purely functional behaviour. Reactive data processing needs state in general but with an exogenous coordination model the coordinated components can be kept free of persistent state. However, this comes at the price of efficiency because state information has to be communicated through the streaming network.

1.4. Structured vs. Non-structured Connectivity

A structured network provides a sense of locality and can be helpful to provide composability and compositionality of sub networks by keeping port-to-port connections local. In contrast to this, a non-structured network would rather be created with global port-to-port connection tables which are hard to read and understand.

For a modular development it is important to understand the behaviour of an isolated sub-entity, independent of where it is going to be integrated, which is why a structured approach is key.

1.5. Real-time Constraints and Analysability of Cyber-pysical Systems

Due to their interaction with the physical environment, some of the services of CPS do have real-time requirements. The specification of real-time properties of streaming networks is discussed by Kirner and Maurer in [3]. Further, the application domain of CPS is often strongly regulated and requires extensive verification of the design and implementation. To enable such verification processes, the system must be made analysable by the underlying design model.

The above mentioned extra-functional requirements layer aims to serve this purpose by introducing an abstraction between the application and the timing requirements. An example of such an abstraction is the Ptides [4] language. This aspect of the model will be comprised in future work and is not further discussed in this paper.

2. Components and Coordination Elements

In the presented coordination concepts, the computational components are not part of the coordination language and are written in a separate programming

language. The coordination components form the routing network which is used to connect multiple computational components and build a streaming network. In order to interface the coordination components, computational components are surrounded by a construct which is called *box* throughout this paper. A streaming network consists of *nets*, *channels*, and *synchronizers* which are either implicit or explicit instances of computational components or coordination components.

Nets and synchronizers have, possibly, multiple input or output ports. The most basic net is the instance of a box declaration. A box declaration assigns the input and output parameters of the computational component with the input and output ports of the net and links the box to the source code of the computational component. A box can be annotated as stateless if the computational component is purely functional. Inside a box declaration, ports can be merged with a *merge-synchronizer* as described in Subsection 2.1.

Nets are hierarchical and can include other nets and networks of nets. By connecting ports of nets together with directed channels a network is created. The connection semantics of nets is explained in Section 3 where network operators are described. The following Subsections describe synchronizers and discuss the problem of flow direction ambiguities.

2.1. Synchronizers

We distinguish between two types of synchronizers: the *copy-synchronizer* (Figure 2) and the *merge-synchronizer* (Figure 3).

The copy synchronizer has no explicit language element but is spawned implicitly whenever more than two ports are connected. It copies messages from its inputs to all of its outputs. The messages are made available at each output port simultaneously and thus, are synchronizing the earliest possible processing. Messages are read in arbitrary order and are processed non-deterministically. A copy-synchronizer allows a one-to-one, one-to-many, many-to-one, and many-to-many mapping of ports.

A merge-synchronizer is spawned explicitly but is only allowed inside a box declaration. It has multiple input ports and only one output port. The merge-synchronizer collects messages at its inputs and as soon as on all input ports a message is available it produces a tuple, containing a message from each input, at its output. This tuple of messages is forwarded to the computational component input where it is interpreted as an ordered set of input parameters. Inputs of the merge synchronizer can be *decoupled*, allowing the synchronizer to trigger, even if no new message is available on such a port. The necessary buffers and triggering semantics for such interfaces are described by the work of Maurer and Kirner in [5].

2.2. Flow Direction Ambiguities

As described in the beginning of this section, channels are directed. As nets and synchronizers can have multiple input and output ports, a connection between two components can be composed out of multiple channels with different directions. This results in a network where the flow-direction is not clearly defined

and channel directions may become ambiguous. Lets assume the box F with the following box declaration:

```
box F (in a, out a, in b, out b) on func_f
```

If this box is instantiated twice, say as the nets H and G, the connection of the two nets would be ambiguous as there are four possible ways to connect them. This is depicted in Figure 1.

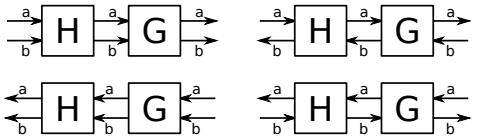

Figure 1. An example of an ambiguous flow direction. The two nets H and G can be connected in four different ways.

Note that the connection of H and G is only ambiguous when inspected locally. By unfolding the complete network and assuming that all ports are connected (this is a requirement for a network to be valid), the direction of each channel is clearly defined. However, this is not sufficient because in a structured program all parts of the program need to be unambiguous, independent of their surrounding elements.

In order to achieve an unambiguous flow direction or to help the programmer to structure the code, ports can be grouped into two separate collections: An *Up-Stream (US)* or a *Down-Stream (DS)* collection. The grouping of the ports into the two collections depends on the context of the program and each collection can hold any number of input or output ports or can be empty. The grouping provides a logical flow-direction due to the constraint that ports in an up-stream of one net can only connect to ports in a down-stream of another net and vice versa. The logical flow-direction is unrelated to the real flow-direction of messages as channels can be of arbitrary direction.

A third collection, called *Side-Ports (SP)*, exists but is explained in more detail in Section 3.4. Figure 4 depicts a schematic representation of a box declaration where ports are grouped into the three different collections.

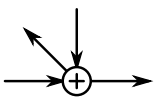

Figure 2. Copy-synchronizer

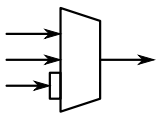

Figure 3. Merge-synchronizer with one decoupled input

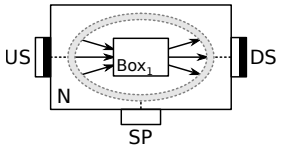

Figure 4. *Box* declaration

3. Network Operators and Port Connections

This section describes the operators that allow to interconnect components in a structured manner. We propose two basic grouping operators that allow to combine two nets by either forcing a connection or preventing a connection. The con-

nective grouping is called *serial composition* and the non-connective grouping is called *parallel composition*. Additionally, an explicit connection of nets via side-ports allows to either, connect a signal to most nets in the network (e.g. clock, reset, or logger), or to realise connections that cannot be achieved with the available network operators.

The main focus of the network operators is twofold: 1. the operators aim at structuring the network by providing a sense of locality, meaning that information necessary to understand a local part of the network is kept local. 2. connections between nets are to be kept explicit and not hidden by an implicit connection semantics of the operators.

A net N can be a single box in the simplest case or a combination of networks via network combinators. The communication interface of each net N consists of a set of ports: $\mathcal{P}(N)$. Each port $p_i \in \mathcal{P}(N)$ is either an input or output port, specified by a mode attribute: $p_i.mode \in \{input, output\}$. Based on that we define predicates for input and output ports as: $\mathcal{I}(N) = \{\ p_i \mid p_i \in \mathcal{P}(N)\ \wedge\ (p_i.mode = input)\ \}$ and $\mathcal{O}(N) = \{\ p_i \mid p_i \in \mathcal{P}(N)\ \wedge\ (p_i.mode = output)\ \}$.

A port p_i is identified by two parts: its signal name $p_i.signal$ and a location identifier $p_i.id$ where the latter is derived from the identifier of the net or box. Ports also have a field that indicates their belonging to a port collection, which can be either up-stream (US), down-stream (DS), or side-port (SP): $p_i.col \in \{US, DS, SP\}$. We use the following predicates for port collections: $\mathcal{US}(N) = \{\ p_i \mid p_i \in \mathcal{P}(N) \wedge (p_i.col = US)\ \}$, $\mathcal{DS}(N) = \{\ p_i \mid p_i \in \mathcal{P}(N) \wedge (p_i.col = DS)\ \}$, and $\mathcal{SP}(N) = \{\ p_i \mid p_i \in \mathcal{P}(N)\ \wedge\ (p_i.col = SP)\ \}$.

3.1. Parallel Composition

The parallel composition uses the operator '|'. The definition of a network N as the parallel composition of networks N_1 and N_2 is written as: '$N = N_1 \mid N_2$'.

No implicit port connection is performed with the parallel composition. Instead, the set of ports of the resulting net $N_1|N_2$ is the union of the set of ports of the two operators N_1 and N_2 with preserving their mode: $\mathcal{P}(N_1|N_2) = \mathcal{P}(N_1) \cup \mathcal{P}(N_1)$. Ports that are grouped in a port collection in the operands N_1 or N_2 are grouped in the same collection in the resulting net $N_1|N_2$: $\mathcal{US}(N_1|N_2) = \mathcal{US}(N_1) \cup \mathcal{US}(N_2)$, $\mathcal{DS}(N_1|N_2) = \mathcal{DS}(N_1) \cup \mathcal{DS}(N_2)$, and $\mathcal{SP}(N_1|N_2) = \mathcal{SP}(N_1) \cup \mathcal{SP}(N_2)$. The parallel composition is schematically represented in Figure 6.

3.2. Serial Composition

The serial composition uses the operator '.'. The definition of a network N as the serial composition of networks N_1 and N_2 is written as: '$N = N_1\ .\ N_2$'.

The down-stream ports of N_1 and the up-stream ports of N_2 of a serial composition $N_1.N_2$ are internally connected: $IntConn(\mathcal{DS}(N_1), \mathcal{US}(N_2))$. However, to be a valid internal connection and thus valid serial composition, the following properties of $\mathcal{DS}(N_1)$ and $\mathcal{US}(N_2)$ must hold:

$$\forall p_i \in \mathcal{DS}(N_1)\ \exists p_j \in \mathcal{US}(N_2).\ (p_i.signal = p_j.signal) \wedge IsCMode(p_i, p_j) \wedge$$
$$\forall p_j \in \mathcal{US}(N_2)\ \exists p_i \in \mathcal{DS}(N_1).\ (p_i.signal = p_j.signal) \wedge IsCMode(p_i, p_j)$$

where the predicate $IsCMode(p, p')$ ensures that input/output are connected:

$$IsCMode(p, p') : (p.mode = input \ \wedge \ p'.mode = output) \ \vee$$
$$(p.mode = output \ \wedge \ p'.mode = input)$$

Knowing which ports are internally connected, the resulting externally visible port sets of $N_1.N_2$ are defined as follows:

- Side-ports of N_1 or N_2 are grouped in the side-port collection of $N_1.N_2$: $\mathcal{SP}(N_1.N_2) = \mathcal{SP}(N_1) \cup \mathcal{SP}(N_2)$.
- Ports in the down-stream collection of N_1 are grouped in the down-stream collection of $N_1.N_2$ and ports in the up-stream collection of N_2 are grouped in the up-stream collection of $N_1.N_2$: $\mathcal{DS}(N_1.N_2) = \mathcal{DS}(N_1)$ and $\mathcal{US}(N_1.N_2) = \mathcal{US}(N_2)$.

For a network expression formed out of network operators to be valid, all ports that are not grouped in the side-port collection must be connected. For a network to be valid, all ports (including ports grouped in the side-port collection) must be connected. The schematic representation of a serial composition is shown in Figure 7.

3.3. Operator Precedence

The order of precedence of the operators is defined as follows: The serial composition precedes the parallel composition.

An example where the precedence is applied is shown in Figure 5. On the left side two sequences $A_1.B_1.C_1$ and $A_2.B_2.C_2$ are created which are then joined by the parallel composition. On the right side the parallel compositions are enforced by the parentheses, resulting in a serial composition with full connectivity.

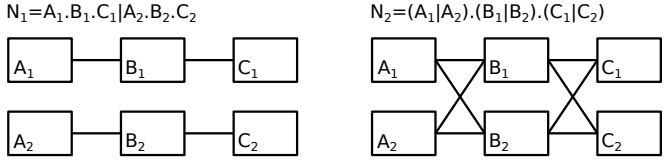

Figure 5. Two examples of connection graphs where the operator precedence is illustrated.

3.4. Side-port Connections

Side-port connections are explicit connections between matching ports in multiple side-port collections of different nets. Lets assume the box declarations

```
1  box F(out a, side in clk)
2  box G(in a, out a, side in clk)
3  box H(in a)
4  box Clk(side out clk)
```

and the two independent nets $F.G.H$ and Clk. An explicit connection of the signal clk can now be declared, connecting all ports in the side-port collections of the nets listed in the declaration block:

```
connect clk {*} // equivalent to 'connect clk {F, G, Clk}'
```

The symbol '*' is used to connect the signal to all nets that have matching ports in their side-port collection but ignores those that have not.

3.5. Scopes

Scoping of network structures is performed by explicit net declarations. A net declaration can contain box declarations, other net declarations, explicit connections, nets, and networks of nets. All elements declared inside a net declaration are not accessible from the outside. A net declaration is instantiated as a net when it is used in a network expression, equivalent to the instantiation of box declarations. A net declaration links, similar to a box declaration, ports available from elements inside the net to ports that are made available on the outside. The ports can optionally be grouped in different collections and can be renamed. A schematic representation is depicted in Figure 8.

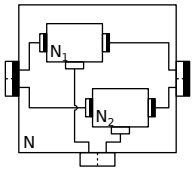

Figure 6. *Parallel composition*

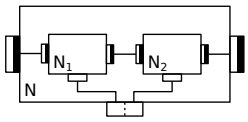

Figure 7. *Serial composition*

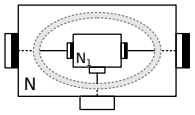

Figure 8. Scoping mechanism of a *Net*

4. Cyber-physical Systems - An Example

In this section we demonstrate vehicle platooning as an example of cyber-physical systems with different interactions of components. The basic idea of vehicle platooning is to coordinate the cruising speed of, in series driving, vehicles to achieve a more resourceful driving. Bergenhem et al. describe different types of platooning systems [6]. In Figure 9 we show a possible interaction scenario of different car components, relevant to vehicle platooning, with a particular focus on the braking mechanism only.

The three horizontally aligned boxes represent different cars driving in series. For Car_i further internals of the braking control are shown. The *anti-lock braking system* (ABS) receives a control signal for a desired braking action but the ABS then decides on its own when to assert and release braking pressure (B) based on feedback over the revolution sensors of the wheels (RS). Besides the manual braking control (MB) we assume an automatic distance control system which uses distance sensors (DS) to measure the distance to its front vehicle and starts automatic braking requests to keep a certain minimal distance.

What we see from this example is that communication between components is bi-directional and individual components act as reactive systems on their own. Such communication patterns cannot be mapped to acyclic directed computation graphs. Figure 10 depicts the car-platooning example with a compositional structure and Figures 11 and 12 show the code describing the problem. The network

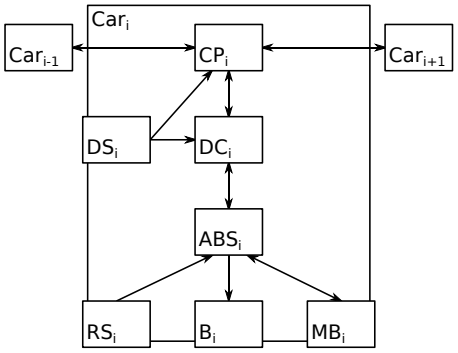

Figure 9. The car platooning example with port-to-port connections.

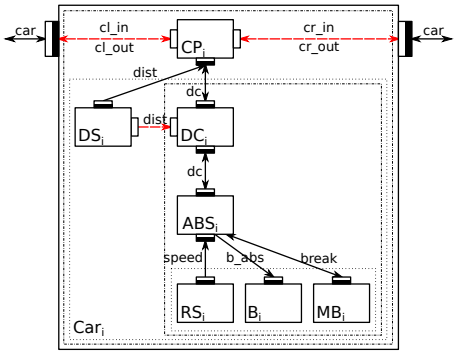

Figure 10. Structured representation of the car platooning example.

operators achieve a structuring of the network by implicitly grouping components together and keeping port connections local. A net declaration *Car* allows to group networks together and create local port-to-port assignments.

```
1   box DC ( side in dist, up down in dc,
          up down out dc)
2     on func_dc
3   box ABS ( in speed, in break, up in
          dc, up out dc, out b_abs, out
          breack)
4     on func_abs
5   stateless box DS (up side out dist)
6     on func_ds
7   stateless box RS (out speed)
8     on func_rs
9   stateless box B (in b_abs)
10    on func_b
11  box MB (in break, out break)
12    on func_mb
```

Figure 11. Global box declarations for the car platooning example.

```
1   net Car { up in car (cl_in), up out
          car (cl_out), down in car (cr_in
          ), down out car (cr_out) }
2   {
3     connect dist {DS, DC}
4
5     box CP ( down in dist, down in dc,
          down out dc, side in cl_in, side
           out cl_out, side in cr_in, side
           out cr_out)
6       on func_cp
7
8     ((RS|B|MB).ABS.DC|DS).CP
9   }
```

Figure 12. The net description for one car.

5. Related Work

A first approach of coordinating streaming applications was introduced with StreaMIT [7], a language that allows to create a loosely structured streaming network by interconnecting computational components, called Filters, with network constructors such as split-join, feedback, or simple one-to-one streams. The fully fledged coordination language S-Net [8] is also based on streaming networks but unlike StreMIT, S-Net is exogenous and achieves a tighter structuring with network operators similar to the operators presented in this paper.

Other than the coordination languages listed above, the concepts presented in this paper target reactive systems with persistent state. Other approaches with the focus on the design of CPS are the language Giotto [9] and the Ptolemy [10] framework. Giotto is based on a time-triggered paradigm and targets applications with periodic behaviour, while the presented coordination model supports mixed timing semantics. The focus of the Ptolemy project lies on providing a unified modelling framework that allows to integrate and compose heterogeneous real-time systems but does not enforce separation of concerns.

6. Summary and Conclusion

In this paper we discussed coordination concepts based on streaming networks suitable for cyber-physical systems. One aspect of the language concepts is to make network structures locally visible to the reader of a coordination program and not to hide connectivity information through implicit connection processes. Another aspect is the enforcement of structure in networks by using a structured programming approach. We showed that this approach, designed for transformational and functional systems, can also be applied for reactive systems where components have persistent state. We applied the language on a car platooning application and presented a concise way of modelling the coordination aspect of the application.

The future work consist of defining a fully fledged coordination language, adding a layer with extra-functional requirement specifications to the model, and building a runtime system for the language.

References

[1] Edward A. Lee. Cyber physical systems: Design challenges. In *Proc. 11th IEEE International Symposium on Object-oriented Real-time distributed Computing*, pages 363–369, Orlando, Florida, USA, May 2008.

[2] Farhad Arbab. Composition of Interacting Computations. In Dina Goldin, Scott A. Smolka, and Peter Wegner, editors, *Interactive Computation*, pages 277–321. Springer Berlin Heidelberg, January 2006.

[3] Raimund Kirner and Simon Maurer. On the Specification of Real-time Properties of Streaming Networks. In *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung*, Kärnten, Austria, October 2015.

[4] Patricia Derler, Thomas Huining Feng, Edward A. Lee, Slobodan Matic, Hiren D. Patel, Yang Zhao, and Jia Zou. PTIDES: A Programming Model for Distributed Real-Time Embedded Systems. Technical Report UCB/EECS-2008-72, EECS Department, University of California, Berkeley, May 2008.

[5] Simon Maurer and Raimund Kirner. Cross-criticality Interfaces for Cyber-physical Systems. In *Proc. 1st IEEE Int'l Conference on Event-based Control, Communication, and Signal Processing*, Krakow, Poland, June 2015.

[6] Carl Bergenhem, Steven Shladover, Erik Coelingh, Christoffer Englund, and Sadayuki Tsugawa. Overview of Platooning Systems. In *Proceedings of the 19th ITS World Congress, Oct 22-26, Vienna, Austria (2012)*, 2012.

[7] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *Compiler Construction*, number 2304 in Lecture Notes in Computer Science, pages 179–196. Springer Berlin Heidelberg, January 2002.

[8] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. Asynchronous Stream Processing with S-Net. *International Journal of Parallel Programming*, 38(1):38–67, February 2010.

[9] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A Time-Triggered Language for Embedded Programming. In *Embedded Software*, number 2211 in Lecture Notes in Computer Science, pages 166–184. Springer Berlin Heidelberg, January 2001.

[10] J. Eker, J.W. Janneck, E.A Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming Heterogeneity - The Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.