Communication Analysis for System on Chip Design *

A. Siebenborn, O. Bringmann, W. Rosenstiel

FZI Forschungszentrum Informatik
Haid-und-Neu-Str. 10-14
76131 Karlsruhe, Germany
[siebenborn,bringmann]@fzi.de

Universität Tübingen
Sand 13
72076 Tübingen, Germany
rosenstiel@informatik.uni-tuebingen.de

Abstract

In this paper we present an approach for analysis of systems of parallel, communicating processes for SoC design. We present a method to detect communications that synchronize the program flow of two or more processes. These synchronization points set the processes into relation and allow the determination of the global timing behavior of such a system. Using the results of our method for communication analysis, we present a new method to detect communications that might produce conflicts on shared communication resources. This information can be used for the assignment of communication resources.

1. Introduction

The complexity of systems is steadily increasing, and already today, there are designs consisting of several processor cores with different instruction sets, communicating by on-chip busses or network-on-chip. The design of such a system starts with a functional description of parallel communicating processes. The single, parallel blocks will be mapped to processor cores or specialized hardware; the communications will be mapped to communication resources, like on-chip busses, network-on-chip, or will be realized by dedicated wires. The temporal behavior of such a system is not easy to determine, especially if communications block the execution of single processes. Whereas for single software tasks, Worst-Case Execution Time (WCET) analysis [7, 8] is sufficient, it is not applicable on systems of communicating processes. In case of shared communication resources, like busses, conflicts on the shared medium may occur that have an effect on the timing behavior of the system. We developed a method for timing analysis that combines a method for WCET analysis with communication analysis to determine the influence of blocking communication on the timing behavior of the system. The information gained can be used to determine the Worst-Case Response Time (WCRT) of the system relative to certain input and output signals. We present a method to identify communications, which produce potential conflicts on a shared communication medium. In further design steps, this information could be used, to map conflicting communications on different independent communication media, or adjust the timing behavior accordingly. The method is not restricted to communication resources and can be applied to other shared resources, if the access is considered as a communication. This makes the method very useful in the area of SoC design.

The paper is structured as follows. In the next section we give an overview on other works in the area of communication analysis. In Section 3 we present our method for communication analysis and illustrate the method by an example. In Section 4 we introduce our method, to detect communications with conflicting access to shared resources.

* This work was partially supported by the DFG priority programs on Embedded Systems SPP 1040 and the BMBF/MEDEA+ project SpeAC

2. Related Work

Existing approaches on communication analysis can be distinguished depending on the underlying model of computation and the addressed application domain. Many approaches introduced in the area of distributed software systems for modeling and analysis of concurrent, communicating systems are based on Petri nets, process algebras and communicating automata. The prerequisite for all presented approaches is the ability to incorporate timing information of the underlying communication network. Therefore, all un-timed approaches (e.g. un-timed Petri nets) are not discussed in the following. Petri nets provide a universal technique for modeling concurrent systems. While early approaches are mainly simulation-based, modern approaches are based on efficient analysis techniques able to cope with timed Petri nets [10, 17]. However, the applied communication types have to be modeled implicitly, so that the analysis is performed without knowledge on concrete communication types, processes, communication channels and the corresponding places and transitions.

A promising approach for analyzing worst-case timing behavior of concurrent systems is based on communicating automata [14], an extension to timed automata [1, 3]. However, these approaches rely also on totally synchronous communications. This quite restrictive model reduces the possible degree of concurrency and is not realistic for communication with buffers and latencies in the communication channel.

Beside all discussed approaches addressing distributed systems, several approaches exist tailored to the requirements of real-time systems. Here, we have to distinguish between hardware-oriented and software-oriented approaches. Typically, hardware-oriented approaches assume a parallel execution of the specified processes on dedicated resources, while software-oriented processes assume a sequential execution of the processes on a single processor by use of task scheduling. The combination of both aspects are tackled by software-oriented approaches addressing parallel and distributed real-time systems, where all specified processes have to be mapped onto several processing elements [18].

A lot of software-oriented approaches abstract from the internal process behavior and operate on a acyclic task graph. This model bases on the assumption, that each task has a statically determined execution time, and each task starts execution once all input signals are available. The main drawback of this model is the missing support of conditional control structures. Therefore, newer approaches extend the basic task graph model by adding control dependencies [11, 16]. But due to the acyclic structure, different communication protocols and data-

dependent loops can not be modeled.

One of the first global hardware-oriented approaches that analyze the timing behavior of a system of communicating processes is presented in [6]. The system is modeled according to its original specification. This approach is able to handle loops by a bottom-up evaluation of communicating loop bodies in the loop hierarchy. It is not applicable for systems with data dependent loops and branches. In addition, it relies on a correct specified system, without data loss and dead locks.

For a restricted class of applications, first analytical approaches exist for timing separation of discrete events [15, 9]. The main problem of these approaches are the exclusion of conditional behavior in the control flow and the restriction to synchronous communications.

Recent approaches address general hardware/software platforms, very often with respect to user-specified I/O event models [12, 5] . The main problems are caused due to the focus on the I/O stream behavior and the missing consideration of functional and communicational dependencies of the application.

In [13] an analysis technique has been presented that combines methods for WCET analysis with an approach for communication analysis for hardware synthesis [4]. However, latencies on the communication channel and possible conflicts on shared communication resources have not been considered until now.

3. Communication Analysis

The approach presented in this paper, allows the validation of the real-time behavior of concurrent software processes. Hereby two methods that work in two different problem domains are combined: (1) communication analysis and (2) static timing analysis. Static timing analysis handles code sequences with control structures, including loops with bounded iteration counts [7, 8]. Communication that blocks process execution and loops with unknown iteration counts can not be handled. On the other hand, for communication analysis only communication points and the timing behavior between those nodes are of interest. For that reason, the problem has to be decomposed for the two analysis domains.

The first step during this decomposition, is to find and classify communications in the system. Usually, communications are encapsulated by functions, like `send()` and `receive()`. Accordingly, these function calls represent the nodes of the communication dependency graph *CDG*, which is the base for communication analysis. An edge between two nodes in the *CDG* exists, if a path in the control flow graph exists that connects the two communication nodes, without including any other communication point. The minimum and maximum latency between two nodes is determined by static timing analysis. For that purpose, a sub graph of the control flow graph has to be build up, which contains a communication node as start and end node. The sub graph contains all paths between the corresponding communication nodes, that do not include further communication nodes. These sub graphs are determined by performing depth-first-search, once starting at the start node, and once starting at the end node, passing edges in the opposite direction. Hereby all reachable nodes are marked. The sub graph contains all nodes that are reachable from the start node and from which the end node can be reached, i.e. the intersection of the two sets of marked nodes.

Considering the synchronization behavior, there exist four basic communication primitives. The first primitive is a communication, where neither the sender, nor the receiver suspends execution and waits for the communication partner. This type of communications effects no synchronization between communication partners and has no influence on the timing behavior of the single processes. Nevertheless, an execution time analysis can help to detect possible data loss, i.e. buffer overflow, if the sender sends more often than the receiver can process the data. Second case is a blocking at both sides of the communication. In this case, synchronization between communicating processes is guaranteed, but also resources may be wasted. The two further possibilities are a non-blocking sender and a blocking receiver and vice versa. Here synchronization between processes is possible, but not necessarily the case. Our communication analysis tackles this type of communication. It allows to determine the communications that synchronize processes and have an influence on the timing behavior. These communications are denoted as *synchronization points SP* in the following. Even the time can be determined during that a process is stalled, due to blocking communication.

For the communication analysis only information on the communications, and the temporal behavior between these communications is of interest. This information can compactly be represented as a communication dependency graph *CDG*. The nodes in the *CDG* represent communication points. This graph contains two types of edges: Edges that represent the control flow between communication points, and edges that represent the communications. It can be defined as follows:

Communication Dependency Graph (CDG) A communication dependency graph is denoted by
$CDG := \langle V_{CDG}, E_{CDG}, E_{COM}, \tau_{CDG}, l_{CDG} \rangle$, where

- V_{CDG} is a set of nodes representing communication nodes or loops with unbounded data-dependent delay.
- $E_{CDG} \subseteq V_{CDG} \times V_{CDG}$ is a set of directed edges describing the precedence dependencies between nodes.
- $E_{COM} \subseteq V_{send} \times V_{rec}$, with $V_{send} = \{v \in V_{CDG} : \tau_{CDG}(v) \in \{send_{async}, send_{sync}\}\}$ and $v_{rec} = \{v \in V_{CDG} : \tau_{CDG}(v) \in \{receive_{async}, receive_{sync}\}\}$ is a set of directed edges describing the communication.
- The function $\tau_{CDG}(v) : V_{CDG} \rightarrow \{send_{async}, send_{sync}, receive_{async}, receive_{sync}, init\}$ denotes the type of each node.
- The edge weights are represented by the function $l_{CDG} : E_{CDG} \rightarrow \mathbb{N}_0 \times \mathbb{N}_0$ with minimal and maximal execution time $l_{CGD}(v_1, v_2) = (cs_{min}, cs_{max})$ between two nodes $v_1, v_2 \in V_{CDG}$.

A communication dependency graph *CDG* is a directed, cyclic graph, which can be constructed based on the *CFG* of each process. The edges e_{com} are given by the communications. Edges e_{cdg} represent the control flow between two communication points in the *CFG*. An edge e_{cdg} between two nodes in the *CDG* exists, if there exists a path in the *CFG* between the corresponding basic blocks. The latencies c_{min}, c_{max} are attributed to the edges e_{cdg}, which are the execution times of the longest and shortest path between the corresponding nodes in the control flow graph.

3.1. Synchronicity Conditions

Based on the communication dependency graph, a condition can be formulated that has to be fulfilled for synchronization points. Each communication pair (v_s, v_r) is a potential candidate for a synchronization point. An obvious condition is, that the blocking communication participant is reached before the non-blocking participant. The set of nodes on the shortest path from v_1 to v_2 is denoted by $path_{min}(v_1 \rightsquigarrow v_2)$. Analogically the set of nodes on the longest acyclic path is denoted by $path_{max}(v_1 \rightsquigarrow v_2)$. The function $L(p)$ means the sum of all edge latencies l on a path p.

A communication $C = (v_s, v_r)$ with $(v_s, v_r) \in E_{COM}$ is a synchronization point if

$$L(path_{max}(v_{sync} \rightsquigarrow v_r)) \leq L(path_{min}(v'_{sync} \rightsquigarrow v_s)) \quad \forall (v_{sync}, v'_{sync}) \in SP_{pre}((v_s, v_r))$$

Herein the set $SP_{prec}((v_s, v_r))$ refers to all previous synchronization points from which the communication nodes v_s or v_r can be reached directly, without passing an other synchronization point. This synchronization condition represents an actual criterion for the verification of existing synchronization points. However, for the detection of synchronization points the criterion is not easy to apply, due to the cyclic dependency that are introduced by loops in the processes.

3.2. Synchronization Point Detection

The main task during the communication analysis is the determination of all synchronization points, such that the condition formulated in section 3.1 holds. Since this condition is based on an already determined set of synchronization points, a constructive algorithm is needed. The algorithm is divided into two phases. In the first phase, initially synchronous communications are determined, which fulfill the synchronization condition between the reset state and the treated communication nodes. In the second phase of algorithm the processes are inspected, after their initiation is completed. The objective is, to determine the minimum number of wait states of each receive node. If the result is negative, then the corresponding communication is not a synchronization point. This can be done by formulating the synchronization conditions as a system of linear equations. As already shown in section 3.1, there is a cyclic dependency introduced by loops. At this point, we introduce the term communication cycle $Cycle(P_i, P_j)$, which is the ordered set of all communications between two processes P_i and P_j. With this ordered set we can define a relation $\prec$, which means for two communications $C_1, C_2 \in Cycle(P_i, P_j)$, with $C_i \prec C_j$, that communication C_1 is reached before C_2 for all possible input data. The cyclic dependency is resolved the way, that first all communications of a communication cycle are assumed to be synchronization points, and the synchronicity condition is checked for the cycle. If one communication does not fulfill the synchronicity condition, it will be removed from the cycle, and all condition are checked again. The method terminates, if all communications of all cycles fulfill the synchronicity condition, such as they form a ring closure. To prove, that the communication points in the detected cycles are really synchronization points the synchronicity condition is checked, based on the initial synchronization points. To describe the problem by a set of equations, minimum and maximum slack variables are introduced, describing minimum and maximum number of wait states at the communication points. The synchronicity condition from section 3.1 is not fulfilled, if the result for a slack variable becomes negative. There are two types of synchronicity equations (SE), corresponding to the minimum slack variables $\underline{x}_i$ and the maximum slack variables $\overline{x}_i$:

$$\begin{aligned} \underline{SE}(C_{isync} \rightsquigarrow C_i) : L(path_{max}(v_{isync} \rightsquigarrow v_r)) \\ = L(path_{min}(v'_{isync} \rightsquigarrow v_s)) + \underline{x}_i \\ \overline{SE}(C_{isync} \rightsquigarrow C_i) : L(path_{min}(v_{isync} \rightsquigarrow v_r)) \\ = L(path_{max}(v'_{isync} \rightsquigarrow v_s)) + \overline{x}_i \end{aligned}$$

In Figure 1 the algorithm of our method is shown. The algorithm is quite similar for the determination of initial synchronization points and repetitive synchronization points. For the initial synchronization point, communication cycles with init-nodes of the *CDG* are considered. For the determination of repetitive synchronization points, the communication cycles do not contain init-nodes.

In a first analysis step, we determine the communications, where program synchronization can be guaranteed. As a result, we get the minimum slack variables $\underline{x}_i$. If the constructed equation system is not solvable, due to an inconsistency, a deadlock in the system is possible. If no synchronization point between processes is found, this indicates possible data loss. The synchronization points determined in the first step, are the base to calculate the maximum slack variables $\overline{x}_i$ in a second step. The variables $\overline{x}_i$ provide us with the necessary information to calculate a WCRT, related to specific input and output signals of the system.

At the first sight it seems to be obvious, that an optimization problem has to be solved to determine all slack variables. But since the calculation of a slack variable is based on the preceding synchronization point, only the paths between this two communications have to be considered, so that only the Gauss algorithm has to be applied, to solve the equation system.

```
function InitSync(CDG) return ISP;
  set τ_C(C) = NC for all communications C ∈ E_COM;
  set τ_C(C) = SP for all a priori known synchronization points;
  set ISP = {C : τ_C(C) = SP};
  compose the set of all communication cycles CYCLE;
  while ∃Cycle ∈ CYCLE ∧ C ∈ Cycle : τ_C(C) = NC do begin
    LES = ∅;
    //setup equation system
    for each Cycle ∈ CYCLE : Cycle ≠ ∅ do
      if ∃C ∈ Cycle : τ_C(C) = SP
           ∧∃C' ∈ nxtCom_Cycle(C) : τ_C(C') = NC then
        LES = LES ∪ {SG(C ⇝ C')};
    solve LES;
    // evaluate solution
    if x_i ≥ 0  ∀x_i ∈ solution(LES) then
      for each x_i ∈ var(LES) do
        τ_C(C_i) = ISP; ISP = ISP ∪ {C_i};
    else
      remove all communication C_j ∈ Cycle_j if: x_j < 0;
  end;
end;
```

Figure 1. ***Algorithm for synchronization point detection***

3.3. Latencies on the communication channel

Until now, we did not consider latencies on the communication channel. However in a real system the transmission of a message will need a certain amount of time, depending on the amount of data, and there will be a certain delay between sending and receiving a message. For that reason, the parameters l_i for the message delay and d_i for the message duration will be added to the communication edges e_{com} of the *CDG*. For the determination of synchronization points, these values have to be taken into account. Communication partners leave synchronization points no longer at the same time, but time shifted by the delay d_i. Also, the communication duration has to be considered for the determination of path latencies between communication points. This becomes clear with the example in Figure 2. In this case the following equation can be set up:

$$\begin{aligned} \underline{SE}_1(I \rightsquigarrow C_1) : & \quad 2 + \underline{x}_1 = 3 + l_1 \\ \underline{SE}_2(I \rightsquigarrow C_2) : & \quad 2 + \underline{x}_1 + d_1 = 7 + l_2 \\ \underline{SE}_3(C_2 \rightsquigarrow C_3) : & \quad 7 + l_3 = 3 + \underline{x}_3 \end{aligned}$$

In equation $\underline{SE}_1$ the delay l_1 has to be added, since the message reaches the receiver delayed by this value, which has to be taken into account for the synchronicity condition. In $\underline{SE}_3$ the delay l_3 has to be added, since S_2 and R_2 are left time shifted by l_3. Besides the latency l_2, in $\underline{SE}_2$ the message duration d_1 of communication C_1 has to be added, while l_1 is contained in x_1.

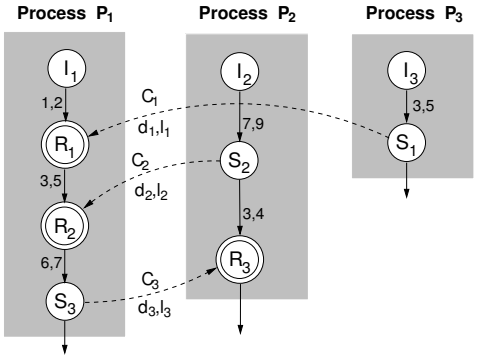

Figure 2. ***Communication Latency and Duration***

3.4. Determination of the WCRT

The determined synchronization points and the slack variables can be used to determine the WCRT of a system. The influence of parallel processes to one process of the system is contained in the slack variables. To determine the worst-case turn around time of one process, only the path latencies of this processes together with the slack variables have to be considered.

3.5. Example

To illustrate the method, we give an example in the following section. Figure 3 shows a system with four communicating processes. The example is subdivided into the following steps:

1. Determine synchronization points and min. slack variables $\underline{x}_i$.
2. Determine max. slack variables $\overline{x}_i$.
3. Determine worst case response times.

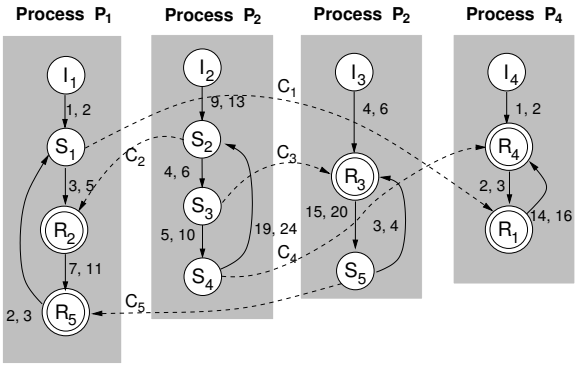

Figure 3. ***Example with four parallel processes***

With respect to the algorithm presented in Figure 5, first all initial synchronization points will be determined.

$$\begin{aligned}
&\underline{SE}_1(I \rightsquigarrow C_1): && 1 = 2 + \underline{x}_4 + 3 + x_1\\
&\underline{SE}_2(I \rightsquigarrow C_2): && 1 + 5 + \underline{x}_2 = 9\\
&\underline{SE}_3(I \rightsquigarrow C_3): && 9 + 4 = 6 + \underline{x}_3\\
&\underline{SE}_4(I \rightsquigarrow C_4): && 9 + 4 + 5 = 2 + \underline{x}_4\\
&\underline{SE}_5(C_2, C_3 \rightsquigarrow C_5): && 11 + \underline{x}_5 = 4 + 15
\end{aligned}$$

These equations lead to the solution $\underline{x}_1 = -20$, $\underline{x}_2 = 3$, $\underline{x}_3 = 7$, $\underline{x}_4 = 16$ and $\underline{x}_5 = 8$. Since $\underline{x}_1$ is a negativ value, communication C_1 is not a synchronization point. However, not any synchronization equation is based on communication C_1, so the other results can be used. Based on these initial synchronization points the repetitive synchronization points are determined, taking backward edges into account:

$$\begin{aligned}
&\underline{SE}_1(C_3 \rightsquigarrow C_3): && 5 + 19 + 4 = \underline{x}_3 + 15 + 4\\
&\underline{SE}_2(C_4 \rightsquigarrow C_4): && 19 + 4 + 5 = 3 + 16 + \underline{x}_4\\
&\underline{SE}_3(C_3, C_5 \rightsquigarrow C_2): && 5 + 19 = 15 + 2 + 5 + \underline{x}_2\\
&\underline{SE}_4(C_2, C_3 \rightsquigarrow C_5): && 11 + \underline{x}_5 = 4 + 15
\end{aligned}$$

This system has the solution $\underline{x}_2 = 2$, $\underline{x}_3 = 9$, $\underline{x}_4 = 9$ and $\underline{x}_5 = 8$. The determined minimum slack variables $\underline{x}_i$ can be used to determine best-case execution times. To determine worst-case execution times, the maximum slack variables have to be determined. The equation system is set up by considering the longest path to the sender and the shortest to the receiver:

$$\begin{aligned}
&\overline{SE}_1(C_3 \rightsquigarrow C_3): && 10 + 24 + 6 = 20 + 3 + \overline{x}_3\\
&\overline{SE}_2(C_4 \rightsquigarrow C_4): && 24 + 6 + 10 = 2 + \overline{x}_1 + 14 + \overline{x}_4\\
&\overline{SE}_3(C_2, C_3 \rightsquigarrow C_5): && 7 + \overline{x}_5 = 6 + 20\\
&\overline{SE}_4(C_3, C_5 \rightsquigarrow C_2): && 10 + 24 = 20 + 3 + 3 + \overline{x}_2
\end{aligned}$$

With this values the maximum turn around time of the system can be determined by considering only the latencies and slack variable of one process. The influence of other processes is contained in the slack variables. The turn around time can be calculated just considering Process 1 as $3 + 3 + \overline{x}_2 + 7 + \overline{x}_5 = 40$. Worst-case latencies between arbitrary communication points in the system can be determined. For that purpose, a path between these nodes has to be chosen, which can contain communication edges in both directions and control flow edges in the right direction. As an example, considering the worst case delay between S_2 and S_5 the following pathes can be chosen:

$$P_1 : S_2 \rightarrow R_2 \rightsquigarrow R_5 \leftarrow S_5$$
$$P_2 : S_2 \rightsquigarrow S_3 \rightarrow R_3 \rightsquigarrow S_5$$

If a path leads from a receiver to a sender, the slack variable of the receiver has to be added. Both paths produce the same result:

$$P_1 : 7 + \overline{x}_5 = 7 + 19 = 26$$
$$P_2 : 6 + 20 = 26$$

4. Access to Communication Channels

The information gained during communication analysis can be used to determine conflicts on shared communication media, e.g. busses. The determined synchronization points set the parallel processes into relation and allow to put all communications into a temporal order. To determine conflicts on shared communication resources it is necessary to define the time interval, during that a channel is used. Depending on the communication protocol, different cases are possible.

1. Only the sender allocates and releases a communication channel.
2. Only the receiver allocates and releases a channel.
3. The channel is allocated by the receiver and released by the sender.
4. The channel is allocated by the sender and stays occupied until the receiver can take the information. If the communication is a synchronization point, the receiver is reached before the sender and the channel is only allocated for the duration of the communication.

In case of non-synchronization points, the third case is the most complicated and needs an extension to the basic algorithm.

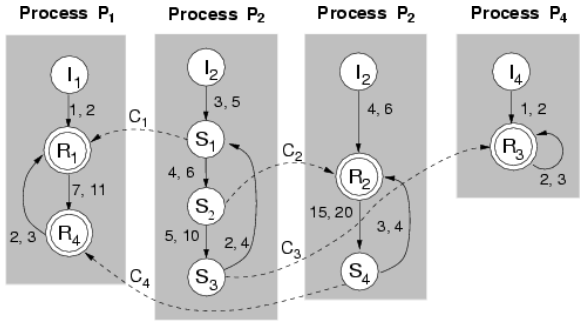

Figure 4. ***Possible conflicts on communication resources***

Basic Algorithm The min. and max. latencies of all paths, together with the determined slack variables allow to calculated absolute time intervals for each communication. However, considering only absolute time intervals to determine possible conflicts on communication resources is not sufficient. For example, the communications C_1 and C_2 in Figure 4 are executed during the time intervals $[3, 5]$ and $[4, 6]$. Even if these intervals overlap, it is obvious, that these communications will not be executed at the same time. That means, that it is necessary to consider also the order of the communications, that is determined by the control flow of the processes. Based on the synchronization points, that set the parallel processes of the system into relation, a global temporal order can be determined. This order can be represented by a graph: Parallel communications can be grouped to one node; edges denote the order of the communication and are determined by the control flow of the program.

Communication Schedule Graph A communication schedule graph of a process is denoted by: $CSG := \langle V_{CSG}, E_{CSG} \rangle$, where

- V_{CSG} is a set of nodes with $v \subseteq E_{COM} : \forall v \in V_{CSG}$ where E_{COM} is the set of communications in the *CDG*.
- $E_{CSG} \subset V_{CSG} \times V_{CSG}$ is a set of directed edges describing the precedence dependencies between nodes.

This graph, denoted as communication schedule graph (CSG), is set-up in two steps:

1. The execution order of the communications is determined, considering only the control flow of the processes.
2. Considering path latencies between communications, potential parallel communications are checked and if necessary, the graph is reordered.

In the first step, only the order of communication points in the *CDG* is considered. During this step, path latencies are ignored. In principle, the algorithm is shown in Figure 5. It starts with initial synchronization points as the first node in the communication schedule graph. The current communication points for each process, i.e. the system state, are stored in an array A. The algorithm terminates, if the sequence of A's is recurring. To test this criterion, A has to be stored in a set S for each iteration of the algorithm. The current A is stored on a stack $\mathbf{V}$, together with the current node v of the CSG. The function $update()$ determines a set $\mathbf{A}$ of system states A, based on a previous system state A_{prev}. Since the control flow graph may include data dependent branches, $update()$ returns a set of state vectors $\mathbf{A}$, one for each branch in the control flow graph. State vectors, that are already contained in S, will not be considered for further iteration steps. Figure 6 shows the CSG for the example in Figure 4. Since there are no data dependent branches

```
A_0 := [I_1, I_2, ..., I_n];
S := {A_0};
V.push(⟨v_0, A_0⟩);
while V ≠ ∅ begin
  ⟨v_prev, A_prev⟩ := V.pop();
  A := update(A_prev);
  for each A_cur ∈ A
    if A_cur ∉ S begin
      v_cur := ∅;
      for each c_send, c_rec ∈ A_cur with ⟨c_send, c_rec⟩ = e ∈ E_com then
        v_cur = v_cur ∪ {e};
      create edge e = ⟨v_prev, v_cur⟩;
      V.push(⟨v_cur, A_cur⟩);
      S := S ∪ {A_cur};
    end;
  end;
end;
```

Figure 5. ***Construction of Commmunication Schedule Graph***

in this example, *update()* returns just one state vector for each iteration of the main loop. The underlined entries mark communciations, where both communication partners are contained in the array. Since the state vector A_3 contains the two communications C_3, C_4, these communications form one node in the CSG and are possibly executed at the same time.

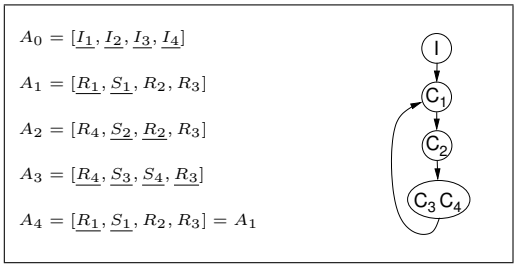

Figure 6. ***Communication Schedule Graph for the Example in Figure 4***

In the second step, with the knowledge on path latencies, and based on the determined synchronization points, potentially parallel communications can be checked, if theire time intervalls are overlapping. Since C_2 in the example is a synchronization point, the control flow will leave S_2 and R_2 at the same time, so only the path latencies $S_2 \rightsquigarrow S_3$ and $R_2 \rightsquigarrow S_5$ have to be considered. Since the time intervalls for these pathes do not overlap ($[5, 10] \cap [15, 20] = \emptyset$), C_3 and C_4 can not be executed at the same time and accordingly the *CSG* can be reordered.

Figure 7 shows an easy example with a data dependend branch. The resulting CSG is shown in Figure 8. Due to the branch $\mathbf{A}$ contains two elements in the second iteration step. In the third iteration step the algorithm terminates, since all elements already appeared, i.e. are contained in S.

Non-Synchronization Points Until now the communications are considered to "happen", when both, receiver and sender of a communication are reached, i.e. both communication partner are contained in the state vector A. However, in the case of non-synchronization points such a system state does not exist. In the example of Figure 3, communication C_1 is not a synchronization point, i.e. the non-blocking sender is reached before the blocking receiver. In this case, the algorithm has to be slightly extended: The sender of non-blocking communications is not written to the state vector, but is stored in a set and is removed from the set, when the receiver is reached. Communication inside this set are considered to be parallel to all other communications. The resulting

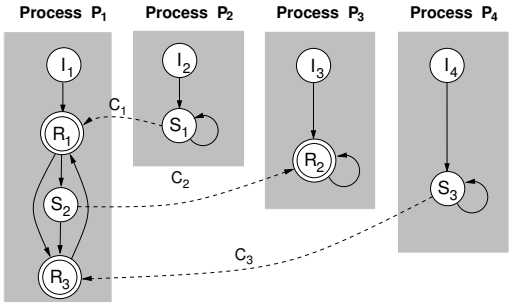

Figure 7. ***Possible conflicts on communication resources***

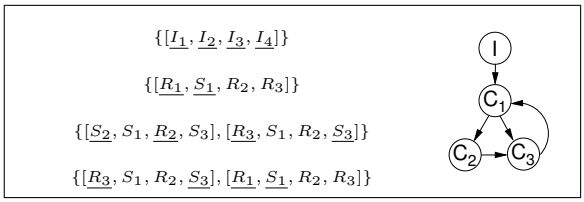

Figure 8. ***Communication Schedule Graph for the Example in Figure 7***

CSG for the example in Figure 3 is shown in Figure 9. Since C_1 is not a synchronization point, S_1 is moved to the set of non-synchronization points N in the first iteration step, where it remains until R_1 is reached. As long as S_1 is contained in N, C_1 is parallel to the other communication and is contained in the nodes of the CSG. In the example S_1 would be removed after the fourth iteration step, the same time it is added again, since the control flow reaches S_1 again.

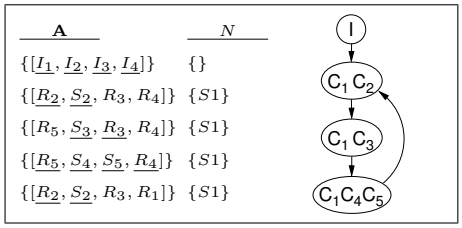

Figure 9. ***Communication Schedule Graph for the Example in Figure 3***

Message Duration Possible parameters for the temporal behavior of communications have been introduced in Section 3.3. The communication latency l_{com} does not influence the execution order of the communications, as long sender and receiver are left at the same time. However, if there is a delay between sending and receiving a message, i.e the sending process leaves a synchronization point before the receiving process, it might be possible, that this communication is parallel to subsequent communications. In this case, the communication delay has to be compared with the latency to the next communication point. If the communication delay is longer than the path latency, the two succeeding communications might be executed in parallel.

5. Conclusion

Our method for communication analysis allows the detection of synchronization points in systems of parallel, communicating processes. These synchronization points set the processes into temporal relation and allow the determination of temporal properties of the system, e.g the WCRT. The information can be used to detect conflicting access to communication resources. This information is important for the design of SoC consisting of several components that comunicate with each other via shared communication resources, like busses or network-on-chip. The method is not restricted to communication resources. It can be applied to determine conflicting access to arbitrary shared resources, if this access is considered as a communication. This opens a broad area for applications of this method.

References

[1] R. Alur. Timed Automata. In *Proceedings of Computer-Aided Verification*, 1999.

[2] J.-Y. Le Boudec and P. Thiran. *Network Calculus - A Theory of Deterministic Queuing Systems for the Internet*. Springer, 2001.

[3] S. Bradley, W. Henderson, and D. Kendall. Using Timed Automata for Response Time Analysis of Distributed Real-Time Systems. In *Proceedings of Workshop on Real-Time Programming WRTP*, 1999.

[4] O. Bringmann, W. Rosenstiel, and D. Reichardt. Synchronisation Detection for Multi-Process Hierarchical Synthesis. In *Proceedings of International Symposium on System Synthesis (ISSS) Hsinchu, Taiwan*, 1998.

[5] S. Chakraborty, S. Künzli, and L. Thiele. A General Framework for Analysing System Properties in Platform-Based Embedded System Designs. In *Proceedings of DATE*, Munich, 2003.

[6] S. Dey and S. Bommu. Performance Analysis of a System of Communicating Processes . In *Proceedings of ICCAD*, 1997.

[7] A. Hergenhan and W. Rosenstiel. Static Timing Analysis of Embedded Software on Modern Processor Architectures. In *Proceedings of the Date 2000 Conference*, March 2000.

[8] A. Hergenhan, A. Siebenborn, and W. Rosenstiel. Studies on Different Modeling Aspects for Tight Calculations of Worst Case Execution Time. In *WIP-Proceedings of the 21th IEEE Real-Time Systems Symposium*, 2000.

[9] H. Hulgaard and S. M. Burns. Efficient Timing Analysis of a Class of Petri Nets. In *Proceedings of Computer-Aided Verification*. Springer-Verlag, 1995.

[10] M. A. Marsan, A. Bobbio, and S. Donantelli. Petri Nets in Performance Analysis: An Introduction . In *Lecture Notes in Computer Science*, volume 1491. Springer-Verlag, 1998.

[11] P. Pop, P. Eles, and Z. Peng. Performance Estimation for Embedded Systems with Data and Control Dependencies. In *CODES*, 2000.

[12] K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. Model Composition for Scheduling Analysis in Platform Design. In *Proceedings of DAC*, 2002.

[13] A. Siebenborn, O. Bringmann, and W. Rosenstiel. Worst-Case Performance Analysis of Parallel, Communicating Software Processes. In *Proceedings of CODES*, 2002.

[14] W. Stark and S. A. Smolka. Compositional Analysis of Expected Delays in Network of Probalistic I/O Automata. In *IEEE Symposium on Logic in Computer Science*, 1998.

[15] E. A. Walkup. *Optimization of Linear Max-Plus Systems with Application to Timing Analysis*. PhD thesis, University of Washington, 1995.

[16] Y. Xie and W. Wolf. Allocation and Scheduling of Conditional Task Graph in Co-Synthesis. In *Proceedings of DATE*, Munich, 2001.

[17] A. Yakovlev, L. Gomes, and L. Lavagno. *Hardware Design and Petri Nets*. Kluwer, 2000.

[18] T.-Y. Yen and W. Wolf. Performance Estimation for Real-Time Distributed Embedded Systems. In *IEEE Transactions on Parallel and Distributed Systems*, volume 9, November 1998.