# Synchronization Detection for Multi-Process Hierarchical Synthesis*

Oliver Bringmann[1,2], Wolfgang Rosenstiel[1,2], Dirk Reichardt[1]

[1] FZI, Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany
[2] Universität Tübingen, Sand 13, 72076 Tübingen, Germany
bringmann@fzi.de, rosen@informatik.uni-tuebingen.de

## Abstract

*Complex system specifications are often hierarchically composed of several subsystems. Each subsystem contains one or more processes. In order to provide optimization across different levels of hierarchy, a synchronicity analysis of the concerned processes has to be performed during high-level synthesis. The first step is the generation of a condensed graph representation of the inter-process communication. This graph is then utilized to detect inter-process communication which can be used to represent synchronization points between two or more processes. A synchronization point represents the starting point of an interval in which the communicating processes run synchronously. This interval is limited by unbounded data-dependent loops, denoted as de-synchronization points. As a result, different processes can only share resources in such an interval.*

## 1 Introduction

State-of-the-art high-level synthesis systems focus mainly on the synthesis of single process specifications. Due to the increasing design complexity, the synthesis of multi-process specifications is becoming more and more important. Moreover, in modern system design, complex systems can be characterized as a hierarchical composition of concurrently executing processes. Therefore, each module of a hierarchical specification contains one or more processes. Such a hierarchical specified system is illustrated in Figure 1. First efforts have been made in synthesizing hierarchical as well as multi-process specifications.

Several approaches have been proposed using the term hierarchical synthesis. The objective of most of the hierarchical synthesis approaches is to reuse already synthesized clusters or modules, during synthesis of the overall system in order to reduce the synthesis complexity.

Hierarchical synthesis approaches based on clustering techniques try to collect operations with a high similarity measure into one cluster, or to use the loop/subroutine hierarchy for clustering. First, each cluster is synthesized separately and then the clustered data-flow graph is synthesized. A lot of different clustering approaches have been presented so far [1, 2, 3, 4, 5].

Hierarchical synthesis approaches, based on reuse of already synthesized modules as register-transfer components, need an extended library model. Here, the synthesized RT descriptions of an arbitrary algorithmic module specification are kept in the library and can then be reused as a "black-box" during synthesis of the overall system [6, 7]. However, most high-level synthesis systems do not provide any optimization across different levels of hierarchy. Especially resources of different processes can not be shared. A first approach using a "white-box" module model, that provides resource sharing across different levels of hierarchy, has been implemented by the CADDY-II system [8, 9]. The CADDY-II system considers the VHDL behavioral specification, the RT structure, and the schedule of the modules during synthesis of the overall system. Therefore, hierarchy can be considered without applying inline-expansion, so that the synthesis time can be kept low.

In the past, first efforts have been achieved in modeling, simulation and synthesis of multi-process descriptions. Most of the approaches address the system level and provide process scheduling [10, 11], or performance estimation [12, 13]. This paper is more about synchronicity analysis of already scheduled processes than process scheduling or performance estimation. Goal of the synchronicity analysis is to find an interval in which the considered processes run synchronously. Such an interval is limited by synchronization and de-synchronization points. A synchronization point denotes an inter-process communication, after that the communication process proceed synchronously. A de-synchronization point denotes an unbounded data-dependent loop. Hence, an appropriate model is needed representing the inter-process communication and their dependencies. Until now, only limited work exists in this area. In [13] a message dependency graph is formed which represents the inter-process communication for each separate basic-block. Therefore, control structures can not be taken into account. This approach aims at the simplification of the interface between concurrent processes. The approach in [12], for determining

* This work is partially supported by the DFG research program "Rapid Prototyping of Embedded Systems with Hard Time Constraints" under Ro1030/4.

timing bounds between consecutive communications does not provide control structures, either. Finally, a very flexible representation of communication has been presented in [14], in order to calculate the worst case execution time of a system of communicating processes. However, this model is limited to specifications without unbounded data-dependent loops.

Our approach allows the synthesis of hierarchical specification, where each module consists of one or more processes (Figure 1). One goal in hierarchical synthesis is to provide resource sharing across different levels of hierarchy [8]. Due to the fact that each module contains at least one process, the resource sharing problem is applied to multi-process specification, which can be hierarchically structured. One topic that has to be solved in this context is the synchronization problem between different processes.

In this paper we describe in more detail how synchronization points and de-synchronization points can be found, in order to support resource sharing in hierarchical synthesis. Section 2 outlines our hierarchical resource sharing approach and gives some relevant definitions. Section 3 addresses the construction of the communication dependency graph, which represents the communication structure of a multi-process specification. The determination of the synchronization and the de-synchronization points is presented in Section 4. Some examples, including the experimental results, are shown in Section 4. Finally, this paper concludes with a summary in Section 5. Note that the synchronicity analysis presented in this paper can also be applied to other topics beyond resource sharing, like minimization of the communication overhead by converting synchronous into asynchronous receive operations.

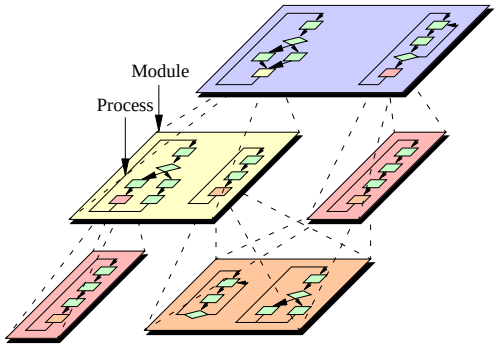



**Figure 1.** Illustration of a Hierarchical Specification

## 2 Problem Definition

In this section, we outline our hierarchical resource sharing approach. Then, the hierarchical schedule is defined containing all information needed for hierarchical resource sharing. Based on the schedule, a more compact graph is derived representing only the information needed for determining the synchronization and de-synchronization points.

### 2.1 Resource Sharing in Hierarchical Synthesis

In hierarchical synthesis, the entire system is synthesized in bottom-up traversal of the hierarchy. After each module is synthesized, the VHDL behavioral description, the RT structure, and the calculated module schedule are saved in the component library. During synthesis of the modules at a higher hierarchical level, the previously synthesized module information can then be used for sharing common submodules across hierarchical levels. Main data structure for resource sharing in hierarchical synthesis is the hierarchical schedule, described in the next subsection. The hierarchical schedule provides information of an operation concerning their scheduled start time, their allocated components, and their enclosed control structure including the iteration count. Based on this information it can be calculated when a submodule is not in use and can be shared. In order to be able to share resources between different processes, first an interval has to be determined, in which the considered processes run synchronously. Such an interval is called *sharing interval*. Within the sharing interval the influence of the control structures has to be considered. This can be done by implicitly enumerating the clock steps of all states of the concerning processes which are competing for the demanded resource. If the intersection of the sets of implicitly enumerated clock steps is empty, then resource sharing can be performed. Further details can be found in [8]. In the context of this paper, we want to describe precisely how the synchronization and de-synchronization points can be calculated, which define the sharing interval.

### 2.2 Hierarchical Schedule

The underlying model of a module is not restricted to specific synthesis limitations. A module may contain one or more processes. Each process consists of a flowgraph, representing the algorithmic specification, a hierarchical schedule, the generated RT structure, and the physical module parameter. Now, the formal definition of a hierarchical schedule is given.

**Definition 1.** A *hierarchical schedule* of a process $P$ is denoted by the tuple $HS(P) := \langle V, E, t, OP, IC, M \rangle$, where

- $V$ is a set of nodes representing clock steps, conditional branches, or nested loops, respectively.
- $E \subset L \times V$ is a set of edges with $L := \{v_l \in V : t(v_l) = loop \vee t(v_l) = branch\}$, where
- the function $t(v_l) \in \{operation, loop, branch\}$ denotes the type of a node $v_l \in V$.
- The relation $OP(v_{op})$, $\forall\ v_{op} \in V : t(v_{op}) = operation$ refers to the scheduled operations of the clock step $v_{op}$.
- The function $IC(v_l)$, $\forall\ v_l \in V : t(v_l) = loop$ returns the minimal iteration count $ic_{min}$ and the maximal iteration

count $ic_{max}$, where *iteration count* denotes the number of iterations of a loop, which may depend on outer loop iterators.

- The function $M(op)$, $\forall\ op \in OP(v_{op})$ refers to the instantiated module or module type of an operation *op*.

The hierarchical schedule $HS(P)$ of a process represents a tree, with the property that only nodes of type *loop* or *branch* can have descendants. The children of one node represent the schedule of this loop, or branch path, respectively. The root node represents the entire process and has an infinite iteration count ($ic_{max} = \infty$). An unbounded iteration count, which can not be statically determined, is denoted by $ic_{max} = u$. In case of a statically determined iteration count, $ic_{min}$ and $ic_{max}$ are equal.

## 2.3 Communication Dependency Graph

In the context of this paper, mainly the dependencies of different inter-process communications and the number of clock steps between consecutive communications are of interest. Such information can compactly be represented as a communication dependency graph, defined as follows:

**Definition 2.** A *communication dependency graph (CDG)* of a process is denoted by $CDG := \langle V, E, t, \delta \rangle$, where

- $V$ is a set of nodes representing communication states or loops with unbounded data-dependent delay.
- $E \subset V \times V$ is a set of directed edges describing the precedence dependencies between communication states or data-dependent loops.
- The function $t(v) \in \{asynchron, synchron, unbounded\}$ denotes the type of each node $v \in V$.
- The edge weights are represented by the function $\delta(v_i, v_j)$ with $(v_i, v_j) \in E$, which returns the number of scheduled clock steps between two communications states or unbounded data-dependent loops of a process.

A communication dependency graph *(CDG)* is a directed, cyclic graph which can be constructed from the hierarchical schedule of each process. In this graph, the control structure of the hierarchical schedule is flattened. As a result, the *CDG* contains no hierarchical structure, so that the calculation of the synchronization points can be performed efficiently. A small example of two *CDG*s constructed out of hierarchical schedules is shown in Figure 2. On the left side the hierarchical schedules of two communicating processes $P_1$ and $P_2$ are shown. For the sake of clarity, the hierarchical schedules are represented as state-transition graphs. On the right side the constructed *CDG*s are given for each process. A dashed line illustrates a communication channel between a send and a receive operation and belongs not to the respective graphs. Three types of nodes can be distinguished. First, the single encircled nodes represent asynchronous send or receive operations without wait functionality. Second, the double encircled nodes indicate synchronous receive operations with wait functionality. Finally, unbounded data-dependent loops are illustrated by double circles with a grey ring and are denoted with U. More details regarding *CDG* construction are shown in the next section.

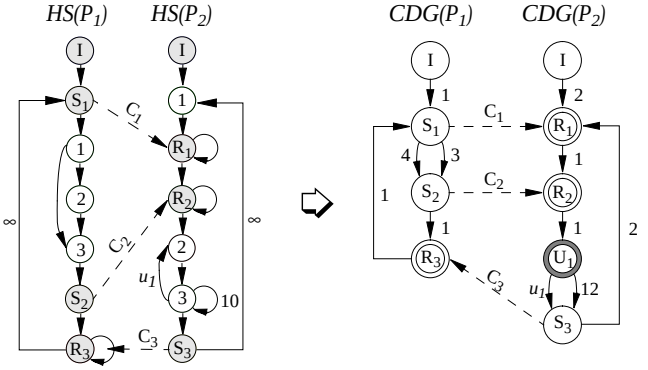



**Figure 2.** *CDGs* Constructed Out of $HS(P_1)$ and $HS(P_2)$

The communication dependency graph is only used for determining the synchronizations points. Further investigations for resource sharing operates on the hierarchical schedule again.

## 2.4 Synchronicity Conditions

A synchronicity condition can be formulated based on the set of communication dependency graphs. This condition has to be fulfilled for all synchronization points. Each communication pair $v_s$, $v_r$ is a potential candidate for a synchronization point. The principal condition for synchronization points is, that the receive node has to be reached before the corresponding send node. Before the definition of a synchronization point can be given, several helpful functions are defined. First, the set of all possible acyclic paths between a start node $v_1$ and an end node $v_2$ of the *CDG* is denoted by $P(v_1 \rightarrow v_2)$. Secondly, the sum of the edge weights $\delta$ of a given path $p \in P$ is given by $d(p)$, representing the sequential time taken. If $p$ contains an edge with weight $u$, then $d(p)$ becomes infinite. Thirdly, the relation $N(p)$ refers to all nodes which are composing the path $p$. Finally, the set of all synchronization points of a given *CDG* is represented by $SP(CDG(v))$, where $CDG(v)$ refers to the *CDG* containing node $v$. Initially, the set $SP$ contains the reset state $I$ of each module.

Now, a test concerning the send node is formulated. Because the receive node has to be reached before the corresponding send node, it is sufficient to consider only the shortest paths between the concerned communication nodes. The shortest path between the nodes $v_1$, $v_2$ is calculated by $P_{min}(v_1 \rightarrow v_2) = \{p \mid d(p) < d(p')\ \forall\ p, p' \in P(v_1 \rightarrow v_2) \wedge p \neq p'\}$.

Based on $P_{min}$ the paths $P_S(v_s)$ from all immediately preceding synchronization points to the considered send node $v_s$ can then be determined by

$$P_S(v_s) = \{P_{min}(v_{sync} \rightarrow v_s) \mid v'_{sync} \notin N(P_{min}(v_{sync} \rightarrow v_s)) \quad \forall\, v_{sync}, v'_{sync} \in SP(CDG(v_s)) \wedge v_{sync} \neq v'_{sync}\}.$$

Accordingly, the paths $P_R$ concerning the receive node $v_r$ can be formulated by using the longest acyclic path $P_{max}$ instead of the shortest path $P_{min}$. Note that the longest acyclic path $P_{max}$ is calculated only during one iteration of the traversed loops. At each backward edge of a complete traversed loop body, the delay of the loop body is multiplied by the iteration count of the corresponding loop, if statically known. Otherwise the longest path is marked as infinite. The backward edges can easily be determined from the hierarchical schedule. Since, the traversed subgraph is a directed acyclic graph with positive weights, the longest path can be calculated in polynomial time based on the algorithm shown in [15], modified with respect to the above mentioned extensions. Hence, all paths $P_R(v_r)$ from the preceding synchronization points to the receive node $v_r$ can be calculated by

$$P_R(v_r) = \{P_{max}(v_{sync} \rightarrow v_r) \mid v'_{sync} \notin N(P_{max}(v_{sync} \rightarrow v_r)) \quad \forall\, v_{sync}, v'_{sync} \in SP(CDG(v_r)) \wedge v_{sync} \neq v'_{sync}\}.$$

Now we are able to give the definition of a synchronization point and a de-synchronization point.

**Definition 3.** The nodes $v_s$, $v_r$ of a communication $C(v_s \rightarrow v_r)$ are called *synchronization points*, if the relation $v_r$ *before* $v_s$ $:= max\{d(p) \mid \forall\, p \in P_R(v_r)\} \leq min\{d(p) \mid \forall\, p \in P_S(v_s)\}$ is fulfilled. A node $v$ with $t(v) = unbounded$ is called a *de-synchronization point*.

The definition of a synchronization point strongly influences the construction as well as the analysis of the communication dependency graph. However, it gives only a criteria for verifying synchronization points and not a technique for determining synchronization points. In Section 4 an algorithm is presented which calculates all synchronization points with respect to the previous definition.

# 3 Construction of the CDG

The communication dependency graph has to be constructed out of the hierarchical schedule of each process. In this graph, a sequence of operations is folded into a single edge. The edge weights represent the respective number of clock cycles needed to perform the sequence of operations. The nodes represent send and receive operations of the hierarchical schedule. In the following, the construction of the *CDG* is given, taken several control structures into account. The construction is performed with respect to definition 3. Due to the limited space, no detailed algorithm for constructing the *CDG* can be presented.

## 3.1 Conditional Branches

In contrast to approaches that calculate the worst case execution path, the synchronization points has to be valid for all alternative paths. Due to the condition of definition 3, it is sufficient to consider only the paths $P_{min}$ and $P_{max}$. Hence, only the shortest and the longest alternative paths have to been constructed in the *CDG*. In case of a nested branch or a consecutive branch structure without communication, the shortest and the longest path, traversing the multiple branch structures, are chosen. This avoids, that the number of alternative paths is increasing, exponentially. If there exists multiple send or receive operations in different alternative paths, then the send and receive nodes can be collapsed according to Section 3.4.

The result quality is not influenced by such a simplification. That is because the calculation of the relation $v_r$ *before* $v_s$ uses the immediately preceding synchronization points which calculation only bases on the shortest and the longest paths.

## 3.2 Bounded Loops

Bounded loops can have a statically determined iteration count with equal bounds $ic_{min}$ and $ic_{max}$ or different bounds $ic_{min}$ and $ic_{max}$, where $ic_{max}$ represents the upper bound and $ic_{min}$ the lower bound.

First, we want to discuss the former loop type in more detail. If the loop body, including their entire nested loop hierarchy, does not contain any communication, then the entire loop is collapsed recursively to a single edge connecting the previous and the next communication nodes. The edge weight becomes the product of the sequential time needed for one iteration of the loop and their iteration count $ic_{max}$. In the case that the nested control hierarchy of a loop contains conditional branches, the edge weight has to be calculated for the shortest as well as the longest alternative path. Note that the edge weight has to be calculated recursively for the entire nested control hierarchy. If the loop body contains a communication node, then this node is created together with an self-referring edge in the *CDG*. This edge is labeled by the sequential time needed for one iteration of the loop.

Second, we want to discuss the latter loop type. In this case, the former technique for transforming a loop is applied as well, with the upper bound $ic_{max}$ and the lower bound $ic_{min}$. As a result, two edges are constructed in the *CDG*, similar to the transformation for conditional branches.

## 3.3 Unbounded Data-Dependent Loops

Loops with unbounded data-dependent iteration count $ic_{max} = u$ represent the de-synchronization points and strongly influence the determination of the synchronization points. Such loops represent the unique feasible node type in the *CDG* beyond the communication nodes. This node type is called *u-node*. During construction of the *CDG*, the u-

nodes can be handled similarly to the communication nodes. However, some exceptions exist. First, if the body of the unbounded loop contains at least one communication node at an arbitrary level of the nested control hierarchy, then the unbounded loop can be treated similarly to bounded loops, containing a communication node. That is because we want to ensure that data is properly communicated between the processes. That means no further messages are sent before the previously sent message has been received via the same communication channel. With this assumption it can be concluded, that both loops, from the send and the receive process, are unbounded and have similar loop conditions. Otherwise, some messages would be lost. If the body of the unbounded loop contains no communication node at an arbitrary level of the nested control hierarchy, then the entire loop body can be disregarded, expect for the minimal sequential time needed to perform one iteration. This value is used as a minimum iteration constraint of a u-node. Second, if an u-node is contained within an alternative path of a conditional branch construct, then this branch construct does not need the alternative longest path, since this path is already covered by the u-node.

### 3.4 Node Collapsing

In order to avoid the exponential growth of alternative paths, two node collapsing rules are given, which can be applied to arbitrary nodes. Multiple u-nodes in alternative paths can be collapsed into a single u-node by applying the following transformation. The incoming edge becomes the weight $min\{d(P_{min}(v_{if} \rightarrow v_u)) \mid \forall\ v_u \in P(v_{if} \rightarrow v_{fi})\}$ and the outgoing edge the weight $d(P_{min}(v_{if} \rightarrow v_{fi})) - min\{d(P_{min}(v_{if} \rightarrow v_u)) \mid \forall\ v_u \in P(v_{if} \rightarrow v_{fi})\}$, where $v_{if}$ and $v_{fi}$ are accessory nodes, denoting the start and the end of a conditional branch construct and are only used for the sake of clarity. Actually, this nodes are only contained in the hierarchical schedule. Nevertheless, because the node collapsing are performed during *CDG* construction, all necessary information are available.

A similar transformation can be applied to communication nodes, with the exception that not only the shortest path but also the longest path has to be considered. The longest path transformation is defined as follows: the incoming edge becomes the weight $max\{d(P_{max}(v_{if} \rightarrow v_c)) \mid \forall\ v_c \in P(v_{if} \rightarrow v_{fi})\}$ and the outgoing edge the weight $d(P_{max}(v_{if} \rightarrow v_{fi})) - max\{d(P_{max}(v_{if} \rightarrow v_c)) \mid \forall\ v_c \in P(v_{if} \rightarrow v_{fi})\}$.

Furthermore, consecutive unbounded loops can be folded into a single u-node by adding the weights of the edges connecting all consecutive u-nodes to the outgoing edge, of the last u-node in the sequence. Note that, communication nodes are not being allowed in the u-node sequence to perform the u-node collapsing. All presented graph transformations are $P_{min}$ and $P_{max}$ path invariant, so that the synchronicity condition $v_r$ *before* $v_s$ is not influenced.

## 4 Synchronization Point Detection

Main task is the determination of all feasible synchronization points such that condition of definition 3 holds. Since the condition bases on an already determined set of synchronization points, a constructive algorithm is needed to find the synchronization points. Our 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. The second phase observes the processes after their initiation is completed. Objective is to determine the 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 equalities. Therefore, the function *D(P)* is used, which constructs a linear function, representing the delay of a path *P* depending on each communication node. A path can be represented as an equality by using a slack variable, describing the wait states of the receive node. After forming the equality system, all communication cycles $C_{Cycle}$ are determined and ordered in decreasing number of contained communication nodes. A communication cycle represents a closed loop of potential synchronization points between two communication nodes. Out of the set $C_{Cycle}$, communication cycles are chosen successively which completely cover the communication nodes. For each chosen cycle, a linear equation system is created. The linear equation system has a totally unimodular coefficient matrix, so that each real solution is also an integer solution. A variable with a negative value denotes a not synchronizing communication. The second phase of the algorithm is shown in the following. The first phase can be handled similar, but instead of the communication cycles, all paths beginning at the reset state are considered.

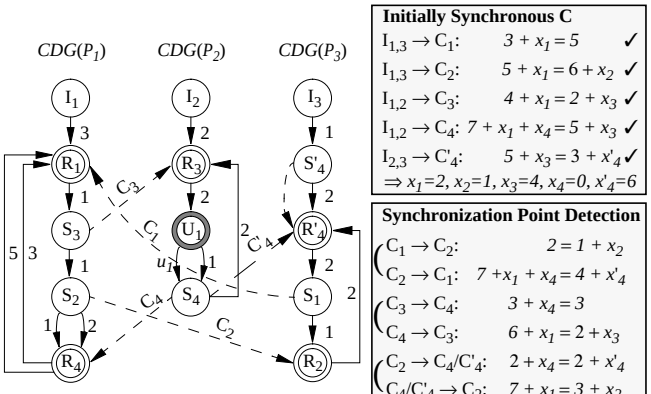
```
function DetectSyncPoint()
  foreach communication C_i(v_s→v_r) with InitSync mark do
    foreach preceding C_j(n_s→n_r) with P(n_s→v_s) ∧ P(n_r→v_r) do
      if D(P_min(n_s→v_s)) = D(P_max(n_r→v_r)) is not false then
        create Eq(C_i, C_j) := D(P_min(n_s→v_s)) = D(P_max(n_r→v_r))
  Determine all communication cycles C_Cycle
  Sort C_Cycle in decreasing order of their number of nodes
  repeat
    get first cycles F_Cycle that cover all C_i(v_s→v_r) with InitSync mark
    Eq_Cyc(F_Cycle) := all equations Eq(C_i, C_j) belonging to F_Cycle
    Evaluate equality system sol := solve(Eq_Cyc(F_Cycle))
    foreach i ∈ variable(sol) with C_i(v_s→v_r) do
      if i < 0 then
        mark C_i(v_s→v_r) with NotSync
        foreach successor C_j(n_s→n_r) with P(v_s→n_s) ∧ P(v_r→n_r) do
          F_Cycle := F_Cycle \ (C_i(v_s→v_r) → C_j(n_s→n_r))
      else
        mark C_i(v_s→v_r) with Sync
  until all C_i(v_s→v_r) have mark ≠ InitSync or F_Cycle was not changed
```

**Algorithm 1.** Synchronization Point Detection (2nd Phase)

## 5 Experimental Results

In this section we present the results applied by our approach to an ethernet controller. We specified the ethernet controller with respect to the description shown in [14]. In that approach, the objective is to determine the worst case execution time of a set of communicating processes. However, the objective differs from our approach. Here, the ethernet controller is used as an example to detect synchronization points in a multi-process specification. In Figure 3 the *CDG*s are shown for the three processes of the ethernet controller. The processes are communicating via the channels $C_1$, $C_2$, $C_3$, $C_4$, and $C'_4$, where the channels $C_4$ and $C'_4$ are representing a communication to multiple receivers. In the first phase all communications are observed, whether they are initially synchronous. An initially synchronous communication is a suitable candidate for a synchronization point. During the main phase the synchronicity condition has to hold for all initially synchronous communication, as well. The upper shaded box shows the initialization phase of the algorithm. All communication nodes are detected as initially synchronous. Note that the variable $u_1$ becomes the minimal value of all alternative edge labels, if it belongs to the path $P_{min}$. A further prerequisite is that all variables must be non-negative, in order to fulfill Definition 3. The lower shaded box describes the linear system of the main phase of the algorithm. In that system, the second equality is false, because it can be transformed to $x_1 = -3$, which contradicts to the non-negativity condition. Hence, just the communication $C_1$ is not a synchronization point, but all the others are.

As a further result, the determined synchronization points can be used to define the sharing interval. The specification contains several add operation. If we combine this approach with the approach in [8], one adder is sufficient for implementing the entire ethernet controller, because the adder can be shared between the processes.




⇒ **Synchronization Points *SP* = {$I_1$, $I_2$, $I_3$, $C_2$, $C_3$, $C_4$, $C'_4$}**

**Figure 3.** Synchronization Points of an Ethernet Controller

## 6 Conclusions

This paper presented a new approach for synchronization point detection in hierarchical synthesis. Our approach supports the analysis of multi-process specification with arbitrary control structures. Especially, loops with unbounded data-dependent delay are supported. The synchronization point detection is implemented in the high-level synthesis CADDY-II in order to provide resource sharing across different levels of hierarchy or process bounds, respectively. Further areas of application are for instance the minimization of the communication overhead by converting synchronous receive operations into asynchronous receive operations.

## 7 References


[1] M. McFarland, T. Kowalski: *Incorporating Bottom-Up Design into Hardware Synthesis.* IEEE Transactions on CAD, vol. 9, pp. 938-950, 1990.

[2] D. E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan, R. L. Blackburn: *Algorithmic and Register-Transfer Synthesis: The System Architect's Workbench.* Kluwer, 1990.

[3] D. Sreenivasa Rao, F. Kurdahi: *Hierarchical Design Space Exploration for a Class of Digital Systems.* IEEE Transactions on CAD, vol. 1, pp. 282-295, 1993.

[4] W. Geurts, F. Catthoor, H. De Man: *Quadratic Zero-One Programming-Based Synthesis of Application-Specific Data Paths.* IEEE Transactions on CAD, vol. 14, pp. 1-11, 1995.

[5] T. Ly, D. Knapp, R. Miller, D. MacMillen: *Scheduling using Behavioral Templates.* Proceedings of DAC, 1995.

[6] D. C. Ku, G. De Micheli: *High-Level Synthesis of ASICs Under Timing and Synchronization Constraints*. Kluwer, 1992.

[7] P. Kission, H. Ding, A. Jerraya: *VHDL Based Design Methodology for Hierarchy and Component Re-Use.* Proceedings of EURO-VHDL, 1995.

[8] O. Bringmann, W. Rosenstiel: *Resource Sharing in Hierarchical Synthesis.* Proceedings of ICCAD, 1997.

[9] O. Bringmann, W. Rosenstiel: *Cross-Level Hierarchical High-Level Synthesis.* Proceedings of D.A.T.E., 1998.

[10] A. Takach, W. Wolf: *Scheduling Constraint Generation for Communicating Processes.* IEEE Transactions on VLSI, vol. 1, no. 2, pp 215-230, 1995.

[11] K. Kuchcinski: *Embedded System Synthesis by Timing Constraints Solving.* Proceedings of ISSS, 1997.

[12] T. Amon, H. Hulgaard, S. Burns, G. Borriello: *An Algorithm for Exact Bounds on the Time Separation of Events in Concurrent Systems.* Proceedings of ICCD, 1993.

[13] D. Filo, D. Ku, C. Coelho, G. De Micheli: *Interface Optimization for Concurrent Systems under Timing Constraints.* IEEE Transactions on VLSI, vol. 1, no. 3, pp. 268-281, 1993.

[14] S. Dey, S. Bommu: *Performance Analysis of a System of Communicating Processes.* Proceedings of ICCAD, 1997.

[15] Y. Liao, C. Wong: *An Algorithm to Compact a VLSI Symbolic Layout with Mixed Constraints.* IEEE Transactions on CAD, vol. CAD-2, no. 2, pp. 62-69, 1983.

[16] W. Pugh: *The Omega Test: a fast and practical integer programming algorithm for dependence analysis.* Proceedings of Supercomputing, 1991.