Worst-Case Performance Analysis of Parallel, Communicating Software Processes [*]

A. Siebenborn, O. Bringmann and 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 a method to perform static timing analysis of SystemC models, that describe parallel, communicating software processes.The paper combines a worst-case execution time (WCET) analysis with an analysis of the communication behavior. The communication analysis allows the detection of points, where the program flow of two or more concurrent processes are synchronized. This knowledge allows the determination of the worst-case response time (WCRT). The method does not rely on restrictions on the system design to prevent deadlocks or data loss. Furthermore possible deadlocks and data loss can be detected during the analysis.

1. INTRODUCTION

The complexity of systems is increasing more and more, and often a single processor is not suitable to cope with high computational demands. Today multi processor systems are used for different applications, e.g. network processing or multimedia. Often these systems consist of DSPs and micro-controllers and interact with dedicated hardware. Increasing system complexity raises the demand for an unambiguous form of specification, that allows the description of systems, consisting of software and hardware. Moreover it should be possible to verify the correctness of such a specification. These requirement are fulfilled by the SystemC modeling platform. SystemC allows to create executable specifications of systems. It is based on the programming language C++, extended by constructs, that allow the description of concurrency, timing and reactivity, all of which are necessary to model parallel systems containing both software and hardware. These extensions are realized by classes, in contrast to adding new syntactic constructs. This way common C++ tools can be used for the design. Though SystemC allows a functional verification of the system by execution of the model, a verification of the temporal behavior of the system, especially the software, is needed. Timing behavior has to be explicitly specified. Execution times for software implementations could be determined by performing static timing analysis. However current methods for timing analysis are not able to handle concurrency and communication. We introduce a new method for static timing analysis for parallel, communicating systems specified by SystemC, the use of the determined timing values for the SystemC model and an analytical method to determine the worst-case response time of such a system. As a side effect we detect possible deadlocks, that result from structural and temporal properties of the system. The paper is structured as follows: In section 2 we give an overview other works in the area of timing analysis. In section 3 we shortly mention some properties of SytemC, especially communication and concurrency. Since we combine two methods, we have to decompose the problem for the two different domains, which is presented in section 4. Section 5 gives an introduction of our method of static timing analysis. In section 6 the communication analysis is presented and illustrated by an example in section 7.

2. RELATED WORK

Objective of timing analysis in the area of embedded systems is to gain information on the correctness of the programs timing behavior. There are several approaches to bound the worst-case execution time of single software tasks. In principle there are two different approaches: Methods performing an explicit path enumeration and methods based on implicit path enumeration. The idea of explicit path enumeration is to use standard graph algorithms to find the longest path in the control flow graph of a program. An example for this approach can be found in [1]. In general algorithms for longest path search are not suitable for cyclic graphs. For that reason loops are replaced by *complex* nodes.

A method based on implicit path enumeration, is presented in [5, 7]. Herein the problem is formulated by integer linear programming (ILP). With this method it is possible to model the whole system, including properties of processor architecture, like pipelining, caches and super-scalar

[*]This work was partially suported by the DFG priority programs on Embedded Systems Ro1030-5/3 and the BMBF/MEDEA+ project SPEAK A-508

.

execution units. Furthermore it is possible to formulate restrictions on the program flow, which result from functional properties of the program, like loop bounds or excluding path dependencies.

Since communication can block the execution of a software process, WCET analysis is not sufficient. In this case the worst-case response time (WCRT) of a system is needed. *State-charts* are a graphical method to specify system behavior. This language allows the description of concurrent processes with communication across hierarchical levels. In [4] a method is presented to validate the timing behavior of state-charts. Since communications are blocking on both sides, every communication represents a synchronization point , which makes a timing analysis easier. However the use of one sided communication leeds to more efficient implementations. Methods allowing a global communication analysis are known from the area of hardware design. However existing approaches often rely on fixed timing behavior between communication points and are restricted on systems without data dependent control structures [6]. In [3] a method is presented to calculate the WCRT of a system of concurrent processes. Yet, this method assumes a correct specified system, without deadlocks and data loss. In [2] a method to detect synchronization points in communicating processes is presented. As a side-effect the method can detect data losses and deadlocks in a system. The information gained by this method can be used for resource sharing during hardware synthesis. However, the worst case response time is not determined by this approach.

3. PARALLEL PROCESSES IN SYSTEMC

The SystemC design platform is a C/C++ based design methodology that allows system specification at several abstraction levels, allowing the specification of both hardware and software. In addition to pure C/C++ descriptions, SystemC allows the description of concurrency, communication and timing. The model can be executed to verify the system behavior. We will consider SystemC version 2.0 specified in [8]. In SystemC concurrency is expressed by *threads*. For communication and synchronization SystemC provides events, interfaces and channels. Events are the basic synchronization primitives and are used to construct communication on higher levels. Events are triggered by the `notify()` method. Process execution can be suspended by the `wait()`-method, resumed by occurance of specified events. The concept of channels provides a very flexible way to specify communication. Simple FIFO communication can be specified as well as higher level communication protocols. Specification of temporal behavior can be realized by the `s_clock` class, which allows to specify a clock with a certain period, or the `wait(100, SC_NS)` method, which suspends the execution of a process, for a specified time (100 nanoseconds in this case). The `wait()` method can be used to specify the timing behavior of software parts, since it is possible to annotate the execution time of code sequences. These execution times can be determined by static timing analysis, which we will show later in the paper.

The communication analysis, that we present in this paper handles communication at a very basic level. In principle we distinguish between three communication primitives:

- Asynchronous communication means, that neither the receiver, nor the sender blocks execution at the communication point, denoted as $(s_{async} \rightsquigarrow r_{async})$.
- In half-synchronous communication, one communication partner blocks at the communication point. This can be the receiver, waiting for a message, or the sender, waiting that the receiver is ready, denoted as $(s_{sync} \rightsquigarrow r_{async})$ or $(s_{async} \rightsquigarrow r_{sync})$
- By full-synchronous communication we mean communication, where both sender and receiver wait for each other at the communication point, denoted as $(s_{sync} \rightsquigarrow r_{sync})$.

The channel concept of SystemC allows the specification of all three communication types. In asynchronous communication, there is no synchronization between threads at all. Full-synchronous communication guarantees process synchronization. Half-synchronous communication can lead to process synchronization under certain conditions. Since the occurance of synchronization influences the systems overall timing behavior, they are worth further examination. In section 6 we will show, what are the conditions for synchronization and how synchronization points can be determined.

4. PROBLEM DECOMPOSITION

The approach presented in this paper allows the validation of the real-time behavior of systems of concurrent software processes. Hereby the WCRT of the system can be determined. Additionally the system is checked for potential data loss and deadlock. While other approaches only discuss systems with synchronization at both sides of communications [4] or simply assume that the rate of sending is less or equal to the rate of receiving [3], our approach finds possible data loss and deadlocks. Data loss occurs when the rate of sending data is higher, than the rate of data consumption. Since production and consumption rates are dependent on execution times, a priori assumptions are difficult to make.

Our approach combines two methods: (1) Communication Analysis and (2) Static Timing Analysis. This two methods work in two different problem domains. Static timing analysis handles code sequences with control structures, including loops with bounded iteration counts. 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. Consequently the first step is, to decompose the problem for the two analysis domains.

The first step is to find and classify communications in the programs. In SystemC communication is encapsulated by channels, interfaces and ports. This way communication points are easy to find. The methods `notify()` and `wait()` give the necessary information for a classification.

Communication points represent the nodes of the communication dependency graph *CDG*, which will be explained in section 6. 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 an other communication node. The minimum and maximum latency between the two nodes is determined by static timing analysis. For that purpose a subgraph of the control flow graph *CFG* is build up, which contains a communication node as start and end node. The subgraph contains all paths between the corresponding communication nodes, which do not include further communication nodes.

5. STATIC TIMING ANALYSIS

The problem addressed in the area of static timing analysis is to bound the execution time of a given sequence of code executed on a given processor. It is assumed that the given piece of code complies the restrictions on tasks, that means uninterrupted execution, no blocking communications and no unbounded loops. The term *processor* includes any architectural components, that influence the runtime of programs. So, not only the execution units, also memory hierarchy and busses are considered. This opens a large design space, that can be exploited at an early design stage. The execution time of a program may vary dependent on input data and the state of the processor system. The objective of static timing analysis is to find a time interval, which includes all possible execution times. The bounds should be as tight as possible to the actual minimum and maximum execution time. The problem is equivalent to the problem of finding the longest path in the program's control flow graph CFG. Since control flow graphs are cyclic graphs in general, this problem is only decidable, if the execution of loops is bounded by additional constraints. Though there exist standard graph algorithms to determine the longest paths in graphs, they are not applicable on cyclic graphs. We use an approach which formulates the problem as an optimization problem with linear constraints. The integer linear programming (ILP) is a flexible way to formulate and solve this kind of problems.

The objective function of the problem is given by the following expression:

$$\sum_{i=1}^{N} c_i x_i \,, \tag{1}$$

where N is the number of basic blocks in the CFG, c_i the execution time or cost of a basic block and x_i its execution count. There are two types of constraints: (1) constraints that describe the structure of the CFG, and (2) constraints that result from program functionality. Details of the method can be found in [5].

6. COMMUNICATION ANALYSIS

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*, defined as follows:

Communication Dependency Graph (CDG). A communication dependency graph of a process 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 channel.
- 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 communication. 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. These latencies are determined by static timing analysis. For that purpose subgraphs of the control flow graph have to be constructed, that contain all possible paths from one communication node to an other in the *CFG*.

6.1 Synchronicity Conditions

A synchronicity condition can be formulated based on the communication dependency graph. 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. 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 $C = (v_s \rightarrow v_r)$ with $(v_s, v_r) \in E_{COM}$ is a synchronization point if

$$\begin{aligned}
&L(path_{max}(v_{sync} \rightsquigarrow v_r)) \leq L(path_{min}(v'_{sync} \rightsquigarrow v_s)) \\
&\quad \forall (v_{sync} \leftrightarrow v'_{sync}) \in SP_{pre}(v_s \rightarrow v_r), \text{ with}
\end{aligned}$$

$$\begin{aligned}
&SP_{pre}(v_s \rightarrow v_r) = \{(v_{sync} \leftrightarrow v'_{sync} \in SP : \\
&\quad \exists p_1 \in paths(v_{sync} \rightsquigarrow v_r), p_2 \in paths(v'_{sync} \rightsquigarrow v_s) \\
&\quad \forall (v_m \rightarrow v_n) \in SP : v_m, v_n \neq v_{sync} \wedge v_m, v_n \neq v'_{syn} \;\wedge \\
&\quad v_m, v_n \notin p_1 \cup p_2 \wedge p_1, p_2 \notin \oslash\} \quad \text{holds.}
\end{aligned}$$

Herein the set $SP_{prec}(v_s \rightarrow v_r)$ 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 applicable. Figure 1 shows the application of the synchronization condition. In this example the following equations can be determined:

$$\begin{aligned}
L(path_{max}(I_2 \rightsquigarrow R_1)) &\leq L(path_{min}(I_1 \rightsquigarrow S_1)) \\
1 &\leq 2 \\
L(path_{max}(S_3 \rightsquigarrow R_1)) &\leq L(path_{min}(R_3 \rightsquigarrow S_1)) \\
1 &\leq 1
\end{aligned}$$

The communication $(S_1 \to R_1)$ is a synchronization point if $(S_3 \to R_3)$ and (I_1, I_2) are synchronization points. Later in the paper we will show, how synchronization points can be determined and how the cyclic dependency introduced by the process loop can be resolved.

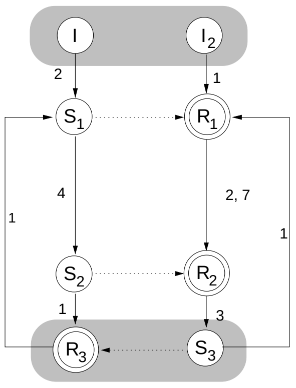

Figure 1: *Communication Dependency Graph*

6.2 Synchronization Point Detection

The main task during the communication analysis is the determination of all feasible synchronization points, such that the condition formulated in section 6.1 holds. Since this condition is based on an already determined set of synchronization points, a constructive algorithm is needed. 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 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 6.1, there is a cyclic dependency introduced by the process loop. 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 6.1 is not fulfilled, if the result for a slack variable is a negative value. There are two types of synchronicity equations, corresponding to the minimum slack variables $\underline{x}_i$ and the maximum slack variables $\overline{x}_i$:

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

Figure 2 shows a part of an example *CDG*. In this figure the influence of a communication from an other communication cycle is illustrated. When communication C_5 is not considered, the following equations can be derived.

$$\begin{aligned}
\underline{SE}(C_3 \rightsquigarrow C_4): &\quad 4 + \underline{x}_4 = 5 \Rightarrow \underline{x}_4 = 1 \\
\underline{SE}(C_1 \rightsquigarrow C_2): &\quad 8 = 1 + 1 + \overline{x}_4 + 1 + \underline{x}_2 \\
&\quad \Rightarrow \underline{x}_2 = -1 \\
\text{with} &\quad 1 + \overline{x}_4 = 7 \Rightarrow \overline{x}_4 = 6
\end{aligned}$$

To determine the maximum path $R_1 \rightsquigarrow R_2$, the maximum slack variable $\overline{x}_4$ has to be considered and for its determination an additional equation is needed.

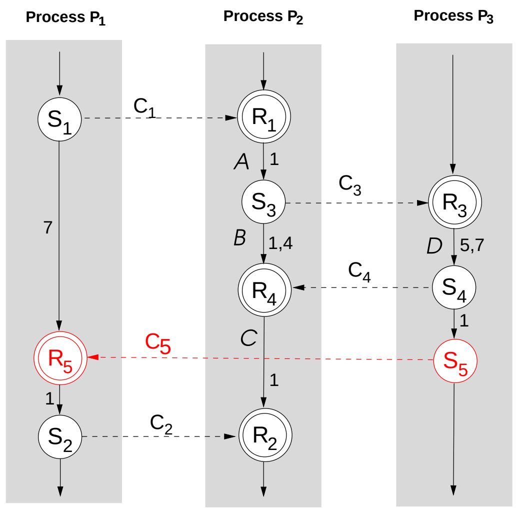

Figure 2: *Interdependencies between communication cycles*

The determination of the maximum and minimum path in such a case can be demonstrated using the example in Figure 2.

$$\begin{aligned}
&L(path_{min}(R_1 \rightsquigarrow R_2)) \\
&\quad = \min A + \max B + \underline{x}_4 + \min C \\
&\quad = \min A + \max B + (\min D - \max B) + \min C \\
&\quad\quad \text{with} \quad \underline{x}_4 = \min D - \max B \geq 0 \\
&\quad = \min A + \min D + \min C
\end{aligned}$$

$$\begin{aligned}
&L(path_{max}(R_1 \rightsquigarrow R_2)) \\
&\quad = \max A + \min B + \overline{x}_4 + \max C \\
&\quad = \max A + \min B + (\max D - \min B) + \max C \\
&\quad\quad \text{with} \quad \overline{x}_4 = \max D - \min B \geq 0 \\
&\quad = \max A + \max D + \max C
\end{aligned}$$

Due to the influence of communication C_4, the path $\max D$ has to be considered to determine the maximum path from R_1 to R_2. The inspection of paths of processes, which are not part of the communication cycle $Cycle(P_i, P_j)$, can be avoided by expressing the minimum path to the send node in the other process by the maximum path to the receive node and the minimum slack variable $\underline{x}_i$.

Though the evaluation of **min** and **max** functions means an additional effort, it leads to a reduction of the number of variables and equations. Taking communication C_5 in the example from Figure 2 into account, the problem can be expressed by the following equations:

$$
\begin{array}{ll}
\underline{SE}(C_3 \rightsquigarrow C_4): & 4 + \underline{x}_4 = 5 \Rightarrow \underline{x}_4 = 1 \\
\underline{SE}(C_1, C_3 \rightsquigarrow C_2): & 7 + \underline{x}_5 = 1 + 5 + 1 \\
 & \Rightarrow \underline{x}_5 = 0 \\
\underline{SE}(C_1 \rightsquigarrow C_2): & 8 + \underline{x}_5 = 1 + 4 + \underline{x}_4 + 1 + \underline{x}_2 \\
 & \Rightarrow \underline{x}_2 = 1
\end{array}
$$

In this case the path over communication C_3 has to be taken into account, to express the synchronicity equation $\underline{SE}(C_1, C_3 \rightsquigarrow C_2)$.

In Figure 6.2 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 of the system.

At the first sight it seems to be obvious, that a optimization problem has to be solved to determine all maximal slack variables. But since the calculation of a slack variable is always 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.

7. EXAMPLE

In this section we present an example applying our approach to the specification of an Ethernet controller, with respect to the description shown in [3].

In Figure 4 the *CDG*s are shown for the three processes. The processes are communicating via the communication channels C_1, C_2, C_3, C_4 and C_4' where the channels C_4 and C_4' represent a communication to multiple receivers. The *CDG* contains three communication cycles:

$$
\begin{array}{rcl}
Cycle_1(P_1, P_2) & = & \{C_3, C_4\} \\
Cycle_2(P_2, P_3) & = & \{C_4'\} \\
Cycle_3(P_1, P_3) & = & \{C_1, C_2\}
\end{array}
$$

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. With respect to the algorithm presented in Figure 6.2, first all initial synchronization points will be determined.

```
function RepetitiveSync(CDG) return $\mathcal{SP}$ $\mathcal{RSP}$;
    compose the set of all communication cycles $\mathcal{CYCLE}$;
    mark all communications $C$ with $analyzed(C)$ = false;
    while $\exists Cycle \in \mathcal{CYCLE} \wedge C \in Cycle : analyzed$ = false do
        $LES$ = setupLES($CDG$);
        solve $LES$;
        // evaluate solution
        if $\underline{x}_i \geq 0 \quad \forall \underline{x}_i \in solution(LES)$ then
            for each $\underline{x}_i \in var(LES)$ do
                if $\tau_C(C_i) \in \{ISP, SP\}$ then
                    set all ISP of current communication cycle to SP
                    set $analyzed(C)$ = true $\forall C \in Cycle : C_i \in Cycle$
                else
                    set $\tau_C(C_i) = RSP \wedge analyzed(C)$ = true;
                    $\mathcal{RSP} = \mathcal{RSP} \cup \{C_i\}$;
                fi;
            od;
        else
            test, if a non-SP communication has influence and correct result;
            remove all communication $C_j \in Cycle_j$ in $Cycle_j$ if: $\underline{x}_j < 0$;
        fi;
    od;
end RepititiveSync;
```

Figure 3: ***Algorithm for synchronization point detection***

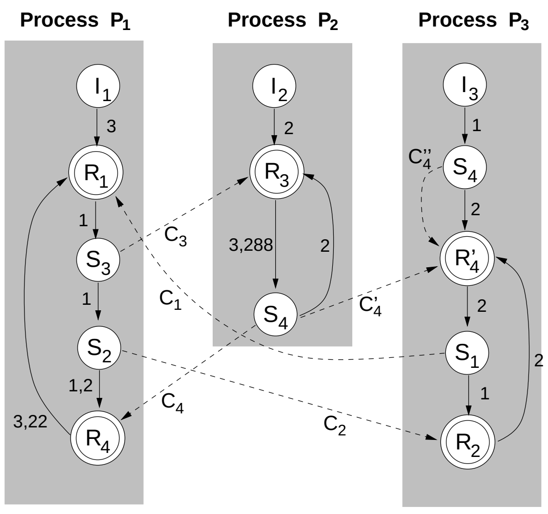

Figure 4: ***Synchronization points of an Ethernet controller***

$$
\begin{array}{ll}
\underline{SE}_1(I_1, I_2 \rightsquigarrow C_3): & 4 + \underline{x}_1 = 2 + \underline{x}_3 \\
\underline{SE}_2(I_1, I_3 \rightsquigarrow C_1): & 5 + \min(\underline{x}_4', \underline{x}_4'') = 3 + \underline{x}_1, \\
 & \text{with} \quad \underline{x}_4'' = 0 \\
\underline{SE}_3(I_2, I_3 \rightsquigarrow C_4'): & 5 + \underline{x}_3 = 3 + \underline{x}_4
\end{array}
$$

This equation system has the solution $\underline{x}_1 = 2$, $\underline{x}_3 = 4$, $\underline{x}_4' = 6$. Thus the communications C_1, C_3 and C_4' are initial synchronization points. It is obvious, that the intra process communication C_2'' has not to be considered for synchronization point analysis. Without the communication C_4'' the system would contain a deadlock, since in this case from the equations $\underline{SE}_1$ and $\underline{SE}_2$ the equation $\underline{x}_3 = \underline{x}'_4 + 4$ can be derived. Inserted in $\underline{SE}_3$ the inconsistent equation $5 + \underline{x}'_5 + 4 = 3 + \underline{x}'_4$.

The second equation system conduces to test the remaining communications points.

$$\begin{array}{ll} \underline{SE}_1(C_3 \rightsquigarrow C_4): & 3 = 3 + \underline{x}_4 \\ \underline{SE}_2(C_1 \rightsquigarrow C_2): & 2 = 1 + \underline{x}_2 \end{array}$$

The solution $\underline{x}_2 = 1$ and $\underline{x}_4 = 0$ indicates, that C_2 and C_4 are initial communication points too. In the second phase the global synchronization points are determined. For the equations paths from the last initial synchronization points to the first communications of each synchronization cycle:

$$\begin{array}{ll} \underline{SE}_1(C_4 \rightsquigarrow C_3): & 23 + \underline{x}_1 = 2 + \underline{x}_3 \\ \underline{SE}'_2(C_2, C_4 \rightsquigarrow C_4): & 2 + \underline{x}_4 = 2 + \underline{x}_4 \\ \underline{SE}_3(C_2 \rightsquigarrow C_1): & 4 + \underline{x}'_4 = 24 + \underline{x}_4 + \underline{x}_1 \\ \underline{SE}_4(C_3 \rightsquigarrow C_4): & 3 = 3 + \underline{x}_4 \\ \underline{SE}_5(C_1 \rightsquigarrow C_2): & 2 = 1 + \underline{x}_2 \end{array}$$

With $SG_2(C'_4 \rightsquigarrow C'_4) : 5 + \underline{x}_3 = 5 + \underline{x}_2 + \underline{x}'_4$ the equation system would have a cyclic dependency. For that reason it is replaced by equation SG'_2 where a path from communication C_2 to C'_4 over communication C_4 is considered. The equations SG_4 and SG_5 are inserted to determine the variables $\underline{x}_2$ and $\underline{x}_4$. The system has the solution $\underline{x}_1 = -20$, $\underline{x}_2 = 1$, $\underline{x}_3 = 1$, $\underline{x}_4 = 0$, $\underline{x}'_4 = 0$. Since x_1 has a negative value, the corresponding communication C_1 is no synchronization point. The communication C_1 is removed from the communication cycle and a new equation system is set up.

$$\begin{array}{ll} \underline{SE}_1(C_4 \rightsquigarrow C_3): & 4 = 2 + \underline{x}_3 \\ \underline{SE}'_2(C_2, C_4 \rightsquigarrow C'_4): & 2 + \underline{x}_4 = 2 + \underline{x}'_4 \\ \underline{SE}_3(C_2 \rightsquigarrow C_2): & 7 + \underline{x}_4 = 5 + x'_4 + \underline{x}_2 \\ \underline{SE}_4(C_3 \rightsquigarrow C_4): & 3 = 3 + \underline{x}_4 \end{array}$$

This system of equations leads to the solution $\underline{x}_2 = 2$, $\underline{x}'_3 = 2$, $\underline{x}_4 = 0$, $\underline{x}'_4 = 0$. Now all variables have positive values and the algorithm terminates.

The determined variables represent the minimal wait cycles at the synchronization points. Based on the determined synchronization points, the maximum value for the wait states will be determined in the next step. These values are needed to determine a WCRT of the system. First the maximum slack variables for the initialization can be determined.This leads to the solution $\overline{x}_1 = 2$, $\overline{x}_3 = 4$, $\overline{x}'_4 = 292$ In the second step the maximal slack variable for the iterative case are determined:

$$\begin{array}{ll} \overline{SE}_1(C_4 \rightsquigarrow C_3): & 4 = 2 + \overline{x}_3 \\ \overline{SE}'_2(C_2, C_4 \rightsquigarrow C_4): & 2 + \overline{x}_4 = 2 + \overline{x}_4 \\ \overline{SE}_3(C_2 \rightsquigarrow C_2): & 26 + \overline{x}'_4 = 5 + \overline{x}_4 + \overline{x}_2 \\ \overline{SE}_4(C_3 \rightsquigarrow C_4): & 288 = 2 + \overline{x}_4 \end{array}$$

The solution in this case is $\overline{x}_2 = 21$, $\overline{x}_3 = 2$, $\overline{x}_4 = 286$, $\overline{x}'_4 = 286$. With this information it is possible to determine maximal execution times between arbitrary points in the system and this way to determine the WCRT of a the system.

8. CONCLUSION

We presented a method to perform timing analysis in systems consisting of parallel software processes. We combined a method for static timing analysis with an approach for communication analysis. The SystemC modeling platform allows the description of such systems. However there is a need for a method, to prove the temporal correctness of such descriptions. The presented method could be applied on SystemC models, and this way it would be possible to verify the temporal correctness of the software in a SystemC model, including the detection of deadlock and data loss. The application of the method has been demonstrated by an example. Besides information an the temporal correctness, the information on control flow synchronization can be used in a further step, to perform an efficient processor allocation in a multiprocessor environment.

9. REFERENCES

[1] P. Altenbernd. On the False Path Problem in Hard Real-Time Programs. In *Proceedings of the 8th Euromicro Workshop of Real-Time Systems*, 1996.

[2] 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.

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

[4] E. Erpenbach and P. Altenbernd. Worst-Case Execution Times and Schedulability Analysis of Statechart Models. In *11th Euromicro Conference on Real Time Systems*, 1999.

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

[6] H. Hulgaard and T. Amon. Symbolic Timing Analysis of Asynchronous Systems. In *IEEE Transactions on Computer-Aided Design*, volume 19. 2000.

[7] Y.-T. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 456–461. IEEE, June 1995.

[8] Open SystemC Initiative (OSCI). Functional Specification for SystemC 2.0. www.SystemC.org, January 2001.