Synchronization after Design Refinements with Sensitive Delay Elements

Tarvo Raudvere, Ingo Sander, Axel Jantsch
Royal Institute of Technology
Stockholm, Sweden
{tarvo,ingo,axel}@kth.se

ABSTRACT

The synchronous computational model with its simple computation and communication mechanism makes it easy to describe, simulate and formally verify synchronous embedded systems at a high level of abstraction. In synchronous models, a local refinement increasing the delay in a single computation block may affect the functionality of the entire model. We provide a synchronization algorithm that preserves the system's functionality after design refinements, by using additional synchronization delays and making some delays sensitive to their input values. The refined and synchronized model stays latency equivalent to the original model. The advantages of our approach are the following: (a) we remain fully within the synchronous model of computation, (b) we preserve the functionality of the existing computation blocks, and (c) we do not require additional computation resources, specific communication protocols, wrapper circuits around computation blocks or schedulers.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids; J.6 [**Computer-aided Engineering**]: Computer-aided Design (CAD)

General Terms

Design, Algorithms

Keywords

System Design, Design Refinement, Synchronization

1. INTRODUCTION

Synchronous computational models are popular in system design targeting safety critical applications and are efficiently used in the aerospace industry [11]. The synchronous hypothesis assumes that the computation in processes and communication between them takes no time. In this kind of models local refinements that increase the delay in some computation block, like the introduction of pipelining and resource sharing, are a potential source of errors due to changed time behavior. The problem becomes more complex if models contain nested feedback loops.

Let's consider the system in Figure 1.a, containing eight combinational functions and five delay elements. Let a refinement increase the delay in fn_6 that is equivalent to add a delay Δ_6 to the model (Figure 1.b). At the first time instant Δ_6 emits its initial value to fn_7 and the result of the computation is thereafter processed by other functions as well. In fact, the latter values are unexpected compared to the behavior of the original model. The algorithm presented in [13] solves the problem as follows. In order to distinguish between expected and unexpected values, refinement added delays are initialized with a special *absent* value ($\perp$). $\Delta^{\perp}$ denotes a delay initialized with $\perp$-value. Functions react to these values as: $fn(\perp,\ldots,\perp)=\perp$. The algorithm guarantees that different types of values never arrive to functions at the same clock cycle. The additional synchronization delay $\Delta^{\perp}_{10}$ ensures that fn_1 receives $\perp$-values from both loops at the second clock cycle. Since loops reproduce $\perp$-values, and $loop_1$ and $loop_2$ have different delays, the algorithm adds more synchronization delays: $\Delta^{\perp}_7$, $\Delta^{\perp}_8$, $\Delta^{\perp}_9$. In the refined and synchronized model fn_1 receives $\perp$-values at every second clock cycles. Although the model processes more values than the original model, the same expected values appear in the same order in both models, i.e., the refined model is latency equivalent to the original model.

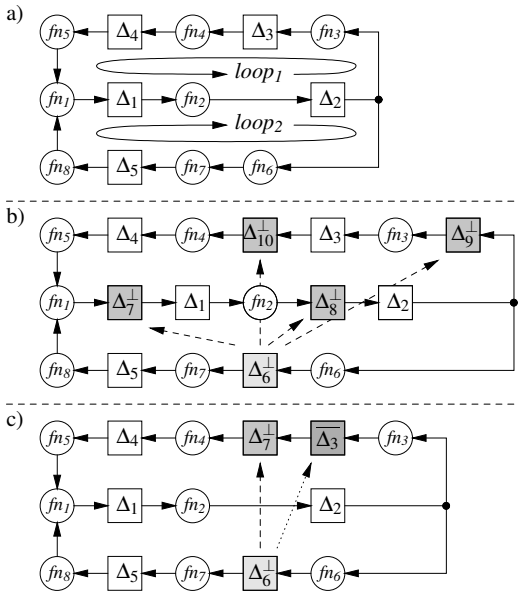

Figure 1: Synchronization in loops with three and four delays

CODES+ISSS'07, September 30–October 3, 2007, Salzburg, Austria.

In order to decrease the number of additional synchronization delays ($\Delta_i^{\perp}$) and to improve the ratio between $\perp$-values and useful data, we propose in this paper a more efficient synchronization algorithm that modifies some delays in the model. The modification makes delay elements sensitive to the types of input values. *A sensitive delay* $\overline{\Delta}$ preserves its state when $\perp$-values arrive to the delay, and instead of the current state value it emits the input value. In other words, a $\perp$-value jumps over the value that arrived at the delay at one clock cycle before. If the original delay with the initial state value st_0 and the input sequence $\{a,b,c,\perp,d,\dots\}$ operates as: $\Delta(st_0,\{a,b,c,\perp,d,\dots\}) = \{st_0,a,b,c,\perp,d,\dots\}$ then the behavior of a sensitive delay is: $\overline{\Delta}(st_0,\{a,b,c,\perp,d,\dots\}) = \{st_0,a,b,\perp,c,d,\dots\}$.

Our algorithm replaces Δ_3 in Figure 1.a with the sensitive delay $\overline{\Delta_3}$ in Figure 1.c and adds $\Delta_7^{\perp}$ to synchronize the refinement added delay $\Delta_6^{\perp}$. Figure 2 shows abstract states of delays in $loop_1$ in the

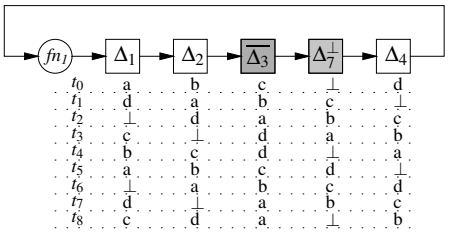

Figure 2: Values stored in delays in the refined and synchronized loop

first nine clock cycles. The characters $(a,b,c,\perp,d)$ corresponding to t_0 are the initial states of delays and fn_1 is viewed as an identity function. Without the modification in Δ_3 fn_1 receives $\perp$-values after every four clock cycles. $Loop_1$ in Figure 1.c including $\overline{\Delta_3}$ and $loop_2$ feed fn_1 synchronously by $\perp$-values after every three clock cycles. The ratio between useful and $\perp$-values and the number of synchronization delays $\Delta^{\perp}$ are clearly improved, compared to the model in Figure 1.b. The model in Figure 1.c is latency equivalent to the original model.

2. RELATED WORK

Many researchers agree that a system design approach must start with a formal system model on a high level of abstraction [8, 6]. The initial model must then be refined manually or automatically [1] into a concrete implementation. Our synchronization technique supports design methodologies that use formal design transformations in a synchronous computational model. The latter model is the base for languages like Lustre [7] and Esterel [3], which have successfully been used for safety-critical industrial applications. The computational model has also similarities to the synthesizable subset in VHDL and Verilog languages that employ a clocked-synchronous computational model, which makes our approach very applicable in practice.

Retiming and pipelining are well-known techniques that address latency and data arrival problems. In order to reduce the circuit area or a critical path retiming algorithms [10, 9] relocate already existing memory elements. Although retiming techniques address synchronization problems, these problems are not caused by additional delays inserted to the model. On the other hand, the introduction of additional delays to the model is elaborated in pipelining transformations. Pipelining in software is simpler, since there different models from the perfectly synchronous one are used. In hardware the data synchronization is solved by a pipeline controller, derived from a high level system specification [15]. Our technique makes it possible to introduce pipelining at system level in synchronous models with nested feedback loops, without adding controllers or changing the used computational model.

In order to avoid synchronizations problems caused by refinements that increase the delays of computation blocks, desynchronization [2, 12] or latency insensitive design [4] (LID) techniques can be applied. The former technique transfers a synchronous model to an asynchronous one, which is less sensitive to delayed data arrival. LID targets the mapping of an IP blocks based synchronous model to hardware, where longer wires entail delayed data arrival. The synchronization problem is solved by (1) wrappers around IP-blocks stalling computation if input data is not available, and (2) by handshake channels and relay stations between IP-blocks that replace synchronous communication. The handshake mechanism distributes stalling messages and a relay station buffers data items if the destination process cannot consume them. In [5] a more advanced approach is described, which replaces LID protocols with schedulers at every IP-block. Although this method drastically simplifies the implementation of LID, it is much more complex to refine a model including schedulers at the system level, compared to the refinement in a pure synchronous model.

Although both of these techniques are common in practice, they have side effects that our synchronization algorithm avoids. We avoid unnecessary discontinuities in the design process caused by changes in the computational model. It is impractical to switch the computational model due to a single local refinement. The use of the same computational model makes it much easier to verify refined models against each other. In addition, verification in deterministic synchronous models by using simulation or formal methods is simpler than in other models. On the other hand our synchronization technique is complimentary applicable within refinements in a synchronous island of GALS model or in an IP block of LID.

3. SYSTEM MODEL

We describe systems in the synchronous model of computation as a set of processes, which communicate through synchronous signals. Processes can be grouped into blocks of processes. Signal s_i is defined as a sequence of events $\{v_0^i, v_1^i, \dots, v_j^i, \dots\}$, where v_j^i is the value of the j-th event (with time tag j) of signal s_i. All signals share the same set of tags for synchronization purposes. The signal direction is from the source process to the destination process, and every process has only one output signal.

There are two kinds of events: (1) *present events* that carry a value and (2) *absent events* that are used only for synchronization. An absent event e_j shows that a signal contains no value at a time instant j. We use the mark $\top$ as an abstract value if we refer to present values, and $\perp$ for absent values. For example, the abstract presentation of signal $\{1_1, 4_2, \perp_3, 2_4, \dots\}$ is $\{\top, \top, \perp, \top, \dots\}$. To extend a data type T to $T_{\perp}$ the $\perp$-value is added to its domain.

A combinational process takes arguments as a dedicated n-input combinational function $f(x_1,\dots,x_n)$ and n input signals. For each tag j, a combinational process consumes from its input signals $s_1,\dots,s_n$ events with the tag j carrying values $v_j^1,\dots,v_j^n$ and produces to its output signal s' an event with the tag j and a value $v'_j = f(v_j^1,\dots,v_j^n)$:

$$P_{comb}(f(x_1,\dots,x_n),s_1,\dots,s_n) = s' = \{v'_0, v'_1, \dots, v'_n, \dots\} = \\ = \{f(v_0^1,\dots,v_0^n), f(v_1^1,\dots,v_1^n), \dots, f(v_j^1,\dots,v_j^n), \dots\}$$

A delay process $P_{\Delta}(st_0,s_1)$ has arguments as an initial state st_0 and an input signal $s_1 = \{v_0^1, v_1^1, \dots, v_j^1, \dots\}$.

$$P_\Delta(st_0, s_1) = s' = \{st_0, v_0^1, v_1^1, \ldots, v_j^1, \ldots\}$$

A finite state machine process P_{FSM} with a state function f_{st}, an output function f_{out}, an initial state st_0, input signals $s_1, \ldots, s_n$ and output signal s' is defined as:

$$\begin{array}{l} P_{FSM}(f_{st}, f_{out}, st_0, s_1, \ldots, s_n) = s' \\ \quad where \\ \quad\quad s' = P_{comb}(f_{out}, s'', s_1, \ldots, s_n) \\ \quad\quad s'' = P_{comb}(f_{st}, (P_\Delta(st_0, s'')), s_1, \ldots, s_n)) \end{array}$$

4. SYNCHRONIZATION

4.1 Definitions

In order to prepare the system to operate with synchronization values ($\perp$), we extend the functionality of combinational processes. If in the original model a function $f(x_1, \ldots, x_n)$ on input values $\{v_1, \ldots, v_n\}$ calculates an output value $f(v_1, \ldots, v_n) = v'$, then the modified function $\overline{f}(x_1, \ldots, x_n)$ operates as follows:

$$\overline{f}(v_1, \ldots, v_n) \quad = \quad \begin{cases} v', \; if \; \forall i, (1 \le i \le n), v_i \ne \perp \\ \perp, \; if \; \exists i, (1 \le i \le n), v_i = \perp \end{cases}$$

The modification in combinational functions applies to all combinational and FSM processes, making them to emit an absent event if any of the input events is not present at some time instant. In addition, we modify the storage elements in FSM processes, such that they do not update their states if an absent event is on their inputs. Since FSM processes do not cause any explicit delay to absent events and the response to input values appears instantly, we call combinational and FSM processes as *computation* processes to differentiate them from delay processes.

In the paper, we use the following terms and functions: A *path* in the system model is a sequence $\{P_1, P_2, ..., P_n\}$ of processes P_i, such that $\forall i, (1 \le i \le n-1)$, exists a signal s_i connecting the output of P_i to an input of P_{i+1}, and $\forall i, j, (1 \le i, j \le n), P_i \ne P_j$. A *loop* is a cyclic path with the same start and end process, and does not include any process twice.

A pair of paths contains two paths, $path_1(P_A, P_B)$ and $path_2(P_A, P_B)$, from process P_A to process P_B, and these paths do not share any other process than P_A and P_B.

Transformation $AbstJmp(P_\Delta \rightarrow P_{\overline{\Delta}})$ replaces a delay process P_Δ with a sensitive delay process $P_{\overline{\Delta}}$, whose reactions to input values are characterized by the following functions $\mathcal{F}_\Delta$ and $\mathcal{F}_{\overline{\Delta}}$, respectively ($st_0$ is the initial value in the delay processes, u_i is an input and v_i is an output value at time instant i):

$$\begin{array}{rcl} \mathcal{F}_{\Delta(st_0)}(u_i) & = & \begin{cases} v_i = st_i \\ st_{i+1} = u_i \end{cases} \\ \mathcal{F}_{\overline{\Delta}(st_0)}(u_i) & = & \begin{cases} if \quad (u_i \ne \perp), \\ \quad\quad v_i = st_i \\ \quad\quad st_{i+1} = u_i \end{cases} \\ & = & \begin{cases} if \quad (u_i = \perp), \\ \quad\quad v_i = u_i \\ \quad\quad st_{i+1} = st_i \end{cases} \end{array}$$

Function $|path_k(P_i, P_j)|_\Delta$ gives the number of delay processes in $path_k$ between processes P_i and P_j.

Function $|path_k(P_i, P_j)|_{\overline{\Delta}}$ gives the number of delay processes in $path_k$ between processes P_i and P_j excluding modified delay processes $P_{\overline{\Delta}}$. The function value is equal to the latency of the path to move an absent value from P_i to P_j.

In order to analyze the delays of paths in the system model, we construct a delay graph $G(W, E)$. The system inputs and outputs are viewed as delay processes $P_{\Delta i}$ within the analysis. To every $P_{\Delta i}$ in the system model corresponds a vertex w_i in G, $w_i \in W$. The graph contains an edge e_{ij} ($e_{ij} \in E$) from vertex w_i to w_j, if there is a path from process $P_{\Delta i}$ to $P_{\Delta j}$ in the system model, and the path contains no other delay processes. Similarly to the system model, a path in the graph is a sequence of vertices, and a loop is a cyclic path. The delay graph is formed so that it abstracts all computation processes and represents only delay processes and paths between them.

4.2 Synchronization requirements

The synchronization algorithm considers two facts: (1) loops reproduce absent events, and (2) there is a path between any two loops in the model. Therefore, if we add a delay process either to a path or a loop, we have to add synchronization delay processes to all loops in the system. These synchronization delays have to be placed so, that multi-input processes connecting loops and paths receive only one type of events at every clock cycle.

In our approach, the shortest loop in the graph determines the ratio between absent and present events processed by every process. If the shortest loop in graph G contains $\mathbf{r}$ delay processes, and a design transformation entails an extra delay process to the system, then after the first refinement the ratio is $\mathcal{R} = (1\perp : \mathbf{r}\top)$. Our algorithm modifies delay processes and adds synchronization delay processes in order to make all computation processes in the model to operate with the same ratio $\mathcal{R}$. If the delay of a loop is equal to $\mathbf{r}$, we add one synchronization delay process to the loop. To a loop with the delay $k * \mathbf{r}, (k \in \mathbb{N})$ [1] we add k synchronization delay processes, locating them so that the delay between two synchronization delay processes is equal to $\mathbf{r}$. In loops, where the number of delays is equal to $k * \mathbf{r} + \delta, (\delta < \mathbf{r}$ *and* $k \in \mathbb{N})$, before adding synchronization delay processes, we modify δ (or $\delta + n * \mathbf{r}, (n \in \mathbb{N})$) delay processes by the transformation *AbstJmp*. After modifications the delays for absent values in all loops are equal to $k * \mathbf{r}, (k \in \mathbb{N})$, i.e., all loops satisfy the following condition (1):

$$|loop_i|_{\overline{\Delta}} \; mod \; \mathbf{r} == 0 \; and \; |loop_i|_{\overline{\Delta}} \ge \mathbf{r}, \; if \; \mathcal{R} = (1\perp : \mathbf{r}\top) \quad (1)$$

In addition, we have to turn attention to multiple acyclic paths that connect two processes P_A and P_B. Paths from P_A to P_B may have different delays, i.e., paths do not contain the same number of delay processes - $|path_1(P_A, P_B)|_\Delta \ne |path_2(P_A, P_B)|_\Delta$. If the paths do not satisfy the following condition (2), different types of events ($\perp$ and $\top$) arrive to P_B at some time instants.

$$\begin{array}{r} (|path_1(P_A, P_B)|_{\overline{\Delta}} - |path_2(P_A, P_B)|_{\overline{\Delta}}) \; mod \; \mathbf{r} == 0, \\ if \; \mathcal{R} = (1\perp : \mathbf{r}\top) \end{array} \quad (2)$$

We say that a system is balanced if all loops and pairs of paths satisfy conditions (1) and (2). Based on these conditions we find the offset δ for loops and pairs of paths. δ shows how many delay processes we have to modify by the transformation *AbstJmp* in order to satisfy either condition (1) or (2). The offset of a $loop_j$ is:

$$\delta_j = |loop_j|_{\overline{\Delta}} \; mod \; \mathbf{r}$$

The offset of a $pair_j$ of $path_a$ and $path_b$ is:

$$\begin{array}{l} \delta_j = min(\delta_{ja}, \delta_{jb}), where \\ \delta_{ja} = (|path_a|_{\overline{\Delta}} - |path_b|_{\overline{\Delta}}) \; mod \; \mathbf{r}, \\ \delta_{jb} = (|path_b|_{\overline{\Delta}} - |path_a|_{\overline{\Delta}}) \; mod \; \mathbf{r} \end{array}$$

Since loops and pairs contain more delay processes than their offset δ_j, we can choose between many different δ element *combinations* of delay processes, which to modify by

[1] $\mathbb{N}$ is the set of natural numbers $\{1, 2, 3, \ldots\}$

AbstJmp. However, it is possible that due to the system structure, any combination of modified delay processes ($P_{\overline{\Delta}}$) and additional synchronization delays ($P_{\Delta}^{\perp}$) do not allow the system to operate with the ratio $(1\perp : \mathbf{r}\top)$. In this case we replace one P_{Δ} process in the shortest $loop_k$ with a sensitive delay process $P_{\overline{\Delta}}$, such that $|loop_k|_{\overline{\Delta}} = (\mathbf{r}-1)$ and try to synchronize the system to operate with the ratio $(1\perp : (\mathbf{r}-1)\top)$. We repeat the last step until the ratio $(1\perp : 1\top)$, which suits to any system[2]. In brief, this ratio means that either to the input or output of every P_{Δ} delay process in the model is added a synchronization delay processes $P_{\Delta}^{\perp}$, and computation processes receive absent events at every second clock cycle.

4.3 Outline of the algorithm

The first task in the synchronization algorithm is to construct the delay graph and to find all loops and pairs of paths. Based on the ratio, determined by the shortest loop, we calculate offsets to all pairs and paths. We apply model checking to find which processes have to be modified in order to get the model balanced. After modifications, in the balanced model all not modified delays get a label. If a refinement adds a delay $P_{\Delta i}^{\perp}$ to the model, then according to the label L_j of the nearest delay process to $P_{\Delta i}^{\perp}$, we add a synchronization delay processes next to all delay processes with labels L_j.

4.4 Finding loops

In order to find all loops we model every vertex w_i in the graph G as a process D_i and every edge e_{ij} as a signal between processes D_i and D_j. Process D_i has a distinct stamp z_i. At every step all processes receive and forward a list of vectors. If an input vector does not contain stamp z_i, process D_i adds z_i to the end of the vector, otherwise the vector is stored at D_i and not forwarded. At the first step only multi-input processes emit vectors with their stamps. A vector with the first stamp equal to z_i received by D_i, contains an ordered sequence of stamps of all processes in a loop. The maximum number of steps, we have to analyze the model to find all loops, is equal to the number of processes D_i.

4.5 Finding pairs of paths

Similarly to the finding loops algorithm we model graph G as a set of processes connected by signals. Process D_i has a stamp z_i. At the first step only these processes whose output is connected to more than one process emit vectors with their stamps. At every further step all processes add their stamps to the end of the received vectors and forward them. Multi-input process D_i discards a received vector, which already contains its own stamp z_i. All other vectors get extended with stamp z_i, stored in the process, and forwarded to the process output. It takes less steps to run the model than is the number of processes D_i in the model. After running the model, we take the stored vectors from multi-input processes and find pairs, which have identical stamps only in the first positions and in the last positions. These vectors contain stamps of pairs of paths.

4.6 Balancing the delays of loops and pairs of path

Based on the vectors of loops and vector pairs of paths found previously, we form a matrix $\mathcal{M}$. Column c_i of $\mathcal{M}$ corresponds to process $P_{\Delta i}$ and row r_j of $\mathcal{M}$ corresponds to $loop_j$ or $pair_j$ of paths. Initially all matrix elements are set to zero. If delay process $P_{\Delta i}$ belongs to $loop_j$, we set the matrix element $m(r_j,c_i) = 1$. If $P_{\Delta i}$ belongs to the $path_a$ in $pair_k$ then $m(r_k,c_i) = 1$, if it belongs to the $path_b$ then $m(r_k,c_i) = -1$, $|path_a|_{\Delta} \geq |path_b|_{\Delta}$; the matrix elements corresponding to the first and the last delay processes in paths forming a pair stay equal to zero.

The sum of values (Σr_j) in r_j gives the delay for absent events in $loop_j$, or the delay difference for absent events in $pair_j$, and presents the value on the left side in conditions (1) and (2) ($\Sigma r_j = |loop_j|_{\overline{\Delta}}$) and ($\Sigma r_j = (|path_a|_{\overline{\Delta}} - |path_b|_{\overline{\Delta}})$). Model M is balanced if all these sums in rows for loops and pairs of paths satisfy conditions (1) and (2), respectively.

We apply the transformation *AbstJmp* to delay processes $P_{\Delta i}$, in order to reduce the delay of a loop or a path for absent events and in such a way to decrease the offset of the respective row in $\mathcal{M}$. The result of the transformation is a sensitive delay process $P_{\overline{\Delta i}}$ that has no delay for absent events. According to the modification in $P_{\Delta i}$ we set all values equal to zero in column c_i in $\mathcal{M}$.

4.7 Finding a proper combination of P_{Δ}-s

A combination of delay processes that have to be modified in order to satisfy conditions (1) and (2) can be found by using a model checker. Columns c_i corresponding to delay processes $P_{\Delta i}$ or $P_{\overline{\Delta i}}$ are modeled as Integer variables x_i in model checking. The model checker can assign values "0" and "1" to x_i-s in a non-deterministic manner. Based on condition (1), for every row r_j in $\mathcal{M}$ representing $loop_j$ we write a boolean expression eq_j:

$$eq_j = (((\Sigma(x_i * m(r_j,c_i)))\ mod\ \mathbf{r}) == 0 \bigwedge (\Sigma(x_i * m(r_j,c_i))) \geq \mathbf{r}$$

Similarly, eq_j for a row r_j corresponding to $pair_j$ is:

$$eq_j = (((\Sigma(x_i * m(r_j,c_i)))\ mod\ \mathbf{r}) == 0$$

The task for the model checker is to check a specification, which says that there does not exist a case where all expressions eq_j of all rows in $\mathcal{M}$ are true at the same time:

$$SPEC\ \neg\mathbf{E}(eq_1 \& eq_2 \& \cdots)$$

If the model checker finds that this specification is not satisfied, it reports a counter-example, i.e., a combination of values, which assigned to x_i-s contradicts the specification. Since the specification is defined through negation, the values of x_i-s in the counter-example satisfy all expressions eq_j at the same time. Process $P_{\Delta i}$ has to be modified by *AbstJmp* if x_i in the counter-example is equal to "0". It is not possible to balance the model for ratio $\mathcal{R} = (1\perp : \mathbf{r}\top)$ if the model checker finds that the property in the SPEC holds. In this case we decrease $\mathbf{r}$ by one and run the model checker again. Although we replace some P_{Δ} delay processes in the original model with the sensitive ones ($P_{\overline{\Delta}}$), the behavior of the modified model is identical to the original one as far no refinement caused or synchronization delay processes $\Delta^{\perp}$ are added to the model.

4.8 Labeling of delay processes

All loops in the balanced model contain $n * \mathbf{r}$ ($n \in \mathbb{N}$) P_{Δ} delay processes and the difference of the number of P_{Δ} processes in paths of a pair is equal to zero or $n * \mathbf{r}$. All P_{Δ} processes in the model get labels from the set *Label* according to their positions. *Label* is an order set containing $\mathbf{r}$ distinct labels in positions $L_0,...,L_{\mathbf{r}-1}$. A new label may be added to position L_j after a design refinement and due to that all former labels in positions $L_j,\ldots,L_{\mathbf{r}-1}$ are shifted to $L_{j+1},\ldots,L_{\mathbf{r}}$. In this paper we use capital letters $(A,B,C,\ldots)$ for labels. Delay processes have identical labels if they have the same delay to a multi-input process. We give the first letter to an arbitrarily chosen P_{Δ} process and mark the rest of P_{Δ}-s according to the following rule. If a processes $P_{\Delta i}$ have got label L_j and there is an edge from w_i to w_k in G then process $P_{\Delta k}$ gets label L_{j+1} (L_0 if

[2] A proof can be found in [13]

$j+1 == \mathbf{r}$). Similarly, process $P_{\Delta l}$ gets label L_{j-1} ($L_{\mathbf{r}-1}$ if $j == 0$) if there is an edge from w_l to w_i.

4.9 Synchronization after design refinements

We model the design transformation caused delay increase in a combinational process as an additional delay process $P_{\Delta^\perp}$ in the same path next to the refined process. In order to synchronize the processing of $\perp$ values, we insert additional $P_{\Delta^\perp}$ processes to the model as described in the following algorithm:

Algorithm 1

Step 1 Add the refinement produced delay process $P_{\Delta^\perp j}$ to the model.

Step 2 Find the closest delay processes $P_{\Delta^\perp k}$ and $P_{\Delta^\perp l}$, which have paths from $P_{\Delta^\perp k}$ to $P_{\Delta^\perp j}$ and from $P_{\Delta^\perp j}$ to $P_{\Delta^\perp l}$, respectively.

Step 3 Take the labels of $P_{\Delta^\perp k}$ and $P_{\Delta^\perp l}$, which have to locate in neighbor positions L_{t-1} and L_t in the set *Label*.

Step 4 Associate a new label with $P_{\Delta^\perp j}$.

Step 5 Preserving the order, shift all labels in positions $L_t, \ldots, L_{\mathbf{r}-1}$ to $L_{t+1}, \ldots, L_{\mathbf{r}}$.

Step 6 Add the new label, associated with $P_{\Delta^\perp j}$ to *Label* in position L_t.

Step 7 Insert a $P_{\Delta^\perp}$ delay process with label L_t to every path, which starts from a process with label L_{t-1}, finishes at a process with label L_{t+1}, and the path does not include any other P_Δ or $P_{\Delta^\perp}$ processes.

So far in the algorithm the system input and output signals are not considered. Since the number of events that a system has to process increases due to the refinement added and synchronization delay processes $P_{\Delta^\perp}$, we have to insert additional synchronization events to input signals as well. An input signal can be viewed as a shift register where from one event e_i enters to the system at every clock cycle. Similarly to delay processes P_Δ we label the input events as well. The event e_0 gets a label based on the label of the closest P_Δ delay process to this input. If the process has got label L_j, then event e_0 has label L_{j-1}, e_1 has label L_{j-2} and e_k has a label L_l, where $l = ((j-(k+1)) \; mod \; \mathbf{r})$. If a refinement adds a $P_{\Delta^\perp}$ process to the model with label L_j, the input signals have to include extra absent events between events with labels L_{j-1} and L_{j+1}. In order to add regularly absent events to the input signals, finite state machine based interfaces can be used. Although absent events appear on the system output signals as well, the refined and synchronized system is latency equivalent to the original system. After excluding the absent events from an output signal, it is identical to the output signal of the original model.

5. CASE STUDY

We illustrate the synchronization algorithm within the design process of a video encoder. The encored is divided into fifteen computation blocks and we assume that all blocks have exactly one clock cycle delay, i.e., in the path between block's input and output is one P_Δ delay process, in every block. The structure of the original model is identical to the delay graph G in Figure 3 leaving out shadowed processes $P^\perp_{\Delta 16}, \ldots, P^\perp_{\Delta 19}$. Let's consider a design refinement that introduces resource sharing in the combinational part of the $block_{11}$ (containing delay process $P_{\Delta 11}$) and therefore adds one clock cycle delay to the block. Since this delay and its initial value is not defined in the system specification, we model this delay as a $P^\perp_{\Delta 16}$ process.

In order to synchronize the model after the refinement we have to find all loops and pairs of paths. There are six loops and fourteen pairs of paths in G. Although in the worst case the number of paths

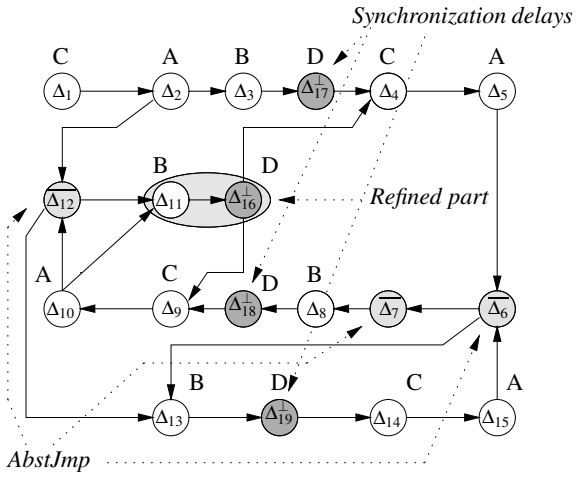

Figure 3: The delay graph G after refinement and synchronization

may grow exponentially with the number of delay processes in the model and to give exponential complexity to our algorithm, in practice there are not direct connections between every two processes and the number of paths stays close to the number of processes. It took only 0.05 seconds to find all loops and pairs in our modeling environment running on a SUN Ultra 80 machine (450MHz CPU and 4GB RAM). Some examples of loops are: $loop_1$ through vertices $\{\Delta_9, \Delta_{10}, \Delta_{11}\}$, $loop_2$ through vertices $\{\Delta_9, \Delta_{10}, \Delta_{12}, \Delta_{11}\}$, $loop_3$ through vertices $\{\Delta_6, \Delta_{13}, \Delta_{14}, \Delta_{15}\}$, and so on. In matrix $\mathcal{M}$ (Figure 4) the first six rows $r_1, \ldots, r_6$ represent loops. If delay Δ_i belongs to $loop_j$, the matrix element $m(r_j, c_i)$ gets value "1". Similarly to loops, we mark the pairs of paths in $\mathcal{M}$. For all pairs we mark the delays of the shorter path with "-1" and the longer path with "1" in matrix $\mathcal{M}$.

r_i \ c_i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Σr_i	δ_i	$\overline{\Sigma r_i}$	$\overline{\delta_i}$
$loop_1$	1								1	1	1					3	0	3	0
$loop_2$	2								1	1	1	1				4	1	3	0
$loop_3$	3					1							1	1	1	4	1	3	0
$loop_4$	4					1	1	1	1	1		1	1	1	1	9	0	6	0
$loop_5$	5			1	1	1	1	1	1	1	1					8	2	6	0
$loop_6$	6			1	1	1	1	1	1	1	1	1				9	0	6	0
$pair_{2,4}$	7		-1								1	1				1	1	0	0
$pair_{2,4}$	8		-1			1	1	1	1	1	1	1	1	1	1	9	0	6	0
$pair_{2,6}$	9		-1	-1	-1							1	1	1	1	1	1	0	0
$pair_{6,13}$	10						1	1	1	1		1				5	2	3	0
$pair_{10,6}$	11				-1	-1					-1	1	1	1	1	1	1	3	0
$pair_{10,9}$	12					1	1	1			-1	1	1	1	1	6	0	3	0
$pair_{10,11}$	13											1				1	1	0	0
$pair_{10,13}$	14				1	1	1				1	-1				3	0	3	0
$pair_{11,6}$	15				-1	-1				1	1		1	1	1	1	4	1	3	0
$pair_{11,9}$	16				1	1	1	1								5	2	3	0
$pair_{11,13}$	17				-1	-1	-1			1	1		1				0	0	0	0
$pair_{12,6}$	18				-1	-1						-1		1	1	1	0	0	0	0
$pair_{12,11}$	19						1	1	1	1				1	1	1	8	2	6	0
$pair_{12,13}$	20				1	1	1				1						4	1	3	0

Figure 4: Matrix $\mathcal{M}$

The shortest loop in G includes three delay processes, which makes the refined and synchronized system to operate with ratio $\mathcal{R} = (1\perp : 3\top)$, $\mathbf{r} = 3$. The sum of elements on every row Σr_i and the offsets δ_i of loops and pairs are given in the right side columns in Figure 4. The Cadence SMV [14] model checker, running on

the SUN machine, created 622 BDD-nodes and spent 0.11 seconds to find that the model is balanced after modifying processes $\{\Delta_6, \Delta_7, \Delta_{12}\}$. Modifications by *AbstJmp* in these processes turn all values in columns c_6, c_7 and c_{12} to zero (the shadowed columns in matrix $\mathcal{M}$). Updated values of Σr_i and δ_i are presented as $\overline{\Sigma r_i}$ and $\overline{\delta_i}$, respectively, in Figure 4.

In the next step we form a three element set $Label = \{L_0 = A, L_1 = B, L_2 = C\}$ and label all Δ_i processes as shown in Figure 3. The refinement in $block_{11}$ adds a delay $\Delta^{\perp}_{16}$ between delay Δ_{11} and delays Δ_4 and Δ_9. Since the refinement is performed between delays with labels B and C, (1) we give a new label D to process Δ_{16}, (2) give position L_2 to D in the set *Label*, (3) shift the label C to position L_3, and (4) insert synchronization delays $\Delta^{\perp}_{17}$, $\Delta^{\perp}_{18}$ and $\Delta^{\perp}_{19}$ into all paths between delays with labels B and C. The new set of labels after the refinement is $Label = \{L_0 = A, L_1 = B, L_2 = D, L_3 = C\}$. The input signal $\{e_0, e_1, e_2, e_3, e_4, e_5, e_6, ...\}$ gets labels $\{e^B_0, e^A_1, e^C_2, e^B_3, e^A_4, e^C_5, e^B_6, ...\}$, and after adding regular $\perp$-events the signal has got the form: $\{\perp^D, e^B_0, e^A_1, e^C_2, \perp^D, e^B_3, e^A_4, e^C_5, \perp^D, e^B_6 ...\}$.

Compared to the LID algorithm [4], our synchronized model after the refinement contains only three additional delay processes and three delay processes are transformed to sensitive delays. Although the LID method does not add any explicit delay process, it adds wrappers around processes and replaces signals between processes with handshake channels between wrappers. In fact, there are combinational logic circuits in wrappers, and buffers in channels and wrappers, which increase the circuit area. In order to implement the absent extension at RT-level, we needed only one bit signal to inform processes about the current data type of an input value ($\perp$ or $\top$). This is equivalent to the one bit signal used to distribute stalling events between computation blocks in LID. The input/output latency of our model is equal the latency of an LID model after the same refinement - both models have to work with ratio: one synchronization (stalling) event per three data events. The LID relay stations in the channels are initialized with stalling values. Since the system contains feedback loops these values are reproduced [4]. Due to the feedback loops, the $\perp$-events are reproduced in our model as well.

The same model synchronized by the algorithm in [13], contains fourteen synchronization delays and operates with ratio $(1\perp : 1\top)$. Compared to the latter, sensitive delays gave 79% improvement in the number of synchronization delays and 67% improvement in the ratio between synchronization and actual data events.

6. CONCLUSION

The introduction of resource sharing and pipelining in computation blocks are only some examples of design refinements, which increase the delay in the refined blocks compared to the original ones. Although the explicit change is made in a single block, it influences the functional behavior of the entire system in the synchronous model of computation. The proposed algorithm solves the synchronization problem and makes it possible to use the synchronous model even at late stages of the design process.

Our synchronization algorithm (1) does not modify the functionality of combinational processes, (2) does not add schedulers or controllers as they are used for process execution in pipelined systems and in data-flow models, and (3) does not introduce wrappers, handshake communication channels or schedulers as used in LID approaches. This leaves the system simpler to analyze, to verify, and to apply the further design refinements. The only resources we add to the model are the regularly placed synchronization delay processes ($\Delta^{\perp}$), and we introduce sensitive delays, which do not delay synchronization events ($\perp$). The proposed algorithm is more efficient compared to the algorithm described in [13], since it requires fewer synchronization delays and allows the system to operate with a better ratio between synchronization and actual data events.

7. REFERENCES

[1] S. Abdi and D. Gajski. Automatic generation of equivalent architecture model from functional specification. In *Proceedings of the 41st Annual Conference on Design Automation, (DAC'04)*. ACM Press, 2004.

[2] A. Benveniste, B. Caillaud, and P. L. Guernic. From synchrony to asynchrony. In *Proceedings of the International Conference on Concurrency Theory*, 1999.

[3] G. Berry, M. Kishinevsky, and S. Singh. System level design and verification using a synchronous language. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'03)*, Washington, DC, USA, 2003. IEEE Computer Society.

[4] L. P. Carloni and A. L. Sangiovanni-Vincentelli. A framework for modeling the distributed deployment of synchronous designs. *Formal Methods in System Design*, 28(2):93–110, 2006.

[5] M. R. Casu and L. Macchiarulo. A new approach to latency insensitive design. In *Proceedings of the 41st Annual Conference on Design Automation (DAC '04)*. ACM Press, 2004.

[6] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, March 1997.

[7] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *Software Engineering*, 18(9):785–793, 1992.

[8] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, December 2000.

[9] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[10] N. Maheshwari and S. S. Sapatnekar. Minimum area retiming with equivalent initial states. In *Proceedings of the International Conference on Computer Aided Design (ICCAD'97)*, 1997.

[11] S. Nadjm-Tehrani and J.-E. Strömberg. Formal verification of dynamic properties in an aerospace application. *Formal Methods in System Design*, 14(2):135–169, March 1999.

[12] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. In *Proceedings of the International Conference on Application of Concurrency to System Design*, St Malo, France, 2005.

[13] T. Raudvere, I. Sander, and A. Jantsch. A synchronization algorithm for local temporal refinements in perfectly synchronous models with nested feedback loops. In *Proceedings of the Great Lakes Symposium on VLSI'07*, Stresa, Italy, March 2007.

[14] The SMV model checker. online [available] http://www-cad.eecs.berkeley.edu/∼kenmcmil/smv/.

[15] M. Weinhardt and W. Luk. Pipeline vectorization. *In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 234–248, Feb 2001.