A case study for the verification of complex timed circuits: IPCMOS [*]

Marco A. Peña[†] and Jordi Cortadella[‡] and Enric Pastor[†] and Alexander Smirnov[‡]

† Department of Computer Architecture	‡ Department of Software
Technical University of Catalonia	Technical University of Catalonia
08860 Castelldefels (Barcelona), Spain	08034 Barcelona, Spain
{marcoa, enric}@ac.upc.es	{jordic, alexs}@lsi.upc.es

Abstract

The verification of a n-stage pulse-driven IPCMOS pipeline, for any $n > 0$, is presented. The complexity of the system is $32n$ transistors and delay information is provided at the level of transistor. The correctness of the circuit highly depends on the timed behavior of its components and the environment. To verify the system, three techniques have been combined: (1) relative-timing-based verification from absolute timing information [13], (2) assume-guarantee reasoning to verify untimed abstractions of timed components and (3) mathematical induction to verify pipelines of any length. Even though the circuit can interact with pulse-driven environments, the internal behavior between stages commits a handshake protocol that enables the use of untimed abstractions. The verification not only reports a positive answer about the correctness of the system, but also gives a set of sufficient relative-timing constraints that determine delay slacks under which correctness can be maintained.

1. Introduction

Verification of concurrent systems typically suffers from the state explosion problem. In systems with a finite number of states, this problem has been alleviated by using symbolic techniques to implicitly enumerate all reachable states [4]. Abstraction has also been a common technique to reduce the complexity of the model, by hiding those implementation details that are irrelevant to the properties begin verified [12].

When time becomes an essential dimension in verification, complexity is drastically affected. The problem of reachability in *timed automata* is proved to be PSPACE-hard [1], where the exponentiality depends on the number of clocks and on the encoding of the maximum values that can be taken by the clocks. This complexity makes the verification of systems with a moderate amount of untimed states almost impractical.

Several approaches have been devised to represent timed states in a succinct form, *e.g.* [3]. However, the incorporation of the timed domain in the representation of the states hampers an efficient representation of large state spaces with BDDs. Even the discretization of time [8] poses serious problems when the number of clocks or the constants of the timing constraints are large.

An interesting approach to face this complexity problem was proposed in [2], where the clocks used during verification and their accuracy are determined dynamically upon demand. In this way, only that timing information relevant to the properties being verified emerges during the calculation of the reachable states.

This paper tackles the verification of complex timed systems by combining three techniques:

1. Iterative verification by automatically abstracting absolute time into relative time [13] and using polynomial algorithms

for absolute timing analysis [10].

2. Assume-guarantee paradigm to perform a hierarchical verification of large systems by means of abstractions.

3. Mathematical induction to prove the correctness of infinite-state systems.

These techniques have been used to verify the IPCMOS control circuit [15], a controller for asynchronous pipelines that can operate at 4GHz and uses a pulse-driven protocol to communicate with the environment. Its correctness highly depends on the delays of the internal gates and the environment.

The main feature of the used relative-time-based verification is the capability of backannotating the timing constraints required for the correctness of the circuit. These constraints indicate the slacks allowable in the delays of the components for which a correct behavior can still be guaranteed. This feature does not seem easily achievable with the strategies used so far for the verification of timed automata.

Section 2 reviews the fundamentals of relative-timing verification [13] and compositional verification. Section 3 describes the IPCMOS architecture and the required verification steps. Section 4 gives the details of the abstractions and steps to verify IPCMOS pipelines. Section 5 describes the verification of one stage.

2. Theoretical background

2.1. Formal verification with relative timing

Modeling formalisms and algorithms for the verification with relative timing are described in detail in [13]. This section describes its fundamentals.

Verification strategy. To verify whether a timed system Y satisfies a safety property P a language inclusion test is posed: $\mathcal{L}(Y) \subseteq \mathcal{L}(P)$, where $\mathcal{L}(Y)$ is the set of all possible behaviors of Y and $\mathcal{L}(P)$ is the set of all possible behaviors satisfying property P. The calculation of the language generated by a timed system is proven to be PSPACE-complete and in practice highly complex in most contexts [1, 7]. To overcome this complexity, [13] avoids calculating the exact timed state space by building conservative approximations of $\mathcal{L}(Y)$ in an iterative manner. Starting from the original system without timing constraints, successive approximations Y_i of Y are constructed, satisfying $\mathcal{L}(Y) \subseteq \mathcal{L}(Y_i)$ and $\mathcal{L}(Y_{i+1}) \subseteq \mathcal{L}(Y_i)$. If the inclusion $\mathcal{L}(Y_i) \subseteq \mathcal{L}(P)$ holds, then $\mathcal{L}(Y) \subseteq \mathcal{L}(Y_i) \subseteq \mathcal{L}(P)$ and the verification succeeds. If not, another approximation Y_{i+1} is generated by adding relative timing constraints [16] until the verification succeeds, or a counterexample is found. At each iteration of the refinement process, relative timing constraints are generated from an *off-line* timing analysis [10] on a set of event structures that covers the traces containing violations of property P.

System model. The system under analysis Y is modeled by means of a *timed transition system* (TTS) [7] composed of a non-empty
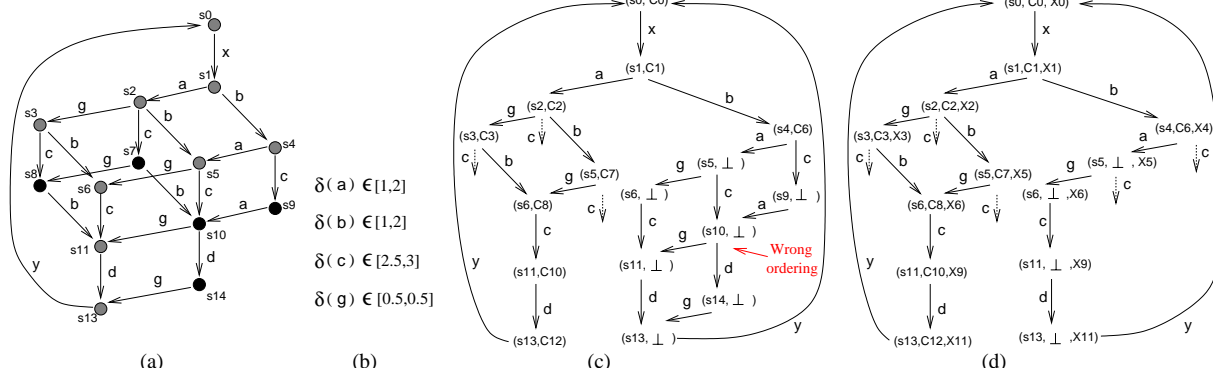
COMPUTER SOCIETY

Figure 1. (a,b) Timed transition system and delay intervals. (b,c) LzTS obtained after first and second compositions.

set of *states* S, a non-empty alphabet of *events* Σ, a *transition relation* $T \subseteq S \times \Sigma \times S$, a set of *initial states* s_{in}, and two functions δ^l, δ^u that associate minimal an maximal delays to the events. We call the subset $Y^- = \langle S, \Sigma, T, \mathsf{s}_{in} \rangle$ the underlying *transition system* (TS) that represents the untimed behavior of the system.

Figures 1(a,b) depict the TTS modeling a system that will be used as introductory example. Figure 1(a) shows the underlying TS, while Figure 1(b) shows the delay intervals of events a, b, c and g. The delay interval for the rest of events is $[0, \infty)$. The timed behavior of the system is highlighted in gray, *i.e.* the reachable states when the specified delays are taken into account.

Assume that the property to verify specifies that event g must always precede event d in any possible trace from state s_0. The property holds in the timed state space since in all possible timed traces (those visiting the gray states), g always fires before d. However, by exploring the untimed state space of Figure 1(a) we see that the property does not hold in state s_{10} if d fires before g. What follows is an overview of the mathematical notions involved in proving that the property actually holds when delays are considered.

A run of the underlying TS Y^- is a sequence $\sigma = \mathsf{s}_1 \xrightarrow{\mathsf{e}_1} \mathsf{s}_2 \xrightarrow{\mathsf{e}_2} \cdots$, such that $\mathsf{s}_1 \in \mathsf{s}_{in}$ and $\forall i \geq 1 : \mathsf{s}_i \xrightarrow{\mathsf{e}_i} \mathsf{s}_{i+1} \in T$. When considering the timing constraints in the TTS, not all runs of Y^- correspond to feasible execution sequences. σ is *timing consistent* with Y if a sequence $t_1 \leq t_2 \leq \cdots$ of real-valued time stamps can be found in which for every $\mathsf{s}_i \xrightarrow{\mathsf{e}_i} \mathsf{s}_{i+1}$ in σ we have $\delta^l(\mathsf{e}_i) \leq t_{i+1} - t_j \leq \delta^u(\mathsf{e}_i)$. The time stamp t_{i+1} is assigned to state s_{i+1} and corresponds to the *firing time* of event e_i. Similarly, t_j corresponds to the time stamp of the state s_j preceding s_{i+1} in which e_i was first enabled (*enabling time*). Thus, the firing time of an event only depends on its enabling time and certain delay amount within the specified bounds.

Lazy transition systems [5] are used to build the sequence of aforementioned approximations for $\mathcal{L}(Y)$. Laziness enables to model time by abstracting absolute timing information into relative timing information. This is accomplished by explicitly distinguishing between the enabling and the firing of an event.

Sequences of events leading to states in which properties do not hold are specified by means of traces with enabling information. Given an alphabet of events Σ, a trace $\theta = E_1 \xrightarrow{\mathsf{e}_1} E_2 \xrightarrow{\mathsf{e}_2} \cdots$ is a sequence such that $\forall i \geq 1 : E_i \subseteq \Sigma$ and $\mathsf{e}_i \in E_i$, where E_i is the set of events enabled when e_i fires. Removing some of the events from Σ enables a trace to capture sets of sequences that share the same enabledness, *i.e.* it is an *enabling compatible mapping* [13]
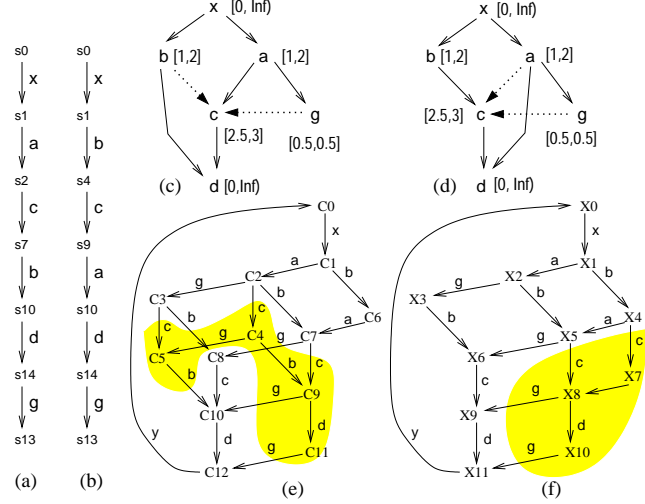

Figure 2. (a,b) Failure traces, (c,d) their corresponding CESs annotated with timing arcs, and (e,f) State space of the CESs (shaded states are unreachable).

between traces. Since the firing time of an event only depends on its enabling time and its delay, the timing analysis on a trace θ' with a subset of events $\Sigma' \in \Sigma$ can also be applied to the original trace θ with the full set Σ; that is, θ is timing consistent iff θ' is timing consistent.

Event structures (CES) are acyclic graphs that capture the causality relations between the set events in a trace. Taking an event structure, which partially specifies the causality in the original system, timing analysis can be performed and the resulting timing constraints incorporated into the specification. The causal event structure $CS_\theta = \langle \Sigma, \prec \rangle$ (where $\prec$ is a precedence relation) generated from trace θ is defined as follows: $\Sigma = \{\mathsf{e}_1, \ldots, \mathsf{e}_n\}$, $\mathsf{e}_i \prec \mathsf{e}_j \Leftrightarrow i < j \wedge \nexists E_k \in \theta : \{\mathsf{e}_i, \mathsf{e}_j\} \subseteq E_k$. By analyzing the *maximal separation time* between pairs of events in a CES [10], it is possible to determine whether two events are ordered in the time domain. Temporal ordering due to event separation is incorporated into the CES by additional temporal relations which represent relative timing constraints [16]. The result is a *lazy event structure* LzCES [13] in which additional temporal relations delay the firing time of events, but never modify their enabling time.

Doing reachability analysis on the resulting LzCES, a new LzTS G is obtained, which contains the derived relative timing
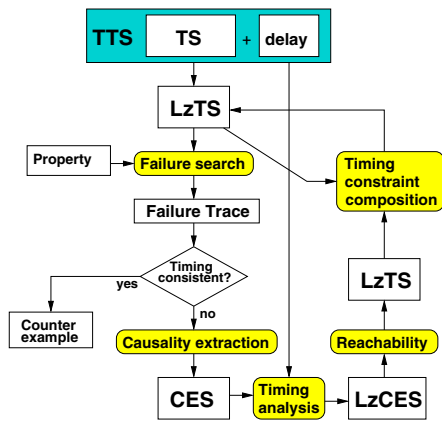
Figure 3. Flow of the verification methodology.

constraints. Refining the behavior of Y by the timing constraints in G is done by calculating the *enabling-compatible product* of Y and G [13]. This product is a particular case of TS product under the restrictions of making synchronization by the *same transitions* and the *same enabling conditions*.

Verification flow. The proposed verification method follows a fully automated iterative approach (see Figure 3). The verification starts by taking a LzTS equivalent to the underlying TS of the system under analysis. In that case the enabling and the firing of all events coincide since no timing information has been considered yet. Given a safety property P, a trace is identified that leads to some state in which P is violated. If the trace is timing consistent then the system violates the required property and the trace provides a counter-example. However, if the trace is not timing consistent, it is used to refine the untimed state space and remove other timing inconsistent traces. Causality information between the events in the trace is extracted and a CES is built from it. Timing analysis on the CES is performed by using the algorithm in [10]. The extracted temporal information is incorporated obtaining a LzCES which is composed with the original LzTS, thus including the temporal information necessary to prove that some of the states in the system are unreachable. The process is repeated until no violation of P exists or a timing consistent failure trace is found.

Going back to the example in Figures 1(a,b), a failure trace from s_0 to s_{10} followed by the firing of d can be found (see Figure 2(a)). From this trace, a CES with the same causality relations is derived (Figure 2(c)). Note that, in the CES, c is triggered by a but not triggered by b, *i.e.* only a subset of the causality relations is captured. Figure 2(e) depicts the state space of the CES. After the timing analysis we add the temporal relations to the CES, denoted as dotted lines in Figure 2(c), which make unreachable all shadowed states in Figure 2(e). Hence, event c is prevented to fire in some states, where its firing would be inconsistent with the timing analysis. Finally, this information is incorporated by composing the original system and the event structure. Some states in the composed system are split into two instances depending on whether they are reached by traces matching (*enabling compatible*) the event structure or not (see states s_5, s_6, s_{11} and s_{13}). Figure 1(c) shows the resulting LzTS, where states annotated with the $\perp$ symbol are not enabling compatible and thus the timing analysis on the CES does not apply. It can be seen that the set of traces is smaller than that of the original system, but larger than that of the actual
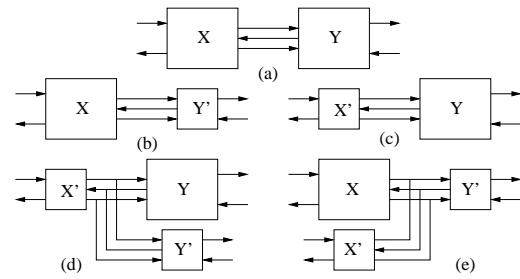

Figure 4. Assume-guarantee verification using abstractions.

state space in Figure 1(a), and that only timing inconsistent traces have been removed.

The first iteration of the algorithm has removed some failure traces but not all of them. Figure 2(b,d,f) depicts one more refinement, with. the final state space in Figure 1(d). In the resulting system all the failure traces have been removed, which proves that the system satisfies the property. Although it is not generally true, in this case the final state space contains exactly the same traces than the actual state space (the gray states) shown in Figure 1(a).

2.2. Compositional verification

The *abstraction* mechanism [12] allows to reduce the size of the state space by removing details irrelevant for proving a given property. When performing abstraction, information about the exact behavior of the system is lost, therefore the truth of some properties cannot be determined by looking only at the abstracted system.

The *assume-guarantee* paradigm [14] exploits the modular structure of systems. It reasons about the correctness of the overall system by checking only the local properties of the components. Unfortunately, a component is designed to operate only in the environment of that system, thus it is unlikely to satisfy any interesting property unless analyzed together with such environment. The *assume-guarantee* technique tackles this intimate relation. In the system of Figure 4 (a), since the behavior of X depends on the behavior of Y, the correctness of X can be proved if certain *assumptions* are satisfied by Y. Then, one must *guarantee* that Y actually meets such assumptions. A similar reasoning can be done in the side of Y. By combining appropriately the assumed and guaranteed properties, it is possible to establish the correctness of the entire system. without building the global state space. Moreover, once a guarantee is proved it can be used as an assumption for a later stage in the verification process. To prevent from erroneous conclusions circularity must be avoided in the reasoning chain. Finally, the assumptions often come in the form of abstractions such that both techniques are combined. Figure 4 depicts the verification scheme for the $X \| Y$ system, using assume-guarantee reasoning with abstractions: in (b) the local properties of X are verified assuming Y' is a valid abstraction of Y; in (c) the local properties of Y are verified assuming X' is a valid abstraction of X; in (d) Y' is checked to be a valid abstraction of Y in order to guarantee (b); and in (e) X' is checked to be a valid abstraction of X in order to guarantee (c). Each "guarantee" verification checks for language containment of the implementation with respect to the abstraction. In our framework, this is performed (following [13]) by checking that any output produced by the implementation can also be produced by the abstraction under the same input stimuli.

Induction is used to prove properties on systems composed of a number of similar components, organized in some inductively
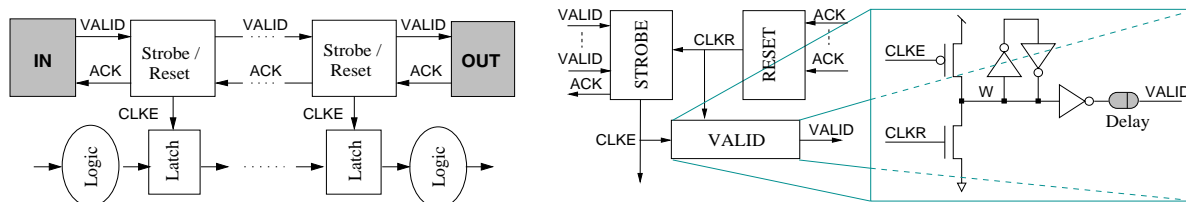
Figure 5. IPCMOS pipeline. Detail of the *Strobe/Reset* **control block and implementation of the** *valid* **module.**
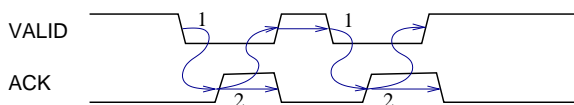

Figure 6. Two phase handshake mechanism.

definable structure like a pipeline, a matrix, etc. These techniques rely on the concept of *invariant* [6] or the so-called *behavioral fixed point* [17], to reason about the behavior of systems with any number of components.

Formal frameworks that support assume-guarantee reasoning with abstractions [11] often rely on: a *preorder relation* $\leq$ and a *composition operator* $\|$ for processes, and a *logic* to specify properties. $X \leq X'$ denotes that the abstraction X' captures more behaviors than X, *i.e.* X *refines* or *implements* X'. Since we verify safety properties, the only condition we have to enforce for the abstraction is that its state space is a superset of that of the original system, so that the observable properties are preserved through the abstractions. This, together with our relative-timing-based verification approach [13], provides a sound framework for performing assume-guarantee verification with abstractions.

3. Verification of IPCMOS circuits

3.1. The IPCMOS architecture

The Asynchronous Interlocked Pipelined CMOS circuit architecture (IPCMOS) [15] is an asynchronous clocking technique for large devices operating at GHz clock frequencies. Thanks to its pulse-based interlocking scheme, it can help addressing design issues such as power consumption, noise reduction and clock synchronization. A single IPCMOS module is relatively small and can be used to build scalable architectures. Figure 5 shows a pipeline composed of IPCMOS blocks and latches, with some details on the internal structure of a single control block.

The IPCMOS block communicates with other blocks via request signals (VALID), acknowledgment signals (ACK) and produces a local data clock signal (CLKE). VALID indicates data availability to the receiver(s), while ACK acknowledges to the sender(s) that data has been received. Generally IPCMOS blocks can be fed multiple ACK and VALID signals to enable safely processing data from multiple sources and feeding the result to multiple destinations.

IPCMOS circuits are *pulse-driven* or *edge sensitive*. Their operation is illustrated by the diagram in Figure 7. It shows how two data items propagate through an IPCMOS pipeline composed of two-stages, S1 and S2. Initially the pipeline is empty: all VALID signals are high, CLKE signals are high and ACK signals are low. As soon as negative pulses are received at all VALID inputs of a stage (only one in case of a pipeline), a positive pulse is generated at the ACK to acknowledge the data receipt. Input data is clocked by a negative pulse on the CLKE signal (produced
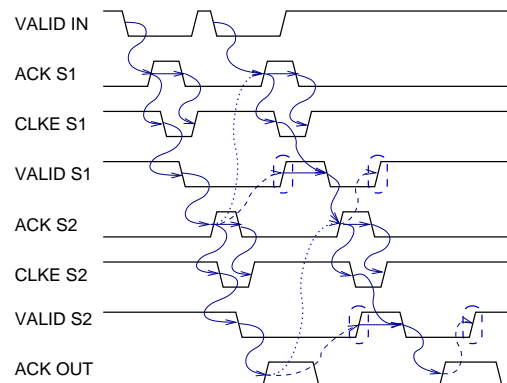

Figure 7. Two stage IPCMOS pipeline waveform.

by the *strobe* module of the IPCMOS architecture in the center of Figure 5). After some delay designed to match the worst case computation time of the logic, a negative pulse is generated by the *valid* module at the VALID line indicating the data availability to the receiver. From this point on, the block waits for positive pulses to be received from data consumers at all ACK inputs (recorded by the *reset* module). Meanwhile VALID input pulses indicating the new input data availability are also "recorded". Hence, new data receipt at every stage is *interlocked* with the acknowledgment of the data by the following stages. The only restriction the IPCMOS modules pose on the environment is the pulse length.

Even though a pulse-driven environment is accepted by each pipeline stage, the internal communication between adjacent stages is performed in a partially handshaked protocol between the positive edges of the pulses (see Figure 6). These additional causality relations enable to abstract the behavior of such components when interacting among them, in such a way that internal timing information can be neglected. This phenomenon considerably simplifies the verification of the pipeline.

The dashed boxes in Figure 7 show the affected edges while the dashed arrows show the restricting causal dependencies that must not exist in the environment (IN, OUT) but take place in the IPCMOS stages (S1, S2). On this diagram we also show that all stages in a sequence cannot be filled with data at the same time, but "bubbles" (empty stages) are needed to propagate data in one direction and the acknowledgment in the other. The causal dependencies demonstrating this fact are dotted in the diagram.

The complexity of a IPCMOS stage depends on the number of data suppliers and receivers. The number of transistors can be computed as: $N_{transistors} = 21 + 7 * N_{inputs} + 4 * N_{outputs}$. A single stage of a linear pipeline contains 32 transistors.

3.2. Verification steps

The correct operation of an IPCMOS pipeline initially empty, is given by the following informal specification (S):

Every data item sent to the pipeline is acknowledged once and only once at every stage.

We verify the correctness of the IPCMOS control circuit, *i.e.* the data path is assumed to be correct. Even though the previous property involves a liveness and a safeness condition, both can be modeled as safety conditions during the calculation of the state space. They can be modeled by means of a deadlock-freeness invariant in the control circuitry of the pipeline, such that the control deadlocks when either some data is not acknowledged or some data cannot move to the next stage.

Additionally, specific conditions about the correct behavior of CMOS circuits must be also ensured. These conditions are described in Section 5.1.

In particular, all properties required in this work have been modeled with very simple temporal expressions that require at most the analysis of 1-step transitions. Therefore, it is not necessary a powerful engine to verify branching time or linear time logic for such task.

Due to the pulse-driven nature of the architecture, its correctness strongly depends on the delay margins associated to the components of the control stage. The goal of the verification is to check whether an IPCMOS pipeline with any number of stages behaves correctly according to S, *i.e.* to check:

$$IN \parallel I_1 \parallel \cdots \parallel I_n \parallel OUT \leq S \qquad (1)$$

for any value of $n > 0$, where I_i are identical instances of the circuit implementation I of a stage. The environment is formed by the data sender *IN* and the data receiver *OUT*, which are indeed part of the specification in the sense that S also specifies the interface behavior of the pipeline, and *IN* and *OUT* can be obtained by simply mirroring such behavior. The verification becomes exponentially more costly as n increases, specially because the communication protocol in both ends of a stage is highly decoupled. Thus, if the verification is carried out using the level of detail provided by I, in practice n cannot go beyond 2 stages. In order to overcome the complexity, the verification of longer pipelines must be carried out using abstractions.

IN and *OUT* operate according to the pulse-based protocol and so does the left side of a stage, whereas the right side of a stage operates according to a two-phase handshake protocol (see Section 3.1). Therefore, the communications between stages I_2 and I_{n-1} inside the pipeline use the handshake scheme (thin arrows in Figure 8), whereas the pulse-based behavior only appears at the extremes of the pipeline (thick arrows in Figure 8). We propose abstractions that hide the pulse-based behavior in a way that all the timing restrictions related to the correctness of such protocol are also encapsulated inside the abstractions (see A_{in} and A_{out} in Figure 8). Therefore, since A_{in} and A_{out} communicate through the handshake protocol, the abstractions can be untimed and *assume-guarantee* reasoning can be used. Hence, we pose the verification of (1) in terms of:

$$A_{in} \parallel A_{out} \leq S \qquad (2)$$

We proceed as follows:

- Build abstractions A_{in} and A_{out} for the components shown in Figure 8. *IN* and *OUT* communicate by pulses with I, whereas I communicates by handshakes with the pipeline.

- Prove (2) *assuming* that A_{in} and A_{out} are correct abstractions of the respective parts of the pipeline.
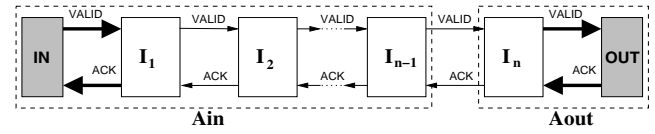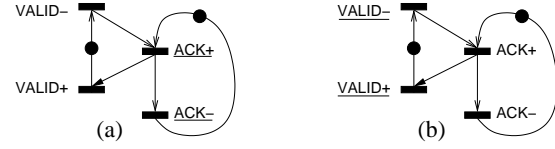

Figure 8. Pipeline verification using abstractions.

Figure 10. STGs for the abstractions A_{in} **(a) and** A_{out} **(b).**

- *Guarantee* the soundness of the abstractions. Discharge the assumptions by proving that A_{in} and A_{out} are correct abstractions of $IN \parallel I$ and $I \parallel OUT$, respectively. Moreover, prove that A_{in} is also a good abstraction of $A_{in} \parallel I$, *i.e.* A_{in} is a *behavioral fixed point* that abstracts the sender *IN* and a chain of n stages.

- Finally, prove the correctness of a pipeline formed by a single fully detailed implementation (I) of a stage. The verification checks if I is a correct CMOS circuit and satisfies S in the given environment, that is $IN \parallel I \parallel OUT \leq S$.

Section 4 covers the first three items, whereas Section 5 shows in detail the use of [13] to perform the proof in the last item.

4. Verification of IPCMOS pipelines

The verification methodology of [13] is used in this section to perform the experiments of the *assume-guarantee* strategy outlined in Section 3.2.

4.1. Abstractions

The models A_{in} and A_{out} must describe the observable behavior of the abstracted parts of the pipeline at a higher level (see Figure 8). The two-phase handshake protocol in the communication interface of the abstractions is modeled by the fact that the rising edge of ACK to acknowledge a data portion is always interlocked within the falling edge and the next rising edge of VALID. The models for the environment (*IN* and *OUT*) used for the verification are shown in Figure 12. Figure 10 depicts the models for the abstractions A_{in} and A_{out}. These models are represented by Signal Transition Graphs (STG), *i.e.* Petri nets in which transitions are interpreted as rising (+) and falling (−) signal transitions. The underlined transitions represent inputs in their respective models.

A_{in}: **abstraction of** *IN*-$I_1 - \cdots - I_{n-1}$. A_{in} hides the pulse-based communication between *IN* and the last stage of the pipeline. A_{in} signals the data availability at the input of the next stage of the pipeline by lowering the output VALID line. VALID is not raised again until the pipeline acknowledges the receipt of the data. The two-phase handshake protocol (see Figure 6) is completed by resetting the VALID line independently of the resetting of the ACK line by the pipeline.

A_{out}: **abstraction of** *I-OUT*. A_{out} hides the pulse-based communication between the last stage of the pipeline and the *OUT* module. A_{out} samples the data available at the end of the pipeline signaled by the low value of VALID, and acknowledges it by producing a positive ACK pulse.
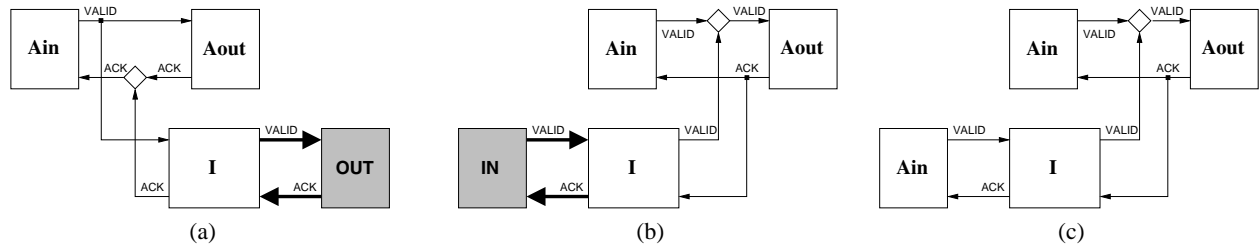
Figure 9. Scheme of the *guarantee* **part of the verification, to prove the correctness of the various abstractions.**

4.2. Assume-guarantee verification

We want to prove that the abstract system built from A_{in} and A_{out} is a good abstraction of an IPCMOS pipeline, *i.e.*:

$$IN \parallel I_1 \parallel \cdots \parallel I_n \parallel OUT \leq A_{in} \parallel A_{out}$$

We use *assume-guarantee* reasoning in five steps in order to carry out the proof. Steps 3 and 4 are the ones that use induction to prove the correctness of an n-stage pipeline, for $n \geq 2$.

Some of the verification steps are graphically depicted in Figure 9. The symbol $\diamond$ models a component that checks that any event produced by the refinement is also produced by the abstraction (i.e. the language produced by the refinement is included in the one produced by the abstraction).

1. Assume: We must prove that the system formed by the abstractions meets the specification of the IPCMOS pipeline, that is: $A_{in} \parallel A_{out} \leq S$. The task is straightforward and completes successfully in a few seconds of CPU time.

2. Guarantee correctness of A_{out}: We prove the correctness of A_{out} with respect to the system formed by a stage I and the OUT module, when I communicates with the rest of the pipeline using the handshake protocol. That is: $A_{in} \parallel I \parallel OUT \leq A_{in} \parallel A_{out}$. For this, the system shown in Figure 9(a) is built. The verification consists in checking the language containment of $I \parallel OUT$ with respect to A_{out}, which is reduced to checking that any output produced by $I \parallel OUT$ can also be produced by A_{out} at the same time instant. In this case, the only relevant output is signal ACK.

3. Guarantee correctness of A_{in} with one stage: We prove the correctness of A_{in} with respect to the system formed by the pulse-based *IN* module and the implementation of a stage of the pipeline I, when I communicates with the next stage in the pipeline using the handshake protocol. That is: $IN \parallel I \parallel A_{out} \leq A_{in} \parallel A_{out}$. For this analysis, the system shown in Figure 9(b) is built. The verification consists in checking that whenever I is ready to change the value of VALID, A_{in} is also ready for that, thus guaranteeing language containment of $IN \parallel I$ with respect to A_{in}.

4. Guarantee A_{in} is a *behavioral fixed point*: The previous proof only guarantees the correctness of A_{in} as an abstraction of *IN* and a single stage. That result serves as the induction hypothesis to prove that A_{in} is a correct abstraction of $IN \parallel I \parallel \cdots \parallel I_{n-1}$, for any $n \geq 2$, as shown in Figure 8. Namely, $A_{in} \parallel I \parallel A_{out} \leq A_{in} \parallel A_{out}$. For this, the system shown in Figure 9 (c) is built and the verification is done similarly to the previous proofs, but now checking signal VALID.

Proofs 3 and 4 prove the correctness of an n-stage pipeline. A_{in} is an abstraction of the *IN* module and a chain of IPCMOS stages

Experiment	CPU time	Refinements
1.	< 1 min.	–
2.	28 min.	7
3.	9 min.	3
4.	10 min.	3
5.	35 min.	40

Table 1. Summary of experimental results

and is said to be a behavioral fixed point [17], since no matter how large n is, A_{in} can be used as a correct abstraction.

5. Guarantee correctness of a 1-stage pipeline: The previous proofs demonstrate the correctness of IPCMOS pipelines with 2 or more stages. It is still needed to prove the correctness of a pipeline with a single stage, that is $IN \parallel I \parallel OUT \leq S$. This step is necessary to consider the case in which a stage is interacting with a pulse-driven environment at both sides. This is the step in which more timing constraints are required to guarantee a correct behavior of the components. Despite of its complexity, since this step also requires the refinement of the model at the level of transistors, we describe it in detail in Section 5.

Table 1 summarizes the results of the five verification steps, using the transyt tool implementing the relative-timing-based verification approach described in [13]. The CPU times, indicated in the second column, have been rounded to minutes and correspond to executions in a 866Mhz PIII computer with 1Gb of RAM running Linux. The number of refinements, indicated in the third column, correspond to the number of iterations needed by transyt to successively incorporate the timing constraints that help pruning the failure traces from the state space of the corresponding models.

In the first experiment, no refinement is required since the verification only consists in computing the untimed state space of the abstractions involved and realizing that no violation of the specification arises. Notice also, that although experiments 2, 3 and 4 require a few refinements the CPU times are comparatively high with respect to that of experiment 5. This is due to the complexity of the required models (see Figure 9) and the resulting BDD explosion when doing reachability analysis.

5. Verification of a 1-stage IPCMOS circuit

This section addresses the verification of the circuit implementation of an IPCMOS pipeline stage (I), in an environment formed by a data sender *IN* and a data receiver *OUT*. We will show the modeling mechanisms to describe the transistor-level circuit, the description of environment and the verification results.

5.1. Modeling CMOS circuits

The behavior of the system is modeled by a TTS with a set of events that update the value of variables and therefore modify the
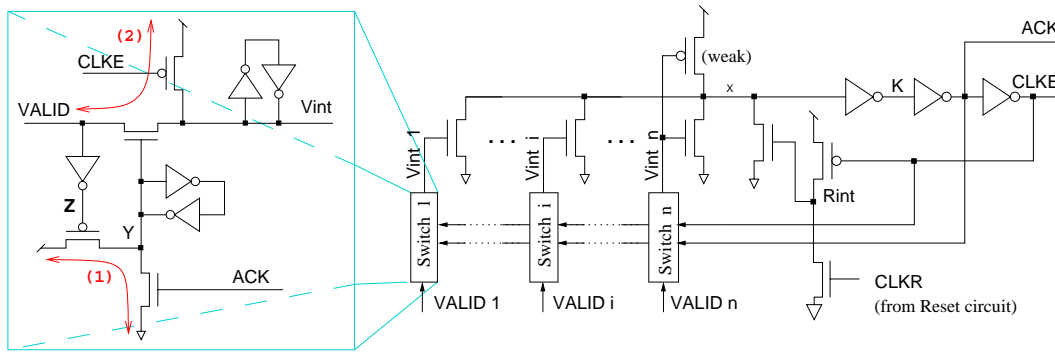
Figure 11. General structure of the *strobe* **(right) and** *strobe switch* **(left) circuits.**

state of the system. We use a boolean variable to model each circuit node and several events that model the rising and falling transitions of the node value. For every event a transition relation is defined, including an enabling condition and a delay interval $[\delta^l, \delta^u]$ specifying the delay bounds of the signal switch once enabled.

A node in the circuit may be driven by stacks of pull-up and pull-down transistors, and possibly pass-transistors. Each stack is modeled by an event that sets the proper value to the variable (*one* for pull-up, *zero* for pull-down and copies the value of another variable for a pass-transistor). Note that we do not consider bidirectional pass-transistors. Delay intervals can be computed using the technology parameters and the fan-out conditions for each signal. The broader the delay intervals for which the circuit is proven to be correct, the more general is the verification, *i.e.* the more robust is the circuit.

As an example, we show the transition relations for signal Y in the *strobe switch* circuit (see Figure 11). Y+, takes place if Y is low and it is pulled up by the p-transistor controlled by Z. Thus, the enabling condition is given by $En(Y+) = \overline{Y} \wedge \overline{Z}$. The enabling condition for Y− is given by $En(Y−) = Y \wedge ACK$ because Y can only be pulled down by the n-transistor controlled by the ACK signal. No specific technology library is used, hence the delay bounds for each stack of transistors to be in the range of [1,2] delay units. Other appropriate delay ranges are used in case of weak transistors or fan-out considerations.

Provided this modeling mechanism, correctness of CMOS circuits can be posed in terms of the following properties:

Persistency. The temporal behavior of a gate is described by the *inertial* delay model. In this model, input pulses shorter than the lower delay bound δ^l are not propagated to the output. Pulses longer than the upper delay bound δ^u are always propagated. However, propagation of pulses with duration between δ^l and δ^u is uncertain and may produce *glitches*, hence signal *persistency* conditions are imposed. Persistency implies that every transition must fire once it is enabled and cannot be disabled by the firing of another transition.

Short-circuits. Custom designs exploit the flexibility of CMOS technology, relaxing the complementarity between the pull-up and pull-down stacks. This introduces a potential source of short-circuits. Although short-circuits can be exploited by considering the pull-up/pull-down relative impedance, generally they are undesirable because they may leave the driven signals undefined.

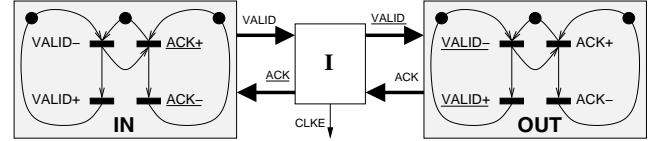The potential short-circuits in the *strobe* circuit of Figure 11 are identified by the following invariants:

Figure 12. Scheme of the verification of an IPCMOS stage.

1. $\overline{Z} \wedge ACK$: Y is pulled down by the n-transistor controlled by ACK and pulled up by the p-transistor controlled by Z.

2. $\overline{VALID} \wedge Y \wedge \overline{CLKE}$: VALID is pulled down by the input and pulled up by the p-transistor controlled by CLKE.

5.2. Modeling the environment

Figure 12 depicts the communication protocol implemented by the *IN* (left) and the *OUT* (right) parts of the environment, in the form of Signal Transition Graphs. The underlined transitions represent the behavior of the circuit stage signals. The *IN* and *OUT* modules operate in a pulse-based manner. Therefore, the environment we use in this experiment is the most general possible to ensure the correct operation of the IPCMOS stage.

The *OUT* module acknowledges the data available at the output of the stage by rising the ACK line. Once this happens, both the stage and the *OUT* module reset the respective VALID and ACK lines independently. A restriction must be imposed to *OUT* to avoid early resetting of ACK. That is, if ACK− arrives too fast after ACK+, the falling edge of ACK may not be properly recorded by the *reset switch* circuit of the stage. Therefore a minimum width is required to the positive pulse of ACK.

The *IN* module notifies new data availability at the input of the stage by lowering the VALID line. The stage acknowledges the incoming data by rising the ACK line. The reset of both lines is carried out independently and no new data can be issued by *IN* until the stage has acknowledged the previous data portion.

5.3. Verification results

Provided the models above, the verification succeeds proving that the invariants characterizing the circuit correctness conditions always hold, *i.e.* the circuit implementing the IPCMOS stage operates correctly in the given *IN-OUT* environment and shows no short-circuits, no persistency violations and no deadlocks. The verification process finishes in less than an hour of CPU time.

The verification succeeds and also provides back-annotation indicating a set of sufficient timing relations between events that guarantee the correctness of the implementation. These relations
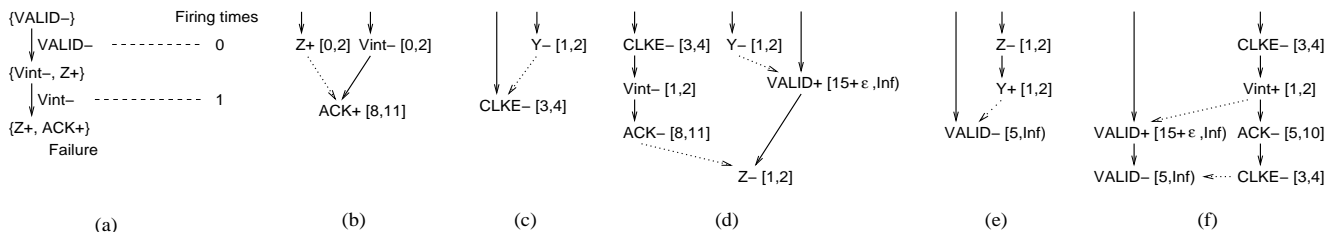
Figure 13. (a) Failure trace and (b) corresponding LzCES. **(c)-(f)** LzCES**s showing other relative timing constraints (dotted arcs) that guarantee correctness.**

are provided as the timed event structures obtained at every iteration of the verification. This information permits to guess about the delay margins allowable to keep the correctness.

Figure 13 shows some of the timing relations obtained during the verification, that guarantee the correctness of the *strobe switch* module (see left side of Figure 11). For example, Figure 13(a) shows a trace leading to a failure situation in which the early firing of ACK+ causes a short-circuit at node Y. Event structure (b) shows the ordering of Z+ and ACK+ in the time domain proving that trace (a) is not timing consistent. This situation corresponds to the case where a falling edge of VALID occurs, followed by the fall of signal Vint and the rise of ACK to indicate the data receipt. Z+ must be faster than ACK+ to avoid the short-circuit at Y corresponding to invariant (1). Event structure (c) shows that after rising signal ACK, the transition Y− turns off the n-transistor isolating Vint from VALID remaining low before it is reset (pulled up) by CLKE−. This ordering is required to guarantee invariant (2). Event structure (d) shows that the event ACK− is ordered with Z− ensuring invariant (1). Indeed it shows that signal ACK always falls before Z thus, avoiding the short-circuit. Event structure (e) shows the ordering relation between CLKE+ and VALID−. The delay of VALID− is set to reset CLKE before the falling edge of VALID. This ordering contributes to guaranteeing invariant (2).

6 Conclusions

A complex example has been presented in which the use of relative-timing-based verification has been crucial to prove the correctness of the system. Although some other parametrized systems have been verified in the past (see *e.g.* [9]), this is the first case in which delay information and refinements down to transistor level have been provided.

A relevant feature of the presented approach is the report of relative timing constraints that describe the slacks that guarantee the correct operation of the system.

The abstractions of different components of the system have still been derived manually. Automatic extraction of timed abstractions is one of the important topics for future work in this area.

References

[1] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] R. Alur, A. Itai, R. Kurshan, and M. Yannakakis. Timing verification by successive approximations. *Information and Computation*, 118(1):142–157, 1995.

[3] O. Bournez and O. Maler. On the representation of timed polyhedra. In *Proc. Int. Conf. on Automata, Languages and Programming (ICALP)*, volume 1853 of *LNCS*, pages 793–807. Springer-Verlag, 2000.

[4] J. Burch, E. Clarke, D. Dill, L. Hwang, and K. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[5] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Lazy transition systems: application to timing optimization of asynchronous circuits. In *Proc. ICCAD*, Nov. 1998.

[6] F. Balarin and A.L. Sangiovanni-Vincentelli. On the automatic computation of network invariants. In *Proc. Int. Conf. on Computer-Aided Verification CAV*, volume 818, pages 234–246, Standford, California, USA, 1994. Springer-Verlag.

[7] T. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for real-time systems. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 353–366, 1991.

[8] T. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *Proc. Int. Conf. on Automata, Languages and Programming (ICALP)*, volume 623 of *LNCS*, pages 545–558. Springer-Verlag, 1992.

[9] C. Leung and M. Greenstreet. A simple proof checker for timing verification. In *ACM Int. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 294–305, Nov. 1995.

[10] K. McMillan and D. Dill. Algorithms for interface timing verification. In *Proc. of the IEEE Int. Conf. on Computer Design*, pages 48–51, 1992.

[11] K. L. McMillan. A compositional rule for hardware design refinement. In *LNCS: Computer-Aided Verication*, volume 1254, pages 24–35. Springer-Verlag, 1997.

[12] T. Melham. Abstraction mechanisms for hardware verification. In *VLSI Specification, Verification and Synthesis*, pages 129–157. Kluwer Academic Publishers, 1988.

[13] M. Peña, J. Cortadella, A. Kondratyev, and E. Pastor. Formal verification of safety properties in timed circuits. In *Proc. Int. Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–11, Apr. 2000.

[14] A. Pnueli. In transition for global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series*. Springer-Verlag, 1984.

[15] S. Schuster, W. Reohr, P. Cook, D. Heidel, M. Immediato, and K. Jenkins. Asynchronous Interlocked Pipelined CMOS Circuits Operating at $3.3 - 4.5$GHz. In *IEEE Int. Solid-State Circuits Conf. (ISSCC)*, pages 292–293, Feb. 2000.

[16] K. Stevens, R. Ginosar, and S. Rotem. Relative timing. In *Proc. Int. Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 208–218, Apr. 1999.

[17] A. Valmari and I. Kokkarinen. Unbounded verification results by finite-state compositional techniques: 10^{any} states and beyond. In *IEEE Int. Conf. on Application of Concurrency to System Design (CSD)*, pages 75–85, Mar. 1998.