Application and Verification of Local Nonsemantic-Preserving Transformations in System Design

Tarvo Raudvere, Ingo Sander, *Member, IEEE*, and Axel Jantsch, *Member, IEEE*

***Abstract*—Due to the increasing abstraction gap between the initial system model and a final implementation, the verification of the respective models against each other is a formidable task. This paper addresses the verification problem by proposing a stepwise application of combined refinement and verification activities in the context of synchronous model of computation. An implementation model is developed from the system model by applying predefined design transformations which are as follows: 1) semantic preserving or 2) nonsemantic preserving. Nonsemantic-preserving transformations introduce lower level implementation details, which are necessary to yield an efficient implementation. Our approach divides the verification tasks into two activities: 1) the local correctness of a refined block is checked by using formal verification tools and predefined properties, which are developed for each nonsemantic-preserving transformation, and 2) the global influence of the refinement to the entire system is studied through static analysis. We illustrate the design refinement and verification approach with three transformations: 1) a communication refinement mapping a synchronous channel to an asynchronous one including a handshake mechanism; 2) a computation refinement, which introduces resource sharing in a combinational computation block; and 3) a synchronization demanding refinement, where an algorithm analyzes the influence of a local refinement to the temporal properties of the entire system and restores the system's correct temporal behavior if necessary.**

***Index Terms*—Design refinement, formal verification, synchronization, system design.**

I. Introduction

A CONTINUOUS trend in design methodologies is to start the design process at a higher abstraction level that allows us to turn more attention to the system functionality, independent of any certain implementation architecture. Since the system description at this level is too abstract for today's synthesis tools, the designer has to create a proper highly detailed model for the soft- and hardware synthesis. Due to the large size of the system and the huge abstraction gap between the initial system model and an implementation model, the verification of these two models against each other is a very complex task.

In this paper, we advocate a system development process which, in the first step, describes the system as a synchronous deterministic model with unlimited resources like ideal functions, memories, and data types. This system model is later on refined by using design transformations that add the necessary implementation details (Fig. 1). Refinement-based design development was introduced to the electronic design automation community by Gajski *et al.* in the context of the SpecC methodology [1]. A final implementation model has to satisfy the given design constraints and to be ready for the synthesis in soft- and hardware. As long as we apply only semantic-preserving transformations, we can assume that the refined models are correct. However, nonsemantic-preserving transformations which, for example, replace an unlimited memory element with a finite one or extend an ideal arithmetic function with an overflow behavior are an inseparable part of the design process. Since the semantics of a model is changed after a nonsemantic-preserving transformation, the verification of its correctness becomes very important.

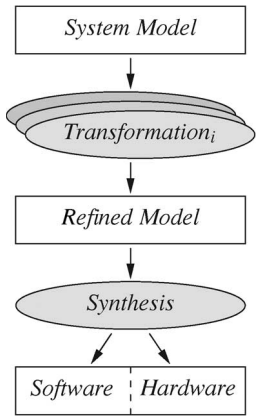

Fig. 1. System design through design transformations.

We propose to verify formally only the local properties of a refined system block immediately after applying a nonsemantic-preserving transformation. Therefore, we define to every nonsemantic-preserving transformation in the design library a set of verification properties and proper abstraction techniques. If the refinement influences the rest of the system, we study this influence by using methods of static analysis. For example, in Fig. 2, a refinement separates process P_4 into processes P_5 and P_6. The verification task is to check whether the pair of new processes calculates the same results as process P_4. The computation in processes P_5 and P_6 may take more time than in process P_4 and, therefore, cause delayed data arrival at the lower input of process P_3. Based on the

Manuscript received December 15, 2006; revised August 31, 2007 and November 13, 2007. This paper was recommended by Associate Editor S. A. Edwards.

The authors are with the School of Information and Communication Technology, Royal Institute of Technology, 164 40 Stockholm, Sweden (e-mail: tarvo@kth.se; ingo@kth.se; axel@kth.se).

Digital Object Identifier 10.1109/TCAD.2008.923249

0278-0070/$25.00 © 2008 IEEE

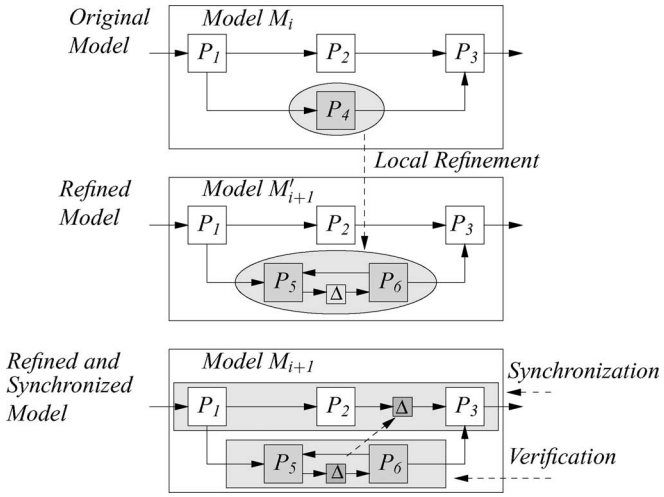

Fig. 2. Local refinement, verification of local properties, and synchronization of the model.

structure analysis of the refined model M'_{i+1}, a synchronization algorithm adds a delay process at the upper input of P_3, which makes the model M_{i+1} latency equivalent to the original one.

In this paper, we present an extended picture about the refinement-based verification approach [2] that we have proposed for formal system design methodologies. In addition to the development of verification properties for local block level correctness checking [2], [3], in this paper, we show how synchronization techniques [4] can be applied for preserving the correct system behavior in the global sense.

In Section II, we give an overview about synchronous languages and system level design methodologies, verification and abstraction techniques, and synchronization approaches. In Section III, we describe the accompanied design and verification process. In Section IV, we introduce the design and verification process through communication, computation, and synchronization demanding refinements. In Section V, we illustrate the verification and synchronization techniques in the context of a digital audio equalizer design process. Section VI concludes this paper.

II. Related Work

Today's design languages, like SystemC, SystemVerilog, or Esterel, are only some examples indicating that the starting point of system design moves to higher levels of abstraction. This corresponds to the following statements of Edwards *et al.* [5]: 1) In a successful system design methodology, the initial system model has to be described at a high abstraction level; 2) in addition, verification has to be done at a higher level of abstraction. At many system level design languages, the system verification issues are kept in mind; and 3) the final implementation has to be derived through correct-by-construction design decisions. A good example is the work in [6], where correct-by-construction design transformations refine the initial model into a final implementation

In this paper, we target the design methodologies, where the design flow starts at a high abstraction level and systems are described with the synchronous model of computation according to the terminology of Lee and Sangiovanni-Vincentelli [7]. In addition to the application of semantic-preserving design transformations, we also allow nonsemantic-preserving transformations. Since the latter class of design refinements by their nature can produce models, which are not correct-by-construction or do not satisfy the given design constraints, the correctness of refined models has to be verified.

Several languages and design methodologies are developed to make the application of formal verification methods easier. Esterel [8] is a synchronous language to describe systems at a high abstraction level. Based on the formal semantics, an Esterel model can be verified through theorem proving or a semantic check method, where the verified property is defined as a syntactic rule. Lustre [9] is a synchronous data flow language that is used to describe the behavior of reactive systems. The model in Lustre can be considered as a finite state structure, where the local states in processes form the global state for the system. In order to verify Lustre models, binary decision diagram (BDD)-based approaches similar to model checking can be used. A good example of verification-oriented languages is Lava [10]. Based on the functional language Haskell, Lava describes both hardware and the required properties that the hardware has to satisfy in the same model. In order to verify formally a Lava program, theorem proving is applied.

In contrast to our approach, these methodologies do not address the application of nonsemantic-preserving refinements nor target the systematic verification of refined models. We give guidelines to the designer in the verification questions providing the right verification and abstraction techniques for predefined nonsemantic-preserving design transformations.

The SpecC methodology addresses transformational design refinement between models at different levels of abstractions. In SpecC, an implementation that has the expected functionality and satisfies the design constraints can be developed automatically. In order to verify that the expected implementation is correct, Abdi and Gajski [11] show that all used transformations preserve the system's correct behavior. Compared to our approach where the designer has to choose between the design transformations but, in such a way, can build different implementations, the SpecC method derives an implementation on a very certain architecture and leaves less freedom to the designer to decide over the implementation options.

In verification, simulation is the most popular technique, but only a limited amount of system behaviors can be observed in this way. In order to have a complete picture of the system behavior, formal methods have to be applied. Theorem proving [12], [13] can be considered as the most powerful verification technique, since even systems with an infinite state space at different levels of abstraction can be analyzed. However, working with theorem proving requires special skills and knowledge, which the designer usually does not have. In contrast, model checking [14] is easy to use and efficient to find errors in finite state systems, but it is sensitive to the number of states in the model. Roughly estimated, the time and memory needed for verification grows exponentially with the number of states. Therefore, all unnecessary behaviors have to be abstracted away before we verify properties in a model.

In order to use model checking efficiently, we define a set of properties to every design transformation and combine the properties with proper abstraction strategies to check the correctness of the refined design block. Quite often, this requires us to devise new abstraction techniques like polynomial abstraction [3] that we have developed for verification of refined arithmetic computation blocks at a high level of abstraction. Transformations that introduce resource sharing, pipelining, and split or merge of processes with arithmetic functionality are only some examples of them.

Several abstraction techniques have been proposed in the past that can solve almost similar verification problems as polynomial abstraction. The idea of uninterpreted function symbols [15] is used to simplify model checking in [16]. The actual values and instructions in a processor are replaced with symbolic ones, and in such a way, an abstract model is created for verification. Clarke *et al.* [17] present a methodology named the counter example-guided abstraction refinement. They start creating the initial abstract model by analyzing the control structure of the system. If a model checker finds a counter example in the abstract model, which is not faulty in the actual system, the abstract model will be refined such that the error is eliminated. This procedure continues until a real error is found or the system correctness is proven.

Depending on the type of verification properties, control or data, abstraction can be applied to respective variables. Hojati and Brayton [18] present how to separate a design into control and data part, and for verification of data-dependent properties, they replace all data variables with one-bit variables. The latter idea is further elaborated in [19], where, accompanied with the interval propagation theory [20], verification of arithmetic system blocks in Very High Speed Integrated Circuit Hardware Description Language is addressed.

All of the previously mentioned abstraction techniques are developed to solve certain verification problems that are not the same as the problem raised by us. Therefore, either they are not as easy to apply as polynomial abstraction or they cannot verify refined models with the ideal data types at a high abstraction level.

Although most of the design transformations refine only one block at a time, the change in a single block may influence the entire system. For instance, the functionality of an arithmetic block remains the same after introducing resource sharing, but due to the feedback signals in resource sharing, the refined block has an internal delay. Therefore, the data delivery to other blocks is delayed, and the system behavior in the synchronous computational model is changed. The described problem is not solvable by the classical retiming techniques [21], since these techniques address synchronization after relocating already existing delay elements. The problem can be used by using desynchronization techniques [22], [23]. These techniques map the original synchronous model to an asynchronous one, for example, to the globally asynchronous and locally synchronous (GALS) model, where the instant when a process is executed depends on the availability of the necessary input data. Although these kinds of models are less sensitive to the computation and communication delays, the switching to an asynchronous model may be impractical since formal verification and formal design transformations in asynchronous models are much more effort than in synchronous models.

An alternative solution to the delay problem is to use the latency insensitive design (LID) [24] technique, which targets the mapping of IP-block-based synchronous models to hardware. Due to the high clock frequency and the difference in wire lengths between IP blocks, concurrently produced data items may reach destination blocks at different clock cycles, which leads to unexpected system behaviors. LID surrounds IP blocks with wrappers that stall processing if input data are not available and replaces synchronous channels between IP blocks by handshake channels including relay stations. The handshake mechanism distributes stalling messages, and a relay station stores data items if the destination process cannot consume them. In order to avoid relay station, bridge-based communication channels can be used between IP blocks [25], or to use schedulers at every wrapper [26]. Similar to GALS models, the computation in a synchronous IP block is executed when the block has received all necessary input data.

Although LID is a common practice in the system-on-chip design implementation, we do not use this approach at system level since models may contain a big number of small computation processes, and it is impractical to equip them with wrappers and schedulers and replace simple synchronous signals with handshake channels. Not only refinement but also verification gets much more complex if the system's functionality includes additional stalling behaviors in a process execution mechanism. In order to avoid discontinuities in the design process caused by switching of computational models, we have proposed a more proper synchronization technique for system level design [4], [27]. Our technique adds only simple delay elements to preserve the concurrent data arrival at destination processes and preserves the same synchronous computational model.

In our previous publications, we have introduced the approach for local verification of refined design blocks after nonsemantic-preserving design transformation [2], the polynomial abstraction technique for computation refinements [3], and the synchronization techniques for temporal refinements [4], [27]. In this paper, we present the complete picture how the local correctness of refined blocks can be checked by using predefined properties and how the influence of a local refinement to the whole system can be compensated by the proposed synchronization techniques.

III. Design and Verification

A. Models and Design Flow

In this paper, we consider a system as a network of concurrent processes that communicate via synchronous signals, as in Lustre, Esterel, and ForSyDe [28]. According to the synchronous hypothesis, computation in processes and communication between them take no time.

A system example in Fig. 3 contains six processes P_i connected by eight signals s_i. A set of processes can be grouped into a block of processes as a process network, and blocks may run at a higher clock speed than the rest of the system. The main classes of processes are as follows: combinational

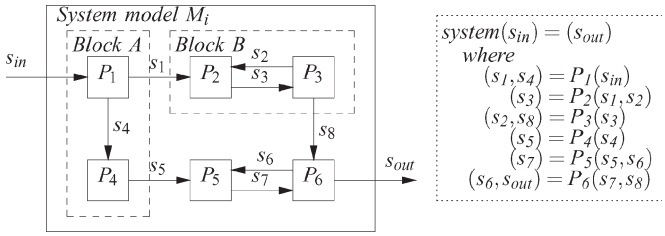

Fig. 3. System model: Blocks, processes, and signals.

processes, finite state machine processes, delay processes, and domain interfaces to adapt clock domains.

A signal is defined as a sequence of events $\{e_0, e_1, e_2, \ldots, e_i, \ldots\}$, where an event e_i carries a value v and the index i is used as a time tag. All signals share the same set of tags for synchronization purposes. The direction of a signal is from the source process to the destination process, and every process has only one output signal. Since for some tags a signal may not contain any value, there is a special absent $(\perp)$ value. A data type D is extended to $D_\perp$ by adding the value $\perp$ to its domain. This allows one to model an aperiodic data rate and to keep the total order of events.

A combinational n-input process takes arguments as a dedicated combinational function f and a set of input signals $s_0, \ldots, s_{n-1}$. For each tag i, the process consumes from the input signals events with the tag i carrying values $v_0, \ldots, v_{n-1}$ and appends to its output signal s_n an event with the tag i and a value $v_n = f(v_0, \ldots, v_{n-1})$. Similarly, a finite state machine process takes arguments as state and output functions, an initial state value, and input signals.

The starting point of the design flow is the system model that describes the system's functionality in the sense of ideal functions. This model, for instance, uses unlimited memory elements and arithmetic functions, which are free of side effects like fixed bit widths and overflow behaviors. The model does not include any nondeterministic behavior and uses the synchronous model of computation. This high abstraction level makes it possible to concentrate on the system functional behavior without considering any lower level implementation details coming from the system structure and architecture.

In order to implement the system in hardware and software, the further design refinement process extends the system model with necessary lower level details concerning a certain design platform. For instance, an implementation model contains only finite data types, bounded memories, and arithmetic functions with overflow behavior. In addition, an implementation model may contain synchronous subdomains operating at different clock frequency. The last stage of the refinement process is an implementation model that can be mapped into soft- and hardware. Hereafter, the classical register transfer level synthesis and compile flow continues.

In order to refine the system model, the designer chooses predefined design transformation rules from the transformation library and applies only these rules to the model. Although the refinement process is not automatic in the sense that the designer has to manually choose transformations, the applications of transformations, abstraction, and verification of the refined models require only minor interaction.

Within the refinement process, it may occur that there is a need for a specific transformation, not yet included in the library. In this case, the transformation has to be created within the design development process and added to the design library. Usually, there are only slight differences between transformations, which belong to the same class, and a new transformation can be easily created by modifying or extending an existing one. Similarly, the same verification properties and abstraction techniques can be used. Thus, the design library grows continuously in time.

Although the design library extending increases the design time, the impact to the development process of the further systems is highly valuable. It is forbidden that the designer refines the model based on the previous knowledge and expertise in an *ad-hoc* manner. If an *ad-hoc* refinement is used, it is the designer's responsibility to find a good method for the verification of the refined model. The refinements and verification strategies that the methodology provides may also be used outside of the formal step-by-step design process. Currently, the design library developed in the context of the ForSyDe methodology [29] is rather small and contains only transformations, which have been defined for the design of small systems within academic research projects.

B. Design Transformations

All design transformation rules in the library are characterized by its name, the required format and constraints to the original process network, the format of the transformed process network, and the impact to the design. Fig. 4 shows the design flow starting from the development of the system model M_0 to an implementation model M_n through transformations T_i. Given a model M_i, a process network PN in M_i, and a transformation rule R, the transformation $T(M_i, R, \mathrm{PN}) \rightarrow M_{i+1} = M[R(\mathrm{PN})/\mathrm{PN}]$ refines the process network PN. The result of the transformation is an intermediate system model M_{i+1} where, in contrast to the model M_i, the process network PN is replaced with $R(\mathrm{PN})$.

According to their characteristics, design transformations are classified as semantic-preserving and nonsemantic-preserving transformations. The former do not change the meaning of the model, rather they change the structure by merging and splitting processes. In general, they are used for optimization of the design. Nonsemantic-preserving transformations change the meaning of a model and introduce new behaviors. Although the semantics of the original model is changed, the refined model may behave the same under given assumptions. For instance, an infinite FIFO buffer can be replaced with a realistic finite one if the data rate on the buffer input does not exceed the limit, which causes the buffer to overflow. Thus, if we know the expected data rate, we can formally prove that the refined model with changed semantics behaves identical to the original one if the buffer size is sufficiently large.

C. Verification

The initial system model that is derived from "an English written text" expresses the ideal functionality of the system.

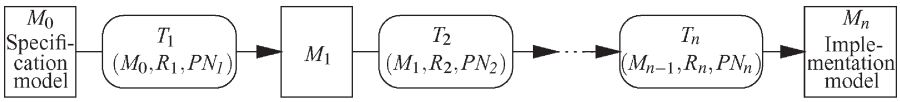

Fig. 4. Design flow.

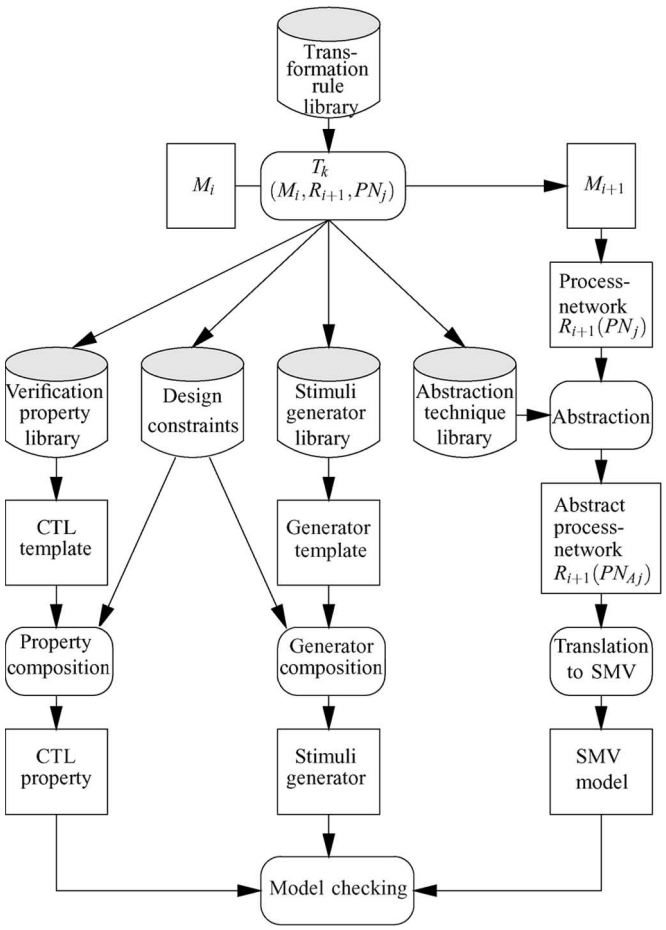

Fig. 5. Verification flow.

Simulation-based verification at this level is much more appropriate than that at lower levels, since we do not need to check for effects caused by limited resources like buffer overflows or the out-of-range values in arithmetic functions with fixed bit widths. Simulation can also be used for the verification of lower level models, but formal methods have to be applied as a complement after nonsemantic-preserving transformations if the exhaustive simulation is possible.

The fact that the system model is refined by using the transformation rules predefined in the transformation library makes it possible to define for every transformation a verification strategy (Fig. 5). Since we know the impact of transformations to the model, we can target exactly the critical properties in verification and do not consider the preserved properties at every step.

In our approach, we define a set of verification properties to every nonsemantic-preserving design transformation rule in the transformation library. The properties are described as temporal logic CTL$*$ expressions. Depending on the design constraints given of the design block under verification, the predefined properties may be incomplete, e.g., we are not aware of the exact bit width of the expected input words to an arithmetic block, or the data rate of a complex asynchronous communication channel. Therefore, we provide property templates in the property library that the designer can easily extend by filling in the exact parameters derived from the design constraints.

Accompanied with the predefined verification properties, the methodology provides abstraction techniques for every specific property in the abstraction library. In fact, the right abstraction methods are very important, since model checking suffers from the state space explosion problem and the design block has to be simplified as much as possible. The application of an unsuitable abstraction may remove an erroneous behavior from the model and render the further model checking useless.

Since we verify refined blocks by locally splitting them out of the system, the input stimuli to these blocks are missing. In some cases, the stimuli can be expressed in the verification property or defined as the input data type of the abstract block. For instance, if a small set of input values is enough to decide over the correctness of a block, the model checker can assign values from an abstract input domain to the block's input in a nondeterministic manner. In order to model periodic patterns as input stimuli, we provide stimuli generators in the generator library that are reconfigurable finite state machines. A designer can parameterize them according to the design constraints, e.g., to generate n input tokens within every m clock cycles.

In this paper, we have used the Symbolic Model Verifier (SMV) model checker [30]. The rules how to map a synchronous model in ForSyDe language to the SMV language can be found in [2]. The hierarchical structure of programs in both languages is quite similar, which makes it simple to map a model from one language to the other. The only limit comes from the ideal data types in the ForSyDe models, which are not supported by SMV—since its models are restricted to be finite. Therefore, the designer has to bind the range of values for all variables before applying model checking.

The complete verification flow is shown in Fig. 5. The transformation rule R_{i+1} refines the process network PN_j of the model M_i into $R_{i+1}(\mathrm{PN}_j)$. According to the transformation rule R_{i+1}, we select necessary properties, abstraction techniques, and input stimuli generators for verification of the refined model. We complete the properties and stimuli generators according to the design constraints and translate the abstract process network $\mathrm{PN}_{\mathrm{Aj}}$ to the SMV language. After that, we run the SMV model checker to verify if the given properties hold or not.

D. Scope and Limitation

Only one design block is refined at time in our approach. All local nonsemantic-preserving design transformations are equipped with verification attributes, which are required to check if the refined blocks behave correctly and satisfy the

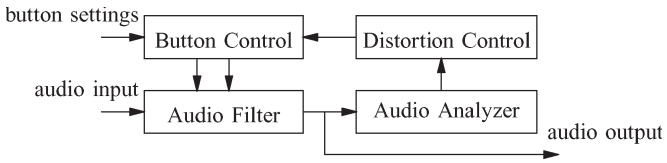

Fig. 6. Main blocks of a digital audio equalizer.

design constraints. Obviously, a local refinement in one domain may cause mismatches in some other domain and lead to unexpected system behaviors. For example, a computation refinement that introduces local resource sharing causes delayed data delivery to destination processes due the increased computation time, which is a change in the communication domain, although the refined block locally behaves as expected. Therefore, the global influence of design transformations has to be studied from the characteristic functions, which are given for each transformation. We illustrate the global influence in the context of a resource sharing refinement in finite-impulse response (FIR) filters. In order to synchronize the entire system after the local refinement, we propose a synchronization technique, which ensures that the system after the local refinement and global synchronization is latency equivalent to the original model.

In the following section, we present the development of verification properties and abstraction techniques in the context of communication and computation refinements and synchronization issues. Nonsemantic-preserving transformations are explained through the refinement of a digital audio equalizer, which is shown in Fig. 6. A communication refinement replaces the synchronous channel between the button control and the audio analyzer with an asynchronous channel. Computation refinements introduce resource sharing in FIR filters of the audio filter and in fast Fourier transform (FFT) unit of the audio analyzer. Due to a temporal change caused by the introduction of resource sharing in the computation blocks, a synchronization technique renders the refined model latency equivalent to the original model.

IV. Local Design Transformations

A. Communication Refinements

The transformation SynchronousChannelToHandshake shown in Fig. 7 is an example of communication refinements that form a special class in the transformation rule library. It is used to prepare a synchronous model for mapping to an asynchronous implementation. The application of this transformation refines a synchronous channel between two computation blocks to an asynchronous communication channel. The refined channel implements a handshake protocol that can be used to model a communication interface between hardware and software domains, for example, adapting the equalizer's button control and distortion control blocks that are implemented in software to the other blocks that are implemented in hardware.

In order to apply this transformation, the original synchronous channel has to fulfill the precondition—the data type of the channel is absent extended ($V_\perp$), i.e., in addition to the values in the domain of V, the cannel can carry absent values. The

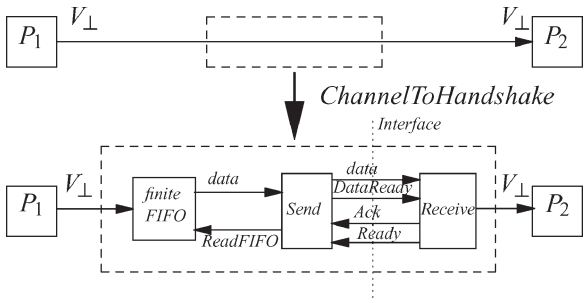

Fig. 7. Refinement into a handshake protocol.

result of the transformation is an asynchronous channel, which is implemented by three processes—FIFO, Send, and Receive. When Send is idle, it tries to read data from FIFO. If the reading was successful, then Send emits the message DataReady to Receive, and after receiving the message Ready, it sends the data. After the data are received, Receive sends a message Ack to Send. If the channel is busy with the transfer of data through the channel, the data items appearing on the channel input will be stored in FIFO.

Obviously, the data transportation through the refined channel requires longer time than in the original channel, since the handshake mechanism makes many steps in Send and Receive. If we model the handshake-protocol-based channel in the synchronous computational model, the data transportation delay of the channel is several clock cycles. Thus, the temporal behavior of the refined model is quite different from the original one. In addition, the size of the buffer has to correspond to the traffic load on the channel input in order to avoid data overflow in the buffer. In order to verify that the refined channel implements the functionality in the design specification and satisfies the given design constraints, the rule SynchronousChannelToHandshake points to four properties as critical issues that the designer should check: 1) reliability; 2) latency; 3) bandwidth; and 4) preservation of data order [2].

The reliability property verifies that all the data items appearing on the channel input will be transferred through the channel without any data loss, except the loss if the FIFO buffer was full at the data arrival.

The reliability property is formulated in CTL$*$ as

$$\begin{aligned}&\texttt{SPEC AG}\,((\texttt{input\_stream.Con} = \texttt{Prst}\ \&\\ &\qquad \texttt{input\_stream.} = 0\ \&\\ &\qquad \texttt{fifo\_size} < \texttt{SIZE} - 1) - >\\ &\qquad \texttt{AF}\,(\texttt{output.Con} = \texttt{Prst}\\ &\qquad\quad \&\ \texttt{output.Val} = 0)).\end{aligned}$$

The property says: "if there is at least one empty slot in the FIFO buffer (SIZE-1) when a data item Prst 0 appears on the channel input, then always this data item will eventually be transferred to the channel output." The data type of the channel is defined as a structure, where the constructor Con separates absent and present values and Val carries the value in present events. Before verifying this property, we have to apply data abstraction to reduce the size of the model. Since the channel is

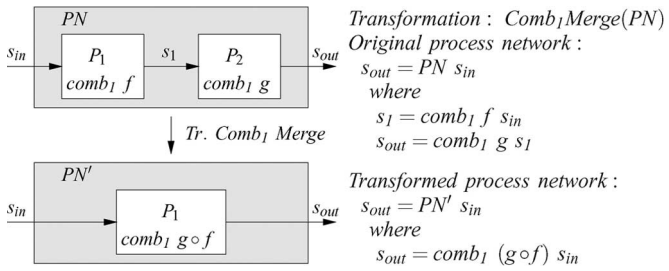

Fig. 8. Transformation: Merging of single-input combinational processes.

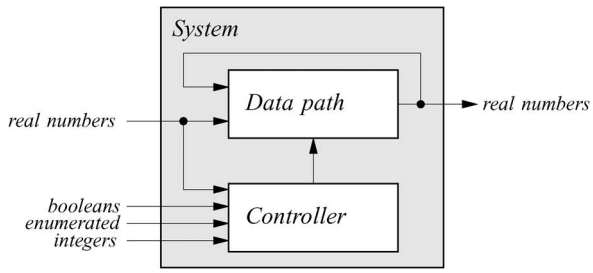

Fig. 9. System structure.

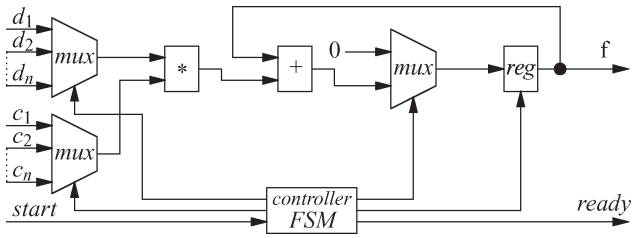

Fig. 10. Sequential model of FIR filter.

insensitive to the exact values in present events and the model checker analyzes all possible system behaviors, it is enough that the channel data type contains the absent value and only two different present values. Prst 0 is the special value described in the CTL property, and Prst 1 represents all the rest of values. Due to the abstract definition of the channel input data type, the model checker can form any sequence with these three values and model them on the channel input.

The latency property checks if the channel can transfer a data item through the channel within a number of clock cycles specified by the designer. In order to find the maximum ratio of present and absent values on the channel input, which does not cause buffer overflow, the bandwidth property is defined. The fourth property checks if the present values on the channel output preserve the same order they have on the channel input. The CTL definitions and a more detailed discussion of the latter properties can be found in [2]. The given properties express a set of general assumptions about correct behavior of a communication channel. Thus, these properties can be used to verify other communication channels that are defined by the designer. However, the verification engineer has to approve the given set of properties for a new channel or to add relevant verification properties.

B. Computation Refinements

We have developed several design transformations that change the model by relocating functionalities from one combinational process to another to balance the computation load or to find an optimal structure. The simplest kinds of transformations are the splitting and merging of combinational processes. For example, a design transformation that merges two combinational processes with one input and one output is shown in Fig. 8. The transformed process that applies the sequential composition $(g \circ f)$ of combinational functions f and g to input values is semantically equivalent to the composition of processes that apply the same functions sequentially. Verification in the transformed process is not essential, since the applied design transformation is semantic preserving.

A final implementation model cannot be derived from the ideal system model without applying nonsemantic-preserving transformations, and therefore, the refined models have to be verified formally. It is not always straightforward to simplify the model for formal verification by using the existing abstraction techniques, and new techniques have to be developed. In the following, we take up the verification of arithmetic computation blocks, where transformations introduce resource sharing for component reuse.

In the system model, most of the arithmetic functions are described as combinational computation blocks using ideal data types, which keeps the model simpler. To implement larger arithmetic circuits as combinational functions is impractical and often restricted by the limited chip area in hardware. Instead, a polynomial specification can be split into smaller operations and implemented on a data path [31]. The transformation CombinationalToResourceSharing refines a combinational arithmetic block to a sequential circuit (Fig. 9) and, in such a way, introduces component reuse. The refined block includes a controller to execute operation on a data path, which contains registers to store intermediate computation results.

Let us consider the refinement of an n-stage FIR filter from the equalizer's audio filter block, which has the functional specification written as the polynomial $F_{\mathrm{FIR}}(c_1, \ldots, c_n, d_1, \ldots, d_n) = \Sigma_{i=1}^{n} c_i * d_i$ that can be modeled as a combinational computation block. The variables c_i and d_i are the ith coefficient of the filter and the $i-1$ clock cycles delayed input value, respectively. A possible sequential implementation of F_{FIR} as a controller and a data path is given in Fig. 10.

The additional signals, start and ready, are used to execute computation and to announce the available result. The controller finite-state machine (FSM) is configured so that the data path calculates a term $c_i * d_i$ at every clock cycle and sums the terms in the internal register reg. The initial value of reg is the constant zero. Due to the additional signals, start and ready, and the increase in the latency compared to the original combinational process, the refined block is not semantically equivalent to the original block.

In order to simplify the verification in sequential computation blocks with ideal data types at a high abstraction level, we have proposed the polynomial abstraction technique [3]. The abstraction technique relies on two theorems: 1) The fundamental theorem of algebra allows one to verify two polynomials by a small number of input assignments and 2) the application of

the Chinese remainder theorem makes it possible to split the verification task into even smaller parts.

Based on the fundamental theorem of algebra [32], we have proven that two multivariable polynomials $S(\overline{x})$ and $I(\overline{x})$, with the degree of x_i equal to k_i, are equivalent if they evaluate pairwise to the same result for any combination of input assignments where every x_i gets $k_i + 1$ different values [3]. Thus, if we have a sequential design implementation $I(\overline{x})$ with a polynomial specification $S(\overline{x})$, we need to find the maximum degree of every variable in these polynomials, and according to the degree, assign a fixed number of different values for the equivalence check. For the degree calculation, we have used a method that is similar to symbolic execution. We analyze the computation on the data path based on the controller behavior starting from the moment when the signal start goes high and finishing when the computation result is ready on the data path output. The degree calculation is based on the mathematical rules which state the following: 1) In multiplication, the degrees of operand polynomials are added and 2) in addition and subtraction, the maximum of operands' degrees of respective terms determine the degrees of the result polynomial. The given theoretical background allows one to reduce the infinite domains of real number input signals to the finite integer ones. Thus, we can use a model checker in order to verify the control structure of the model based on the circuit's input/output functionality.

Although the ranges of input values are reduced and finite, some values in the data path may still grow quite large. For instance, the degree of x in $P(x) = x^{10}$ is 10 that allows one to apply 11 different input values $(0, 1, \ldots, 10)$ for verification. As the binary presentation of the greatest value, $P(10) = 10^{10} \approx 2^{33}$, requires 33 bits, we need additional state space reduction. Therefore, we apply the Chinese remainder theorem to the first abstract model.

According to the theorem, two functions $f(\overline{x})$ and $g(\overline{x})$ calculate the same result in the region $0 \leq (f(\overline{v}), g(\overline{v})) < (m_0 m_1, \ldots, m_n)$ if, for every relative prime number $m_i \in m_0, m_1, \ldots, m_n$ and every input assignment $(\overline{v})$, $(f(\overline{v}) \mod m_i) = (g(\overline{v}) \mod m_i)$.

By using this knowledge, we can replace the system verification for the domain $\{0, \ldots, \prod_{i=1}^{n} m_i\}$ to n verifications with $m_i + 1$ element input domains $\{0, \ldots, m_i\}$. In order to perform the series of verifications, we have to change the assignment statements reg_value := input_value to reg_value := (input_value $\mod m_i$). Instead of verifying directly the function $P(x) = x^{10}$, we verify 11 significantly smaller models $P(x) = (x^{10}) \mod m_i$, $m_i \in \{2, 3, 5, \ldots, 31\}$, since $\prod_{i=1}^{11} m_i \approx 2^{37} \leq 2^{33}$. To present the largest value $31 = 2^5$ appearing in these models requires only 5 bits compared to the original 33 bits.

C. Synchronization Demanding Refinements

Our design library provides several transformations, which relocate delay elements to balance the delays in computation blocks. A transformation, which moves a delay from the output to the inputs of a combinational process, is shown in Fig. 11. Depending on the initial values of the delay processes

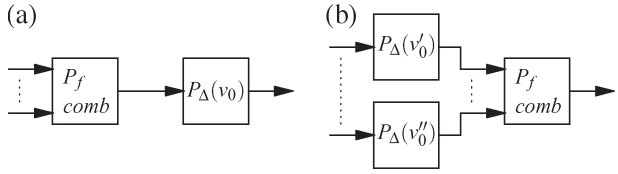

Fig. 11. Transformation: Move delay to input (a) original model and (b) refined model.

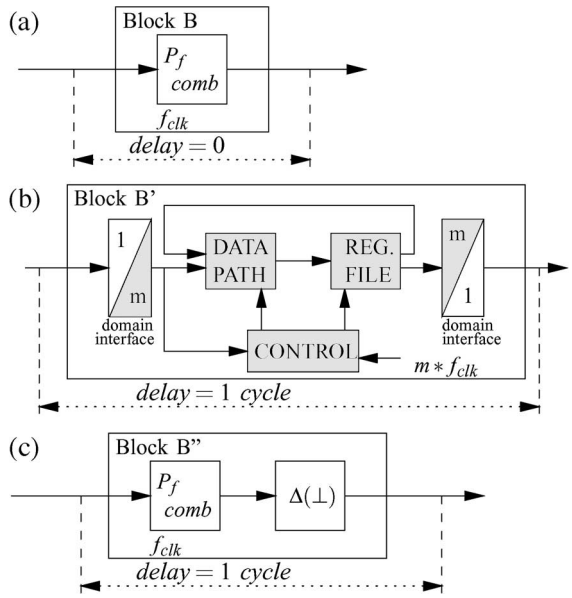

Fig. 12. Resource sharing leads to several clock domains and introduces an extra delay.

in the refined model, the transformation can be classified as nonsemantic preserving or semantic preserving. If the same initial values are used in the original and in the refined models $(v_0 = v_0' = \cdots = v_0'')$, then the transformation is nonsemantic preserving since the first output events of these two models as the reactions to the same input signals are potentially different. However, the recalculation of the initial values, such that $f(v_0', \ldots, v_0'') = v_0$, by using retiming techniques renders the transformation semantic preserving.

Although retiming techniques can restore the system behavior after relocating already existing delay elements, the introduction of new delay elements, like in pipelining and resource sharing refinements, requires other approaches for data synchronization. We classify transformations, which add a delay to a computation block as temporal refinements.

Let us take up the refinement of a combinational function to a sequential implementation that was considered in the previous section. One possibility is to model the sequential circuit in a clock domain with a higher clock rate, as shown in Fig. 12. According to the synchronous hypothesis, the computation in processes takes no time, and the values on a process output appear at the same moment when the input values arrive. Although the block B' operates m times faster, there is still a noticeable delay caused by the feedback loop, and according to the synchronous hypothesis, the blocks B and B' cannot be identical. If we consider that the delay in the refined model is one clock cycle, the block B' corresponds to B'' instead of

B. Thus, the refinement changes the temporal properties of the original block.

Temporal refinements add delay elements, and their initial values change the behavior of the model. Due to the additional values, multi-input processes, which have one input connected to a refined combinational process, receive a different set of values in the first clock cycle than in the original model. One way to solve this kind of synchronization problem is to use the LID or similar methods. Since these methods add handshake-protocol-based communication channels or use schedulers, making the further system refinement and verification more complex, we have proposed an alternative synchronization technique [4]. In the following, we explain our approach, which uses only simple delay elements (synchronization delays) and keeps the model latency equivalent to the original model after temporal refinements.

The main aim of our approach is to initialize the temporal refinement added delays and synchronization delays with synchronization values ($\perp$), which are distinct from the actual data values ($\top$) processed by the initial system model. Let $\Delta(\perp)$ be a delay process whose initial value is $\perp$. We extend all combinational and FSM processes P_j so that their reaction to $\perp$ values is a $\perp$ value in the same clock cycle, i.e., $P_j(e^1, \ldots, e^n) = \perp$, if $\forall i (1 \leq i \leq n)$, $e^i = \perp$. In addition, FSM processes do not update their states if $\perp$ values are on their inputs. Due to this extension, $\perp$ values are reproduced in loop-back structures. We say that a system is $\perp$ consistent if each process at every time instant receives only values of the same type ($\top$ or $\perp$). Our algorithm initializes the refinement produced delay processes with $\perp$ values and extends the system with $\perp$-delay processes so that the system stays $\perp$ consistent.

Let us consider the following terms for further discussion.

1) A path is a sequence of processes connected by signals that have the same direction from one end to the other.
2) A loop is a cyclic path including no process twice.
3) A common source is a process, whose output signal is connected to inputs of more than one process.
4) A common destination is a multi-input process.
5) A pair of paths contains two paths having one common source and one common destination, and no other process belongs to both of the paths.

In order to preserve the latency equivalence of synchronous models, the delay of the paths that feed a common destination process [process P_4 in Fig. 13(a)] may only be increased equally. If a refinement adds a $\perp$-delay process to one path (P_5 in path$_1$), the rest of the paths have to be extended with $\perp$ delays as well (P_6 in path$_2$). The synchronization procedure gets much more complex if the system contains multiple feedback loops. If we insert a $\perp$-delay process [process P_{10} in Fig. 13(b)] to a path (path$_3$) that feeds a loop (loop$_1$), it is necessary to add a $\perp$-delay process (process P_{11}) to the loop as well. Otherwise, the common destination process of the path and the loop receives in some clock cycles different types of events that make the system $\perp$ inconsistent. Since the $\perp$ values will be reproduced in the loop, the feeder system part has to deliver regularly $\perp$ values to the loop. The regularity is denoted as a pattern.

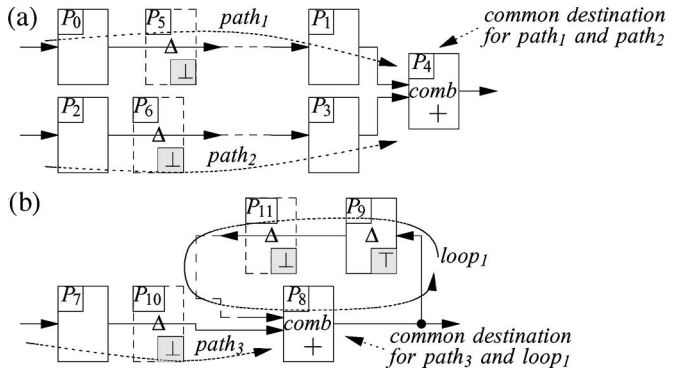

Fig. 13. Synchronization of (a) paths and (b) loops.

Pattern: A pattern is defined as a minimal sequence $V = \{v_0, \ldots, v_i, \ldots, v_{n-1}\}$ of values $\top$ and $\perp$, which cannot be constructed by a single repetitive subpart $\{v_0, \ldots, v_i\}$.

For a process, the pattern shows in which order $\perp$ and $\top$ values arrive.

Pattern Equivalence: Two patterns V^a and V^b are equivalent if they have the same length n, and there exists an integer constant k such that for all $i \in \{0, \ldots, n-1\}$, $v_i^a = v_j^b$, where $j = ((i + k) \mod n)$.

The pattern $V_1 = \{\top, \top, \perp, \perp\}$ is equivalent to $V_2 = \{\top, \perp, \perp, \top\}$ and not to the pattern $V_3 = \{\top, \perp, \top, \perp\}$. Elements in equivalent patterns can be rotated but not shuffled.

We assume that the system is connected, and because of that, all processes have to operate with equivalent patterns. The pattern length is determined by the delays of all loops and the delay differences in all pairs of paths. According to the algorithm in [4], the pattern length N is the greatest common integer divisor of the mentioned delays. The algorithm creates an ordered N-element set of labels and gives a label L_i to all delay processes in the original model according to the following rule. The labeling starts from an arbitrarily chosen delay process. If a delay process $\Delta_i(\top)$ has a label L_k and there is a path from $\Delta_i(\top)$ to $\Delta_j(\top)$ without including any other delay process, then $\Delta_j(\top)$ gets label $L_{((k+1) \mod N)}$. Let the capital letters A, B, and C be used to label delay processes in the original model. If after labeling a temporal refinement adds a delay process $\Delta_i(\perp)$ with a new label D to the model, so that the closest delays to $\Delta_i(\perp)$ have labels B and C, then the synchronization algorithm adds a synchronization delay to all paths between delay processes with labels B and C.

The labeling is illustrated on the system shown in Fig. 14(a) that contains eight combinational processes and four $\top$-delay processes. The system contains two feedback loops, loop$_1$ = $\{p_1, p_2, \ldots, p_{11}\}$ and loop$_2$ = $\{p_4, p_5, \ldots, p_9\}$, having delays $D_1^{\text{loop}} = 4$ and $D_2^{\text{loop}} = 2$. There are two pairs of paths: from P_0 to P_6 and from P_9 to P_4. The path$_1$ runs through $P_1, P_2, \ldots, P_5$, and path$_2$ is the direct connection between P_0 and P_6. The delay difference of them is $D_{1,2}^{\text{path}} = 2$. The delay difference of path$_3$, $(P_{10}, P_{11}, P_1, P_2, P_3)$, and path$_4$, the direct connection from P_9 to P_4, is $D_{3,4}^{\text{path}} = 2$. The greatest common integer divisor for the found values is $N = 2$. $L_0 = A$ and $L_1 = B$ are the ordered set of labels. P_3 and P_8 get the label $L_0 = A$, and P_5 and P_{11} get $L_1 = B$. Since the system input

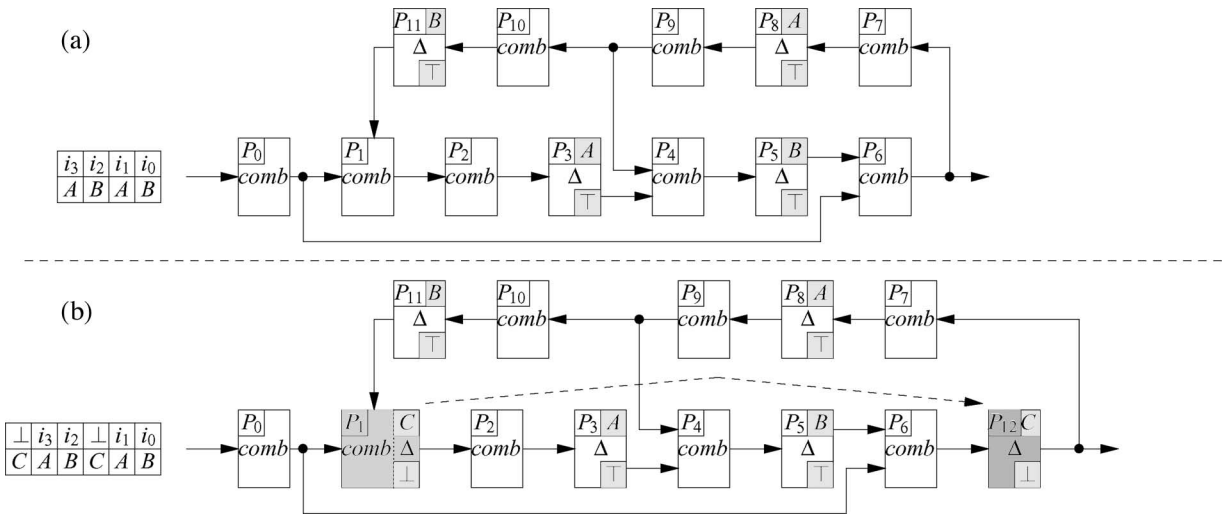

Fig. 14. Synchronization example: (a) initial model after labeling and (b) refined model.

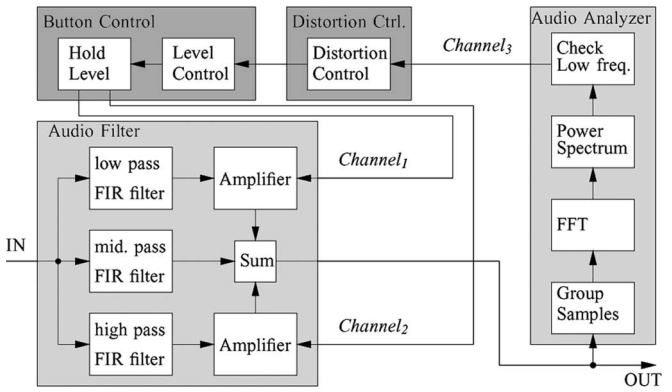

Fig. 15. Digital audio equalizer.

can be considered as a shift register, which emits one value at every clock cycle, the input events are labeled similarly to the delay process.

Let a temporal refinement in P_1 add a delay $\Delta(\bot)$ at the output of P_1. The new delay gets label C, which obtains the position L_2 since the new delay is on the path from P_{11} labeled with B to P_3 labeled with A. All processes have to operate with patterns equivalent to $\{\top, \top, \bot\}$, and therefore, the algorithm inserts a synchronization delay process on every path after B delays and before A delays. Thus, the synchronization process P_{12} can locate anywhere between P_5 and P_8. The input signal is extended according to the labels as well. The refined and synchronized model is latency equivalent to the original one, i.e., these two models produce the actual data values in the same order on the same input signal.

V. Refinement and Verification of a Digital Audio Equalizer

We illustrate the application of the described verification and synchronization algorithms within the design process of a digital audio equalizer (Fig. 15). The equalizer operates in the following manner. It divides the audio input signal into three frequency bands and amplifies the lower and higher bands according to the button settings in the button control block. The sum of the amplified signals gives the equalizer output. The audio analyzer observes the bass level in the output signal. If the basses are too strong, the distortion control block adjusts the bass amplification level to protect the speakers.

TABLE I
Verification Time and the Number of BDD Nodes for Checking Computation Refinements in the FIR-Filter and FFT Blocks

Functionality	CPU time (sec.)	BDD nodes
FIR – filters	3×29.4	$1.4M$
FFT – block	1.5	$135k$

A. Refinement of FIR-Filters and FFT Blocks

In the initial specification model, all signal processing blocks are described as combinational processes, which use ideal data types and arithmetic functions. Instead of implementing the FFT and FIR-filter blocks as large combinational circuits, we introduce resource sharing in these blocks similar to the approach in Section IV-B. All these blocks will be implemented as sequential circuits, containing a controller that executes basic arithmetic operations in a data path and stores intermediate computation results in a register file (Fig. 12). The sequential blocks are separated by clock domain interfaces from the rest of the design and run at higher clock rates than the rest of the system, such that they have one clock cycle delay compared to the original combinational blocks. In order to verify the correctness of the sequential blocks, we applied the polynomial abstraction technique that reduces the input signals of the FIR filters and FFT to one bit variables. The SMV model checker spent all together only 1.5 min and created less than 1.5 million BDD nodes for verification, as shown in Table I.

B. Synchronization

After introducing resource sharing in the FIR-filter blocks, the latency of these blocks is larger than that in the original model. In order to keep the refined model latency equivalent to the original one, we apply the synchronization algorithm

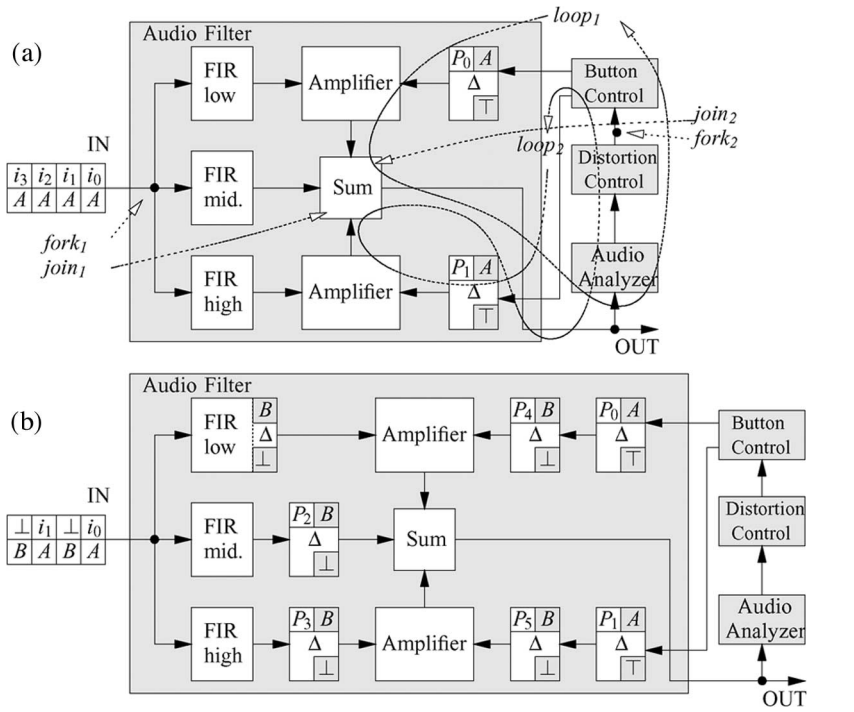

Fig. 16. Audio equalizer: (a) initial model (b) after the clock domain refinement of the process $\mathrm{FIR_{low}}$.

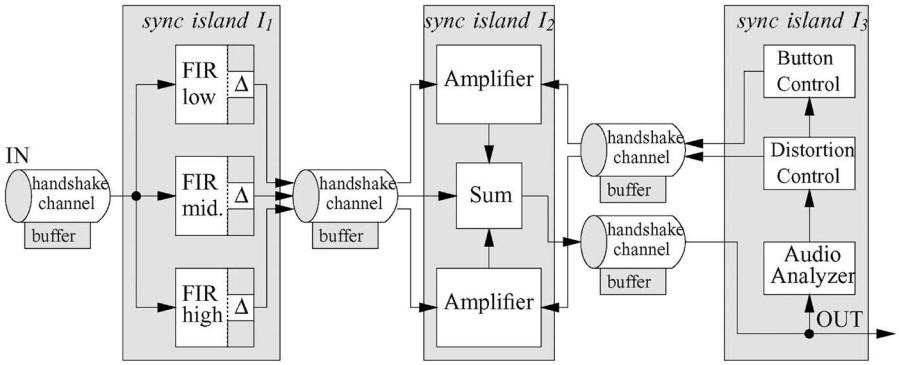

Fig. 17. Implementation of the audio equalizer as LID model.

described in Section IV-C. In the following, we explain the algorithm steps after the first refinement that is applied to $\mathrm{FIR_{low}}$.

The refined model contains two loops ($\mathrm{loop_1}$ and $\mathrm{loop_2}$ in Fig. 16) with one delay process in each. The delay difference of common source/destination paths between respective fork and join points is in all cases equal to zero. Since the greatest common divisor of the found values is one $(N = 1)$, the delay processes P_0 and P_1 and input events are labeled with the same label A.

The $\perp$ delay added to the process $\mathrm{FIR_{low}}$ by the refinement gets a new label B. According to the synchronization algorithm, the new content of set of labels is $\{L_0 = B, L_1 = A\}$, $N = 2$, and processes operate with patterns equivalent to $\{\top, \perp\}$. In order to synchronize the system, we have to insert $\perp$-delay processes with label B between A delays and between all input events. $\mathrm{FIR_{mid}}$ and $\mathrm{FIR_{high}}$ are refined in a similar way, and the existing $\perp$-delay processes P_2 and P_3 are encapsulated in the refined blocks. A prototype tool implemented in the ForSyDe modeling environment required less than 0.5 s to find a proper location of synchronization delay processes and the extension to the input signals.

In order to compare our synchronization technique with LID or GALS approaches, a general structure of an LID- or GALS-model-based solution is shown in Fig. 17. The model is divided into the following: 1) three asynchronously communicating synchronous islands or 2) three synchronous blocks of LID, communicating via handshake channels.

Although the LID method does not add any explicit delay process for synchronization, there are buffers in every channel and wrapper, 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 in LID that is used to distribute stalling events between computation blocks. The input/output latency of LID and our model were the same, and both models had to work with two times more values than the original model. The LID relay stations

TABLE II
VERIFICATION TIME AND THE NUMBER OF BDD NODES FOR CHECKING Channel_1 AND Channel_2

Property	CPU time (sec.)	BDD nodes
Property 1 : Reliability	0.37	32490
Property 2 : Latency	0.11	3111
Property 3 : Bandwidth	0.38	34748
Property 4 : Preserving Order	2.56	163219

TABLE III
VERIFICATION TIME AND THE NUMBER OF BDD NODES FOR CHECKING Channel_3

Property	CPU time (sec.)	BDD nodes
Property 1 : Reliability	1.08	49533
Property 2 : Latency	0.26	11772
Property 3 : Bandwidth	0.25	7496
Property 4 : Preserving Order	4.96	181870

in the channels are initialized with stalling values. Since the system contains feedback loops, these values are reproduced [33]. Due to the feedback loops, the synchronization values are reproduced in our model as well.

The LID and GALS approaches are common in IP-block-based design, although they may not be the best candidates for refinements in small synchronous blocks, since very small and disproportionate islands are created, like I_1 and I_3. Due to the complex communication mechanism between synchronous islands, formal verification and formal refinements in this kind of models are considerably more complex. In addition, our model is more expressive, since the impact of refinements is explicit. The labels indicate relations between refined computation blocks and paths with increased delays. The explicit synchronization delay processes can be shifted to proper positions and reused for further refinements. The synchronization delay processes can be mapped onto latches in the hardware model.

C. Refinement of Communication Channels

The initial system model in our design methodology is described by means of synchronous design blocks that communicate via synchronous signals. In order to develop an implementation, which contains parts in software and hardware domains, the synchronous communication channels Channel_1, Channel_2, and Channel_3 between these two domains have to be replaced with asynchronous ones. According to the computation load in the equalizer blocks, we implement the audio filter and audio analyzer blocks in hardware and the rest of the design in software [34]. Let us consider the channel refinement between button control and audio filter in the following. Under assumptions that 1) processes P_0 and P_1 store the amplification levels until an update comes from the button control, 2) the button settings are changed rarely, and 3) also the distortion control interaction to the amplification process occurs seldom, we can apply the transformation SynchronousChannelToHandshake (Section IV-A) to the synchronous Channel_1 and Channel_2. After the design decision about the number of slots in the finite FIFO buffer (eight slots) of the refined channel and the expected data rate (maximum four messages within 36 clock cycles), we verified the given properties. The verification time and the number of created BDD nodes are shown in Table II.[1]

The third channel Channel_3 between software and hardware domains was refined in a similar way. The only difference between this channel and the rest is that Channel_3 can transfer several values in parallel as a packet instead of a single value as C Channel_1, Channel_2. The verification time and the number of BDD nodes required to check the same properties for Channel_3 are given in Table III.

[1]Due to the improved input stimuli generators in our verification library, the resource demands are different from the experiments in [2], although the quality of the verification is the same.

VI. CONCLUSION

In this paper, we present a general verification concept in transformational system design methodologies, which we have integrated into ForSyDe. The idea of predefined verification properties and abstraction techniques makes the use of formal verification in the system design more efficient and tractable to designers. We avoid the verification of a very detailed implementation model against the ideal specification. Instead, the verification is done in a systematic way—after applying nonsemantic-preserving design transformations, we check the correctness of the predefined properties in refined models. Caused by the state space limits of formal verification tools and the huge size of today's systems, we verify formally only the local properties of a refined block and leave the global influence to static analysis techniques. For refinements in arithmetic computation blocks, we have developed the polynomial abstraction technique, and in order to compensate for the influence of local refinements to the entire system, we have proposed a novel synchronization algorithm.

Although the development of the design and verification methodology is a time-consuming process, including the development of design transformations, verification properties, and abstraction techniques, the use of this methodology will in most cases significantly decrease the entire design time. In particular, verification time, which takes a considerably large amount of the design time, will shorten compared to the application of formal verification in an *ad hoc* style.

The future work in the ForSyDe framework is to extend the transformation rule library in sight of some target design platform. This includes the development of verification properties and selection of good abstraction techniques for these rules. One of the future plans is to turn the attention to other design issues than only the system's functional and temporal correctness. For example, the application of power consumption analysis in the system refinement phase can help us to choose between alternative design transformations and, in such a way, to develop a more efficient final implementation.

REFERENCES

[1] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*. Norwell, MA: Kluwer, 2000.

[2] T. Raudvere, I. Sander, A. K. Singh, and A. Jantsch, "Verification of design decisions in ForSyDe," in *Proc. CODES+ISSS*, Newport Beach, CA, Oct. 2003, pp. 176–181.

[3] T. Raudvere, A. K. Singh, I. Sander, and A. Jantsch, "System level verification of digital signal processing applications based on the polynomial abstraction technique," in *Proc. ICCAD*, San Jose, CA, Nov. 2005, pp. 285–290.
[4] T. Raudvere, I. Sander, and A. Jantsch, "A synchronization algorithm for local temporal refinements in perfectly synchronous models with nested feedback loops," in *Proc. GLSVLSI*, Stresa, Italy, Mar. 2007, pp. 353–358.
[5] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: Formal models, validation, and synthesis," *Proc. IEEE*, vol. 85, no. 3, pp. 366–390, Mar. 1997.
[6] T. Seceleanu, "Systematic design of synchronous digital circuits," Ph.D. dissertation, Univ. Turku, Turku, Finland, 2001.
[7] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 17, no. 12, pp. 1217–1229, Dec. 1998.
[8] G. Berry, M. Kishinevsky, and S. Singh, "System level design and verification using a synchronous language," in *Proc. ICCAD*, 2003, pp. 433–439.
[9] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic, "Translating discrete-time Simulink to Lustre," *ACM Trans. Embed. Comput. Syst.*, vol. 4, no. 4, pp. 779–818, Nov. 2005.
[10] S. Singh, "System level specification in Lava," in *Proc. DATE*, 2003, pp. 370–375.
[11] S. Abdi and D. Gajski, "Verification of system level model transformations," *Int. J. Parallel Program.*, vol. 34, no. 1, pp. 29–59, Feb. 2006.
[12] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas, "Effective theorem proving for hardware verification," in *Proc. 2nd Int. Conf. TPCD—Theory, Practice and Experience*, 1994, pp. 203–222.
[13] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: A theorem prover for program checking," *J. ACM*, vol. 52, no. 3, pp. 365–473, May 2005.
[14] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1512–1542, Sep. 1994.
[15] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessors control," in *Proc. 6th Int. Conf. CAV*, 1994, pp. 68–80.
[16] S. Berezin, A. Biere, E. M. Clarke, and Y. Zhu, "Combining symbolic model checking with uninterpreted functions for out-of-order processor verification," in *Proc. 2nd Int. Conf. FMCAD*, 1998, pp. 369–386.
[17] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Proc. 12th Int. Conf. CAV*, 2000, pp. 159–169.
[18] R. Hojati and R. K. Brayton, "Automatic datapath abstraction in hardware systems," in *Proc. 7th Int. Conf. Comput.-Aided Verification*, 1995, vol. 939, pp. 98–113.
[19] V. Paruthi, N. Mansouri, and R. Vemuri, "Automatic data path abstraction for verification of large scale designs," in *Proc. ICCD Topic: Verification and Test*, 1998, pp. 192–194.
[20] E. Hansen, "A generalized interval arithmetic," in *Proc. Int. Symp. Interval Mathematics*, 1975, vol. 29, pp. 7–18.
[21] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5–35, 1991.
[22] A. Benveniste, B. Caillaud, and P. L. Guernic, "From synchrony to asynchrony," in *Proc. Int. Conf. Concurrency Theory*, 1999, pp. 162–177. [Online]. Available: citeseer.ist.psu.edu/benveniste99from.html
[23] D. Potop-Butucaru and B. Caillaud, "Correct-by-construction asynchronous implementation of modular synchronous specifications," in *Proc. Int. Conf. Appl. Concurrency Syst. Design*, St. Malo, France, 2005, pp. 48–57.
[24] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 9, pp. 1059–1076, Sep. 2001. [Online]. Available: citeseer.ist.psu.edu/carloni01theory.html
[25] S. Suhaib, D. Mathaikutty, D. Berner, and S. Shukla, "Validating families of latency insensitive protocols," *IEEE Trans. Comput.*, vol. 55, no. 11, pp. 1391–1401, Nov. 2006.
[26] M. R. Casu and L. Macchiarulo, "A new approach to latency insensitive design," in *Proc. 41st Annu. DAC*, 2004, pp. 576–581.
[27] T. Raudvere, I. Sander, and A. Jantsch, "Synchronization after design refinements with sensitive delay elements," in *Proc. CODES+ISSS*, Salzburg, Austria, Oct. 2007, pp. 21–26.
[28] I. Sander and A. Jantsch, "System modeling and transformational design refinement in ForSyDe," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 23, no. 1, pp. 17–32, Jan. 2004.
[29] I. Sander, "System Modeling and Design Refinement in ForSyDe," Ph.D. dissertation, Royal Inst. Technol., Stockholm, Sweden, 2003.
[30] *The SMV model checker*. [Online]. Available: http://www-cad.eecs.berkeley.edu
[31] A. Peymandoust and G. D. Micheli, "Using symbolic algebra in algorithmic level DSP synthesis," in *Proc. 38th Annu. DAC*, 2001, pp. 277–282. [Online]. Available: citeseer.ist.psu.edu/peymandoust01using.html
[32] J. E. Eaton, "The fundamental theorem of algebra," *Amer. Math. Mon.*, vol. 67, no. 6, pp. 578–579, 1960.
[33] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "A framework for modeling the distributed deployment of synchronous designs," *Form. Methods Syst. Des.*, vol. 28, no. 2, pp. 93–110, Mar. 2006.
[34] Z. Lu, I. Sander, and A. Jantsch, "A case study of hardware and software synthesis in ForSyDe," in *Proc. 15th ISSS*, Kyoto, Japan, Oct. 2002, pp. 86–91.

Tarvo Raudvere received the Dipl.Eng. and M.Sc. degrees in computer and system engineering from Tallinn University of Technology, Tallinn, Estonia, in 1998 and 2000, respectively, and the Ph.D. degree in electronic system design from the Royal Institute of Technology, Stockholm, Sweden, in 2007.

Currently, he is a Test Engineer of embedded systems in Tallinn. His research interests include system level design, formal verification, and latency insensitive design.

Ingo Sander (S'97–M'02) received the M.Sc. degree in electrical engineering from the Technical University of Braunschweig, Braunschweig, Germany, in 1990, and the Ph.D. degree from the Royal Institute of Technology, Stockholm, Sweden, in 2003.

He is currently a Senior Lecturer with the School of Information and Communication Technology, which is part of the Royal Institute of Technology. His research interests include design methodologies and architectures for embedded systems.

Axel Jantsch (M'97) received the Dipl.Ing. and Dr. Tech. degrees from the Technical University of Vienna, Vienna, Austria, in 1988 and 1992, respectively.

Between 1993 and 1995, he received the Alfred Schrödinger scholarship from the Austrian Science Foundation as a Guest Researcher with the Royal Institute of Technology (KTH), Stockholm, Sweden. From 1995 to 1997, he was with Siemens Austria, Vienna, as a System Validation Engineer. Since 1997, he has been with KTH as an Associate Professor, where he became a Full Professor in electronic system design in December 2002 with the School of Information and Communication Technology. He is heading a number of research projects involving a total number of ten Ph.D. students, in two main areas: system modeling and networks on chip. He has published over 160 papers in international conferences and journals and one book in the areas of VLSI design and synthesis, system level specification, modeling and validation, HW/SW codesign and cosynthesis, reconfigurable computing, and networks on chip.

Dr. Jantsch has served on a number of technical program committees of international conferences such as FDL, DATE, CODES+ISSS, SOC, NOCS, and others. He was the TPC Chair of SSDL/FDL in 2000, TPC Cochair of CODES+ISSS in 2004, and General Chair of CODES+ISSS in 2005. Since December 2002, he has been the Subject Area Editor for the *Journal of System Architecture*.