(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
22 November 2001 (22.11.2001)

PCT

(10) International Publication Number
WO 01/88765 A2

(51) International Patent Classification[7]: G06F 17/50

(21) International Application Number: PCT/US01/14973

(22) International Filing Date: 8 May 2001 (08.05.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

09/566,684	8 May 2000 (08.05.2000)	US
09/566,692	8 May 2000 (08.05.2000)	US
09/566,683	8 May 2000 (08.05.2000)	US
09/566,682	8 May 2000 (08.05.2000)	US

(71) **Applicant** *(for all designated States except US)*: **REAL INTENT, INC.** [US/US]; 3910 Freedom Cricle, #102A, Santa Clara, CA 95054 (US).

(72) **Inventors; and**
(75) **Inventors/Applicants** *(for US only)*: **NARAIN, Prakash** [IN/US]; 237 Clifton Avenue, San Carlos, CA 94070 (US). **KUMAR, Rajiv** [IN/US]; 2889 Agate Avenue, Santa Clara, CA 95051 (US). **BEARDSLEE, John, M.** [US/US]; 431 Sherwood Way, Menlo Park, CA 94025 (US). **RANJAN, Rajeev, K.** [IN/US]; 2737 Mauricia Avenue, Apartment A, Santa Clara, CA 95051 (US). **MORRISON, Christopher, R.** [US/US]; 927 Primrose Avenue, Sunnyvale, CA 94086 (US).

(74) **Agents: MALLIE, Michael, J.** et al.; Blakely, Sokoloff, Taylor & Zafman LLP, 7th floor, 12400 Wilshire Boulevard, Los Angeles, CA 90025 (US).

(81) **Designated States** *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) **Designated States** *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:
— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

WO 01/88765 A2

(54) **Title:** INTENT-DRIVEN FUNCTIONAL VERIFICATION OF DIGITAL DESIGNS

(57) **Abstract:** A method and apparatus are provided that facilitate analysis of the intended flow of logical signals between key points in a design. According to one aspect of the present invention, hardware design defects can be detected using a novel Intent-Driven Verification process. First, a representation of a hardware design and information regarding the intended flow of logical signals among variables in the representation are received. Then, the existence of potential errors in the hardware design may be inferred based upon the information regarding the intended flow of logical signals by (1) translating the information regarding the intended flow of logical signals into a comprehensive set of checks that must hold true in order for the hardware design to operate in accordance with the intended flow of logical signals, and (2) determining if any of the checks can be violated during operation of circuitry represented by the hardware design.

INTENT-DRIVEN FUNCTIONAL VERIFICATION OF DIGITAL DESIGNS

COPYRIGHT NOTICE

Contained herein is material that is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction of the patent disclosure by any person as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all rights to the copyright whatsoever.

BACKGROUND OF THE INVENTION

Field of the Invention

The invention relates generally to the field of design verification. More particularly, the invention relates to a new approach for functional verification of digital designs.

Description of the Related Art

The objective of design verification is to ensure that errors are absent from a design. Deep sub-micron integrated circuit (IC) manufacturing technology is enabling IC designers to put millions of transistors on a single IC. Following Moore's law, design complexity is doubling every 12-18 months, which causes design verification complexity to increase at an exponential rate. In addition, competitive pressures are putting increased demands on reducing time to market. The combination of these forces has caused an ever worsening "verification crisis".

Today's design flow starts with a specification for the design. The designer then implements the design in a language model, typically Hardware Description Language (HDL). This model is typically verified to discover incorrect input/output (I/O) behavior via a stimulus in expected results out paradigm at the top level of the design.

By far the most popular method of functional verification today, simulation-based functional verification, is widely used within the digital design industry as a method for finding defects within designs. A very wide variety of products are available in the market to support simulation-based verification methodologies.

However, a fundamental problem with conventional simulation-based verification approaches is that they are vector and testbench limited.

Simulation-based verification is driven by a testbench that explicitly generates the vectors to achieve stimulus coverage and also implements the checking mechanism. Testbenches create a fundamental bottleneck in simulation-based functional verification. In order to verify a design hierarchy level, a testbench must be generated for it. This creates verification overhead for coding and debugging the testbench. Hence, a significant amount of expensive design and verification engineering resources are needed to produce results in a cumbersome and slow process.

Several methods have been attempted by Electronic Design Automation (EDA) companies today in order to address the shortcomings of simulation. However, none of these attempts address this fundamental limitation of the process. For example, simulation vendors have tried to meet the simulation throughput challenge by increasing the performance of hardware and software simulators thereby allowing designers to process a greater number of vectors in the same amount of simulation time. While this does increase stimulus coverage, the results are incremental. The technology is not keeping pace with the required growth rate and the verification processes are lagging in achieving the required stimulus coverage.

Formal verification is another class of tools that has entered the functional verification arena. These tools rely on mathematical analysis rather than simulation of the design. The strong selling point of formal verification is the fact that the results hold true for all possible input combinations to the design. However, in practice this high level of stimulus coverage has come at the cost of both error coverage and particularly usability. While some formal techniques are available, they are not widely used because they typically require the designer to know the details of how the tool works in order to operate it. Formal verification tools generally fall into two classes: (1) equivalence checking, and (2) model checking.

Equivalence checking is a form of formal verification that provides designers with the ability to perform RTL-to-gate and gate-to-gate comparisons of a design to determine if they are functionally equivalent. Importantly, however, equivalence checking is not a method of functional verification. Rather, equivalence checking

merely provides an alternate solution for comparing a design representation to an original golden reference. It does not verify the functionality of the original golden reference for the design. Consequently, the original golden reference must be functionally verified using other methods.

Model checking is a functional verification technology that requires designers to formulate properties about the design's expected behavior. Each property is then checked against an exhaustive set of functional behaviors in the design. The limitation of this approach is that the designer is responsible for exactly specifying the set of properties to be verified. The property specification languages are new and obscure. Usually the technology runs into capacity problems and the designer has to engage with the tools to solve the problems. There are severe limits on the size of the design and the scope of problems that can be analyzed. For example, the designer does not know which properties are necessary for complete analysis of the design. Further, specifying a large number of properties does not correlate well with better error coverage. Consequently, model checking has proven to be very difficult to use and has not provided much value in the verification process.

In view of the foregoing it would be desirable to create a verification methodology to create high quality designs without the need for simulation testbench and to increase the productiveness of design engineers by minimizing the tool setup effort and report processing effort. In particular, it would be advantageous to abstract the internal details of the tools from the user to make these tools more accessible to designers. For example, it would be advantageous to allow a verification methodology to be employed while allowing the designer to think about characteristics of the design. In this manner, the designer can think in terms of time progression of data at various design elements (entities) rather than the implementation of the verification tool. Additionally, rather than requiring the designer to write properties in a complex and arcane language, it would be advantageous to automatically formulate a list of verification checks that, if satisfied, would guarantee the absence of errors in a design with a high level of confidence. Finally, it would be advantageous to maintain and utilize relationships among the list of automatically generated verification checks to

facilitate error reporting and to prune the error space for more efficient run-time processing.

BRIEF SUMMARY OF THE INVENTION

A method and apparatus are described that facilitate analysis of the intended flow of logical signals between key points in a design. According to one aspect of the present invention, hardware design defects can be detected using a novel Intent-Driven Verification process. First, a representation of a hardware design and information regarding the intended flow of logical signals among variables in the representation are received. Then, the existence of potential errors in the hardware design may be inferred based upon the information regarding the intended flow of logical signals by (1) translating the information regarding the intended flow of logical signals into a comprehensive set of checks that must hold true in order for the hardware design to operate in accordance with the intended flow of logical signals, and (2) determining if any of the checks can be violated during operation of circuitry represented by the hardware design. Advantageously, in this manner, the designer is not required to manually code individual monitors for each property he/she would like to verify. Rather, verification cycle time and resource requirements are reduced by allowing the designer to simply annotate a language representation of the hardware design under test with information regarding the desired/expected interaction between components of the design and/or the designer's expectations for acceptable functional behavior (in terms of the expected state of variables at various points in the control flow structure of a finite state machine associated with the hardware design representation, for example) and the generation of a comprehensive set of checks for identifying the intent gap is automatically performed.

Other features of the present invention will be apparent from the accompanying drawings and from the detailed description that follows.

BRIEF DESCRIPTION OF THE SEVERAL VIEWS OF THE DRAWINGS

The present invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings and in which like reference numerals refer to similar elements and in which:

Figure 1 is a block diagram illustrating an exemplary architecture of an Intent-Driven Verification system according to one embodiment of the present invention.

Figure 2 is a high-level flow diagram illustrating an Intent-Driven Verification processing according to one embodiment of the present invention.

Figure 3 is an example of a computer system upon which one embodiment of the present invention may be implemented.

Figure 4A is a high-level block diagram illustrating an exemplary Sentry verification entity according to one embodiment.

Figure 4B is a schematic diagram illustrating an exemplary implementation of a Sentry verification entity according to one embodiment.

Figure 4C illustrates what is meant by concept of flow of logical signals according to one embodiment of the present invention.

Figure 5 is a flow diagram that illustrates the automatic formulation of design verification checks according to one embodiment of the present invention.

Figure 6A is a block diagram illustrating a design example.

Figure 6B is a block diagram illustrating the design example of **Figure 6A** with the inclusion of Sentry verification entities.

Figure 7 is a state diagram illustrating the operation of the bus interface units of **Figures 6A and 6B.**

Figure 8 is a simplified state diagram illustrating the operation of the round robin arbiter of **Figures 6A and 6B.**

Figures 9A - 9E represent an exemplary RTL source code representation of the design example of **Figure 6B.**

Figure 10 is a block diagram that conceptually illustrates linking processing according to one embodiment of the present invention.

Figure 11 is a high-level flow diagram that illustrates linking processing according to one embodiment of the present invention.

Figure 12 is a flow diagram that illustrates processing block 1130 of **Figure 11** according to one embodiment of the present invention.

Figures 13A – 13E illustrate an exemplary text report file for the annotated RTL of **Figures 9A - 9E** according to one embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

A method and apparatus are described that facilitate analysis of the intended flow of logical signals between key points in a design. Embodiments of the present invention seek to solve or at least alleviate the above-referenced problems of conventional verification approaches by employing a revolutionary approach for functional verification. Importantly, this approach enables verification of design intent prior to simulation and synthesis. Identifying and eliminating design defects early in the design cycle eliminates costly down-stream design iterations resulting in dramatically shorter design verification cycles.

Broadly stated, embodiments of the present invention allow errors in a hardware design to be discovered by checking a representation of the hardware design against design intent. In one embodiment, a comprehensive set of design verification checks including behavioral integrity checks and temporal integrity checks are automatically generated based upon a language representation of a hardware design and information regarding the design intent, e.g., the intended flow of logical signals among a plurality of variables in the language representation of the hardware design. The design intent may be communicated to a hardware design verification tool, for example, by way of inline annotations embedded within the hardware design description. Advantageously, in this manner, the design verification process is streamlined thereby allowing quality levels to be achieved faster and with reduced resource requirements.

According to one embodiment, a novel "verification entity," referred to as a Sentry™ verification entity, is provided that facilitates modeling and verification of what the designer intended to build (SENTRY is a trademark or registered trademark of Real Intent, Inc. of Santa Clara, California). For example, feedback may be provided to the designer regarding differences between what has actually been created by a language representation of a hardware design, such as a Register Transfer Language (RTL) source code representation, and the design intent, e.g., the intended flow of logical signals in a the hardware design.

Briefly, Sentry verification entities may conceptually be thought of as objects, which are embedded in a design during the verification process. Sentry verification entities are not structural entities – they are not included in the final hardware. Rather,

Sentry verification entities are special purpose objects for support of verification. Sentry verification entities provide a mechanism for expressing design intent. Expressed design intent (i.e., design intent that is explicitly stated by the designer) and implied design intent (i.e., expected behaviors that should occur within standard design practices although not explicitly stated by the designer) are checked against what the design actually accomplishes. As will be described further below, in one embodiment, Sentry verification entities are embedded within a model of the design under test to represent sentinel variables in the design through which logical signals in the design pass. For example, a Sentry verification entity may be used to explicitly associate state information with a sentinel variable independent of the value of the sentinel variable. The state information may indicate whether or not the sentinel variable is active or inactive at various points in the control flow structure of a finite state machine associated with the hardware design representation. Consequently, the integrity of the data flow can be verified by confirming checks that are expressed as a function of the states associated with one or more sentinel variables.

According to another embodiment, a comprehensive set of design verification checks may be formulated by applying predetermined properties to an annotated hardware design representation. The application of predetermined properties, such as conflicting assignments (CA), block enable (BE), assignment execution (AX), constant value memory element (CME), constant value variable (CV), Sentry activate always off (AAO), loss of valid data (LVD), assertion correctness (AC), and access of invalid data (AID), to signal propagation among all the sentinel variables results in an exhaustive list of checks that confirm whether or not the predetermined properties hold true for the intended flow of logical signals. Advantageously, in this manner, a designer gets the benefits of a comprehensive Intent-Driven Verification™ (IDV™) methodology by simply identifying sentinel variables and providing information regarding intended temporal behaviors and/or relationships (INTENT-DRIVEN VERIFICATION and IDV are trademarks or registered trademarks of Real Intent, Inc. of Santa Clara, California.)

According to yet another embodiment, an ordered representation of the comprehensive set of design verification checks can be created to allow conclusions to be drawn about the hardware design. For example, dependency relationships among the comprehensive set of design verification checks may be determined. A check is

dependent upon a set of other checks if it is impossible to violate the first without violating at least one or more checks from the set. After the dependency relationships have been determined, this linking information may be used to facilitate error reporting or to streamline check processing. Notably, reporting of multiple intent violations due to a common design defect may be avoided thereby containing redundant failures commonly produced by prior art simulators. Additionally, if there is no way to violate the current check being verified without also violating one or more other previously processed checks, there is no point in reporting the violation of the current check since this would not communicate any new information to the user. Finally, assuming it has already been determined that a first check cannot be violated, during the processing of a subsequent check, if the current search path violates the first check, then the current search path can be abandoned since there is no solution along the current search path. In this manner, checks may be more efficiently processed and the user is not inundated with redundant, cumulative information.

In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the present invention may be practiced without some of these specific details. In other instances, well-known structures and devices are shown in block diagram form.

The present invention includes various steps, which will be described below. The steps of the present invention may be performed by hardware components or may be embodied in machine-executable instructions, which may be used to cause a general-purpose or special-purpose processor programmed with the instructions to perform the steps. Alternatively, the steps may be performed by a combination of hardware and software.

The present invention may be provided as a computer program product that may include a machine-readable medium having stored thereon instructions, which may be used to program a computer (or other electronic devices) to perform a process according to the present invention. The machine-readable medium may include, but is not limited to, floppy diskettes, optical disks, CD-ROMs, and magneto-optical disks, ROMs, RAMs, EPROMs, EEPROMs, magnet or optical cards, flash memory, or other type of media / machine-readable medium suitable for storing electronic instructions.

Moreover, the present invention may also be downloaded as a computer program product, wherein the program may be transferred from a remote computer to a requesting computer by way of data signals embodied in a carrier wave or other propagation medium via a communication link (e.g., a modem or network connection).

For convenience, embodiments of the present invention will be described with reference to Verilog and VHDL. However, the present invention is not limited to any particular representation of a hardware design. For example, the language representation of a hardware design may be in the C programming language, C++, derivatives of C and/or C++, or other high-level languages. In addition, while embodiments of the present invention are described with reference to functional verification of hardware designs, aspects of the present invention are equally applicable to other types of design activities as well, such as hardware synthesis, design optimization, design simulation and performance analysis. Furthermore, while embodiments of the present invention are described with reference to the provision of augmented design information by way of hardware description language (HDL) annotations, it is contemplated that the augmented design information could reside in a file separate from the file containing the HDL. Alternatively, a new HDL could be developed having semantics capable of capturing the augmented design information.

Terminology

Before describing an illustrative design verification environment in which various embodiments of the present invention may be implemented, brief definitions of terms used throughout this application are given below.

A "design" is defined as a description of a collection of objects, such as modules, blocks, wires, registers, etc. that represent a logic circuit.

A design may be expressed in the form of a language. For example, a hardware description language (HDL), such as Verilog or VHDL can be used to describe the behavior of hardware as well as its implementation.

As used herein an "annotation" refers to text embedded in a language construct, such as a comment statement or remark. Most programming languages have a syntax for creating comments thereby allowing tools designed to read and/or process the programming language to read and ignore the comments.

"Simulation" is generally the process of evaluating design behavior for a set of input conditions to draw approximate conclusions about the behavior of many different attributes of the design.

"Formal analysis" generally refers to the process of analyzing a design for all possible input conditions to derive definite conclusions about the behavior of an attribute with respect to the design.

"Functional verification" generally refers to the process of applying stimuli to a design under test with appropriate checking mechanisms to either detect a defect in the design or to establish that the functionality performs as expected. Typically, the three key components of a functional verification process are the applied stimulus, the checking mechanism, and the user's ability to both run the process and debug the results. As will be described later, the effectiveness of a functional verification process may be measured in terms of the following three metrics: (1) error coverage, (2) stimulus coverage, and (3) usability.

The term "design intent" generally refers to what the designer intends in terms of the interaction between components of the design and the designer's expectations regarding acceptable functional behavior. For example, the designer may intend a particular flow of logical signals among the variables of an RTL design description (the

intended flow). Traditionally, design intent has referred to input constraints, internal constraints, and/or cause and effect modeling. In contrast, as used herein the term "design intent" includes "implied design intent" and additional forms of "expressed design intent." Expressed design intent generally refers to design intent explicitly conveyed by a designer by way of intent modeling semantics in a control file or embedded within annotations of RTL source code (the hardware design representation), for example. Examples of expressed design intent regarding the intended flow of logical signals may include the intended temporal behaviors (e.g., the ACK signal must go high within 4 cycles of the REQ signal) and the intended data flow relationships (e.g., the data being loaded from the input port is the data intended for transfer by the driver of the input port). Implied design intent generally refers to design intent that is inferred from the design description including expected behaviors that should occur within standard design practices.

A "Sentry verification entity" or "verification entity" generally refers to an object that may be embedded within a hardware design to facilitate the modeling of design intent. As described further below, Sentry verification entities provide a mechanism by which state information can be associated with variables in a representation of a hardware design. According to various embodiments of the present invention, Sentry verification entities may be used to verify a design by allowing design intent (both expressed and implied) to be checked against what the design actually accomplishes.

A "property" is a condition or statement, typically expressed in terms of a relationship among a group of one or more signals in a hardware design, that must hold for the hardware design to function as intended. According to one embodiment, violations are reported at the property-level rather than at the more detailed level of design verification checks.

The term "design verification check" or simply "check" generally refers to a mechanism for verifying a property. Properties may be composed of one or more checks, which are the atomic units that are verified. That is, properties may be broken down into one or more checks. Since properties and checks are sometimes equivalent (i.e., when a property comprises a single design verification check), these terms may at

times be used interchangeably. Examples of checks include, but are not limited to: monitors, Boolean conditions, sensitized path conditions, and the like. According to one embodiment, a comprehensive set of checks may be automatically formulated based upon a representation of the hardware design and information regarding the intended flow of logical signals among a plurality of variables in the representation.

Check A is said to be "dependent" upon checks B, C, and D if it is impossible to violate check A without violating at least one of checks B, C, and D. If check A is dependent upon check B, C and D, then a "dependency relationship" exists between check A and checks B, C, and D.

Characteristics of Effective Functional Verification

As mentioned earlier, the effectiveness of a functional verification process may be measured in terms of error coverage, stimulus coverage, and usability. Error coverage refers to the ability of the verification methodology to identify a broad range of potential design defects. A verification process should be effective in identifying all potential failures in a hardware design. This is achieved by creating a checking mechanism. In traditional practice, the checking mechanism is created by comparing the hardware design representation, e.g., RTL source code, to an independently created design model. This independent model is used to predict the response of the design for comparison. Given that the purpose of the verification process is to assure design quality, it is important to build a very comprehensive checking mechanism.

Stimulus coverage refers to the ability of the verification methodology's ability to achieve broad coverage of the input domain. Preferably, the verification process should examine all possible combinations of input sequences for the design under test. High stimulus coverage is critical to verification, particularly with designs containing large numbers of states. Yet, all existing technologies fall short here. The ideal verification process should achieve exhaustive stimulus coverage for any design.

Usability refers to the overall ease and effectiveness of user interaction with the verification tool. The verification process should be simple and effective to use. To accomplish this, setup costs for running the process should be minimized, and the reported information should be well organized to allow easy debugging. The process

should fit well into existing design methodologies and should require a minimum amount of training in order to be used effectively. Usability should be given high priority when developing any functional verification system and should not be an afterthought that is secondary to the underlying technology.

Intent-Driven Verification Overview

Intent-Driven Verification (IDV) is a revolutionary approach for functional verification of hardware designs. IDV ranks high on all three of the above-described metrics and successfully integrates the technologies required for an effective functional verification system together into a complete package. Briefly, IDV identifies the "intent gap" between what a designer intends to build (design intent) and what has actually been created within the language representation of the hardware design (design implementation), such as RTL source code. Advantageously, by simply augmenting the RTL source code or associated control file with information indicative of the designer's intent, a comprehensive set of checks can be automatically formulated and verified. This provides a novel and effective way to identify design problems early in the development process. Additionally, IDV is scalable, provides fast isolation of the source of defects, and significantly reduces the overall verification effort, thus enabling IC design projects to dramatically reduce their time to market.

An Exemplary Intent-Driven Verification System

Figure 1 is a block diagram illustrating an exemplary architecture of an Intent-Driven Verification (IDV) system according to one embodiment of the present invention. According to the embodiment depicted, the IDV system 100 includes an annotated hardware design representation reader 120. The annotated hardware design representation reader 120 may be a conventional RTL reader with the additional ability to recognize and process augmented design information. Alternatively, the annotated hardware design representation reader 120 may be a conventional C or C++ parser capable of processing the augmented design information. In one embodiment, the augmented design information comprises annotations embedded within the hardware design representation 105, such as an annotated RTL source code file 105 containing special verification semantics (directives). Exemplary directives and their syntax are described further below. In alternative embodiments, the augmented design information may be included in a control file associated with the hardware design representation 105, such as control file 115.

The IDV system 100 also includes a control file reader 130. The control file reader 130 may be a conventional control file reader. Alternatively, in the case that the augmented design information (e.g., information regarding design intent, such as the intended flow of logical signal among variables in the hardware design representation) is to be provided by way of control file 115, then the control file reader 130 additionally includes parsing functionality enabling the control file reader 130 to recognize and process the augmented design information.

According to the architecture depicted, the IDV system 100 also includes a model builder 145, a design intent analyzer 140, a property manager 165, an analysis engine 180, and a report manager 170. The model builder 145 receives output of the hardware design representation reader 120, the control file reader 130, and a cell library (not shown) and builds an internal representation of the hardware design, a model. Additionally, based upon the designer's expressed design intent and/or implied intent, the model builder embeds special verification entities into the model for the purpose of identifying the intent gap between what the designer intended to build (design intent) and what has actually been created within the source code (design implementation).

These special embedded objects are referred to as Sentry verification entities. Details regarding the use, functionality, and implementation of Sentry verification entities are presented below.

The design intent analyzer 140 automatically produces a comprehensive set of design verification checks, such as behavioral integrity checks and temporal integrity checks, based upon a predetermined set of properties and the Sentry verification entities. For example, for each Sentry verification entity in the model, the design intent analyzer 140 may automatically create checks to verify that there is no access of invalid data by the Sentry verification entity and that no active data from the Sentry verification entity is lost. Other properties and verification of these properties, in terms of checks and Sentry verification entities, are discussed below. Consequently, by merely associating the Sentry attribute with a variable in an RTL representation of the hardware design and providing minimal additional information regarding the desired/expected interaction between components of the design and regarding expectations for acceptable functional behavior, the design intent analyzer 140 enables verification while the design is being developed and facilitates detection of design defects at the RTL-level. Advantageously, in this manner, the designer is not required to manually code individual monitors for each property he/she would like to verify. Rather, verification cycle time and resource requirements are reduces by allowing the designer to simply associate the Sentry attribute with certain important variables, sentinel variables (e.g., by "declaring" the sentinel variables as Sentry verification entities), provide minimal additional information regarding the desired/expected interaction between components of the design (such as an indication regarding the expected state of the sentinel variables at various points in the control flow structure of a finite state machine associated with the hardware design representation), and the generation of a complete set of checks involving the sentinel variables is automatically performed. Importantly, the use of Sentry verification entities to enable automatic formulation of a comprehensive set of design verification checks not only dramatically reduces verification effort and time but also significantly increases design robustness.

The property manager 165 maintains information regarding relationships among the checks generated by the design intent analyzer and additionally maintains information regarding relationships among properties and whether the properties are

satisfied, violated, or whether the results are indeterminate or conditional upon one or more other properties. According to one embodiment, the property manger 165 and analysis engine 180 cooperate to determine dependency relationships among the comprehensive set of design verification checks. As described further below, the dependency relationships (or "linking" information) may be used to facilitate error reporting or used to streamline check processing.

The analysis engine 180 verifies the model produced by the model builder by testing for violations of the properties. Preferably, the analysis engine 180 employs an analysis-based technique, such as an integration of simulation and formal analysis methodologies, so as to maximize stimulus coverage for the design under test. However, in alternative embodiments, simulation, functional verification, or other well-known verification methodologies may be employed individually.

The report manager 170 provides feedback to the designer, in the form of a report file 175, for example, regarding potential design defects. It should be noted that a single error in the hardware design might lead to a violation of 25 to 30 or more different design verifications checks. There is no point in reporting every violation if the user has already been notified that an error exists. Therefore, according to one embodiment, an intent violation hierarchy may be maintained when reporting design defects. In this manner, redundancy of reported information is contained. Multiple intent violations due to a common defect are not reported. This minimizes the amount of data to be analyzed to detect defects.

High-Level Intent-Driven Verification Processing

Figure 2 is a high-level flow diagram illustrating Intent-Driven Verification processing according to one embodiment of the present invention. The Intent-Driven Verification process generally breaks down into three fundamental technologies, intent capture, intent gap detection, and intent violation reporting. A method for capturing the designer's intent for the design (intent capture) is represented by processing blocks 210 – 230, detection of areas within the design source code that differ from the original design intent (intent gap detection) comprises processing block 240, and identification and reporting of the space between the designer's intent and the design's implementation (intent violation reporting) is represented by processing block 250. Briefly, after a complete view of the overall design intent is captured, e.g., after the two forms of intent are integrated together by the design intent analyzer 140, the analysis engine 180 may begin identifying individual intent violations. When the analysis is complete, the report manager 170 provides feedback to the user in a well-organized manner. In one embodiment, the processing blocks described below may be performed under the control of a programmed processor, such as processor 302. However, in alternative embodiments, the processing blocks may be fully or partially implemented by any programmable or hardcoded logic, such as Field Programmable Gate Arrays (FPGAs), TTL logic, or Application Specific Integrated Circuits (ASICs), for example.

Intent-Driven Verification processing begins at processing block 210 where a language representation of a hardware design under test is received. The language representation may be RTL source code, such as Verilog or VHDL. Preferably, to take advantage of the control flow structure of the language representation, augmented design information is provided by inline annotations to the language representation of the hardware design, e.g., annotated RTL 105. However, in alternative embodiments, design intent may be provided in the control file 115.

In this example, at processing blocks 220 and 230, the design intent is captured in two basic forms: expressed and implied. Expressed design intent is design intent that is explicitly stated by the designer. The implementation of IDV described herein may use an elegant RTL-based method to express this intent in an annotated RTL source code file, such as annotated hardware description representation 105, and/or the

control file 115. Regardless of their source, the expressed design intent is determined based upon the annotations. Importantly, the form of intent modeling provided herein is very intuitive and does not require the users to learn a new language. Additionally, this intent modeling captures the intent in a concise manner and thereby minimizes setup effort. Typically, the expressed intent defines the interaction between components of the design and/or the designer's expectations regarding acceptable functional behavior. Implied design intent may be inferred directly from the design description. Advantageously, implied intent capture infers numerous and complex design behaviors automatically without requiring a designer to explicitly state them. Typically, this consists of expected behaviors that should occur within standard design practices.

At processing block 240, the intent gap is identified based upon the design intent and the design implementation. In the example depicted, the two forms of design intent are integrated and used as input to processing block 240. In alternative embodiments, the intent gap may be determined for only a single form of design intent or one or more other combinations of subsets of expressed and implied design intent. At any rate, preferably, the intent gap is statically identified using functional verification techniques. The intent gap is the distance between a design's intent and the actual implementation. According to one embodiment, IDV detects the intent gap in terms of individual intent violations. Intent violations are the result of defects within the design implementation. Importantly, because the intended behavior of the design is well understood, intent violations isolate design defects with high localization to their source.

At processing block 250, feedback is provided to the user. For example, one or more report files 175 may be created that provide information regarding the individual intent violations identified in processing block 240. Preferably, the redundancy of information is contained and the intent violation data is organized into a format that provides useful and highly accurate debugging information to a designer. Additionally, sequential debugging information may be provided along with detailed explanations of any identified intent violations. An exemplary report file format is illustrated in **Figures 13A – 13E.**

An Exemplary Computer Architecture

Having briefly described an exemplary verification system in which various features of the present invention may be employed and high-level Intent-Driven Verification processing, an exemplary machine in the form of a computer system 300 representing an exemplary workstation, host, or server in which features of the present invention may be implemented will now be described with reference to **Figure 3**. Computer system 300 comprises a bus or other communication means 301 for communicating information, and a processing means such as processor 302 coupled with bus 301 for processing information. Computer system 300 further comprises a random access memory (RAM) or other dynamic storage device 304 (referred to as main memory), coupled to bus 301 for storing information and instructions to be executed by processor 302. Main memory 304 also may be used for storing temporary variables or other intermediate information during execution of instructions by processor 302. Computer system 300 also comprises a read only memory (ROM) and/or other static storage device 306 coupled to bus 301 for storing static information and instructions for processor 302.

A data storage device 307 such as a magnetic disk or optical disc and its corresponding drive may also be coupled to computer system 300 for storing information and instructions. Computer system 300 can also be coupled via bus 301 to a display device 321, such as a cathode ray tube (CRT) or Liquid Crystal Display (LCD), for displaying information to an end user. For example, graphical and/or textual depictions/indications of design errors, and other data types and information may be presented to the end user on the display device 321. Typically, an alphanumeric input device 322, including alphanumeric and other keys, may be coupled to bus 301 for communicating information and/or command selections to processor 302. Another type of user input device is cursor control 323, such as a mouse, a trackball, or cursor direction keys for communicating direction information and command selections to processor 302 and for controlling cursor movement on display 321.

A communication device 325 is also coupled to bus 301. Depending upon the particular design environment implementation, the communication device 325 may include a modem, a network interface card, or other well-known interface devices, such

as those used for coupling to Ethernet, token ring, or other types of physical attachment for purposes of providing a communication link to support a local or wide area network, for example. In any event, in this manner, the computer system 300 may be coupled to a number of clients and/or servers via a conventional network infrastructure, such as a company's Intranet and/or the Internet, for example.

Modeling Intended Flow of Logical Signals in a Hardware Design

Currently available verification methodologies lack suitable mechanisms to allow designers to adequately express their design intent. Rather, existing functional verification methodologies, such as model checking, generally require a designer to formulate properties about the design's expected behavior. Consequently, the designer is responsible for exactly specifying the set of properties to be verified. One problem with this approach, however, is that the designer may not know or even be capable of knowing which properties are necessary for a complete analysis of the design. Additionally, the property specification languages are new and obscure and typically require the designer to know the details of how the verification tool works in order to operate it.

Briefly, the intent modeling techniques described herein seek to raise the level of abstraction of the verification process to increase the productiveness of existing design environments/design verification tools. In particular, there is no need for the designer to have any knowledge regarding the internal details of tools employing these intent modeling techniques. Rather, the designer is free to focus on characteristics of the design under test. For example, according to one embodiment, incorporation of Sentry verification entities into a hardware design representation only requires simply annotations indicative of the expected state of variables at various points in the control flow structure of a finite state machine associated with the hardware design representation.

Referring now to **Figure 4A**, a high-level block diagram illustrating an exemplary Sentry verification entity 400 will now be described. At this level, the Sentry verification entity 400 may be conceptually thought of as an object that enables explicit association of state information with variables of a language description of a

hardware design. This association may be accomplished by "declaring" a variable, e.g., an interconnect in the hardware design representation through with logical signals pass, as a Sentry verification entity resulting in the creation of a sentinel variable. Data in 420 represents a flow of data signals into to the sentinel variable (e.g., an assignment of a data value to the sentinel variable); and data out 440 represents a flow of data signals out of the sentinel variable (e.g., an assignment of the current data value of the sentinel variable to another variable in the representation of the hardware design). As described further below, one or more control signals 410 may be employed to represent the intended flow of logical signals among sentinel variables by associating expected states with the sentinel variables at different points of the designs control flow. Additionally, design verification checks, expressed as a function of the expected states of the sentinel variables, may be automatically formulated thereby allowing the integrity of the data flow to be verified by confirming whether or not any positive checks (e.g., checks that indicate a design defect when there is a solution) are violated or whether or not a counterexample may be found for any negative checks (e.g., checks that indicate a design defect when there is not a solution).

Figure 4B is a schematic diagram illustrating an exemplary implementation of a Sentry verification entity 400 according to one embodiment. In the embodiment depicted, the Sentry verification entity 400 includes a state latch 445 to store the current state 435 of a variable. The Sentry verification entity 400 also includes state maintenance logic 455 for maintaining the state 435 or updating the state 435 based upon a deactivate control signal 411 and an activate signal 412. In this example, the state maintenance logic 455 comprises an AND gate 415 and an OR gate 425. The AND gate 415 receives the current state 435 of the corresponding variable from the state latch 445 and an inverted version of the deactivate control signal 411. The OR gate 425 receives the output of the AND gate 415 and the activate control signal 412. Consequently, the state maintenance logic 455 outputs the same state input from the state latch 445 when both the activate control signal 412 and the deactivate control signal 411 are logic 0 thereby holding the variable's current state. The state 435 transitions to active, e.g., logic 1, if the activate control signal is logic 1; and the state 435 transitions to inactive, e.g., logic 0, if the deactivate control signal is logic 1. In

this example, the input combination of logic 1 on both the activate control signal 412 and the deactivate control signal 411 is invalid. Importantly, while for simplicity the present example is illustrated with two states, active and inactive, the concept of associating state information with a variable in a hardware design representation by Sentry verification entities is extensible to more than two states. Also, it is appreciated that the implementation of the Sentry verification entity 400 may have many equivalent logical representations than that depicted in **Figure 4B**. The implementation of **Figure 4B**, therefore, is to be considered merely representative of one or the many various possible logically equivalent implementations.

Figure 4C illustrates what is meant by concept of intended flow of logical signals according to one embodiment of the present invention. In one embodiment, if a path (not necessarily a "sensitized path") exists between two sentinel variables, then there is a flow of logical signals from one to the other. In this example, since the outputs of sentinel variables 401, 402, and 403 are all electrically coupled to the input of sentinel variable 405 in the resultant hardware component, design verification checks associated with sentinel variable 405 will be a function of at least the state of sentinel variables 401, 402, and 403. Notably, in this example, there is not always a "sensitized path" between each of the sentinel variables 401, 402, and 403 and sentinel variable 405 as the output of any given sentinel variable 401, 402, and 403 does not always have an effect on the value of sentinel variable 405. Consequently, it may be said that a design structure implements the intended flow of logical signals between key elements (e.g., sentinel variables) in the design where the intended flow of logical signals consists of: (1) The logical signal at a key destination design element, at a given time, resulted from the logical signals at a set of key source design elements, at the same or earlier time, and would be different or will not change if a key source signal was different or did not change respectively; (2) The logical signal at a key destination design element, at a given time, independent of the logical signals at a set of key source design elements, at the same or earlier time, and will not change if a key source signal was different, (3) The logical signal at a key source design element, at a given time, should not govern the logical signal at a key destination design element. This intended

flow of logical signals can be inferred from the states associated with the key variables and the rules for correct design behavior.

Exemplary Annotations

Various statements that may be used to make up annotations will now be described. According to one embodiment, annotations can be entered in one of two ways:

(1) A Single Line *// vx statement;*

(2) Multiple Lines
```
/* vx code ON
        statement;
        statement;
*/
```

According to one embodiment, any synthesizeable Verilog can also be entered within the context of an annotation to create additional validation checks. Importantly, by placing these annotations within the RTL code itself, the designer performs three tasks in one:

1. Documents the source code through the assertions
2. Identifies assumptions made during block development
3. Prepares the RTL for validation

Several built-in constructs may be used in annotations to simplify assertion coding. The examples listed in **Table 1** cover common behaviors that many signal groups exhibit. In these examples, the syntax for an annotation comprises a comment delimiter (e.g., "//" or a "/*" "*/" pair), a directive marker (e.g., "vx"), a directive (e.g., "onehot"), and optional directive parameters (e.g., variables i, j, and k).

Annotation	Meaning
// vx onehot (i,j,k);	Exactly one of i, j, or k must be high
// vx onecold (i,j,k);	Exactly one of i, j, or k must be low
// vx onetrue (i,j,k);	At most one of i, j, or k can be high
// vx onefalse(i,j,k);	At most one of i, j, or k can be low

Table 1: Built-in Assertions

Such assertions allow the designer to explicitly specify design constraints for annotated signals. According to one embodiment, techniques are also provided for implicit design constraints based upon the control flow in the circuit. These implicit constraints are thought to be key to both block and interface validation.

In any design, signals are used to coordinate information exchange between different sections of the circuit. Such signals may be the contents of a data bus, or the handshaking signals between two interacting state machines. These key signals are referred to herein as sentinel variables of the design. The purpose of other logic in the circuit is to ensure proper interaction between these signals. A design failure is nothing but the incorrect exchange of data among sentinel variables.

According to one embodiment, sentinel variables are identified in annotations by the "sentry" directive. Sentry verification entities may be used to identify a sentinel signal and to model its behavior. In the examples described herein, Sentry verification entities have two states, active and inactive, that are controlled by "activate" and "deactivate" directives presented in vx annotations. The various Sentry verification entity controls, e.g., in the form of annotations, are placed in the control flow of the design so that the state of the Sentry verification entity is dependent on the state of the design. The Sentry verification entity should be activated when its corresponding sentinel variable carries valid information that is utilized somewhere in the design, and deactivated when the data is invalid or not used. Importantly, according to one embodiment, IDV automatically validates multiple forms of data exchange between

sentinel variables to ensure that the exchange is always valid. This checking mechanism works both in block-level and full-chip validation.

The designer is responsible for identifying signals that should be modeled as Sentry verification entities. Common sentinel variables in a design are: handshaking signals, data buses, state variables, and output ports. After sentinel variables have been identified, the designer may add appropriate annotations to the design representation.

Any signal (wire, register, or I/O) may be declared as a Sentry verification entity. A Sentry verification entity may also be declared on a sub-range of a vector. Each Sentry verification entity declaration should also identify a clock signal that controls when the corresponding sentinel variable may change its value. The following are examples of sentry annotations:

```
reg a;              // vx sentry a: clk;
wire b;             // vx sentry b: clk;
reg [0:15] c;       // vx sentry [1:2] c: clk;
```

According to one embodiment, Sentry verification entities alternate between active and inactive states based upon the control flow in the design. For instance, a data bus controlled by a state machine may be declared as a Sentry verification entity. When the state machine is in the idle state, the data bus Sentry verification entity is deactivated. Once the state machine enters a transmitting state, the data bus Sentry verification entity is activated. Any other Sentry verification entities in the design that try to access the data bus while it is deactivated will be flagged as an error, such as a "bad data access" error. Similarly, a design defect is indicated if no other Sentry verification entities in the design are accessing the data bus while it is active. For example, a "data loss" may be reported.

According to one embodiment, Sentry verification entities may be activated and deactivated using the annotations in **Table 2**.

Annotation	Meaning
// vx activate(a);	Activate Sentry *a*:

// vx deactivate(a);	Deactivate Sentry *a*:

Table 2: Exemplary Sentry verification entity controls

Sentry annotations may be placed within the control flow of the design so that the state of the Sentry verification entity is dependent on the control state. Sentry annotations can also be coded using the "`//vx code ON`" annotation thereby allowing the designer to code the Sentry verification entity states within their own annotation block separate from the source code for the hardware design representation.

Formulation of Design Verification Checks

Figure 5 is a flow diagram that illustrates the automatic formulation of design verification checks according to one embodiment of the present invention. As described above, properties may be verified by breaking them down into one or more checks and performing verification processing at the check-level. Briefly, according to this example, a set of predetermined properties are applied to each sentinel variable in the design representation. More specifically, the predetermined properties are applied to signal propagation among the sentinel variables to produce an exhaustive list of checks that confirm whether or not the predetermined properties are violated for the intended flow of logical signals.

According to the embodiment depicted, design verification check formulation begins at processing block 510 where a model of the hardware design under test is received. Preferably, the model includes embedded objects, such as Sentry verification entities, associated with each sentinel variable to facilitate check generation. At processing block 520, the model is traversed to locate the next sentinel variable. Upon locating the next sentinel variable, at processing block 530, design verification checks for the sentinel variable are automatically formulated based upon the predetermined properties. For example, each property may be expressed as one or more checks. Further, many of the checks may be expressed at least partially as a function of the states of one or more sentinel variables as described further below.

At processing block 540, a determination is made whether or not the model traversal is complete. If so, processing proceeds to processing block 550. Otherwise,

processing returns to processing block 520. At processing block 550, the design verification checks generated in processing block 530 are added to the property manager for subsequent linking and analysis, for example, which are described further below.

Exemplary Properties / Design Verification Checks

Empirical analysis performed by the assignee has shown that verification of a relatively small set of properties at key points of a hardware design representation (e.g., handshaking signals, data buses, state variables, and output ports) can guarantee the absence of errors in the hardware design with a high level of confidence.

An exemplary set of properties will now be described. Importantly, however, the present invention should not be construed as being limited to this particular set of properties. In various circumstances, a subset of the properties described below may be sufficient to achieve the desired level of confidence. Additionally, the set of properties described herein is not intended to be an exhaustive list. It is contemplated that Sentry verification entities and the verification techniques described herein would be equally useful in the context of a larger set of properties or even a completely different set of properties.

In one embodiment, the predefined set of properties applied to signal propagation among the sentinel variables of the hardware design representation include: access of invalid data (AID), loss of valid data (LVD), assertion correctness (AC), conflicting assignments (CA), block enable (BE), assignment execution (AX), constant value memory element (CME), constant value variable (CV), Sentry activate always off (AAO).

AID (Access of Invalid Data) is an undesirable condition. A Sentry verification entity that is in the valid state should not receive data that flowed through a Sentry verification entity that was in the invalid state. Therefore, a solution to one or more checks representing a violation of this property is indicative of a design defect. According to an embodiment of the present invention, an approach for finding AID is to search for an input sequence for the sentinel variable under verification, starting from the reset state, that will: (1) set the corresponding destination Sentry verification entity in the valid state, (2) create a sequentially sensitized path from a source Sentry verification entity to the Sentry verification entity under verification, and (3) set the source Sentry verification entity in the invalid state at the time when the sensitive data was flowing through the destination Sentry verification entity.

LVD (Loss of Valid Data) is another undesirable condition. Any data flowing through a Sentry verification entity that is in the valid state must be received by some Sentry verification entity that is in the valid state. Therefore, a solution to one or more checks representing a violation of this property is indicative of a design defect. According to an embodiment of the present invention, an approach for finding LVD is to search for an input sequence for the sentinel variable under verification, starting from the reset state, that will: (1) set the corresponding source Sentry verification entity in the valid state, and (2a) block the sensitized paths between the source Sentry verification entity and all possible destination Sentry verification entities, or (2b) set the destination Sentry verification entities with sensitized paths from the source Sentry verification entity, in the invalid state.

AC (Assertion Correctness) is a desired condition. An assertion represents a condition the designer explicitly indicated should occur at a particular point in the design. Consequently, a counterexample for compliance with the assertion is indicative of a design defect. According to an embodiment of the present invention, an approach for determining an AC violation is to search for an input sequence for the sentinel variable under verification, starting from the reset state, that will be a counterexample for the assertion.

The remaining properties (i.e., conflicting assignments, block enable, assignment execution, constant value memory element, constant value variable, and Sentry activate always off) may be verified in a similar manner. However, for sake of brevity, checking algorithms for these properties will not be described. Rather, only brief descriptions will be provided. CA (Conflicting Assignments) is an undesirable condition where a wire in the design can be driven by multiple conflicting drivers. BE (Block Enable) checks if the condition enabling the execution of a certain block of code in the HDL will never be enabled. AX (Assignment Execution) checks to see if each logical value can be assigned to a variable through each assignment. CME (Constant Value Memory Element) checks to see if a memory element in the design will always hold a constant value. CV (Constant Value Variable) checks to see if a variable in the design will always hold a constant value. AAO (Sentry Activate Always Off) is an

undesirable condition indicative of an error in the specification of the design intent where a sentry verification entity is never set in active state.

An Exemplary Design Example

In the design example illustrated by **Figures 6 - 9**, it is assumed that there is a need for three independent units, such as microprocessors, client A 605, client B 610, and client C 615, that are required to share access to the same memory 640, such as a synchronous RAM. In this design example, it is assumed that the memory 640 requires a single read/write signal. The following data/control signals are therefore needed from each unit: (1) Address - 10 bits, (2) Write data - 8 bits, (3) Read data - 8 bits, and (4) Read/write - 1 bit. As a result, an arbiter, such as round robin arbiter 636, has been designed that accepts data from each unit 605, 610, and 615 and arbitrates to determine which one is granted access to the memory 640 at any one time. Each unit 605, 610, and 615 will initiate a memory request signal when it wants access to the memory 640 and will deactivate it when finished. If more than one unit 605, 610, and 615 requests the bus 641 at the same time, access should be grated on a "round-robin" basis so that no one unit 605, 610, or 615 is locked out while another has continuous access. Continuous access is granted to any one unit 605, 610, or 615 for a period of time, up to a number of clock cycles separately programmable from client A's data bus 606. When a programmable "watch dog" time has not been set, an 8 clock cycle delay should default. For speed purposes, the use of Tri-state buffers, rather than multiplexers is desirable.

Figure 6B is a block diagram illustrating the design example of **Figure 6A** with the inclusion of Sentry verification entities at appropriate points of the circuit corresponding to the RTL representation of **Figures 9A – 9E**. Thus, **Figure 6B** conceptually illustrates the internal model representation after the model builder 145 embeds Sentry verification entities 645, 650, 655, 660, 665, and 670; and as seen by the design intent analyzer 140 and the analysis engine 180.

In this example, the Sentry verification entities 645, 650, 655, 660, 665, and 670 have been chosen to (1) verify the operation of the signals between clients 605, 610, and 615 and bus interface units (BIUs) 620, 625, and 630, respectively; and (2)

verify the operation of the signals between the BIUs 620, 625, 630 and the round robin arbiter 635. The data signals (i.e., dataA, dataB, and dataC) flowing from clients 605, 610, and 615 to bus interface units (BIUs) 620, 625, and 630, respectively, are declared as sentinel variables at line 20 of **Figure 9A**. The data signals (i.e., dataOut) flowing from BIUs 620, 625, and 630 to clients 605, 610, and 615, respectively, are declared as sentinel variables at line 50 of **Figure 9B**. The data signals (i.e., dataA, dataB, and dataC) received by the round robin arbiter 635 are declared as sentinel variables at line 115 of **Figure 9C**.

The states associated with the Sentry verification entities 645, 650, 655, 660, 665, and 670 at different points of the control flow are controlled by way of annotations at line 21 of **Figure 9A**, at lines 73, 81, and 91 of **Figure 9B**, at line 131 of **Figure 9C**, and at lines 156, 169, and 180 of **Figure 9D**. In this example, therefore, the designer has expressed his/her intent that (1) the Sentry verification entities associated with dataA, dataB, and dataC remain active constantly in module main; (2) on reset, the Sentry verification entity associated with dataOut is deactive; (3) the Sentry verification entity associated with dataOut is deactive when the corresponding BIU is in the "NO_REQ" state and active when the corresponding BIU is in the "GRANTED" state; (4) the Sentry verification entities associated with dataA, dataB, and dataC are deactive during state transition of the round robin arbiter 635; and (5) the Sentry verification entities associated with dataA, dataB, and dataC are activated when the round robin arbiter is in "stateA", "stateB", and "stateC", respectively, thereby allowing only one of dataA, dataB, and dataC to be active at any given time.

An exemplary text report file illustrating the verification results for the annotated RTL of **Figures 9A - 9E** will be described below with reference to **Figures 13A – 13E**.

Linking

As mentioned above, the property manager 165 may track information regarding relationships among the various design verification checks generated by the design intent analyzer. In the embodiments described herein, the linking process is performed concurrently with verification of the checks. In alternative embodiments,

however, linking and verification processing may be performed separately. In any event, the dependency relationships (or "linking" information) may be used to facilitate error reporting or used to streamline check verification processing.

Figure 10 is a block diagram that conceptually illustrates linking processing according to one embodiment of the present invention. According to this example, link processing involves both the property manager 165 and the analysis engine 180. The property manager 165 provides the next check 1005 to be processed to the analysis engine 180; and the analysis engine 180 returns link information 1010, such as whether or not the currently processed check is dependent upon another check and if so, which check. The analysis engine 180 includes a linking process 1020 and a local model 1025. The local model 1025 is initialized with a copy of the model 1015 generated by the model builder. However, during link processing, the local model is modified to disable various checks.

Figure 11 is a high-level flow diagram that illustrates linking processing according to one embodiment of the present invention. At processing block 1110, the next design verification check is received by the analysis engine 180 from the property manager 165. In order to avoid circular dependency relationships, those design verification checks that are already known to be dependent upon the received design verification check are disabled in the local model 1025 at processing block 1120. The received design verification check is evaluated at processing block 1130 and appropriate linking is also performed. Then, at processing block 1140, all of the design verification checks are enabled in anticipation of further check processing. Finally, at decision block 1150, a determination is made whether or not there are more checks to be processed. If so, control flow returns to processing block 1110. Otherwise, link processing is complete.

Figure 12 is a flow diagram that illustrates processing block 1130 of **Figure 11** according to one embodiment of the present invention. Processing of the received design verification check begins at processing block 1231 where the previous partial solution is expanded to create a current partial solution. Next, at processing block 1232, the current partial solution is evaluated; and a determination is made at processing block 1233 whether or not the current partial solution violates an

independent check. If it is determined that the current partial solution violates an independent check, then processing proceeds to processing block 1235. Otherwise, processing branches to decision block 1234.

At decision block 1234 the current partial solution is tested to determine whether or not it is complete. If the current partial solution is complete, then the currently processed check is marked as an independent check and processing continues with processing block 1238. Otherwise, if the current partial solution is not complete, then processing returns to processing block 1231.

Assuming the current partial solution has been found to violate an independent check, then at processing block 1235, the current check is temporarily locally marked as being dependent upon the independent check. At processing block, a determination is made whether or not the current partial solution can be rectified to remove the violation of the independent check. If the current partial solution can be so rectified, then processing proceeds to processing block 1237. Otherwise, processing continues with processing block 1238. At processing block 1237, the current solution is rectified to remove the violation and control flow returns to processing block 1231.

Finally, at processing block 1238, the local dependency information is used to update the master dependency information in the property database 160. In this manner, an intent violation hierarchy may be built for use during reporting of design defects.

Intent Violation Reporting

Figures 13A – 13E illustrate an exemplary text report file for the annotated RTL of **Figures 9A - 9E** according to one embodiment of the present invention. In this example, a summary of the functional checks is listed at lines 5 through 17. The summary indicates a total of five failed checks, three inconclusive checks, twelve interface checks, and 144 passed checks (some of which may be conditional). A summary of the failed checks is listed at lines 27 through 38. One block enable violation is reported at line 31, one assignment execution violation is reported at line 32, and three occurrences of loss of valid data are reported at line 38 of **Figure 13A**.

Exemplary detailed debugging information regarding the property violations is shown in **Figure 13B**.

In the foregoing specification, the invention has been described with reference to specific embodiments thereof. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the invention. The specification and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense.

CLAIMS

What is claimed is:

1. A method of detecting errors in a hardware design comprising:
receiving a representation of a hardware design and information regarding the intended flow of logical signals among a plurality of variables in the representation;
inferring the existence of potential errors in the hardware design based upon the information regarding the intended flow of logical signals by
translating the information regarding the intended flow of logical signals into a plurality of checks, each of the plurality of checks representing a condition that must hold true in order for the hardware design to operate in accordance with the intended flow of logical signals, and
determining if any of the plurality of checks can be violated during operation of circuitry represented by the hardware design.
2. The method of claim 1, further comprising identifying one or more key points in the hardware design based upon the information regarding the intended flow of logical signals.
3. The method as in claim 1 or 2, wherein the hardware design is expressed in a hardware description language (HDL).
4. The method of claim 3, wherein information regarding the intended flow of logical signals is captured by semantics of the HDL.
5. The method as in any of the preceding claims, further including determining the intended flow of logical signals through one or more of the plurality of variables by interpreting the information regarding the intended flow based upon its context.
6. The method of claim 5, wherein the context of the information regarding the intended flow comprises its position within the control flow structure.
7. The method of claim 3, wherein the HDL comprises VHDL.
8. The method of claim 3, wherein the HDL comprises Verilog.

9. The method as in any of the preceding claims, further including extracting the information regarding the intended flow from one or more inline annotations embedded within the representation of the hardware design.

10. The method of claim 9, wherein the one or more inline annotations have a predetermined format including a comment start marker, a directive marker, and a directive name.

11. The method of claim 10, wherein the directive marker comprises "vx".

12. The method as in any of the preceding claims, wherein the information regarding the intended flow of logical signals among one or more of the plurality of variables is provided by way of a file separate from that containing the hardware design.

13. The method as in any of the preceding claims, further comprising reporting the existence of one or more potential errors in the hardware design.

14. The method as in any of the preceding claims, wherein the information regarding the intended flow of logical signals among a plurality of variables in the representation includes information regarding a state associated with a variable of the plurality of variables.

15. The method of claim 14, wherein the plurality of checks include one or more rules relating to signal propagation, and wherein the method further comprises evaluating whether or not a signal associated with the variable is propagated in accordance with the one or more rules.

16. The method of claim 15, wherein one or more of the plurality of checks are violated if the signal is capable of causing data flow activity in the hardware design while the variable is in the inactive state.

17. The method of claim 15, wherein one or more of the plurality of checks are violated if the signal is incapable of causing data flow activity in the hardware design while the variable is in the active state.

18. The method as in any of the preceding claims, wherein the information regarding the intended flow of logical signals includes an indication of one or more conditions under which each of the plurality of variables are to be associated with each of a plurality of states, and wherein conclusions regarding each of the plurality of checks capable of being inferred based upon the states

associated with the plurality of variables during propagation of the logical signals.

19. The method of claim 18, wherein the plurality of checks includes a first check relating to access of inactive data by any of the plurality of variables in the representation.

20. The method as in claim 18 or 19, wherein the plurality of checks includes a second check relating to loss of active data from any of the plurality of variables in the representation.

21. The method as in any of the preceding claims, wherein said determining if any of the plurality of checks can be violated during operation of circuitry represented by the hardware design employs formal techniques.

22. The method as in any of the preceding claims, wherein said determining if any of the plurality of checks can be violated during operation of circuitry represented by the hardware design employs a simulation-based approach.

23. The method as in claims 3-8, further comprising determining a failure model by automatically generating the plurality of checks, including one or more behavioral integrity checks and one or more temporal integrity checks, based upon the HDL and the information regarding the intended flow of logical signals among the plurality of variables in the representation, each of the plurality of checks symptomatic of one or more errors in the hardware design.

24. The method as in any of the preceding claims, wherein the plurality of variables represent interconnects in the hardware design through which the logical signals pass, the method further comprising explicitly associating state information with one or more variables of the plurality of variables independent of the values of the one or more variables and in accordance with the intended flow of the logical signals.

25. The method of claim 24, wherein the representation contains inline annotations, wherein the state information comprises an active state and an inactive state, and wherein the method further includes:

causing a variable of the one or more variables to be associated with the active state in response to a first directive included within the inline annotations; and

causing the variable to be associated with the inactive state in response to a second directive included within the inline annotations.

26. The method of claim 25, wherein the active state represents a state in which those of the logical signals passing through the associated variable are intended to reach one or more other variables of the plurality of variables while at least one of the one or more other variables is in the active state, the inactive state representing a state in which those of the logical signals passing through the associated variable are not intended to reach other variables of the plurality of variables while the other variables are in the active state.

27. The method as in any of the preceding claims, further including evaluating whether inactive data is accessed by determining if a value received by a first variable of the plurality of variables is governed by a second variable of the plurality of variables while the second variable is in an inactive state.

28. The method as in any of the preceding claims, further including evaluating whether active data is lost by determining if a value on one or more of the logical signals governed by a first variable of the plurality of variables while the first variable is in an active state is not received by at least one other variable of the plurality of variables while the at least one other variable is in an active state.

29. The method as in any of the preceding claims, further comprising linking two or more checks of the plurality of checks by determining dependency relationships among the two or more checks.

30. The method of claim 29, further comprising:
for each check of the plurality of checks
disabling those of the other plurality of checks that are dependent on the check,
evaluating the check, and
enabling all of the plurality of checks.

31. The method of claim 30, wherein said evaluating the check includes:
expanding a current partial solution to create a new partial solution;
evaluating the new partial solution;

determining whether or not the new partial solution will violate another independent check, and

if it is determined that the new partial solution will violate the other independent check and if the new partial solution cannot be rectified to remove the violation, then treating the check as dependent upon the other independent check.

32. The method as in claims 29-31, further comprising using the dependency relationships to facilitate error reporting.

33. The method of claim 31, wherein the method is performed in the context of simulation of a circuit, and the method further includes rectifying the new partial solution to remove the violation by abandoning the current input from the sequence of inputs and restoring the state of the circuit to an earlier state.

34. The method of claim 31, wherein the method is performed in the context of formal analysis, and the method further includes rectifying the new partial solution to remove the violation by abandoning the current search path for the solution and restoring the search to build a different solution.

35. The method as in claims 29-34, further comprising containing the reporting of redundant failures by suppressing reporting of multiple property violations associated with the plurality of checks that are due to a common design defect of the hardware design.

36. An apparatus for detecting errors in a hardware design, the apparatus comprising:

a storage device having stored therein one or more design analysis routines for verifying the design integrity of a hardware design described in a hardware description language (HDL), conforming to a particular methodology, and employing a particular form of annotations to the HDL; and

a processor coupled to the storage device for executing the one or more design analysis routines to receive the hardware design, receive one or more indications of design intent, produce a failure model, and detect the existence of potential errors in the hardware design, where:

the one or more indications of design intent are received by extracting the one or more indications of design intent from the annotations to the HDL,

the failure model is produced by generating a plurality of checks based upon the HDL and the design intent, each of the plurality of checks being symptomatic of one or more errors in the hardware design, and

the existence of the potential errors in the hardware design, if any, are detected by analyzing each check of the plurality of checks.

37. The apparatus of claim 36, wherein the one or more design analysis routines, when executed, further identify one or more key points in the hardware design based upon the design intent.

38. The apparatus as in claims 36 or 37, wherein the one or more indications of design intent comprise information regarding a state associated with a variable in the HDL.

39. The apparatus as in claims 36-38, wherein the plurality of checks include one or more rules relating to signal propagation, and wherein said detecting the existence of potential errors in the design, if any, by analyzing each check of the plurality of checks includes evaluating whether or not a signal associated with the variable is propagated in accordance with the one or more rules.

40. The apparatus of claim 39, wherein one or more of the plurality of checks are violated if the signal causes data flow activity in the hardware design while the state of the variable is inactive.

41. The apparatus as in claims 39 or 40, wherein one or more of the plurality of checks are violated if the signal does not cause data flow activity in the hardware design while the state of the variable is active.

42. The apparatus as in claims 36-41, wherein the HDL comprises VHDL.

43. The apparatus as in claims 36-41, wherein the HDL comprises Verilog.

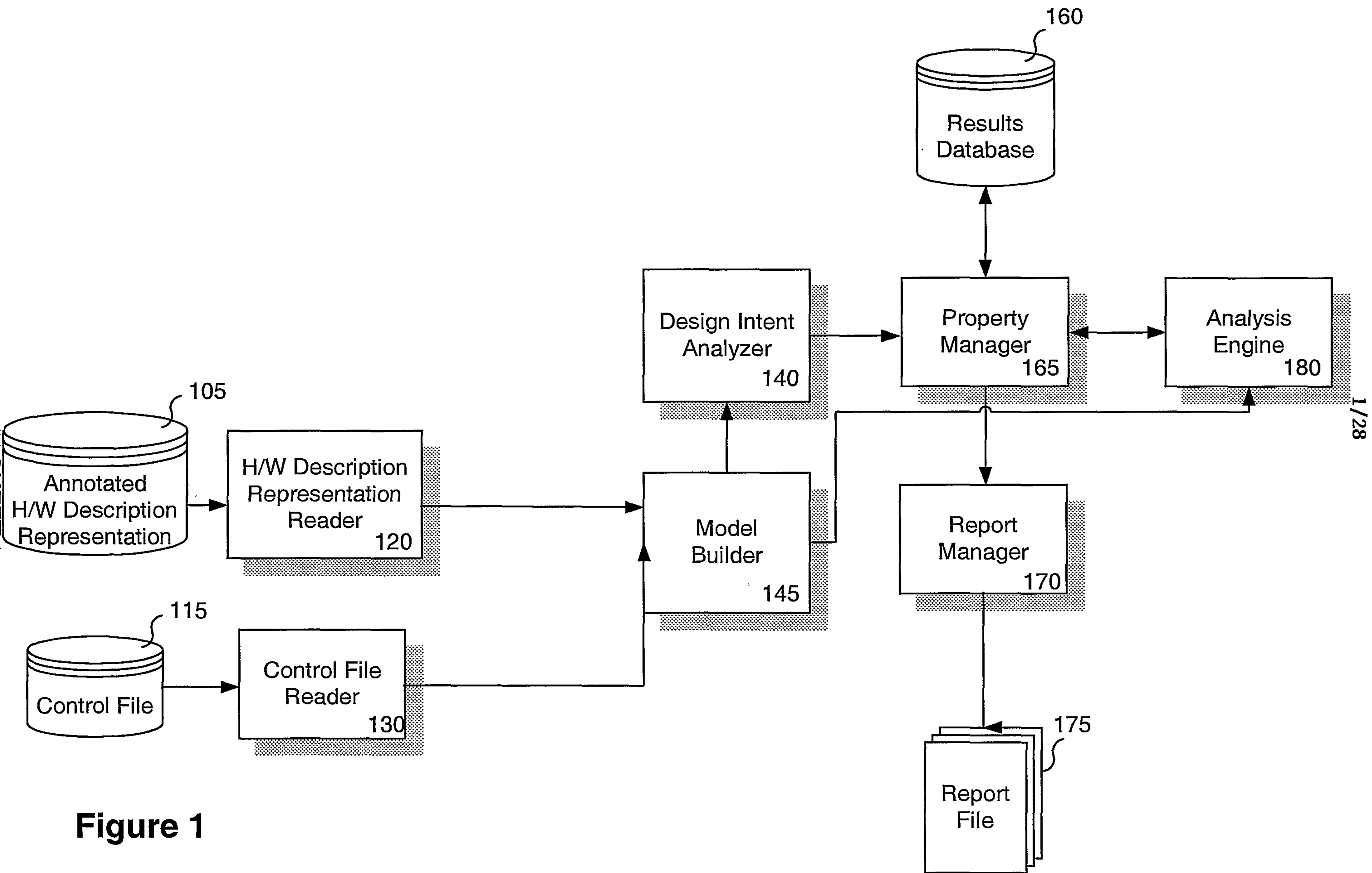

Figure 1

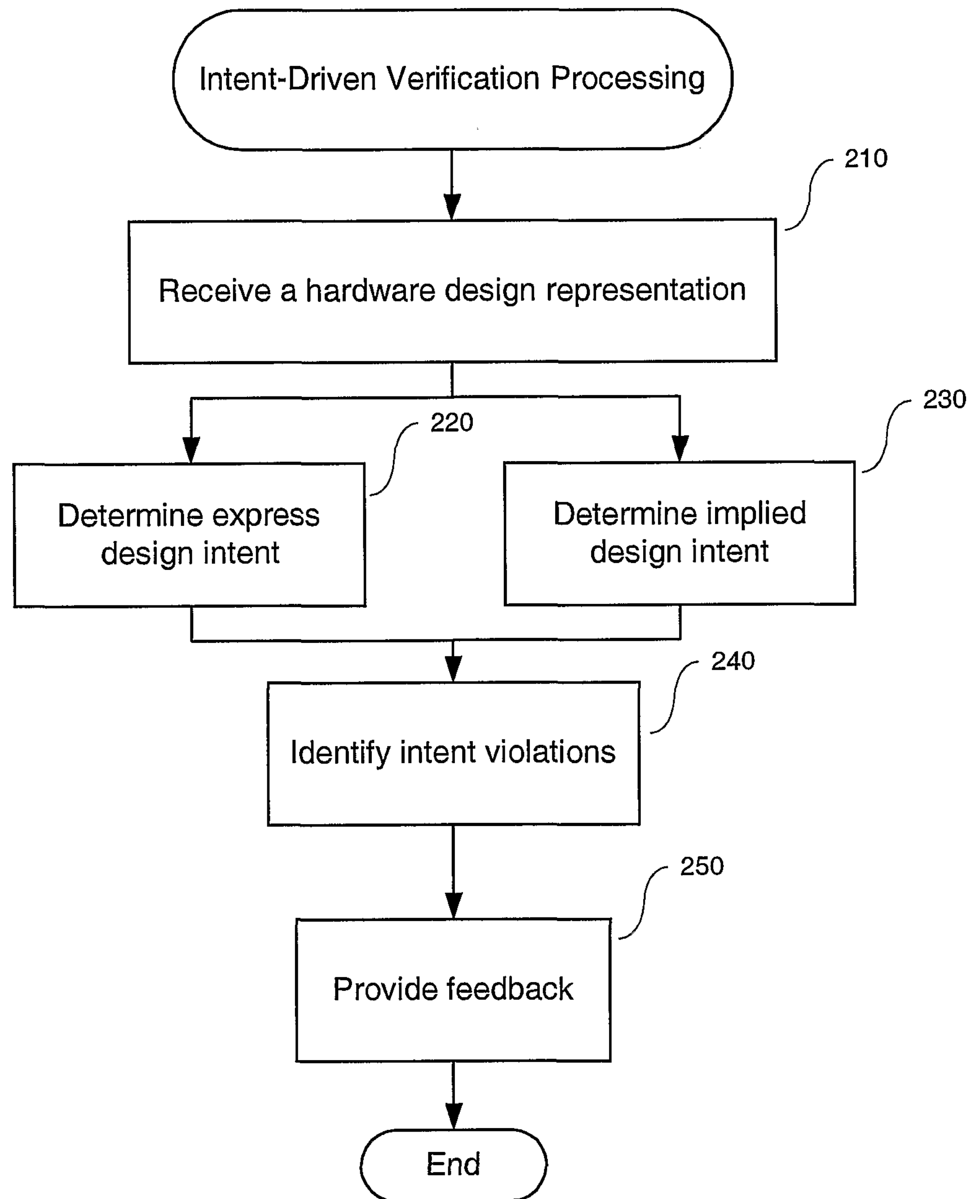

Figure 2

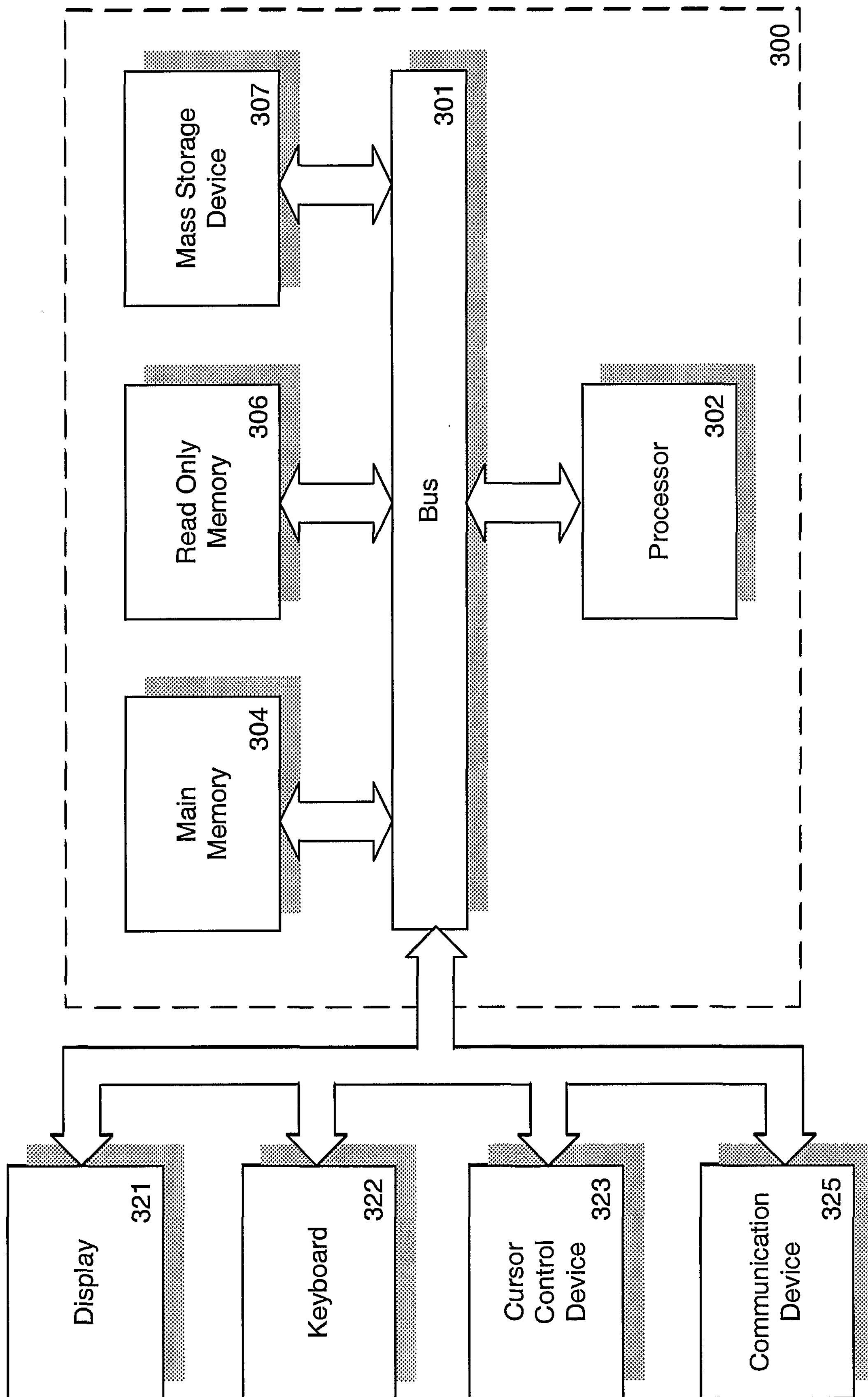

Figure 3

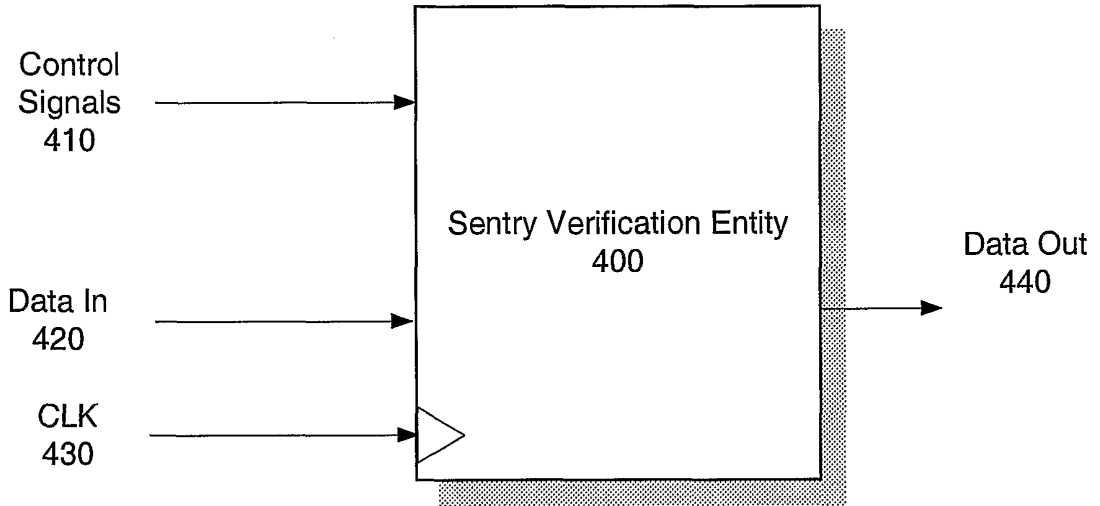

Figure 4A

Sentry Verification Entity 400

State
Latch
445

State 435

CLK
430

415

425

Deactivate
411

Activate
412

State Maintenance Logic 455

Data In
420

Channel

Data Out
440

Figure 4B

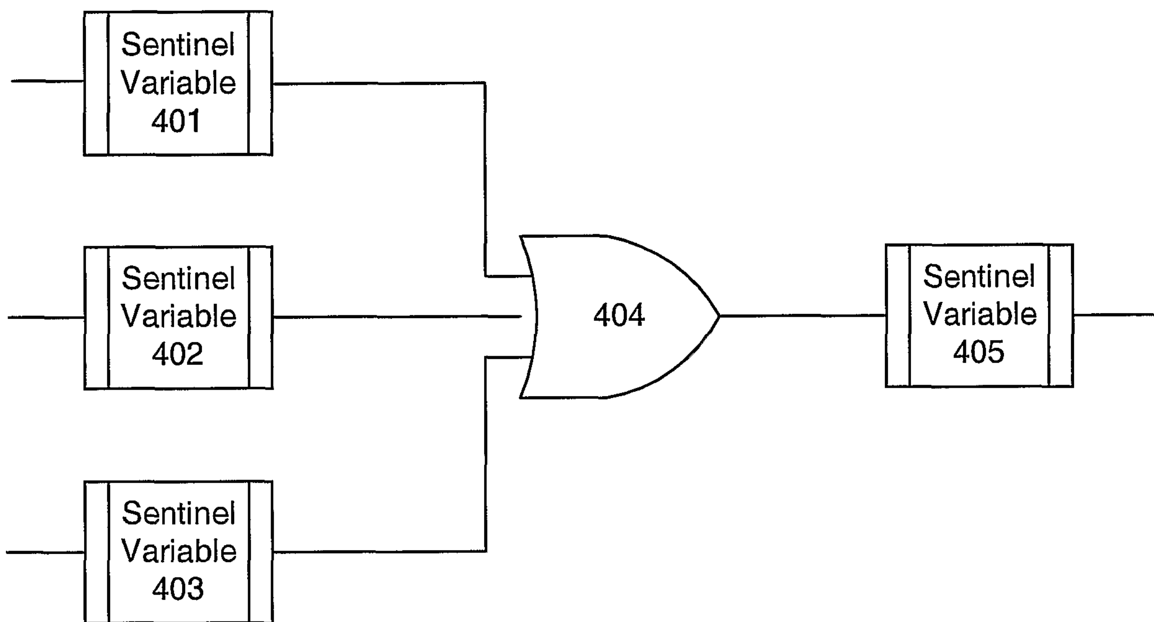

Figure 4C

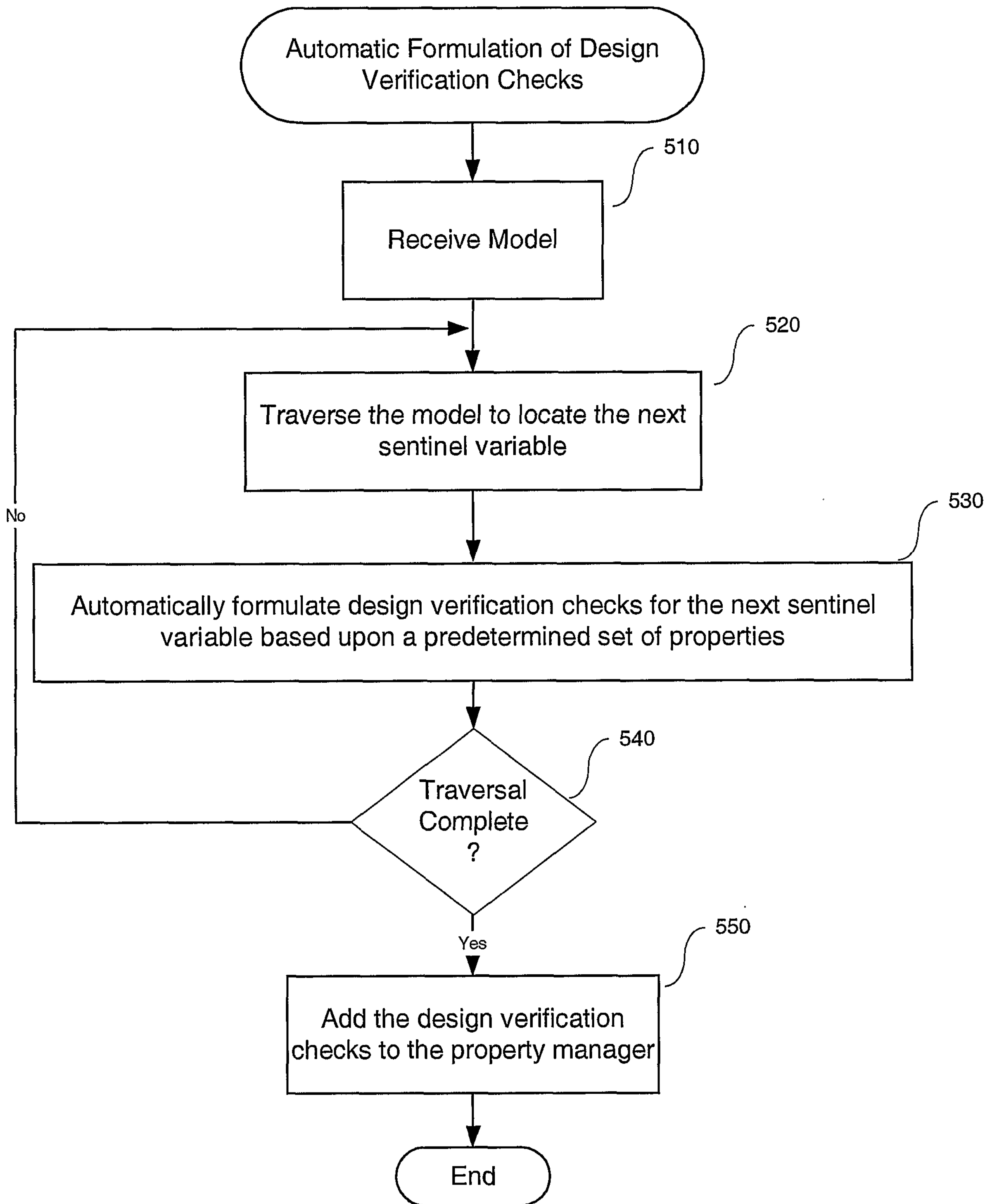

Figure 5

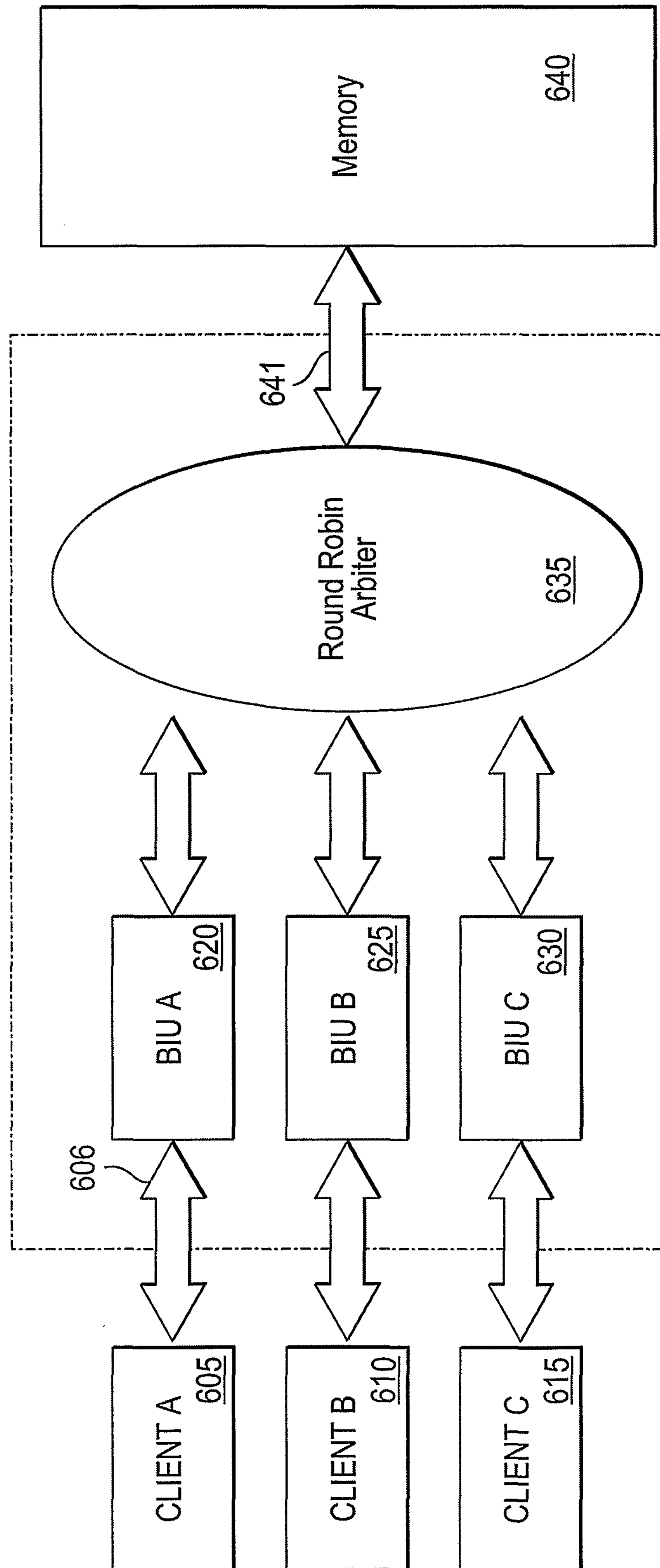

Fig. 6A

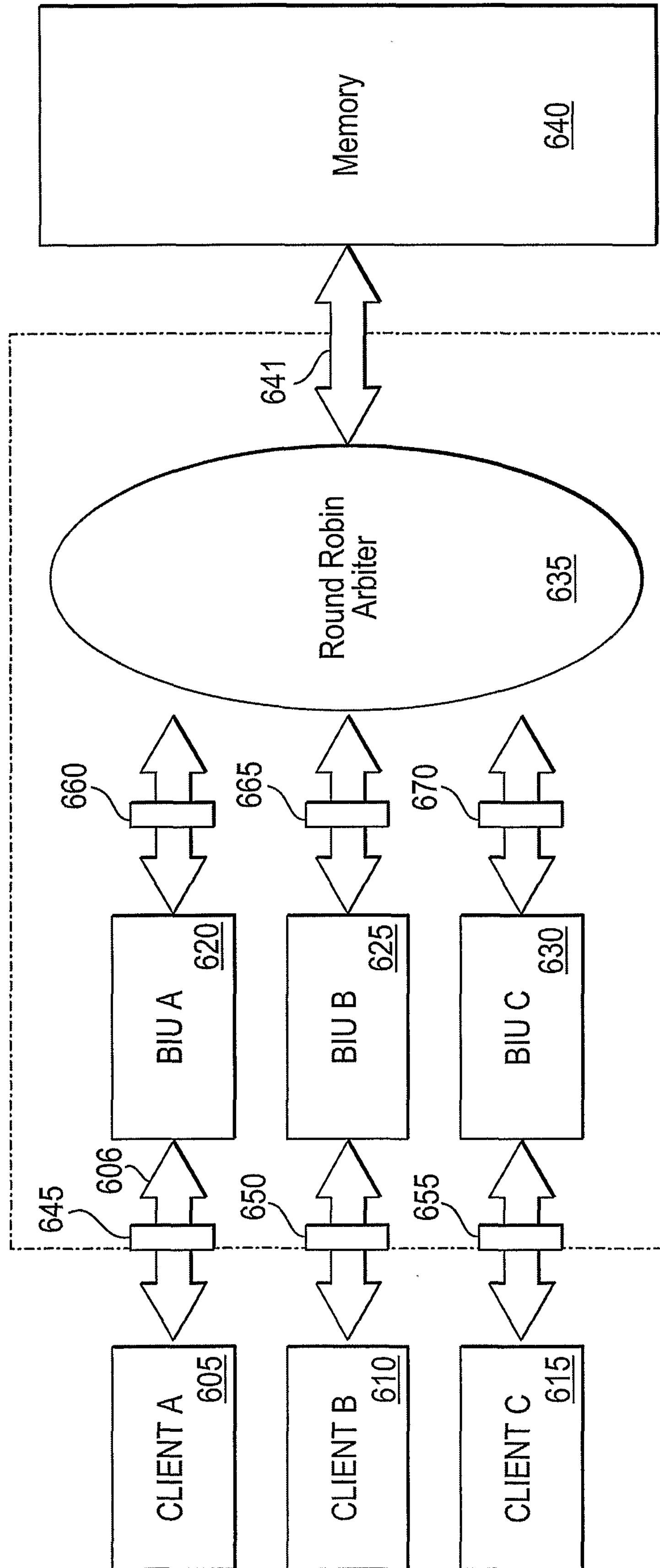

Fig. 6B

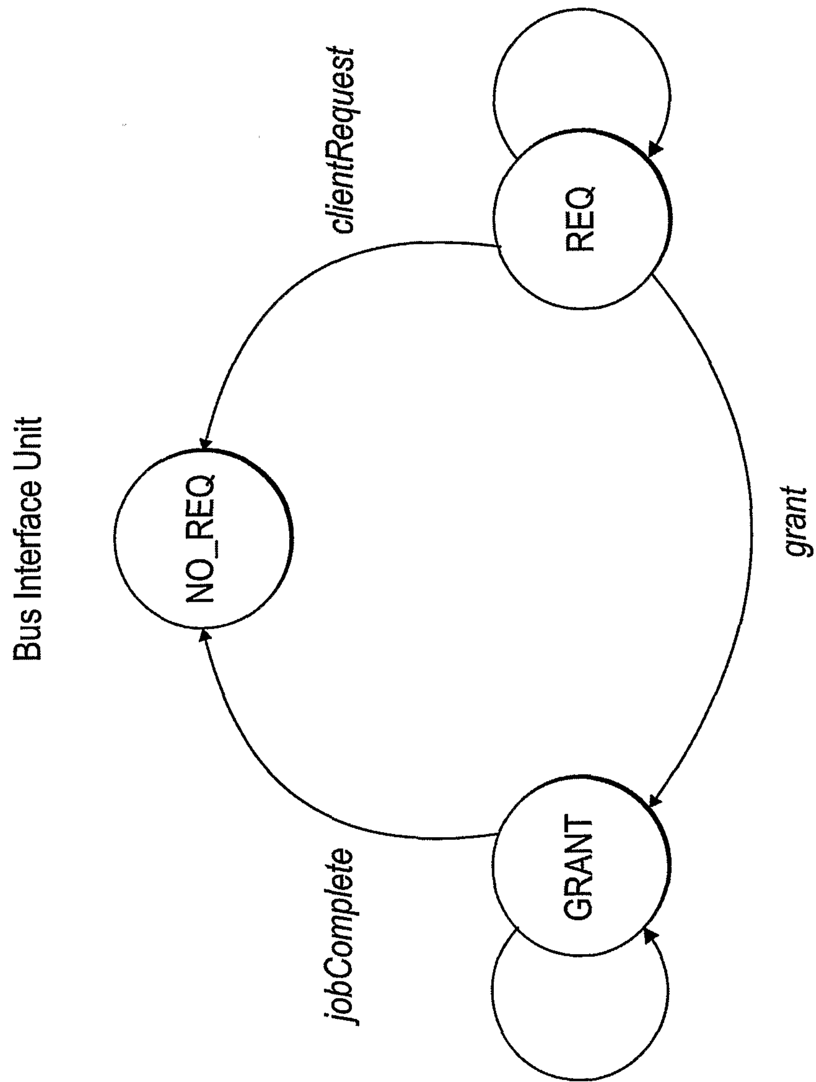

Fig. 7

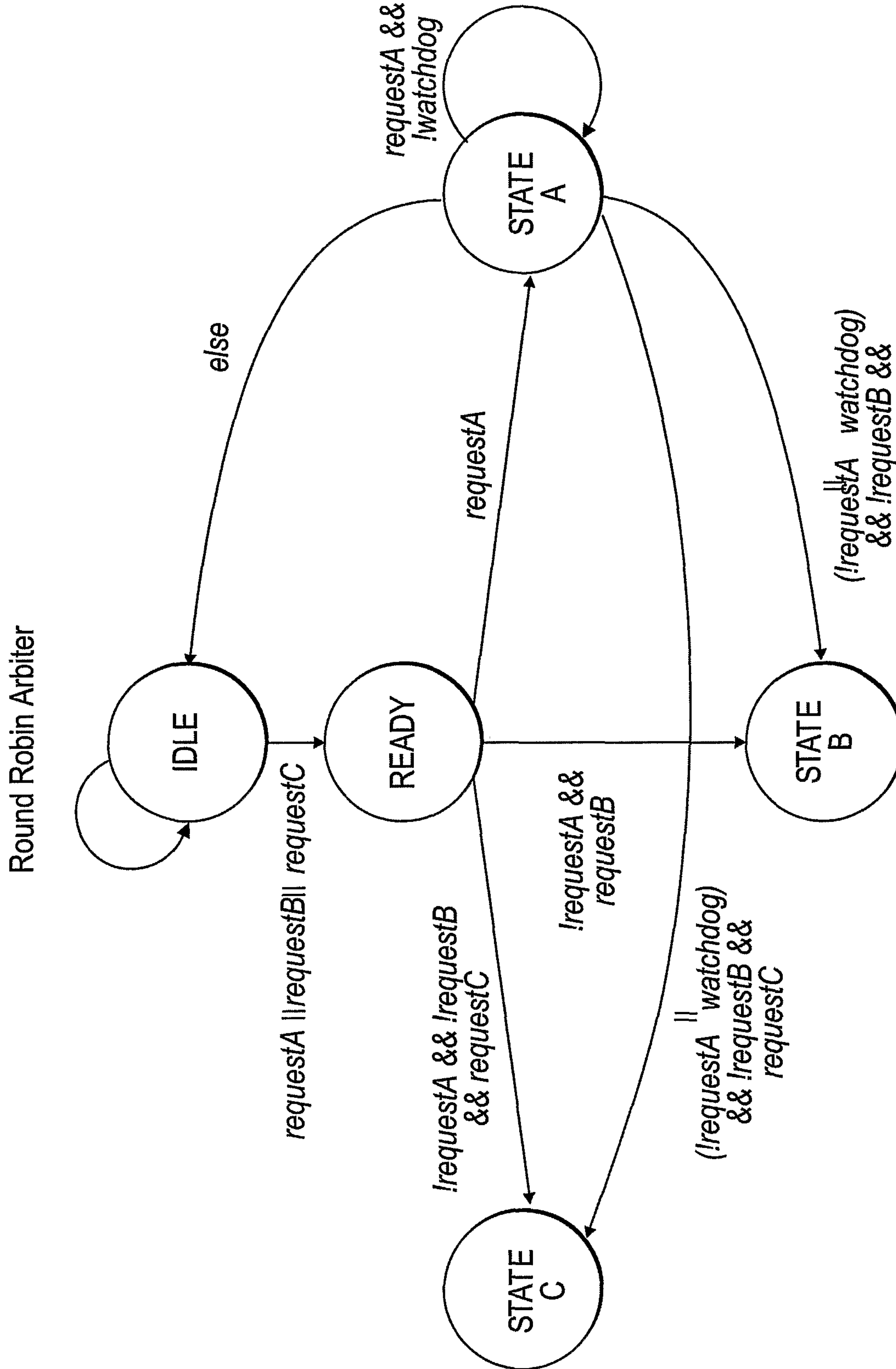

Fig. 8

```
1.    `define DEFAULT_MAX_ACCESS_TIME 1'b1
2.
3.    module main(clk, reset,
4.            dataA, clientRequestA, jobCompleteA, clientGrantA,
5.            dataB, clientRequestB, jobCompleteB, clientGrantB,
6.            dataC, clientRequestC, jobCompleteC, clientGrantC,
7.            memoryData);
8.
9.            input           clk, reset;
10.           input [0:0]     dataA, dataB, dataC;
11.           input           clientRequestA, clientRequestB, clientRequestC;
12.           input           jobCompleteA, jobCompleteB, jobCompleteC;
13.           output          clientGrantA, clientGrantB, clientGrantC;
14.           output [0:0]    memoryData;
15.
16.           wire [0:0]      dataOutA, dataOutB, dataOutC;
17.
18.           // Put sentries at the boundary signals and make them active constantly.
19.
20.           // vx sentry dataA, dataB, dataC:clk;
21.           // vx always activate(dataA,dataB,dataC);
22.
23.           // Create three instances of the bus interface
24.
25.           BusInterface busInterfaceA(clk, reset, clientRequestA, jobCompleteA,
26.                   dataA, dataOutA, requestA, grantA,
27.                   clientGrantA);
28.
29.           BusInterface busInterfaceB(clk, reset, clientRequestB, jobCompleteB,
30.                   dataB, dataOutB, requestB, grantB,
31.                   clientGrantB);
32.
33.           BusInterface busInterfaceC(clk, reset, clientRequestC, jobCompleteC,
34.                   dataC, dataOutC, requestC, grantC,
35.                   clientGrantC);
36.
37.           RoundRobinArbiter arbiter(clk, reset,
38.                   dataOutA, requestA, grantA,
39.                   dataOutB, requestB, grantB,
40.                   dataOutC, requestC, grantC,
41.                   memoryData);
42.
43.   endmodule // main
```

Figure 9A

```
44. module Businterface(clk, reset,clientRequest, jobComplete,
45.         data, dataOut, request, grant, clientGrant);
46.
47.         input           clk, reset, clientRequest, job complete
48.         input [0:0]     data;
49.         output [0:]     dataOut
50.         // vx sentry dataOut:clk;
51.
52.         output          request;
53.         input           grant;
54.         output          clientGrant;
55.
56.         parameter       No_REQ = 2'b00;
57.         parameter       REQ = 2'b01;
58.         parameter       GRANTED = 2'b10
59.
60.         req [1:0]       state, nextState;
61.         // vx flop state;
62.
63.         //assign request = ((state == REQ)  (state == GRANTED));
64.         assign request = ((state == REQ !((state == GRANTED) && !jobComplete));
65.         assign dataOut = data;
66.         assignclientGrant = grant
67.
68.         always @(state or reset or clientRequest or jobComplete or grant)
69.         begin
70.                 if(reset)
71.                 begin
72.                         nextState = NO_REQ;
73.                         // vx deactivate(dataOut);
74.                 end
75.                 else
76.                 begin
77.                         nextState = state;
78.                         case (state)
79.                                 NO_REQ;
80.                                 begin
81.                                         // vx deactivate(dataOut);
82.                                         if (clientRequest0 nextState = REQ;
83.                                 end
84.                                 REQ:
85.                                 begin
86.                                         // vx assert("env1", clientRequest && ljobComplete):
87.                                         if (grant) nextState = GRANTED;
88.                                 end
89.                                 GRANTED:
90.                                 begin
91.                                         // vx activate(dataOut);
```

Fig. 9B

```
92.                                     // vx assert("env2", !clientRequest);
93.                                       if (jobComplete|| !grant)
94.                                           nextState = NO_REQ;
95.                                 end
96.                            endcase // case(state)
97.                       end // else
99.                  end // always
100.                 always @ (posedge clk)
101.                 state <+ nextState;
102.  endmodule // BusInterface
```

Fig. 9B
(Cont.)

```
103.  module RoundRobinArbiter(clk, reset,
104.          dataA, requestA, grantA,
105.          dataB, requestB, grantB,
106.          dataC, requestC, grantC,
107.          writeData);
108.
109.          input           clk, reset;
110.          input [0:0]     dataA, dataB, dataC;
111.          input           requestA, requestB, requestC;
112.          output          grantA, grantB, grantC;
113.          output [0:0]    writeData;
114.
115.          // vx sentry dataA, dataB, dataC: clk;
116.
117.          reg [0:0]       watchDogTimer, nextWatchDogTimer;
118.          reg [2:0]       state, nextState;
119.          // vx flop state, watchDogTimer;
120.          wire [0:0]      maxAccessTime;
121.
122.          parameter       idle = 3'b000;
123.          parameter       ready = 3'b001;
124.          parameter       stateA = 3'b010;
125.          parameter       stateB = 3'b011;
126.          parameter       stateC = 3'b100;
127.
128.          // next state logic
129.          always @(reset or requestA or requestB or requestC or state or watchDogTimer)
130.          begin
131.                  // vx deactivate(dataA,dataB,dataC);
132.                  if (reset) begin
133.                          nextState= idle;
134.                          nextWatchDogTimer = 1'b0;
135.                  end
136.                  else begin
137.                          nextState= idle; // default state
138.                          // by default timer should increment
139.                          nextWatchDogTimer = watchDogTimer+1'b1;
```

Figure 9C

```
140.
141.    case (state)
142.        idle:
143.        begin
144.            nextWatchDogTimer = 1'b0;
145.            if (requestA||requestsB||requesteC) nextState = ready;
146.        end
147.        ready:
148.        begin
149.            if (requestA) nextState = stateA;
150.            else if (requestB) nexState = stateB;
151.            else if (requestC) nextState = stateC;
152.            else nextState + idle;
153.        end
154.        stateA:
155.        begin
156.            // vx activate(dataA);
157.            nextState= statA;
158.            // if request has been disabled or the max access time has
159.            // reached change state.
160.            if ((requestA == 1'b0)|| (watchDog Timer == maxAccess Time)) begin

161.                nextWatchDogTimer = 1'b0;
162.                if (requestB) nextState = stateB;
163.                else if (requestC) nextState = stateC;
164.                else nextState = idle;
165.            end
166.        end
167.        stateB
168.        begin
169.            // vx activate(dataB);
170.            nextState= stateB;
171.            if ((requestB == 1'b0)|| (watchDogTimer == maxAccessTime)) begin
```

Fig. 9D

```
172.                    nextWatchDogTimer = 1'b0;
173.                    if ((requestB == 1'b0)|| (watchDogTimer == maxAccessTime)) begin
174.                    else if (requestA) nextState= stateA;
175.                    else nextState= idle;
176.                end
177.            end
178.            stateC:
179.            begin
180.                // vx activate(dataC);
181.                nextState= stateC;
182.                if (requestC == 1'b0)|| (watchDogTimer == maxAccessTime))begin
183.                    nextWatchDogTimer = 1'b0;
184.                    if (requestA) nextState= stateA;
185.                    else if (requestB) nextState= stateB
186.                    else nextState== idle;
187.                end
188.            end
189.        endcase // case(state)
190.      end
191.   end // always
```

Fig. 9D
(Cont.)

```
192.        // state transition
193.        always @(posedge clk)
194.        begin
195.            state <= nextState;
196.            watchDogTimer <= nextWatchDogTimer;
197.        end
198.        // outputs
199.        assign grantA = (nextState == stateA);
200.        assign grantB = (nextState == stateB);
201.        assign grantC = (nextState == stateC);
202.        assign writeData = (grantA ? dataA : `DATA_WIDTH'bz);
203.        assign writeData = (grantB ? dataB : `DATA_WIDTH'bz);
204.        assign writeData = (grantC ? dataC : `DATA_WIDTH'bz);
205.        assign maxAccessTime = `DEFAULT_MAX_ACCESS_TIME;
206.        // vx always onetrue("grant", grantA, grantB, grantC);
207.    endmodule // RoundRobinArbiter
```

Figure 9E

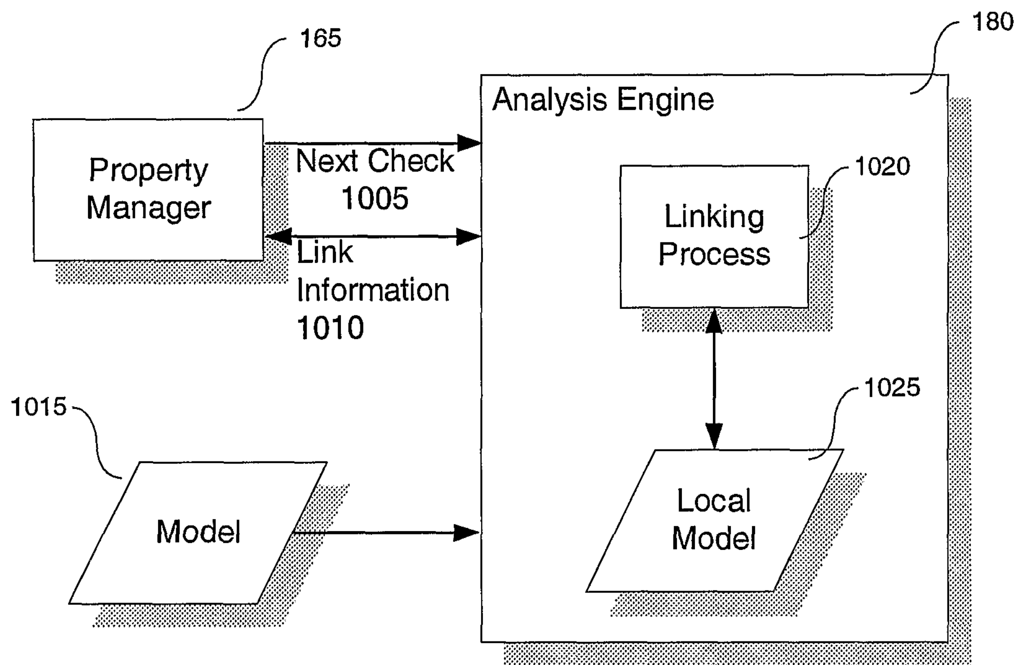

Figure 10

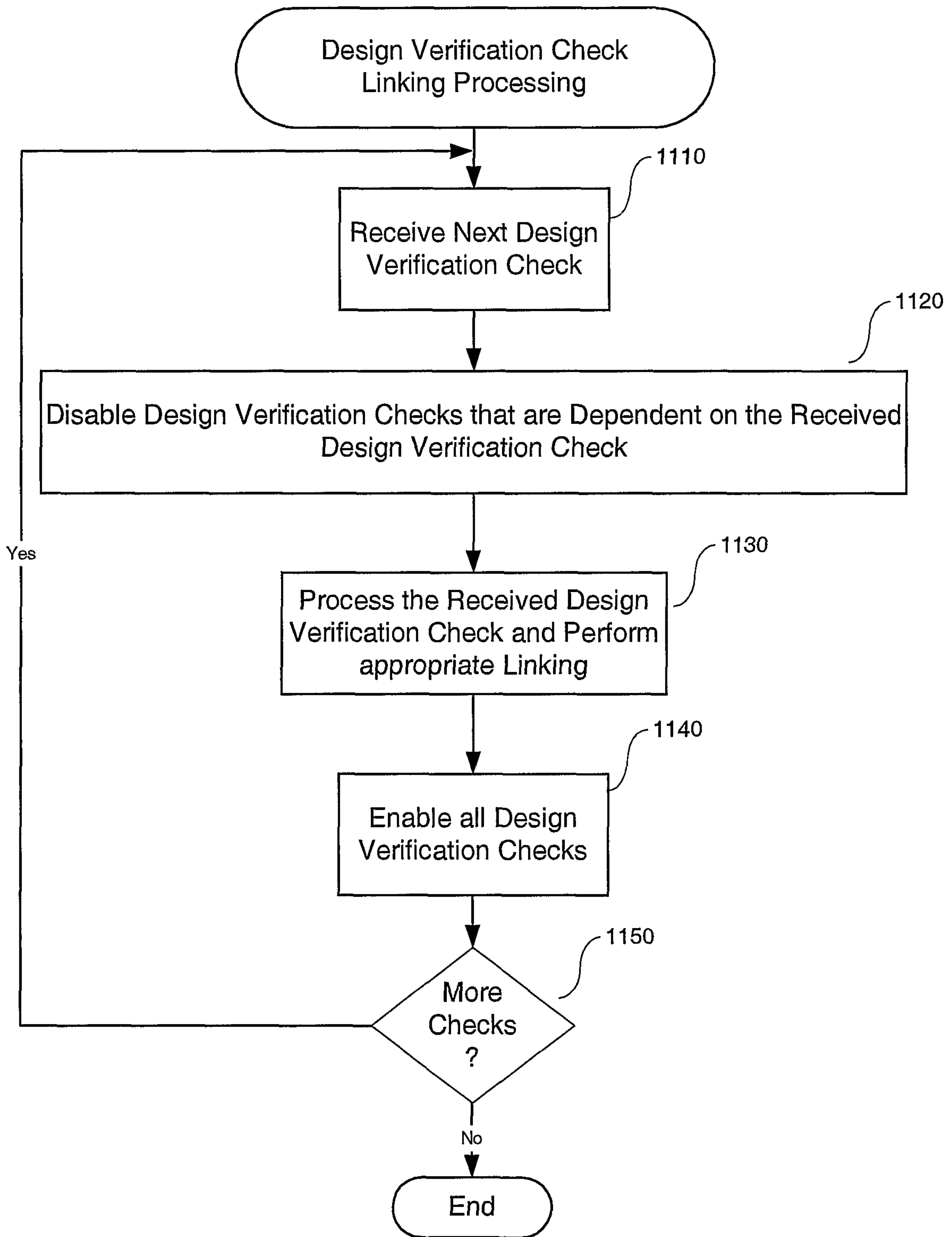

Figure 11

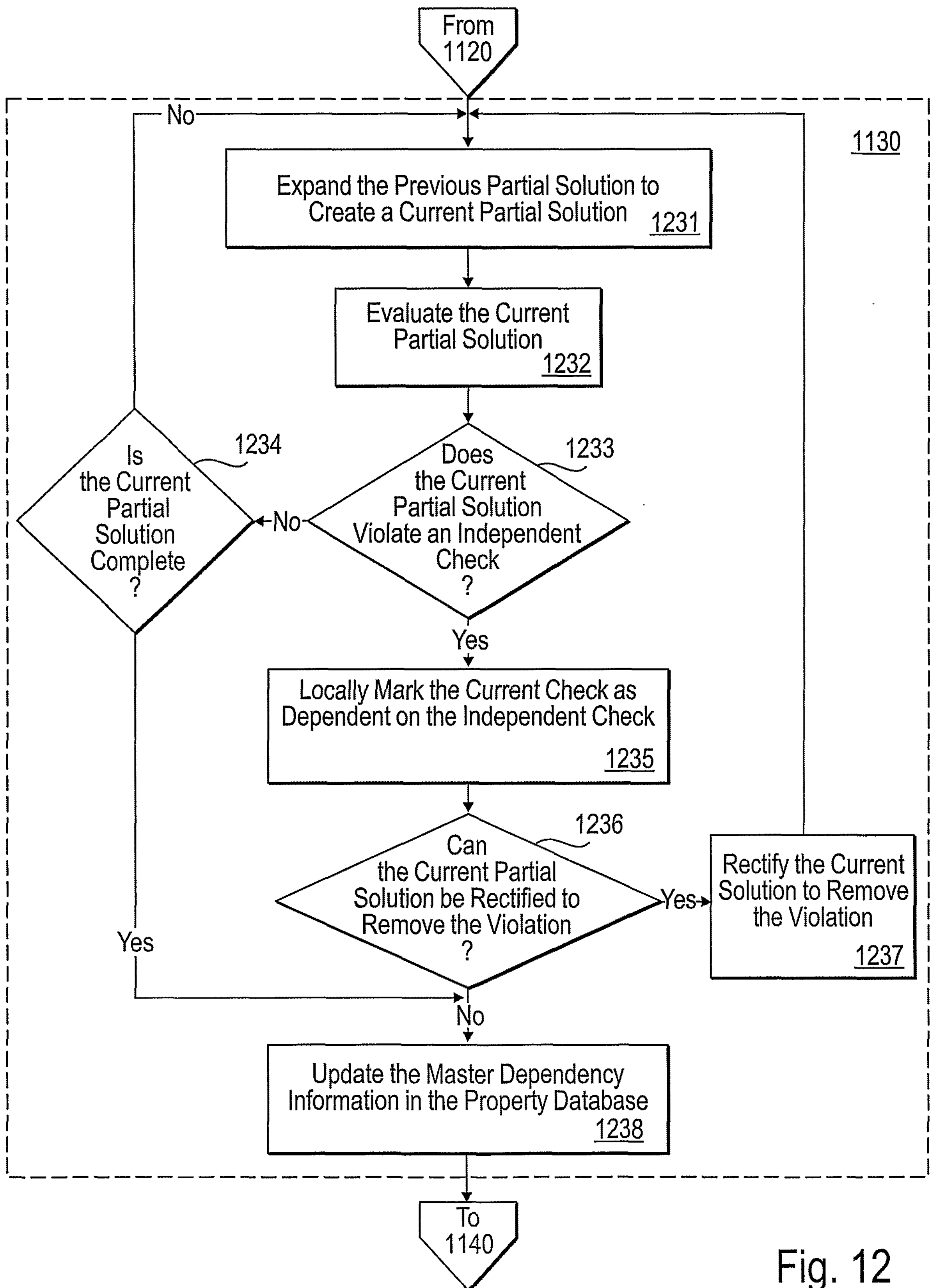

Fig. 12

```
1.   ===================================
2.   Validation Results for Module "main"
3.   ===================================
4.
5.   ========================
6.   Functional Checks Summary
7.   ========================
8.   Failed checks (FAILED)                                    = 5
9.   Bounded failed checks (BOUNDED_FAIL)                      = 0
10.  Inconclusive checks (INCONCLUSIVE)                        = 3
11.  Secondary failed checks (SECONDARY)                       = 0
12.  Interface checks (INTERFACE)                              = 12
13.  Skipped checks (UNPROCESSED/DISABLED/DEFERRED)            = 0
14.  Bounded passed checks (BOUNDED_PASS)                      = 0
15.  Passed checks (PASSED/USER PASSED/CONDITIONAL)            = 144
16.  ----------------------------------------------------
17.  Total checks                                              = 164
18.
19.  ==========================
20.  RTL Characteristics Summary
21.  ==========================
22.  Number of inferred flops                = 10
23.  Number of inferred latches        = 0
24.  Number of static X sources        = 0
25.  Number of non-resettable flops    = 0
26.
27.  =================================
28.  Summary of failed checks by type:
29.  =================================
30.  [CA]  = 0   (out of 39)
31.  [BE]  = 1   (out of 40)
32.  [AX]  = 1   (out of 15)
33.  [CME] = 0   (out of 10)
34.  [CV]  = 0   (out of 29)
35.  [AC]  = 0   (out of 7)
36.  [AAO] = 0   (out of 6)
37.  [AID] = 0   (out of 9)
38.  [LVD] = 3   (out of 9)
```

Figure 13A

```
39.
40.       Block enable [BE] : failed check
41.       ---------------------------------------
42.       FAILED      BE: arbiter.nextState_bit0_AX_6
43.                     : Enable condition for block with assignment "arbiter.nextState_bit0_AX_6" is always
off
44.                     : Block with assignment "arbiter.nextState_bit0_AX6" is defined on line 191 of "rr3.v"
45.
46.       Assignment execution [AX] : 1 failed check
47.       -----------------------------------------------------
48.       FAILED     AX: arbiter.nextWatchDogTimer_bit0_AX_1
49.                   : Assignment "arbiter.nextWatchDogTimer_bit0_AX1" only has a constant 1"b1 value
50.                   : Assignment "arbiter.nextWatchDogTimer_bit0_AX_1 is defined on line 179 of"rr3.v"
51.
52.       Loss of valid data [LVD] :3 failed checks
53.       -------------------------------------------------
54.       FAILED    LVD :dataA[0]
55.                  : Loss of valid data has been detected on wire "dataA[0]"
56.                  : Wire "dataA[0]" is defined on line 40 of "rr3.v"
57.                  : VCD file is "roundRobin3/main/trace221.vcd"
58.       FAILED    LVD :dataB[0]
59.                  : Loss of valid data has been detected on wire "dataB[0]"
60.                  : Wire "dataB[0]" is defined on line 41 of "rr3.v"
61.                  : VCD file is "roundRobin3/main/trace223.vcd"
62.       FAILED    LVD :dataC[0]
63.                  : Loss of valid data has been detected on wire "dataC[0]"
64.                  : Wire "dataC[0]" is defined on line 42 of "rr3.v"
65.                  : VCD file is "roundRobin3/main/trace225.vcd"
```

Fig. 13B

```
66.  ============================================
67.  Summary of bounded failed checks by type:
68.  ============================================
69.  [CA] = 0      (out of 39)
70.  [BE] = 0      (out of 40)
71.  [AX] = 0      (out of 15)
72.  [CME] = 0     (out of 10)
73.  [CV] = 0      (out of 29)
74.  [AC ] = 0     (out of 7)
75.  [AAO] = 0     (out of 6)
76.  [AID] = 0     (out of 9)
77.  [LVD] = 0     (out of 9 )
78.
79.  ============================================
80.  Summary of inconclusive checks by type:
81.  ============================================
82.  [CA] = 0      (out of 39)
83.  [BE] = 0      (out of 40)
84.  [AX] = 0      (out of 15)
85.  [CME] = 0     (out of 10)
86.  [CV] = 0      (out of 29)
87.  [AC] = 0      (out of 7)
88.  [AAO] = 0     (out of 6)
89.  [AID] = 0     (out of 9)
90.  [LVD] = 0     (out of 9 )
91.
```

Fig. 13C

```
92.   Loss of valid data [LVD] :  inconclusive checks
93.   ---------------------------------------------
94.   INCONCLUSIVE LVD   :busInterfaceA.dataOut[0]
95.                      : Valid data loss check on wire 'busInterfaceA.dataOut[0]"did not complete
96.                      : Wire "busInterfaceA.dataOut[0]" is defined on line 80 of "rr3.v"
97.   INCONCLUSIVE LVD   :busInterfaceA.dataOut[0]
98.                      : Valid data loss check on wire 'busInterfaceA.dataOut[0]"did not complete
99.                      : Wire "busInterfaceA.dataOut[0]" is defined on line 80 of "rr3.v"
100.  INCONCLUSIVE LVD   :busInterfaceA.dataOut[0]
101.                     : Valid data loss check on wire 'busInterfaceA.dataOut[0]"did not complete
102.                     : Wire "busInterfaceA.dataOut[0]" is defined on line 80 of "rr3.v"
103.
104.  ==========================================
105.  Summary of bounded failed checks by type:
106.  ==========================================
107.  [CA] = 0     (out of 39)
108.  [BE] = 0     (out of 40)
109.  [AX] = 0     (out of 15)
110.  [CME] = 0    (out of 10)
111.  [CV] = 0     (out of 29)
112.  [AC ] = 0    (out of 7)
113.  [AAO] = 0    (out of 6)
114.  [AID] = 0    (out of 9)
115.  [LVD] = 0    (out of 9 )
```

Fig. 13C
(Cont.)

```
116.    ==========================================
117.    Summary of bounded failed checks by type:
118.    ==========================================
119.    [CA] = 0      (out of 39)
120.    [BE] = 0      (out of 40)
121.    [AX] = 0      (out of 15)
122.    [CME] = 0     (out of 10)
123.    [CV] = 0      (out of 29)
124.    [AC ] = 6     (out of 7)
125.    [AAO] = 0     (out of 6)
126.    [AID] = 3     (out of 9)
127.    [LVD] = 3     (out of 9 )
128.
129.    Assertion correctness [AC] : 6 interface checks
130.    -----------------------------------------------
131.    INTERFACE     AC     :busInterfaceA.env1
132.                         : Correctness check on assertion 'busInterfaceA.env1" is at the interface
133.                         : Assertion "busInterfaceA.env1 is defined on line 121 of "rr3.v"
134.    INTERFACE     AC     :busInterfaceA.env1
135.                         : Correctness check on assertion 'busInterfaceA.env2" is at the interface
136.                         : Assertion "busInterfaceA.env2 is defined on line 127 of "rr3.v"
137.                         : Check is used by conditional checks:
138.    AID                  :arbiter.dataA[0]
139.    INTERFACE     AC     :busInterfaceA.env1
140.                         : Correctness check on assertion 'busInterfaceB.env1" is at the interface
141.                         : Assertion "busInterfaceB.env1 is defined on line 121 of "rr3.v"
142.    INTERFACE     AC     :busInterfaceB.env2
143.                         : Correctness check on assertion 'busInterfaceB.env2" is at the interface
144.                         : Assertion "busInterfaceB.env2 is defined on line 127 of "rr3.v"
145.                         :Check is used by conditional checks:
146.    AID                  :arbiter.dataB[0]
147.    INTERFACE     AC     :busInterfaceC.env1
148.                         : Correctness check on assertion 'busInterfaceC.env1" is at the interface
149.                         : Assertion "busInterfaceC.env1 is defined on line 121 of "rr3.v"
```

Fig. 13D

```
150.  INTERFACE   AC    :busInterfaceC.env2
151.                    : Correctness check on assertion 'busInterfaceC.env2" is at the interface
152.                    : Assertion "busInterfaceA.env1 is defined on line 121 of "rr3.v"
153.                    : Check is used by conditional checks :
154.  AID               :arbiter.dataA[0]
155.
156.  Access of invalid data [AID] : 3 interface checks
157.  ------------------------------------------------------------
158.  INTERFACE   AID   :dataA[0]
159.                    : Wire "dataA[0]" has accessed invalid data from the interface
160.                    : Wire "dataA[0]" is defined on line 40 of "rr3.v"
161.  INTERFACE   AID   :dataB[0]
162.                    : Wire "dataB[0]" has accessed invalid data from the interface
163.                    : Wire "dataB[0]" is defined on line 41 of "rr3.v"
164.  INTERFACE   AID   :dataC[0]
165.                    : Wire "dataC[0]" has accessed invalid data from the interface
166.                    : Wire "dataC[0]" is defined on line 42 of "rr3.v"
```

Fig. 13D (Cont.)

```
167.
168.  Loss of valid data [LVD] : 3 interface checks
169.  -------------------------------------------------------
170.  INTERFACE   LVD   :arbiter.dataA[0]
171.                      : Wire "arbiter.dataA[0]" has loss of valid data at the interface
172.                      : Wire "arbiter.dataA[0]" is defined on line 141 of "rr3.v"
173.  INTERFACE   LVD   :arbiter.dataB[0]
174.                      : Wire "arbiter.dataB[0]" has loss of valid data at the interface
175.                      : Wire "arbiter.dataB[0]" is defined on line 142 of "rr3.v"
176.  INTERFACE   LVD   :arbiter.dataC[0]
177.                      : Wire "arbiter.dataC[0]" has loss of valid data at the interface
178.                      : Wire "arbiter.dataC[0]" is defined on line 143 of "rr3.v"
179.
180.  =========================================
181.  Summary of bounded failed checks by type:
182.  =========================================
183.  [CA] = 0     (out of 39)
184.  [BE] = 0     (out of 40)
185.  [AX] = 0     (out of 15)
186.  [CME] = 0    (out of 10)
187.  [CV] = 0     (out of 29)
188.  [AC ] = 0    (out of 7)
189.  [AAO] = 0    (out of 6)
190.  [AID] = 0    (out of 9)
191.  [LVD] = 0    (out of 9 )
192.
193.  [...]
194.
```

Fig. 13E

```
195.  ==========================================
196.  Summary of bounded failed checks by type:
197.  ==========================================
198.  [CA] = 39     (out of 39)
199.  [BE] = 39    (out of 40)
200.  [AX] = 14    (out of 15)
201.  [CME] = 10    (out of 10)
202.  [CV] = 29    (out of 29)
203.  [AC ] = 1    (out of 7)
204.  [AAO] = 6    (out of 6)
205.  [AID] = 6   (out of 9)
206.  [LVD] =6   (out of 9 )
207.
208.  [...]
209.
210.  ============================
211.  List of inferred flops
212.  ============================
213.  bus.InterfaceA.state[0]
214.  bus.InterfaceA.state[1]
215.  bus.InterfaceB.state[0]
216.  bus.InterfaceC.state[1]
217.  bus.InterfaceC.state[0]
218.  bus.InterfaceC.state[1]
219.  arbiter.state[0]
220.  arbiter.state[1]
221.  arbiter.state[2]
222.  arbiter.watchDogTimer[0]
```

Fig. 13E (Cont.)