



US 20130298102A1

(19) United States

(12) Patent Application Publication
Kadkade et al.

(10) Pub. No.: US 2013/0298102 A1

(43) Pub. Date: Nov. 7, 2013

(54) INPUT SPACE REDUCTION FOR
VERIFICATION TEST SET GENERATION

(71) Applicant: Mentor Graphics Corporation, (US)

(72) Inventors: Sudhir D. Kadkade, Lake Oswego, OR
(US); Clifton A. Lyons, JR.,
Wilsonville, OR (US); Kunal P.
Ganeshpure, Wilsonville, OR (US)(73) Assignee: Mentor Graphics Corporation,
Wilsonville, OR (US)

(21) Appl. No.: 13/875,143

(22) Filed: May 1, 2013

Related U.S. Application Data

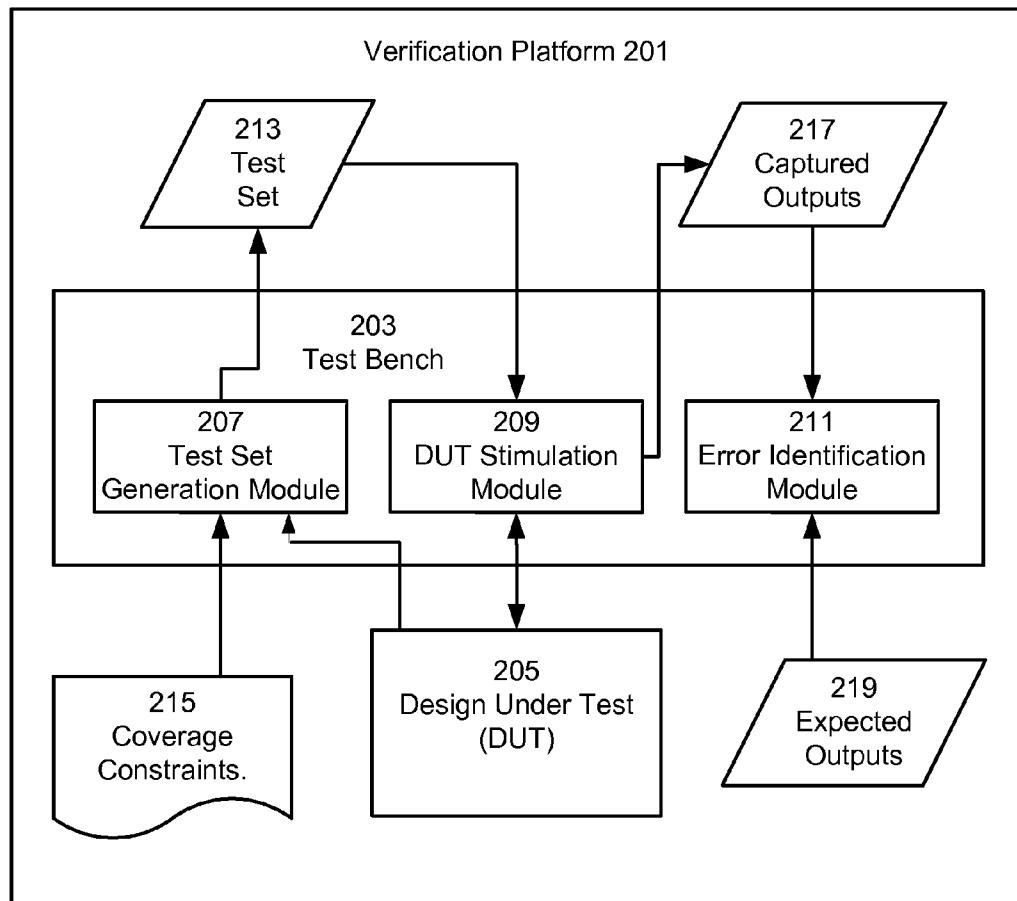
(60) Provisional application No. 61/641,222, filed on May
1, 2012.

Publication Classification

(51) Int. Cl.
G06F 17/50 (2006.01)
(52) U.S. Cl.
CPC *G06F 17/50* (2013.01)
USPC 716/136

(57) ABSTRACT

Various embodiments provide for the determination of a test set that satisfies a coverage model, where portions of the search space need not be searched in order to generate the test set. With various embodiments, a search space defined by a set of inputs for an electronic design and a coverage model is identified. The search space is then fractured into subspaces. Subsequently, the subspaces are solved to determine if they include at least one input sequence that satisfies the coverage constraints defined in the coverage model. The subspaces found to include at least one input sequence that satisfies these coverage constraints, are then searched for unique input sequences in order to generate a test set. Subspaces found not to include at least one input sequence that satisfies the coverage constraints may be excluded from the overall search space.



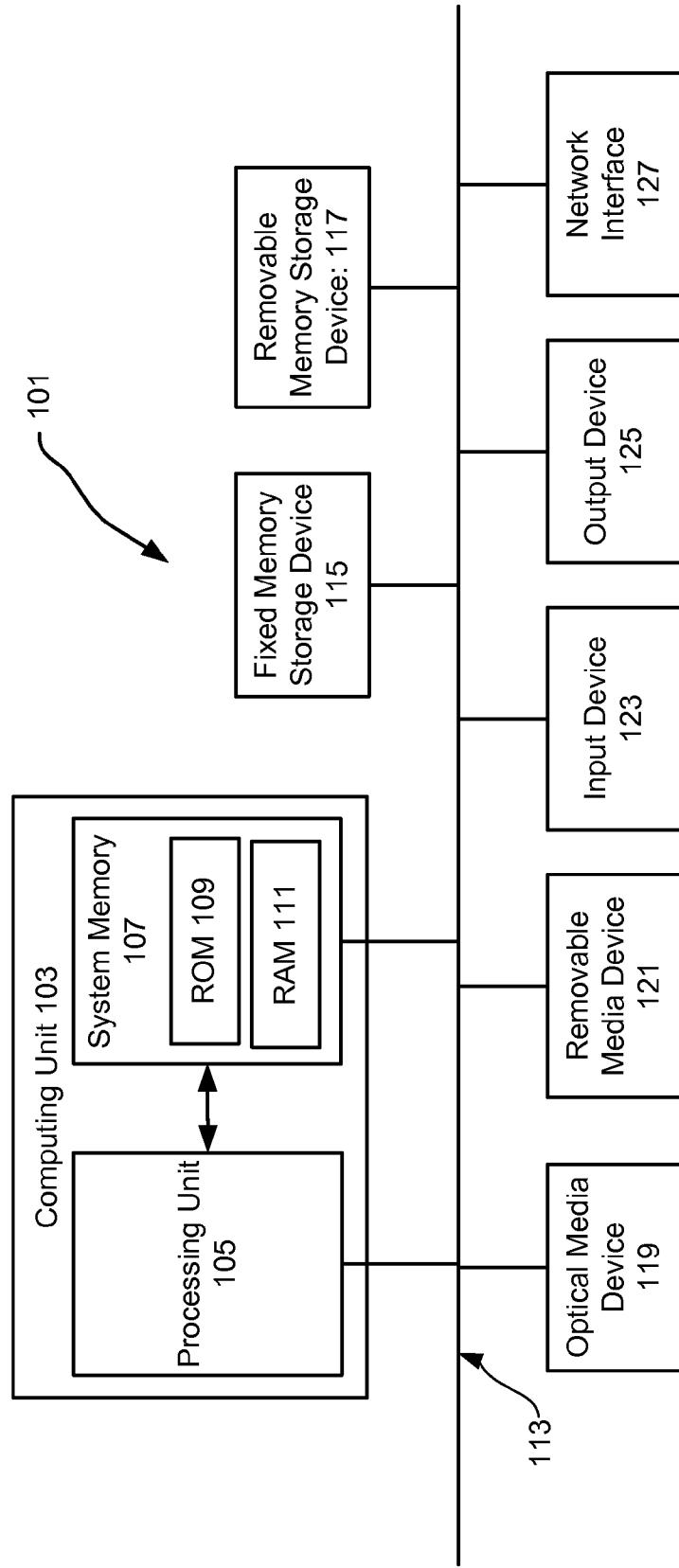
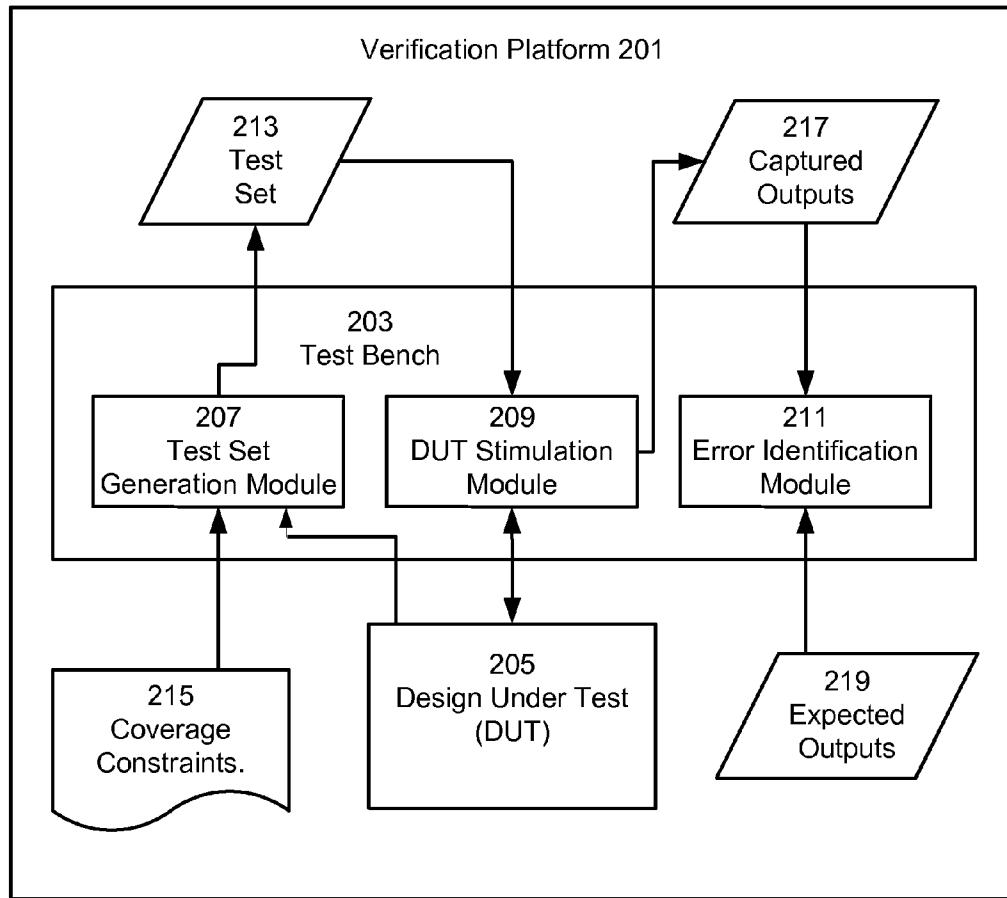
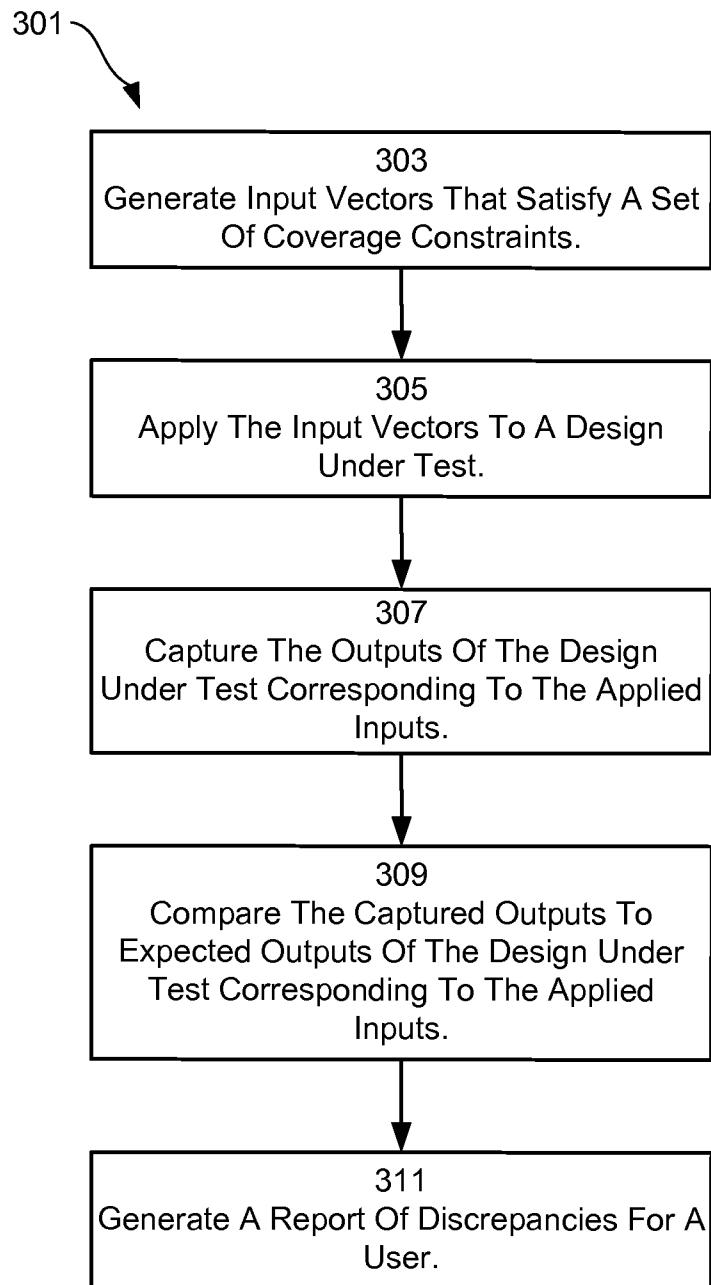
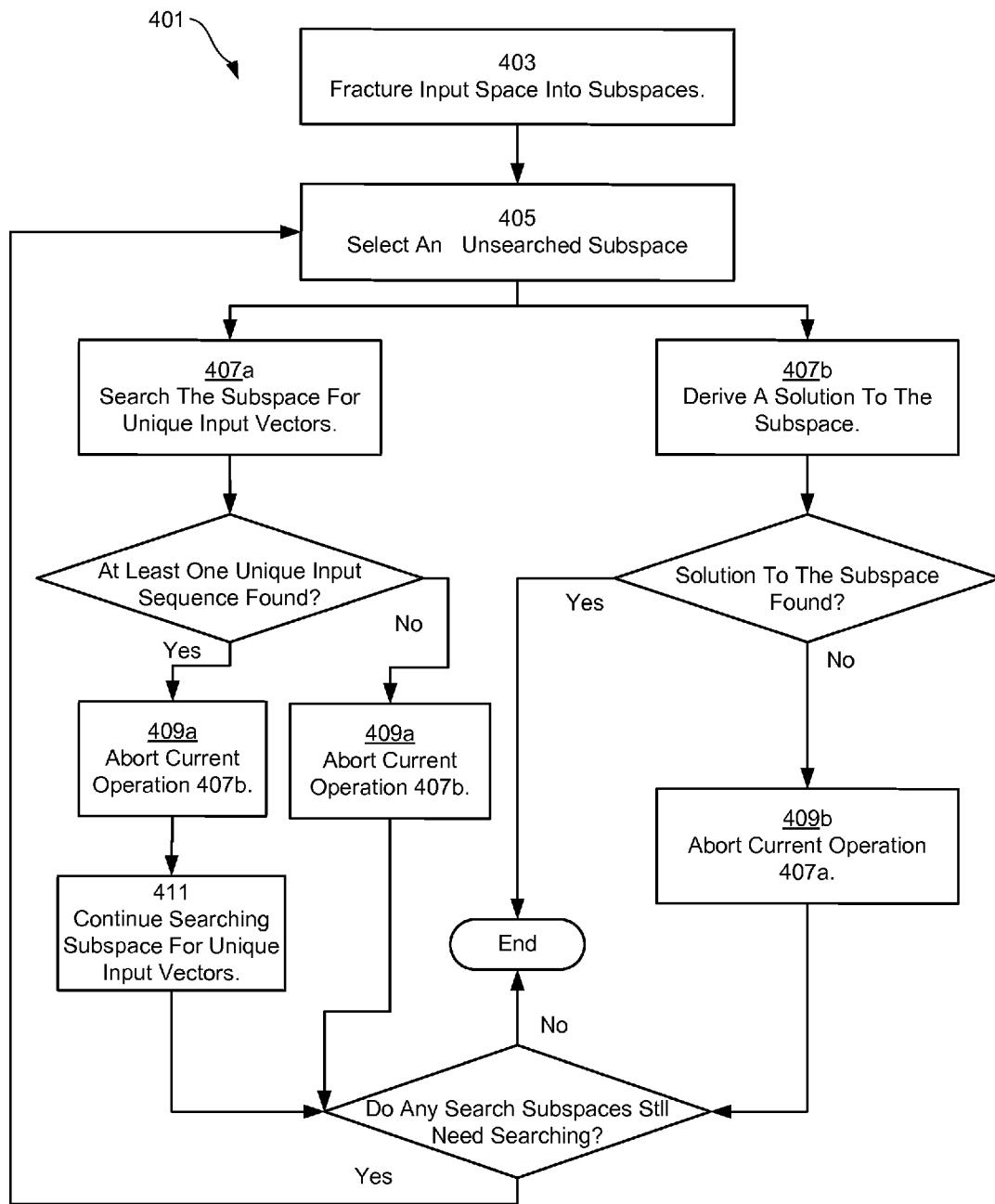
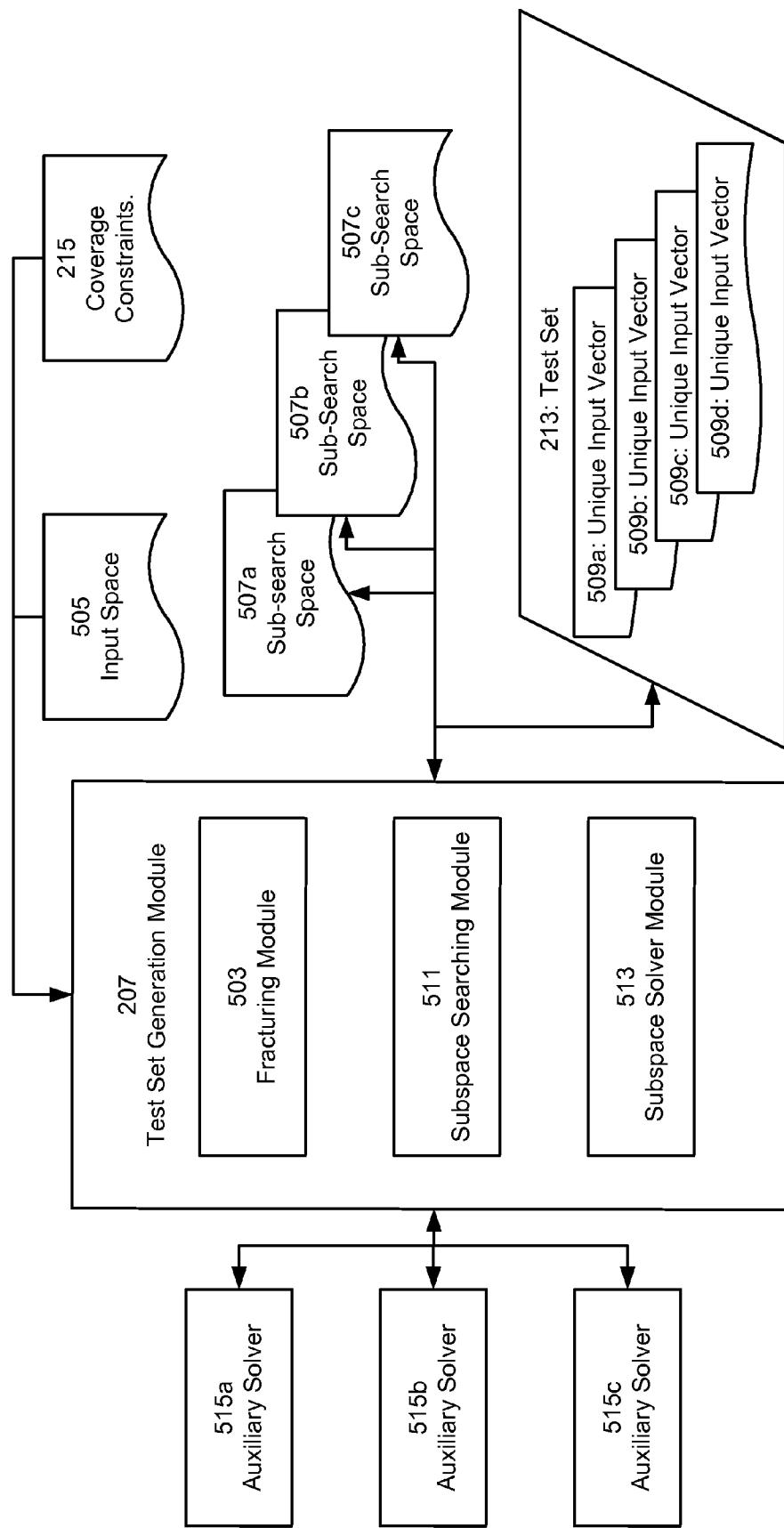


FIGURE 1

**FIGURE 2**

**FIGURE 3**

**FIGURE 4**

**FIGURE 5**

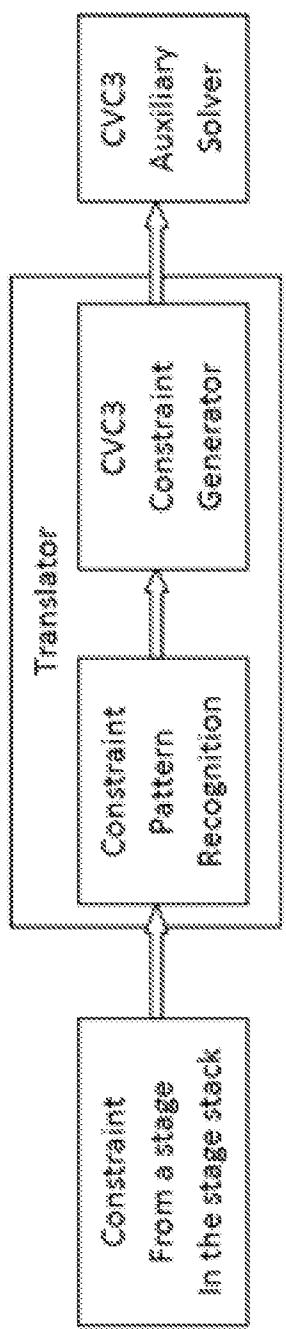


FIGURE 6

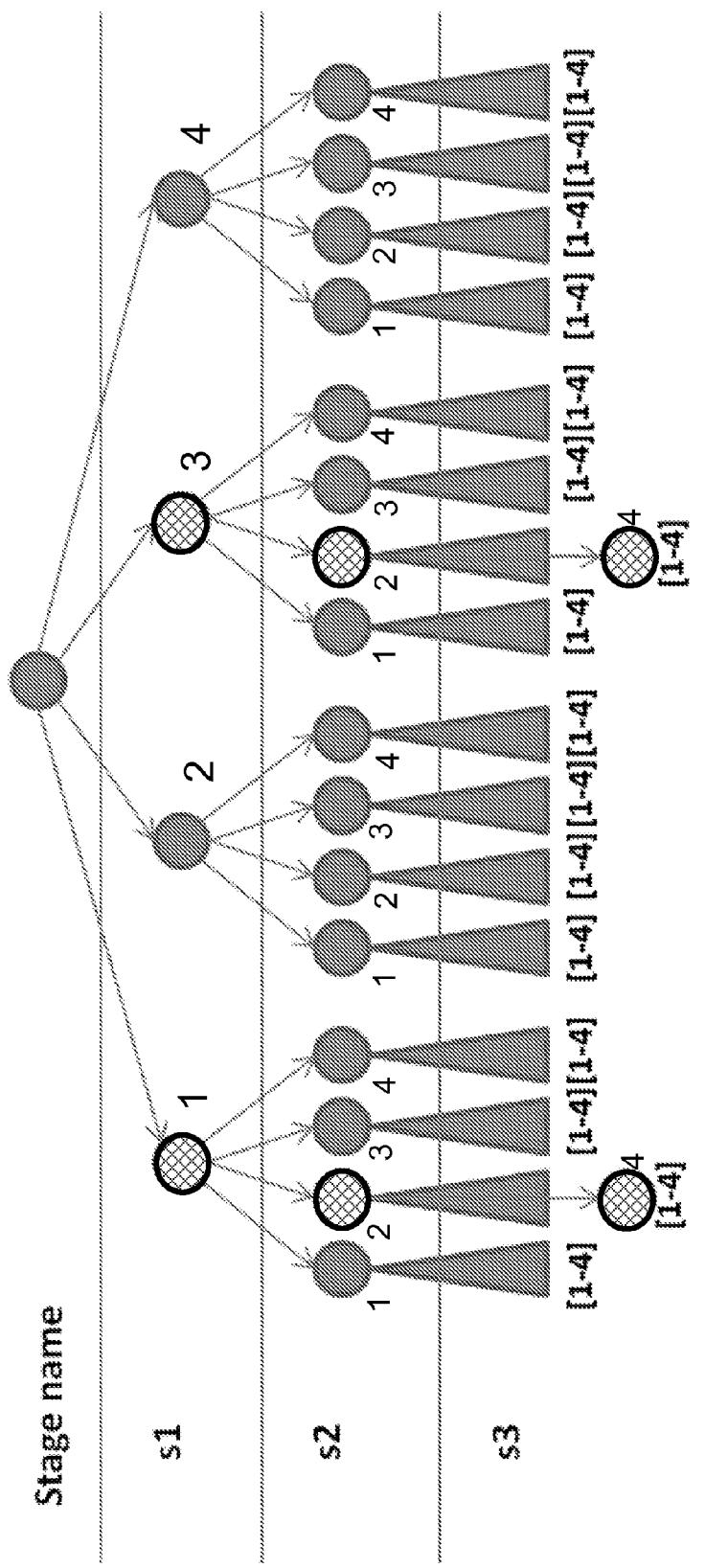
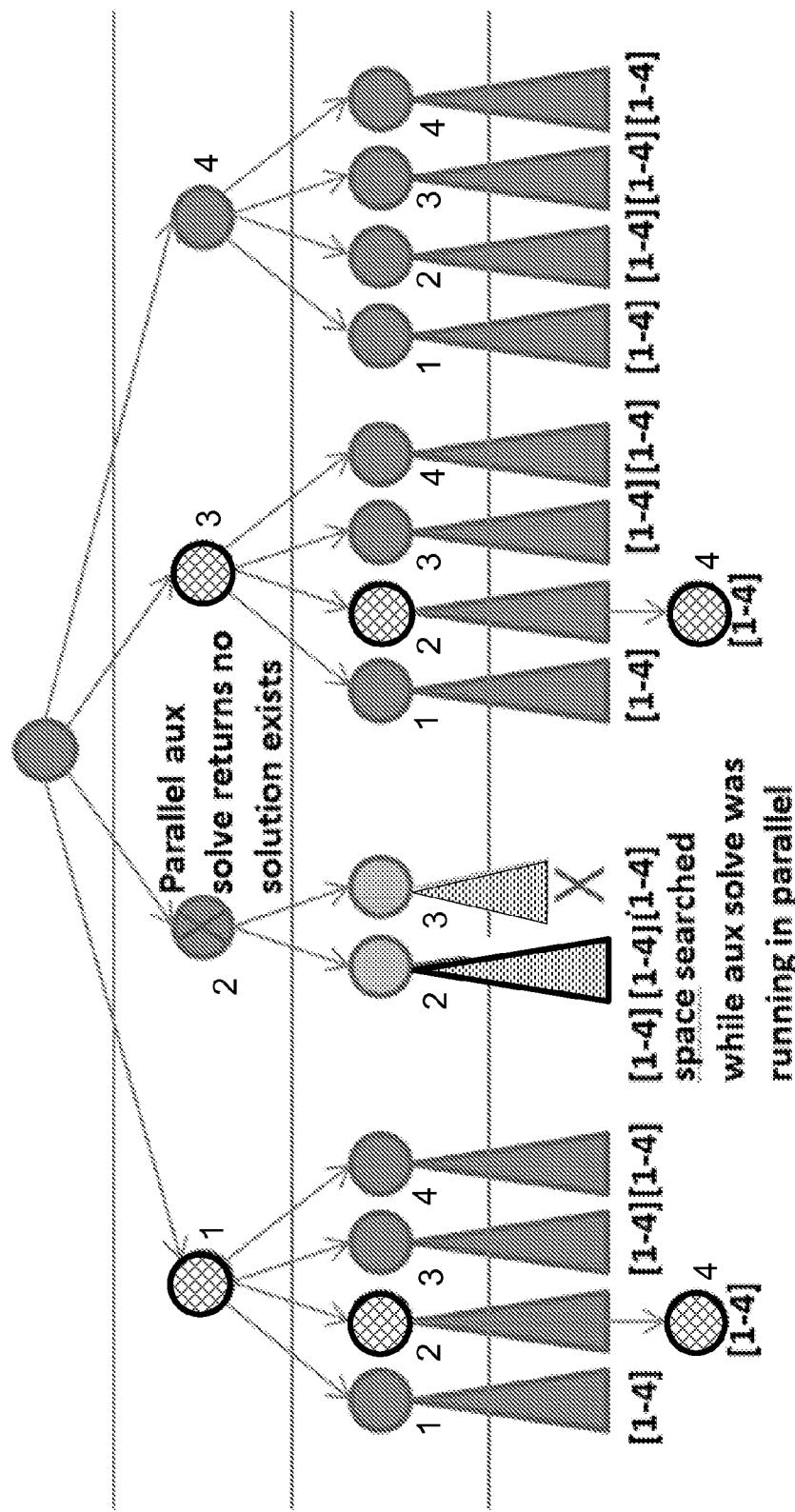


FIGURE 7

**FIGURE 8**

INPUT SPACE REDUCTION FOR VERIFICATION TEST SET GENERATION

CROSS REFERENCE TO RELATED APPLICATIONS

[0001] This application claims priority to provisional application No. 61/641,222, filed on May 1, 2012, which is hereby incorporated by reference in its entirety for all purposes.

FIELD OF THE INVENTION

[0002] The present invention is directed towards the verification of electronic designs. More specifically, various embodiments are applicable to generating test sets, which satisfy specified verification criteria.

BACKGROUND OF THE INVENTION

[0003] Electronic devices are used in a variety of products, from personal computers to automobiles to toys. There are various different types of electronic devices, such as, for example, an integrated circuit. Furthermore, as those of skill in the art will appreciate, electronic devices can be connected, to form other electronic devices or systems. The designing and fabricating of electronic devices typically involves many steps, sometimes referred to as the "design flow." The particular steps of a design flow often are dependent upon the type of electronic device, its complexity, the design team, and the fabricator that will manufacture the device.

[0004] Several steps are common to many design flows. Initially, the specification for a new design is expressed, often in an abstract form and then transformed into lower and lower abstraction levels until the design is ultimately ready for manufacture. The process of transforming the design from one level of abstraction to another is referred to as synthesis. At several stages of the design flow, for example, after each synthesis process, the design is verified. Verification aids in the discovery of errors in the design, and allows the designers and engineers to correct or otherwise improve the design. The various synthesis and verification processes may be facilitated by electronic design automation (EDA) tools.

[0005] Synthesis and verification processes applied to modern electronic designs may be quite complex and may include many different steps. An illustrative design flow, for an integrated circuit, for example, can start with a specification for the integrated circuit being expressed in a high-level programming language, such as, for example, C++. This level of abstraction is often referred to as the algorithmic level. At this abstraction level, the functionality of the design is described in terms of the functional behavior applied to specified inputs to generate outputs. The design will then be synthesized into a lower level of abstraction, typically, one of various logic level of abstraction having different amounts of detail. At this level of abstraction, the design may be expressed in a hardware description language (HDL) such as Verilog, where the circuit is described in terms of both the exchange of signals between hardware registers and the logical operations that are performed on those signals. At this stage, verification is often performed to confirm the functional behavior of the design, i.e. that the logical design conforms to the algorithmic specification.

[0006] After the logical design is verified, it is synthesized into a device design. The device design, which is typically in the form of a schematic or netlist, describes the specific electronic components (such as transistors, resistors, and

capacitors) that will be used in the circuit, along with their interconnections. Verification is again performed at this stage in order to confirm that the device design conforms to the logical design, and as a result, the algorithmic specification.

[0007] Once the components and their interconnections are established, as represented by the device design, the design is again synthesized, this time into a physical design that describes specific geometric elements. The geometric elements define the shapes that will be created in various layers of material to manufacture the circuit. This type of design often is referred to as a "layout" design. The layout design is then used as a template to manufacture the integrated circuit. Verification is again performed, to ensure that the layout design conforms to the device design.

[0008] Although there are different methods of performing verification, various embodiments disclosed herein are directed towards verification processes that "exercise" a design by applying input to the design and capturing the output resulting from application of the input. The applied input is often referred to as an input vector. The captured output then is compared to the output the design should have produced according to the input vector and the specification. Various technologies exist for exercising a design. For example, the response (i.e. the output) of the design to the input vector may be simulated in software. In some cases, the output may be captured from an emulator that is emulating the design with the input vector as stimulus for the emulation. In other aspects, a prototype may also be used to generate the output. Those of ordinary skill in the art will appreciate that combinations of simulation, emulation, and prototyping could be used in various combinations during verification and that various combinations of technologies can be employed to implement a verification system as described here.

[0009] Verification, in various aspects, consists of applying multiple input vectors sequentially (where an input vector is a tuple of values for input variables,) referred to as the test set and capturing each resulting output, referred to as the output set. The individual outputs from the output set then are compared to the corresponding expected outputs. There are many ways to generate the input vectors to include in a test set. For example, directed tests, that is, where the input vectors are directly specified by a designer can be employed. Random combinations of inputs can also be selected and used to form input vectors. One could generate a test set that corresponds to all possible input combinations. The set of all possible input vectors to a modern electronic design may be so large however, that it is not computationally feasible to exhaustively test the design in this manner. As a result, another approach to generating input vectors for verification is often used.

[0010] Each input of a design has a domain. The domain specifies the set of possible values that may be applied to the input corresponding to the domain. Sometimes, there are constraints, which restrict the combinations of input values that may be used to generate an input vector. These constraints may include legality constraints (e.g., which may be derived from the design's operating specification), apparatus constraints (e.g., which may be derived from the specifications of the equipment used to exercise the design), and/or coverage constraints (e.g., which may be derived from a verification plan). Other types of constraints may also be used to limit or restrict the input values, which may be used to generate input vectors.

[0011] The process of finding a tuple of values (e.g., a vector), which satisfies a set of constraints, is often referred to as “constraint solving”. The process of finding a set of tuples of values, within which every tuple satisfies a set of constraints, is sometimes referred to as “constraint exploration.”

[0012] Conceptually, to find a solution (or a set of solutions) to a constraint set, one could iterate through the set of possible value tuples (i.e. the cross products of the domains of the variables) and test each tuple against the constraints, selecting only those tuples that satisfy every constraint, until the desired number of solutions is found. In practice, for modern electronic devices, the size of the set of possible value tuples (sometimes referred to as the “input space”) is so large and the fraction of acceptable tuples is so small that the iteration approach is not practical. Instead, computational techniques such as satisfiability analysis are used to locate acceptable tuples within the input space. Those of ordinary skill in the art will appreciate that such computational techniques generally run considerably faster if the size of the input space can be reduced. Therefore, methods that can identify solution-free regions of an input space that can be pruned without affecting the verification results are of considerable interest.

[0013] Example embodiments of the disclosure discuss methods and techniques for pruning the input space as indicated above. Although such discussion is generally made with reference to coverage constraints, those of ordinary skill in the art will appreciate that other types of constraints may be used without departing from the spirit and scope of the invention.

BRIEF SUMMARY OF THE INVENTION

[0014] Various embodiments provide for the determination of a test set that satisfies a constraint set, where portions of the input space are excluded from being searched in order to generate the test set.

[0015] With various embodiments, an input space defined by a set of inputs for an electronic design and a constraint set are identified. The input space is then fractured into subspaces. Subsequently, the subspaces are solved to determine if they include at least one input vector that satisfies the constraints defined in the constraint set (e.g., state space solving). The subspaces found to include at least one input vector that satisfies these constraints, are then searched for unique input vectors to include in the test set (e.g., constraint exploration). Subspaces found not to include at least one input vector that satisfies the constraints, are excluded from being searched. In certain variations, the constraint set is defined by a coverage model.

[0016] In various embodiments, operation of the methods and techniques disclosed herein may be repeatable. More particularly, given the same input space and constraint set, the results will be reproducible over multiple iterations of the methods detailed herein. Furthermore, the results may be reproducible using different ones of the methods detailed herein.

[0017] In various embodiments, the search of a particular subspace for unique input vectors will be initiated while the subspace is being solved as stated above. If it is determined by the solver that no solutions exist, then the search of the subspace is aborted. With some implementations, multiple different solvers will be used to solve a subspace simultaneously. In response to one of the solvers determining that no solution exists, the search of the subspace by all solvers is aborted.

[0018] In certain variations, the constraint set is defined by a coverage model. These and additional embodiments may be further understood from the following detailed disclosure of illustrative embodiments.

BRIEF DESCRIPTION OF THE DRAWINGS

[0019] Variations are described by way of illustrative implementations shown in the accompanying drawings in which like references denote similar elements, and in which:

[0020] FIG. 1 illustrates a computing device according to one or more embodiments.

[0021] FIG. 2 illustrates a verification platform according to one or more embodiments.

[0022] FIG. 3 illustrates a method of performing verification according to one or more embodiments.

[0023] FIG. 4 illustrates a method of generating a test set according to one or more embodiments.

[0024] FIG. 5 illustrates component of the verification platform of FIG. 2 in greater detail according to one or more embodiments.

[0025] FIG. 6 illustrates a block diagram of constraint translation according to one or more embodiments.

[0026] FIG. 7 illustrates a search tree for a constraint problem example according to one or more embodiments.

[0027] FIG. 8 illustrates a search tree for a constraint problem example according to one or more embodiments.

DETAILED DESCRIPTION OF THE INVENTION

[0028] The operations of the disclosed implementations may be described herein in a particular sequential order. However, it should be understood that this manner of description encompasses rearrangements, unless a particular ordering is required by specific language set forth below. For example, operations described sequentially may in some cases be rearranged or performed concurrently or in a different sequential order. Further, the methods disclosed may be used in conjunction with other methods disclosed; however, For the sake of simplicity, the illustrated flow charts and block diagrams typically do not show the various ways in which particular methods can be used in conjunction with other methods.

Illustrative Operating Environment

[0029] As the techniques of the present invention may be implemented using computer executable instructions being executed on a computer, the components and operation of a programmable computer system on which various embodiments may be employed is described. Accordingly, FIG. 1 shows an illustrative computing device 101. As seen in this figure, the computing device 101 includes a computing unit 103 having a processing unit 105 and a system memory 107. The processing unit 105 may be any type of programmable electronic device (e.g., a microprocessor) for executing machine executable instructions (e.g., software). The system memory 107 may include both read-only memory (“ROM”) 109 and random access memory (“RAM”) 111. Both or either the ROM 109 and the RAM 111 may store the instructions for execution by the processing unit 105.

[0030] The processing unit 105 and the system memory 107 are connected, either directly or indirectly, through a bus 113 or alternate communication structure, to one or more peripheral devices. For example, the processing unit 105 or the system memory 107 may be directly or indirectly con-

nected to one or more additional devices, such as; a fixed memory storage device **115**, for example, a magnetic disk drive; a removable memory storage device **117**, for example, a removable solid state disk drive; an optical media device **119**, for example, a digital video disk drive; or a removable media device **121**, for example, a removable floppy drive. The processing unit **105** and the system memory **107** also may be directly or indirectly connected to one or more input devices **123** and one or more output devices **125**. The input devices **123** may include, for example, a keyboard, a pointing device (such as a mouse, touchpad, stylus, trackball, or joystick), a scanner, a camera, and a microphone. The output devices **125** may include, for example, a monitor display, a printer and speakers. With various examples of the computing device **101**, one or more of the peripheral devices **115-127** may be internally housed with the computing unit **103**. Alternately, one or more of the peripheral devices **115-127** may be external to the housing for the computing unit **103** and connected to the bus **113** through, for example, a Universal Serial Bus ("USB") connection.

[0031] With some implementations, the computing unit **103** may be directly or indirectly connected to one or more network interfaces **127** for communicating with other devices making up a network. The network interface **127** translates data and control signals from the computing unit **103** into network messages according to one or more communication protocols, such as the transmission control protocol ("TCP") and the Internet protocol ("IP"). Also, the interface **127** may employ any suitable connection agent (or combination of agents) for connecting to a network, including, for example, a wireless transceiver, a modem, or an Ethernet connection.

[0032] It should be appreciated that the computing device **101** is shown here for illustrative purposes only, and it is not intended to be limiting. Various embodiments of the invention may be implemented using one or more computers that include the components of the computing device **101** illustrated in FIG. 1, which include only a subset of the components illustrated in FIG. 1, or which include an alternate combination of components, including components that are not shown in FIG. 1. For example, various embodiments of the invention may be implemented using a multi-processor computer, a plurality of single and/or multiprocessor computers arranged into a network, or some combination of both.

[0033] As stated above, various embodiments of the invention may be implemented using a programmable computer system executing software instructions, a computer readable medium having computer-executable software instructions stored thereon, or some combination thereof. The software instructions may be stored on one or more computer readable media or devices, such as, for example, the system memory **107**, or an optical disk for use in the optical media device **119**. The software instructions, in various embodiments are stored in non-transitory media. More specifically, in some embodiments, the software instructions are not stored as a carrier wave or some other transitory signal.

Coverage Based Verification

[0034] As detailed above, various embodiments provide methods and apparatuses for generating a test set to be used for verifying an electronic design. Specifically, the test set may be generated that includes unique input vectors that satisfy the coverage model by excluding a search all portions of an input space defined by the potential inputs to the electronic design. FIGS. 2 and 3 illustrate a verification platform

201 and a method **301** of performing verification that may be provided by various implementations of the present invention.

[0035] As can be seen from FIG. 2, the verification platform **201** includes a test bench **203** and a design under test (DUT) **205**. With various instances of the invention, the DUT **205** will be implemented by an electronic design simulator, such as, for example, the Questa simulator available from Mentor Graphics Corporation of Wilsonville, Oreg. In some instances, the DUT **205** will be implemented by an electronic design emulator, such as, for example, the Veloce emulator available from Mentor Graphics Corporation of Wilsonville, Oreg. In various instances, the DUT **205** will be implemented by a prototype of the electronic design for which the DUT **205** represents. Still, with some embodiments, combinations of these different embodiments may be used. For the balance of this disclosure however, embodiments are described, for purposes of clarity only, that the DUT **205** is implemented in a simulator.

[0036] In some implementations, the DUT **205** may be a software program. As those of ordinary skill in the art will appreciate, software programs are also verified during development. This verification process shares similarities to the hardware verification process detailed above. For example, a software program will have a set of valid inputs (e.g., button press, menu selection, key input, etc.). Furthermore, the software program may have a set of inputs with which the designer is particularly interested in testing. As such, coverage constraints may be specified for the software program, and then various embodiments of the present disclosure may be implemented to determine a test set for the software program as detailed herein.

[0037] The test bench **203** includes a test set generation module **207** configured to derive a set of input vectors (i.e. the test set **213**) for the DUT **205** that satisfy a set of coverage constraints **215**. The derivation of input vectors by the test set generation module **207** will be discussed in greater detail below. The test bench **203** further includes a DUT stimulation module **209** configured to apply the input vectors from the test set **213** to the DUT **205** (e.g., drive signals onto inputs of the DUT). The DUT stimulation module **209** is also configured to capture the responses of the DUT **205** as it is tested with the test set **213** as input. These captured responses are referred to as the captured outputs **217**. Additionally, an error identification module **211** is provided, which is configured to compare the captured outputs **217** to a set of expected outputs **219**. The error identification module **211** is further configured to identify any discrepancies in the comparison and report them as potential errors in the DUT **205**.

[0038] As can be further seen from FIG. 2, the test set generation module **207** may, in some implementations, generate tests based on the state of the DUT **205**. For example, during a verification process, the state of the DUT **205** will change in response to various factors (e.g., inputs applied, etc.). As such, the test set generation module **207** may account for these changing states in generating inputs to include in the test set **213**.

[0039] FIG. 3 shows the method **301**, which may be applied using the verification platform described above. As can be seen from this figure, the method **301** includes an operation **303** for generating the test set **213**, that is, for generating input vectors that satisfy the coverage constraints **215**. An operation **305** for applying the test set **213** to the DUT **205** and an operation **307** for recording the captured outputs **217** is also

provided. An operation for comparing the captured outputs 217 to the expected outputs 219 and an operation 311 for outputting a result (e.g., generating a report) of any discrepancies from this comparison are then provided.

Coverage Based Input Set Generation

[0040] As stated, various embodiments provide methods and apparatuses for generating a test set to be used for verifying an electronic design. Certain examples generate the test set without searching all portions of an input space defined by the potential inputs to the electronic design. Furthermore, the test set generation module 207 may be configured to generate the test set 213 in this manner, such as, for example, by application of the operation 303.

[0041] An electronic design, which the DUT 205 may represent, may have multiple input ports, with multiple different inputs allowed per port. For discussion purposes, each input port may be represented by an input variable, where each input variable may have multiple possible values. For example, the DUT 205 may represent an electronic design with 5 input ports where each input is binary. Representing the input ports as variables a-e, the set of all possible inputs for the DUT 205 may then be represented as the set of all possible combinations of (a, b, c, d, e) where the variable values are either logic 0 or logic 1. Each input combination (e.g., [0, 0, 0, 0, 1]) is referred to here as an input vector. The set of all possible input vectors then is referred to as the input space. For this simple example, there are only 32 different input combinations, that is, there are 32 unique input vectors in the input space. However, if the variable values were allowed to be anything between 0 and 9, then the possible combinations increase to 10,000. For a design where there are 20 input ports and 10 possible input values per port, there are over 10^{19} potential input combinations.

[0042] Modern electronic designs may include hundreds of input ports, with many different combinations of potential input values per port. As such, it is not practical to derive and then test all possible unique input vectors in a verification process. In order to reduce the number of input vectors needed to test a design, input ports, input values, and combinations of input ports and values that exercise particular functionality are identified. The input space is then restricted based on these identified input ports and input values. This is done by writing a set of constraints that restricts the values that each input variable can take on. More particularly, given the example above, where there are 5 possible inputs, represented by variables a-e, and where the possible input values are allowed to be an integer value between 0 and 9, the input space may be restricted by the constraint that the input variable "a" only be allowed to take on values of (1, 2, 3, 4, 5). Accordingly, the input space is now "restricted" based on this constraint, referred to herein as the coverage constraint, and the number of unique input vectors that satisfy this constraint is less than the total number of unique input vectors in the entire input space.

[0043] The operation 303 of FIG. 3 then derives the unique input vectors within the input space that satisfy the coverage constraints. Similarly, the test set generation module 207 of FIG. 2 is configured to determine the unique input vectors within the input space that satisfy the coverage constraints. As stated, these unique input vectors form the test set 213.

[0044] FIG. 4 illustrates a method 401 that may be provided by various implementations of the present invention to identify these unique input vectors in a search space defined by the

coverage constraints 215 and the DUT 205. FIG. 5 illustrates the test set generation module 207 in greater detail. As can be seen, the test set generation module 207 includes an input space fracturing module 503. FIG. 5 shows an input space 509, which corresponds to the inputs and possible input value of the DUT 205. Accordingly, the input space fracturing module 503 may be configured to split the search space 505 into subspaces 507.

[0045] The method 401 includes an operation 403 for fracturing the input space 505 into the sub spaces 507. With various implementations, the search space is fractured by input variables. Using the system and constraints described above as an example, with inputs a-e having possible values 0-9, and where the variable a is constrained between 1 and 5, the input space 505, may be fractured by the variable a, into the following 5 sub spaces 507: (1, b, c, d, e), (2, b, c, d, e), (3, b, c, d, e), (4, b, c, d, e), and (5, b, c, d, e). More sophisticated methods of fracturing may be used. For example, the input space 505 may be fractured based on multiple variables. Furthermore, an order of which variable the input space is fractured on first may be selected.

Search Space Reduction Using Auxiliary Solvers

[0046] An operation 405 for selecting an unsearched one of the subspaces 507 and an operation 407a for searching the selected subspace 507 for unique input vectors 509 are provided. The test set generation module 207 shown in FIG. 5 further includes a subspace searching module 511, which is configured to search the sub spaces 507 for unique input vectors 509. Any unique input vectors 509 found during the search are then added to the test set 213.

[0047] The method 401 further includes an operation 407b for deriving a solution to the sub space 507 selected by the operation 405, and for which the operation 407a is searching for unique input vectors. With various implementations, the operations 407a and 407b are executed simultaneously, such as, for example, by executing them on separate programmable computer systems (e.g. the programmable computer system 101 detailed above). In various implementations, the operation 407b will determine if at least a single solution to the selected subspace 507 exists. More specifically, 407b will determine if at least one input vector satisfying the coverage constraints 215 exists within the selected subspace 507.

[0048] As detailed, the operation 407a searches the selected subspace 507 for unique input vectors 509, that is, input vectors that satisfy the coverage constraints, but are not yet included in the test set 213. The operation 407b however, derives a solution to the selected subspace 507, to determine if at least one input vector, not necessarily a unique input vector 509 exists within the subspace 507. Accordingly, as stated, the test set generation module 207 includes the subspace searching module 511, which may be configured to search a sub space 507 for unique input vectors. Additionally, a subspace solver module 513 is included in the test set generation module 207. Auxiliary solvers 515 are also provided, which the subspace solver module 513 may use as resources to solve a subspace 507. With various implementations, multiple auxiliary solvers 515 may be used, sometimes simultaneously, by, for example, executing the solvers on separate computer systems. More specifically, the subspace solver module 513 may use multiple different auxiliary solvers 515 simultaneously to determine if a solution to the

selected subspace **507** exists. Subsequently, whichever solver determines that a solution does not exist first will trigger the operation **409b**.

[0049] Operations **409a** and **409b** are provided for aborting the operations **407b** and **407a** respectively. As can be seen, if the operation **407b** determines that no solutions to the selected subspace **507** exist (i.e. that no input vectors satisfying the coverage constraints **215** exist within the subspace **507**) then an operation **409b** for aborting the current operation **407a** is provided. If the operation **407b** determines that a solution does exist, then the operation **407a** is allowed to continue.

[0050] Similarly, in the event that a unique input vector **509** is found by the operation **407a**, and then the currently executing operation **407b** is aborted by the operation **409a** and an operation **411** for continuing to search the selected subspace **507** for unique input vectors **509** is provided. Additionally, if no input vectors are found within the subspace by the operation **407a**, the operation **409a** may be used to abort the operation **407b**. Once either the currently selected subspace **507** is searched or it is determined that there are no input vectors satisfying the coverage constraints **215**, it is checked to see if any subspaces **507** still need searching. If so, then the method **401** returns to operation **405**. If not, then the method **405** ends.

[0051] With some implementations of the disclosed subject matter, the methods detailed in FIG. 4 and elsewhere herein, may be performed in parallel. More specifically, multiple iterations of the method **401** may be performed concurrently. Additionally, the methods detailed herein may be performed in a hierarchical fashion. More particularly, an input space (e.g., the input space **505**) may be fractured at different levels, forming a hierarchy of subspaces **507**. Unique input vectors within these subspaces may be identified using the techniques disclosed herein, by for example, initiating multiple instances of the method **401** using the hierarchical subspaces. As one non-limiting example, if a first subspace at one level of hierarchy and a second subspace at a lower level of hierarchy were being searched and it was determined that no solutions existed for the higher level subspace (i.e., the higher level subspace did not include any input vectors that satisfy the constraints), then the search of both the higher and lower level subspace may be aborted.

[0052] With various implementations of the disclosed technology, the auxiliary solvers **515** may adhere to a numerical solver standard, such as, for example, DIMACS or SMT-LIB. In further implementations, the auxiliary solvers **515F** may be of the following types of numerical solvers, CVC3, ABSolver, CVC4, MiniSMT, Yices, or Z3. The CVC3 solver and associated technical details are addressed in greater detail in *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, by Clark Barrett and Cesare Tinelli, volume 4590 of *Lecture Notes in Computer Science*, pages 298-302. Springer, July 2007. Berlin, Germany, which article is incorporated entirely herein by reference.

[0053] In various implementations the subspaces **507** may be transformed into a particular format (e.g., suitable for use by the auxiliary solvers **515**), such as, for example, bit vector format. In some examples, the constraints associated with a particular subspace **507** may be translated into a corresponding set of constraints, such as, for example, in bit-vector representation. With further implementations, the set of constraints may also be simplified. With some implementations, the translator works by generating a constraint expression tree

and propagating the expression tree in the post-order fashion. As shown below, constraint translation may consist of (i) the Constraint Pattern Recognizer and the (ii) Constraint Generator (e.g., CVC3 Constraint Generator). The constraint pattern recognizer identifies some constraint patterns involving non-linear operators that can be converted to simpler operations. The Constraint Generator may translate theses identified sub-expressions into the simpler form. If a sub-expression cannot be simplified or is not identified by the Constraint Pattern Recognizer then the Constraint Generator may translate the original constraints without simplification.

[0054] FIG. 6 illustrates a block diagram of constraint translation. In FIG. 6, constraints from a stage (e.g., in a stage stack) is received by the Constraint Pattern Recognizer that identifies some constraints. The Constraint Generator (e.g., CVC3) receives the recognized constraints and translates them to simpler operations. The Constraint Generator may use an auxiliary solver.

[0055] Table 1 below shows the patterns that may be recognized by the Constraint Pattern Recognizer and the corresponding bit vector constraints generated by the Constraint Generator (e.g., CVC3).

Variables and Domains:

[0056] A: [0 . . . 255]

[0057] Bit Vector Representation: $A_{BV} = a^7, a^6, a^5, a^4, a^3, a^2, a^1, a^0$

TABLE 1

Patterns Recognized By Constraint Pattern Recognizer		
inFact Constraints	CVC3 Bit Vector Constraints	Comments
$A \% 2^n$	$A_{BV} \& (2^n - 1)$	Bitwise And Operation
$A / 2^n$	$A_{BV} \gg n$	Arithmetic Shift right operation
$A * 2^n$	$A_{BV} \ll n$	Shift left operation
$2^n \ll A$	2^{A+n}	
$2^n \gg A: A < n$	2^{n-A}	

[0058] The following is one example with two constraints. The values of variables a_1 , a_2 and a_3 can be represented using 3 bits. The constraint C1 says that a_2 and a_3 can only have a single “1”.

Variables and Domains:

[0059] $a_1: [1 \dots 4]$

$a_2: [1 \dots 4]$

$a_3: [1 \dots 4]$

Constraints:

[0060] C1: $(a_2 \wedge (a_2 - 1)) \wedge (a_3 \vee (a_3 - 1)) == 0$ /* a_2 and a_3 have only one “1”*/

C2: $(a_1 \% 2) \vee (a_2 \% 4) \vee (a_3 \% 8) == 7$

Transformed Constraints:

[0061] C1: $(a_{2BV} \& (a_{2BV} - 1)) \mid (a_{3BV} \& (a_{3BV} - 1)) == 000$

C2: $(a_{1BV} \& 001) \mid (a_{2BV} \& 011) \mid (a_{3BV} \& 111) == 111$

[0062] As can be seen, the constraint C1 could not be recognized to have any specific patterns. Hence it was translated without any simplification. But for constraint C2 the mod operator is eliminated as recognized by the masking operation being done using the mod operator.

[0063] In some implementations, operation of the methods and techniques disclosed herein may be repeatable. More particularly, given the same input space **505** and coverage constraints **215**, the results will be reproducible. The concept of repeatability is discussed below.

Repeatability 1. Introduction:

[0064] As indicated, various embodiments may provide repeatability for a given input space and set of constraints. Accordingly, for a given random seed and a given problem the constraint solver should generate the same solution order among multiple solver runs. Repeatability is an important consideration for testability and correctness of the solver. It is important to note that the following discussion is provided for purposes of explanation only and is not intended to be limiting.

Example 1

[0065] The following constraint problem will be used as an example for the rest of the section.

Variables and Domains:

[0066] $a_1: [1 \dots 4]$
 $a_2: [1 \dots 4]$
 $a_3: [1 \dots 4]$

Constraints:

[0067] C1: $(a_2 \wedge (a_2 - 1)) \wedge (a_3 \vee (a_3 - 1)) == 0$ /* a_2 and a_3 have only one “1”*/

C2: $(a_1 \% 2) \vee (a_2 \% 4) \vee (a_3 \% 8) == 7$

Complexity: Total number of possible solutions: $4^3 = 64$

Solutions:

[0068] $\{a_1, a_2, a_3\} = \{1, 2, 4\}$ and $\{3, 2, 4\}$

[0069] FIG. 7 illustrates an example search tree for the above constraint problem example. In FIG. 7, the search space for the above problem with the solution values shown in hatch shade. The fracturing order is $\{a_1, a_2, a_3\}$ which is done the stages $\{s_1, s_2, s_3\}$. The values next to the circles show the values assigned to the variables $\{a_1, a_2, a_3\}$. At each stage, after assigning a variable value the heuristic solver runs an optimizer to propagate the assignment to all the constraints and simplifies them. This is not shown separately in the figure and is the part of the edges coming out of the circles. The triangles at the bottom represent all the values of a_3 which are searched in the stage s_3 . The node on the top denoted by “r” is the root node. We will use the following notation:

$\{a_1, a_2, a_3\}$: This represents the values of a_1 , a_2 and a_3 which are being assigned at various stages $\{s_1, s_2, s_3\}$. For example $\{2, 3, 1\}$ means that in stage s_1 value of $a_1=2$, in stage s_2 value of $a_2=3$ and in stage s_3 value of $a_3=1$.

$\{a_1, -, -\}$: This means that the value of a_1 is assigned in stage s_1 but a_2 and a_3 are unassigned.

$\{a_1, a_2, \Delta\}$: This means that we have assigned the values of a_1 and a_2 in stages s_1 and s_2 and we are searching for all the values of a_3 in the stage s_3 . Similarly, we can have $\{a_1, \Delta, \Delta\}$ or $\{\Delta, \Delta, \Delta\}$.

$\{a_1, a_2, \wedge\}$: This represents the condition where we have assigned the values of a_1 and a_2 in the stages s_1 and s_2 , but we will not end up searching all the values of a_3 in stage s_3 because the auxiliary solver (see FIG. 6) returned that no solution exists in the search space $\{a_1, a_2, \Delta\}$.

$\{a_1, \wedge, X\}$: This indicates that we all the values of a_3 and the remaining values of a_2 will be skipped.

[0070] FIG. 8 illustrates a search tree for the above constraint problem example. In FIG. 8, an auxiliary solver is launched in the stage s_2 for the value of $a_1=2(\{2, -, -\})$. While the auxiliary is running, the heuristic solver continues searching for $\{2, 2, \Delta\}$ as shown by the dot filled longer triangle. The auxiliary solver returns with a no-solution exists, when the heuristic solver is in $\{2, 3, \wedge\}$ as shown by the “X”. The heuristic solver aborts the search, pops the stage stack and continues searching for another value in stage s_1 .

2. Problem Definition

[0071] In order to obtain repeatability of solution order in the constraint solutions, constraint random solver employs a pseudo random number generator in order to assign a variable value at each stage of the search tree. The order of search progression depends on the random number sequence. A pseudo random number generator produces a unique sequence for a given seed parameter. For a given fracturing order of the solution space, the solution order is unique to the random number seed. The order of search progression is unique to a given pseudo random number sequence in case the search space is static. Table 2 shows the search space progression for a given pseudo-random sequence for the constraint problem in Example 1.

TABLE 2

Search space progression for a given pseudo random number sequence		
pseudo-random sequence	Search progression	Comments
1	$\{1, -, -\}$	Push stage stack
4	$\{1, 4, -\}$	Push stage stack
2	$\{1, 4, 2\}$	Push stage stack
1	$\{1, 4, 1\}$	
2	$\{1, 4, 2\}$	
4	$\{1, 4, 4\}$	
3	$\{1, 4, 3\}$	Pop Stack
2	$\{1, 2, -\}$	Push Stack
4	$\{1, 2, 4\}$	Solution Found

[0072] The use of an auxiliary solver sub-system used to improve the performance of constraint solution speeds up the search progression of the heuristic solver by indicating whether a solution exists in a given sub-space or not. Consequently, the heuristic does not search the sub-space if an auxiliary solve indicated that there was no solution. As the use of auxiliary solves changes the search progression, it puts the following feature on repeatability. This feature says that the final solution set and the solve order is independent of the following scenarios:

- Enabled or disabling the auxiliary solve mechanism: The final solution should not be affected by the search space pruning when the auxiliary solver mechanism is activated.
- The timing of the results returned from the auxiliary solves: As the auxiliary solves are invoked as separate processes, the system should be tolerant of the non-determinism of the solve execution time.

[0073] In the previous implementation activating the auxiliary solve mechanism breaks repeatability due to the fact that the future order of search progression was dependent on the current search space pruning decisions. This is because it reused the same pseudo random number generator across stages without initializing it with a new seed value. This is a

problem when the auxiliary solve mechanism asynchronously eliminates a sub-space without progressing the pseudo random number generator by the corresponding amount, hence affecting the solution order. Table 3 shows this scenario where we see the effect on the search space progression when auxiliary solve mechanism is activated for the Example 1 above. An auxiliary solve starts in the third row in $\{1, 4, -\}$. It returns with a solution not found when in $\{1, 4, 2\}$. This causes it to abort search for the space $\{1, 4, \Delta\}$. Now instead of searching for space $\{1, 2, -\}$ it searches for $\{1, 1, -\}$ because the next random number generated is “1” instead of a “2”.

TABLE 3

Effect on search space progression due to auxiliary solves					
pseudo-random sequence	Without Auxiliary Solve Mechanism		With Auxiliary Solve Mechanism		
	Search progression	Comments	Search progression	Comments	
—	{—, —, —}	Push stage stack	{—, —, —}	Push stage stack	
1	{1, —, —}	Push stage stack	{1, —, —}	Push stage stack	
4	{1, 4, —}	Push stage stack	{1, 4, —}	Push stage stack and launch auxiliary solve	
2	{1, 4, 2}		{1, 4, 2}	Aux. solve returned Solution not found so pop stage stack	
1	{1, 4, 1}		{1, 1, —}	Push Stage stack	
2	{1, 4, 2}		{1, 1, 2}		
4	{1, 4, 4}		{1, 1, 4}		
3	{1, 4, 3}	Pop Stack	{1, 1, 3}		
2	{1, 2, —}	Push Stack	{1, 1, 2}		
4	{1, 2, 4}	Solution Found	{1, 1, 4}		

[0074] The non-determinism of auxiliary solves can also effect the solution order. The auxiliary solver is started in a separate process in parallel while the heuristic solver is searching in the current sub-space. Consequently, the time it takes for the auxiliary solver to solve a given problem depends on how the process is scheduled by the operating system. If the auxiliary solve returns with no-solution exists in a given sub-space before the heuristic solver finishes, then the heuristic solver stops searching in the given sub-space and starts searching in a next sub-space. The next random number in the sequence determines the next sub-space to be searched. For a solver run, with the auxiliary solve mechanism deactivated, the heuristic solver will finish searching within the current sub-space where no-solution exists. At this time the pseudo-random number generator may return a different value hence selecting a sub-space different from the one selected when the auxiliary solver mechanism is activated. This can be seen in the Table 4 below which compares the search progression of two cases of auxiliary solve mechanism where the auxiliary solver finishes at different times.

TABLE 4

Search space progression when auxiliary solves finishes at different times					
pseudo-random sequence	Case1: With Auxiliary Solve Mechanism		Case2: With Auxiliary Solve Mechanism		
	Search progression	Comments	Search progression	Comments	
—	{—, —, —}	Push stage stack	{—, —, —}	Push stage stack	
1	{1, —, —}	Push stage stack	{1, —, —}	Push stage stack	
4	{1, 4, —}	Push stage stack and launch auxiliary solve	{1, 4, —}	Push stage and stack launch auxiliary solve	
2	{1, 4, 2}	Aux. solve returned Solution not found so pop stage stack	{1, 4, 2}		
1	{1, 1, —}	Push Stage stack	{1, 4, 1}	Aux. solve returned Solution not found so pop stage stack	
2	{1, 1, 2}		{1, 2, —}	Push stage stack	
4	{1, 1, 4}		{1, 2, 4}	Solution Found	
3	{1, 1, 3}		{1, 2, 3}		
2	{1, 1, 2}		{1, 2, 2}		
4	{1, 1, 4}		{1, 2, 4}		

3. Solution

[0075] In order to make the solve order independent of non-determinism and search space pruning done by the auxiliary solve mechanism, we use separate pseudo-random number generators for each stage. These generators are initialized by a seed which is passed on by the previous stage and is based on the value to which a variable is assigned in the previous stage. Consider the following Table 5, which shows the random sequence generated at the level s_i based on the branch taken in the level s_{i-1} . As a result of this, the random number sequence at s_{i-1} does not get effected by the fact that the auxiliary solve mechanism might be pruning out the solution space at the stage s_i without searching it completely.

TABLE 5

Random number sequence at s_i when a branch is taken at s_{i-1}	
Brach at s_{i-1} $a_i = 1$	Random sequence at s_i $\{1, 2, 4, 3\}$
2	{2, 3, 1, 2}
3	{1, 3, 4, 2}
4	{4, 1, 2, 3}

[0076] This enables us to launch auxiliary solves at any stage without causing problems with the solution ordering. This is shown in Table 6, which compares the sequence of operations involved in launching jobs at various stages of the search. In Case1 we are always launching jobs in the stage s_1 while in Case 2 we are doing the same for the stage s_2 . The random number generators at various stages initialized based on Table 5. It also shows the solve progression when the auxiliary solve mechanism is deactivated (Column 1). It can be seen that the solution order remains the same. Moreover, in Case1 even when the Job2 takes longer time than usual to return a solution, the solution order does not change.

TABLE 6

Solve progression comparison when jobs are launched at stage s_1 and stage s_2				
Without Aux solve	Case1: With Aux being launched at stage s_1		Case2: With Aux being launched at stage s_2	
	Solve Progression	Aux Solve Result	Solve Progression	Aux Solve Result
{4, 4, Δ}	{4, Δ, X}	Job1: No solution exists	{4, 4, Δ}	Job1: No solution exists
{4, 1, Δ}	—	—	{4, 1, Δ}	Job2: No solution exists
{4, 2, Δ}	—	—	{4, 2, Δ}	Job3: No solution exists
{4, 3, Δ}	—	—	{4, 3, Δ}	Job4: No solution exists
{2, 2, Δ}	{2, 2, Δ}	Job2: No solution exists	{2, 2, Δ}	Job5: No solution exists
{2, 3, Δ}	{2, 3, Δ}	—	{2, 3, Δ}	Job6: No solution exists
{2, 1, Δ}	{2, X, X}	—	{2, 1, Δ}	Job7: No solution exists
{2, 2, Δ}	—	—	{2, 2, Δ}	Job8: No solution exists
{3, 1, Δ}	{3, 1, Δ}	Job3: At least one solution exists	{3, 1, Δ}	Job9: No solution exists
{3, 3, Δ}	{3, 3, Δ}	—	{3, 3, Δ}	Job10: No solution exists
{3, 4, Δ}	{3, 4, Δ}	—	{3, 4, Δ}	Job11: No solution exists
{3, 2, Δ}	{3, 2, Δ}	—	{3, 2, Δ}	Job12: At least one solution exists
{1, 1, Δ}	{1, 1, Δ}	Job4: A solution exists	{1, 1, Δ}	Job13: No solution exists
{1, 2, Δ}	{1, 2, Δ}	—	{1, 2, Δ}	Job14: At least one solution exists
{1, 4, Δ}	{1, 4, Δ}	—	{1, 4, Δ}	Job15: No solution exists
{1, 3, Δ}	{1, 3, Δ}	—	{1, 3, Δ}	Job16: No solution exists

[0077] Although certain devices and methods have been described above in terms of the illustrative embodiments, the person of ordinary skill in the art will recognize that other embodiments, examples, substitutions, modification and alterations are possible. It is intended that the following claims cover such other embodiments, examples, substitutions, modifications and alterations within the spirit and scope of the claims.

1. A method comprising:
identifying a set of input vectors within an input space of a design;
dividing the set of input vectors into multiple subsets of the input vectors;
determining, using a first satisfiability solver, that at least one of the subsets does not provide a solution to a set of input constraints; and
searching, using a second satisfiability solver, the multiple subsets for unique input vectors that satisfy the set of input constraints, wherein the searching excludes a sub-searching of the at least one of the subsets.
2. The method of claim 1, wherein the set of input constraints make up a coverage model of the design
3. The method of claim 2, wherein the design is a micro-circuit design.
4. The method of claim 1, wherein the searching and the determining occur simultaneously.

5. The method of claim 4, wherein the searching excludes the sub-searching of at least one of the subsets by aborting the sub-searching of the at least one subset in response to the determining that the at least one of the subsets does not provide the solution to the set of input constraints.

6. The method of claim 1, wherein the first satisfiability solver is a numerical solver and the second satisfiability solver is a heuristic solver;

7. The method of claim 6, wherein the heuristic solver searches one of the subsets in hierarchical stages, each stage using a different random seed.

8. One or more computer readable media comprising computer executable instructions, that when executed by one or more computers, cause the one or more computers to:

- identify a set of input vectors within an input space of a design;
- divide the set of input vectors into multiple subsets of the input vectors;
- determine, using a first satisfiability solver, that at least one of the subsets does not provide a solution to a set of input constraints; and
- search, using a second satisfiability solver, the multiple subsets for unique input vectors that satisfy the set of input constraints, wherein the search excludes a sub-search of the at least one of the subsets.

9. The one or more computer readable media of claim 8, wherein the set of input constraints make up a coverage model of the design

10. The one or more computer readable media of claim 9, wherein the design is a microcircuit design.

11. The one or more computer readable media of claim 8, wherein computer executable instructions, when executed by one or more computers, cause the one or more computers to perform the determining and the searching simultaneously.

12. The one or more computer readable media of claim 11, wherein the search excludes the sub-search of the at least one of the subsets by aborting the sub-search of the at least one

subset in response to the determining that the at least one of the subsets does not provide the solution to the set of input constraints.

13. The one or more computer readable media of claim **8**, wherein the first satisfiability solver is a numerical solver and the second satisfiability solver is a heuristic solver;

14. The one or more computer readable media of claim **13**, wherein the heuristic solver searches one of the subsets in hierarchical stages, each stage using a different random seed.

15. An apparatus comprising:

one or more processors; and

one or memory storing software, that when executed by

one or more processors, cause the apparatus to:

identify a set of input vectors within an input space of a design;

divide the set of input vectors into multiple subsets of the input vectors;

determine, using a first satisfiability solver, that at least one of the subsets does not provide a solution to a set of input constraints; and

search, using a second satisfiability solver, the multiple subsets for unique input vectors that satisfy the set of input constraints, wherein the search excludes a sub-search of the at least one of the subsets.

16. The apparatus of claim **15**, wherein the set of input constraints make up a coverage model of the design

17. The apparatus of claim **16**, wherein the design is a microcircuit design.

18. The apparatus of claim **15**, wherein the search and the determine occur simultaneously.

19. The apparatus of claim **18**, wherein the search excludes the sub-search of the at least one of the subsets by aborting the sub-search of the at least one subset in response to the determining that the at least one of the subsets does not provide the solution to the set of input constraints.

20. The apparatus of claim **15**, wherein the first satisfiability solver is a numerical solver and the second satisfiability solver is a heuristic solver;

[text missing or illegible when filed]

* * * * *